



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Feature Model-based Software Product Line Testing

VOM FACHBEREICH 18
ELEKTRO- UND INFORMATIONSTECHNIK
ZUR ERLANGUNG DER WÜRDE
EINES DOKTOR-INGENIEURS (DR.-ING.)
GENEHMIGTE DISSERTATION

VON

DIPL.-INFORM. SEBASTIAN OSTER

GEBOREN AM

22. NOVEMBER 1981 IN HAMBURG

REFERENT: PROF. DR. RER. NAT. A. SCHÜRR
KORREFERENT: PROF. DR. RER. NAT. U. GOLTZ

TAG DER EINREICHUNG: 23. AUG. 2011
TAG DER MÜNDLICHEN PRÜFUNG: 16. DEZ. 2011

HOCHSCHULKENNZIFFER D17
DARMSTADT 2012

Dedication

For the ones I love the most.
Oya, Mikael, Alisha, Barbara, and Wolfgang.
Your love is inspiring and motivating.

ABSTRACT

Software Product Line (SPL) engineering is a popular approach for the systematic reuse of software artifacts across a very large number of similar products. SPLs are gaining widespread acceptance and various domains already apply SPL engineering successfully to address the well-known needs of the Software Engineering community, such as increasing quality, saving costs for development and maintenance, and decreasing time-to-market. The central aspect of systematic reuse is the concept of variability often leading to an enormous number of possible products

In the automotive sector, we are increasingly encountering a situation where a single electronic control unit (ECU) may be instantiated in at least 10,000 different ways, and the software running on a network of more than 50 ECUs in a single car may exist in millions of different configurations. As a result, we are confronted with a world where any instance of a certain brand of car possesses a unique configuration of the embedded software of all its ECUs. At a first glance, SPL engineering seems to result in a major benefit for the automotive sector as well as for other industrial domains, allowing the combination of mass production and product customization. The reverse side of the coin is the challenge to assure the quality of each derivable product of the SPL e.g. via testing activities.

Testing all products of an SPL individually is generally not feasible. This thesis contributes an approach to significantly reduce the test effort for SPL testing. Faults are likely to be revealed at execution points where features exchange information with other features or influence one another. Therefore, a criterion for test adequacy is to cover as many interactions among different features as possible, thus, increasing the probability of finding bugs based on feature interaction. We present a novel approach to generate a representative set of products of the SPL required for comprehensive coverage of feature interactions. The features of the feature model are combined in products using a combinatorial strategy assuring a certain degree of feature interaction coverage. For this purpose we introduce a graph transformation-based algorithm to translate the feature model into a binary constraint satisfaction problem and also an algorithm combining constraint solving techniques with a feature combination strategy to generate the representative set of products. A mapping between the feature model and a reusable test model allows for generating test cases for each product automatically. We implemented our approach as a tool chain and applied it to three different industrial SPLs for evaluation purposes. The results suggest that with our approach higher coverage of feature interactions is achieved at a fraction of cost and time when compared with the state-of-the-art approach of testing all derivable products.

SFTWARE-Produktlinien (SPL)-Engineering ist ein populärer Ansatz für die systematische Wiederverwendung von Software-Artefakten über eine große Menge von sich ähnelnden Produkten. Die industrielle Verwendung von SPLs steigt zunehmend. Eine Vielzahl von verschiedenen Domänen setzen bereits erfolgreich auf SPL Engineering mit der Zielsetzung, die bekannten Software-Engineering-Anforderungen, wie steigende Qualitätsansprüche, Kostenreduzierung für Entwicklung und Wartung und der Verkürzung der Time-to-Market, zu erfüllen. Zentraler Bestandteil der systematischen Wiederverwendung ist das Konzept der Variabilität, welches häufig zu einer enorm hohen Anzahl von möglichen Produkten führt.

Im Automobilbereich werden die Entwickler zunehmend mit der Situation konfrontiert, in der ein elektronisches Steuergerät (ECU) bis zu 10.000 verschiedene Konfigurationsmöglichkeiten besitzt. Bei durchschnittlich mehr als 50 ECUs in einem Auto ergeben sich Millionen von möglichen Konfigurationen. Folglich hat bereits jedes Auto bei bestimmten Modellen einiger Automobilhersteller einen individuellen Softwarestand. Auf den ersten Blick scheint das SPL-Engineering einen immensen Vorteil für den Automobilbereich und viele andere Industriezweige zu bringen, da dieser Ansatz die Kombination von Massenproduktion und Produkt-Customizing bietet. Die Kehrseite der Medaille ist die Herausforderung, die Qualität jedes einzelnen Produktes sicherzustellen, z.B. durch Tests.

Das individuelle Testen aller Produkte ist in der Regel nicht möglich. Diese Arbeit stellt einen Ansatz zur signifikanten Reduktion des Testaufwands für SPLs vor. Fehler treten erwartungsgemäß dort auf, wo Features Informationen/Daten austauschen oder sich beeinflussen. Die Abdeckung solcher Interaktionen erscheint als geeignetes Kriterium für adäquates Testen, um möglichst viele interaktionsbasierte Fehler zu finden. In dieser Arbeit wird eine Methodik zur Generierung einer repräsentativen Menge von Produkten vorgestellt. Die Features des Featuremodells werden gemäß einer kombinatorischen Test Strategie zu Produkten kombiniert, um einen gewissen Grad an Feature-Interaktionen abzudecken. Dazu wird das Featuremodell durch Graphtransformation in ein binäres Constraint Satisfaction Problem übersetzt, welches dann durch eine Kombination von Constraint Solver und kombinatorischem Testen gelöst wird, um die repräsentative Menge von Produkten zu generieren. Ein Mapping zwischen den Features und einem wiederverwendbaren Testmodell erlaubt die automatische Generierung von produktspezifischen Testfällen für jedes Produkt der SPL. Der gesamte Ansatz wird als Werkzeugkette implementiert und anhand von drei verschiedenen industriellen SPLs evaluiert. Die Ergebnisse zeigen, dass mit diesem Ansatz eine höhere Abdeckung von Fehlern durch das Abdecken von Feature Interaktionen erreicht werden kann. Gleichzeitig werden Kosten und Zeit im Vergleich zum standardmäßigen SPL-Testverfahren, dem Testen einzelner Produkte, reduziert.

PUBLICATIONS

Some ideas and figures within this thesis have appeared previously in the following publications. A complete list of my publications is available online under <http://www.es.tu-darmstadt.de/mitarbeiter/sebastian-oster>.

Book Chapters:

- S. Oster, A. Wübbeke, G. Engels, A. Schürr: **Model-Based Software Product Lines Testing Survey**, in: J. Zander, I. Schieferdecker, P. Mosterman (eds.): *Model-based Testing for Embedded Systems*, CRC Press/Taylor & Francis, 339–381, 2011.

Articles:

- G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, Y. Traon: **Pairwise Testing for Software Product Lines: Comparison of Two Approaches**, in: *Software Quality Journal - Special issue on Quality Engineering for Software Product Lines*, Heidelberg: Springer Verlag, 2011.

Refereed Conference And Workshop Papers:

- S. Oster, M. Lochau, M. Zink, M. Grechanik: **Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations**, in: *Proceeding of the 15th International Software Product Line Conference (SPLC'11), FOSD 2011 Workshop Proceedings*, 2011.
- S. Oster, I. Zorcic, F. Markert, M. Lochau: **"MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing"**, in: K. Czarnecki, U. Eisenecker (eds.), *4th International Workshop on Variability Modelling of Software-Intensive Systems*, Namur, Belgium, 79–82, 2011.
- S. Oster, F. Markert, P. Ritter: **"Automated Incremental Pairwise Testing of Software Product Lines"**, in: *Proceedings of the 14th International Software Product Line Conference*, Heidelberg: Springer Verlag, 196–210, 2010.
- S. Oster, P. Ritter, A. Schürr: **"Featuremodellbasiertes und kombinatorisches Testen von Software-Produktlinien"**, in: *Proceedings of the Software Engineering 2010; GI-Edition Lecture Notes in Informatics, Vol. 159*, Gesellschaft für Informatik, 177–188, 2010.
- S. Oster, A. Schürr: **"Architekturgetriebenes Pairwise-Testing für Software-Produktlinien"**, in: *Software Engineering 2009 Workshop: Produkt-Variabilität im gesamten Lebenszyklus*, 131–134, 2009.

- S. Oster, F. Markert, A. Schürr: "**Integrated Modeling of Software Product Lines with Feature Models and Classification Trees**", in: Proceedings of the 13th International Software Product Line Conference (SPLC'09), MAPLE 2009 Workshop Proceedings, Springer Verlag, 75–82, 2009.
- A. Schürr, S. Oster, F. Markert: "**Model-Driven Software Product Line Testing: An Integrated Approach**", in: Proceedings of the 36th International Conference on Current Trends in Theory and Practice of Computer Science; Lecture Notes in Computer Science (LNCS), 112-131, 2009.
- S. Oster, A. Schürr, I. Weisemöller: "**Towards Software Product Line Testing using Story Driven Modelling**", in: U. Aßmann, J. Johannes, A. Zündorf (eds.), Proceedings of the 6th International Fujaba Days, Technische Universität Dresden, 48-51, 2008.

CONTENTS

I INTRODUCTION	1
1 INTRODUCTION	3
1.1 Motivation	4
1.2 Contribution	6
1.2.1 Benefit for Industry	8
1.2.2 Classification	8
1.3 Outline	8
1.4 Running Example: Body Comfort System	12
II BACKGROUND AND RELATED WORK	15
2 SOFTWARE PRODUCT LINES	19
2.1 Software Product Line Fundamentals	21
2.1.1 Variability and Reuse	23
2.1.2 Variability Mechanism and Binding	24
2.2 Variability Representation	25
2.2.1 Feature Models	26
2.2.2 Alternative Representations	29
2.3 Challenges of SPL Engineering	30
3 SOFTWARE TEST	31
3.1 Terms and Definitions	32
3.2 Testing Techniques	34
3.2.1 Static and Dynamic Techniques	34
3.2.2 Test Levels	36
3.3 Model-based Testing	36
3.4 Combinatorial Testing	38
3.5 Quality of Tests	42
3.5.1 Coverage Criteria	42
3.5.2 Mutation Testing	44
4 RELATED WORK: SOFTWARE PRODUCT LINE TESTING	45
4.1 Reuse-Techniques	47
4.1.1 Model-based Testing of SPLs	48
4.1.2 Variability and Mapping	56
4.1.3 Lessons Learned - Reuse-Techniques	58
4.2 Subset-Heuristics	58
4.2.1 Representative Sets	59
4.2.2 Combinatorial Interaction Testing	59
4.2.3 Subset of Test Cases	60
4.2.4 Lessons Learned - Subset-Heuristics	61
5 SUMMARY PART II	63

III	CONCEPT AND THEORY	65
6	FEATURE MODEL-BASED TESTING	69
6.1	Formal Definition of Feature Models	70
6.2	Feature Interaction Coverage	72
6.3	Feature Model as Propositional Formula	77
7	COMBINATORIAL SPL TESTING	81
7.1	Constraint Satisfaction Problem	82
7.2	Feature Model to CSP Transformation	84
7.2.1	Flattening	85
7.2.2	Value Extraction	95
7.2.3	CSP Extraction	97
7.2.4	Cartesian Flattening	98
7.3	Subset Extraction	99
7.3.1	Subset Selection Pseudocode	101
7.3.2	Subset Extraction for BCS	107
8	MODEL-BASED SPL TESTING	109
8.1	150% Test Model	111
8.2	100% Models and Test Case Generation	112
8.3	Discussion of the 150% Test Model Philosophy	114
9	SUMMARY PART III	117
IV	IMPLEMENTATION AND EVALUATION	119
10	MOSO-POLITE TOOL CHAIN	123
10.1	Feature Modeling	124
10.1.1	Feature Modeling Tool Comparison	124
10.1.2	pure::variants Overview	126
10.2	Combinatorial Subset Selection	127
10.2.1	CSP Extraction	129
10.2.2	MOFLON - SDM	132
10.2.3	Subset Extraction	137
10.3	Model-Based Test Case Generation	139
10.3.1	Rational Rhapsody	139
10.3.2	ATG	142
10.3.3	Alternative Tooling	142
10.4	Testing the Implementation	143
10.4.1	Systematic Validation	143
10.4.2	Semantical Equivalence	144
10.4.3	Consistency	145
10.4.4	Completeness	145
10.4.5	Evaluation of Efficiency	146
11	EVALUATION	149
11.1	BCS Case Study	149
11.1.1	Pairwise Feature Interaction Coverage	154
11.1.2	Limitations of Pairwise	156

11.2 Adam Opel AG - Instrument Panel Cluster	157
11.2.1 Feature Model Extraction	157
11.2.2 Results of the IPC Case Study	158
11.3 Danfoss - Automation Drive	159
11.3.1 Options and their Configuration	161
11.3.2 Results of the Automation Drive Case Study	161
11.4 Threats to Validity and Comparison with Related Approaches	163
12 SUMMARY PART IV	169
V CONCLUSIONS	171
13 CONCLUSIONS	173
13.1 Discussion	175
13.2 Future Work	177
BIBLIOGRAPHY	179
APPENDIX	199

LIST OF FIGURES

Figure 1	Overview of our contribution	7
Figure 2	Overview of Parts	9
Figure 3	Overview of Chapters	11
Figure 4	The four ECUs of the BCS [MLD ⁺ 09]	13
Figure 5	Overview - Part II	17
Figure 6	Software Product Line development process by [PBvdL05] . .	21
Figure 7	Feature model of the BCS	27
Figure 8	Classification of test techniques	34
Figure 9	V-Model	36
Figure 10	Model-based testing overview [PP04]	37
Figure 11	Categories of combinatorial testing strategies	40
Figure 12	Conceptional process model for Software Product Line testing [OWES11]	50
Figure 13	CADeT process model [OWES11]	51
Figure 14	ScenTED process model [OWES11]	52
Figure 15	Hartmann process model [OWES11]	53
Figure 16	Model checking process model [OWES11]	54
Figure 17	Reusing state machines process model [OWES11]	55
Figure 18	Overview - Part III	68
Figure 19	Feature model of our running example	73
Figure 20	Relations in logic	78
Figure 21	Cross-tree constraints in logic	79
Figure 22	Feature model to CSP transformation	85
Figure 23	Subtree for flattening	85
Figure 24	Transformation rule pulling up an <i>optional</i> child beneath a <i>mandatory</i> parent	86
Figure 25	Transformation rule pulling up an <i>optional</i> child beneath an <i>optional</i> parent	87
Figure 26	Transformation rule pulling up an <i>optional</i> child beneath an <i>alternative</i> parent	87
Figure 27	Transformation rule pulling up an <i>optional</i> child beneath an <i>or</i> parent	87
Figure 28	Transformation rule pulling up a <i>mandatory</i> child beneath arbitrary parents	88
Figure 29	Transformation rule pulling up an <i>alternative</i> child beneath arbitrary parents	89

Figure 30	Transformation rule pulling up an <i>or</i> child beneath arbitrary parents	91
Figure 31	Subset of the BCS feature model	92
Figure 32	Step 1 of the flattening	92
Figure 33	Step 2 of the flattening	92
Figure 34	Step 3 of the flattening	93
Figure 35	Procedure to proof the semantical equivalence	94
Figure 36	Rules for value extraction for <i>mandatory</i> (1), <i>optional</i> (2), <i>alternative</i> (3), and <i>or</i> (4) features	96
Figure 37	The FMCSF of the BCS subset	97
Figure 38	Comparison of the flattening approaches of an <i>alternative</i> parent with <i>alternative</i> child elements	98
Figure 39	Activity diagram subsetExtraction()	103
Figure 40	Activity diagram of buildConfiguration()	104
Figure 41	Activity diagram of fillConfiguration()	106
Figure 42	Statechart test model of <i>ManPW</i>	112
Figure 43	Mapping example by means of the BCS alarm system functionality	113
Figure 44	Detailed overview of the MoSo-PoLiTe approach	117
Figure 45	Overview - Part IV	122
Figure 46	Overview of the MoSo-PoLiTe tool chain	124
Figure 47	pure::variants overview [pG11]	127
Figure 48	BCS in pure::variants	128
Figure 49	Tool chain dataflow	129
Figure 50	Data structure of the eclipse plugin	129
Figure 51	Data structure of the FMCSF	131
Figure 52	Steps of the SDM-based flattening handling a <i>optional</i> child node	133
Figure 53	Detailed view of the SDM transformation processing an <i>optional</i> child	134
Figure 54	Runtime Flattening for manual Java implementation and SDM-based	136
Figure 55	Visualization of the runtime delta	137
Figure 56	Subset of configurations in pure::variants	138
Figure 57	Three of 17 configurations of the combinatorial subset in pure::variants	138
Figure 58	BCS statechart example in Rhapsody	140
Figure 59	Examples for mapping features to statechart artifacts in Rhapsody	141
Figure 60	Example for deriving a 100% test model out of a 150% test model	142
Figure 61	Valid and invalid pairs of features	145
Figure 62	Statistics - Pairwise	147

Figure 63	Statistics - Threewise	147
Figure 64	Comparison of pairwise and threewise feature interaction coverage for seven randomly generated feature models	148
Figure 65	Outline of the evaluation chapter	151
Figure 66	BCS case study	152
Figure 67	T-wise coverage of k-wise interactions	156
Figure 68	Feature model of the Opel SPL case study	158
Figure 69	Feature model of the Danfoss use case	160
Figure 70	Feature model describing the Options-FM	162
Figure 71	Comparison to Jenny	164
Figure 72	MoSo-PoLiTe process model	166
Figure 73	Comparison - MoSo-PoLiTe and related model-based SPL testing approaches	173
Figure 74	Appendix: Metamodel for the SDM implementation	210
Figure 75	Appendix: Iteration through the feature model including pattern matching	211
Figure 76	Appendix: SDM of the flattening process	212
Figure 77	Appendix: Transformation rule for <i>mandatory</i> children	213
Figure 78	Appendix: Transformation rule for <i>optional</i> child features	214
Figure 79	Appendix: Transformation rule for child features within <i>or</i> <i>alternative</i> groups	215

LIST OF TABLES

Table 1	All possible combinations of the input parameters p1, p2, and p3	39
Table 2	Comparison of combinatorial algorithms able to handle constraints	41
Table 3	Comparison of model-based test approaches for SPL [OWES11]	57
Table 4	Comparison of the different subset-heuristics	61
Table 5	Categories of feature interactions	76
Table 6	Categories of feature interactions	100
Table 7	Example for the pairsToCover HashMap	102
Table 8	Possible configurations of the BCS-small feature model	108
Table 9	Feature model tool comparison	126
Table 10	Feature models from SPLOT research [SR11]	146

Table 11	Overview of the configurations generated by MoSo-PoLiTe	150
Table 12	Triples of feature interaction for (1) { <i>CLS, AutPW, RCK</i> }	153
Table 13	Triples of feature interaction for (1) { <i>CLS, ManPW, RCK</i> }	153
Table 14	Triples of feature interaction for { <i>CLS, AutPW, LED</i> }	154
Table 15	Feature interaction coverage under pairwise in the statechart	154
Table 16	BCS statechart coverage under pairwise	158
Table 17	Correlation between test coverage and testing effort by using MoSo- PoLiTe	163
Table 18	Execution Times for pairwise generation on feature models. Key: CP = Cell Phone, SH = Smart Home, MT= Model Transformation, ES= Electronic Shopping	165
Table 19	Test generation characteristics	166
Table 20	Comparison of model-based test approaches for SPL [OWES11] including MoSo-PoLiTe	168

LISTINGS

Listing 1	Subset extraction pseudocode - I	103
Listing 2	Subset extraction pseudocode - II	104
Listing 3	Subset extraction pseudocode - III	105
Listing 4	Flattening pseudocode	130
Listing 5	Generated Source Code for child <i>optional</i> step 4	134
Listing 6	Manually written source code for child <i>optional</i>	135
Listing 7	Appendix: Generated Source Code for child <i>optional</i>	217

Part I

INTRODUCTION

INTRODUCTION

Software Product Line (SPL) engineering is a popular approach for the systematic reuse of software artifacts across a very large number of similar products. SPLs are gaining widespread acceptance and various domains already apply SPL engineering successfully to address the well-known needs of the Software Engineering community, including increasing quality, saving costs for development and maintenance, and decreasing time-to-market [CNo1]. SPLs offer a systematic reuse of software artifacts within a range of products sharing a common set of features (i.e., units of functionality) [Grioo]. According to IEEE a feature is *a distinguishing characteristic of a software item (e.g. concerning its performance, portability, or functionality)* [ANS83].

The central aspect of systematic reuse is the concept of variability. This concept provides the possibility to define particular artifacts (features) for the entire SPL as not necessarily being part of each product. Variability specifies the point at which features are selected in combination with other features [CE00, pages 93-95]. The stakeholders of the SPL generally define where variability occurs and decide which features are variable e.g. optional or alternate. Variability leads to a combinatorial explosion of possible products from one SPL. For instance, an SPL with around 200 optional features may lead to over 10^{60} different products.

The concept of Product Lines is not new and engineers in various domains, such as the automotive sector, have adopted this concept of development for the last few decades, to benefit from the advantages that SPL engineering offers. However, with regard to software, systematic reuse, including variability concepts, is still challenging and a relatively new problem for the automotive sector. One main reason for the increasing need of systematic variability management in software is the fact that the majority of modern features in a car are based on software. Thus, variability moves from mechanics and hardware to software [Bos05].

Due to variability, we are increasingly encountering a situation in the automotive sector where a single electronic control unit (ECU) may be instantiated in at least 10,000 different ways, and the software running on a network of more than 50 ECUs in a single car may exist in millions of different configurations. As a result, we are confronted with a world where any instance of a certain brand of car possesses a unique configuration of the embedded software of all its ECUs. At a first glance, this situation seems to result in a major benefit for the automotive sector and other industrial domains, allowing the combination of mass customization and the ability to produce individual, customer-specific configurations.

The reverse side of the coin is the challenge to assure the quality of each derivable product of the SPL. Here, two major problems occur:

- The number of configurable products is almost unlimited and
- test engineers generally have a limited period of time to execute tests for a specific product; thus the question arises about what should/can be tested during that limited period to ensure sufficient coverage and to find faults within the system.

Thus, **testing each individual product thoroughly is not feasible** under the above mentioned circumstances. Engineers from the Software Engineering Community and from various industrial domains are seeking methods to reduce the effort of testing SPLs.

1.1 MOTIVATION

In single system development, testing consumes between 25% and 50% of the development costs [LL05]. In SPL engineering, testing consumes even more resources, due to variability [McGo1]. Furthermore, testing is even more critical because a fault within a certain functionality can spread over thousands or even millions of products which reuse this functionality.

Faults are likely to be revealed at execution points, where features exchange information with one another or influence each other [Bin99, page 557]. Thus, *interacting features* is a foundation of a fault model for SPLs. A feature interaction occurs when one or more features modify or influence the behavior of other features [JZ98]. Covering all possible feature interactions is one possible criteria for test coverage for an SPL, which is an important metric of software quality [ZHM97], since it indicates thoroughness of testing. Achieving higher coverage is correlated with the probability of detecting more defects [POC93, NA09, Kim03, CL05] and increasing software reliability [MLBK02, CLW96]. Even though it is agreed that coverage alone may not always be a strong indicator of software quality [KBP01, page 181], it is a general consensus that achieving higher coverage is desirable for gaining confidence in software [POC93, CLW96].

One industrially used procedure is to generate a product of the SPL under test including all features—a so-called 150% product. Testing this product would then result in testing the entire SPL. Or more specifically, this would be equal to testing all features in interaction with each other. However, testing a 150% product has two major drawbacks:

- A 150% configuration can rarely be generated because features may exclude each other and are not combinable or executable within one configuration.
- This procedure is not effective since it does not cover situations when features have to function without each other. Fault hiding is then a significant issue.

Thus, to ensure the functionality of the entire SPL, every possible feature combination would need to be tested. We will refer to this technique as N-wise testing, where N is the number of features within the SPL. Unfortunately, N-wise testing would result in testing every possible product; this is often not feasible.

Another option is to test a subset of possible products. Instead of testing all possible products of an SPL, and thus all potential interactions, only a subset is selected for testing purposes. According to Scheidemann [Scho7], decreasing the number of configurations for testing purposes allows for the configurations to be tested individually. Scheidemann names the following success stories for testing SPLs with a limited number of configurations:

- driver assistance SPL of Robert Bosch GmbH [PBvdL05]
- home entertainment SPL of Philips [Treo4]
- Avaya Labs SPL [GLRW04]

Thus, it seems to be promising to reduce the number of configurations for testing purposes and this is the central topic of this thesis. The following research questions (RQs) motivate this thesis:

1. Can we test an entire SPL without testing each possible product?
2. How can we apply lessons learned from the software testing community to decrease the test effort for SPLs?
3. How should we systematically select a subset of possible products for testing purposes with regard to feature interaction?
4. What is the effect of testing for feature interactions in the SPL context?
5. How can we reuse test artifacts to test the product of an SPL?
6. What is necessary to support industry with a suitable tool chain in relation to these RQs?

One possible solution is to calculate a subset of products which is representative for the entire SPL under test by means of a certain coverage criterion e.g. feature interaction coverage. We will refer to this technique as T-wise testing where T is smaller than N or, more simply, as **subset-heuristics**. Achieving higher test coverage when testing for feature interactions means that testers produce various products of the SPL with which they can execute tests for features and their interactions. The faster these testers achieve higher coverage of feature interactions, the lower the cost of testing [Kan96], because testers can concentrate sooner on other aspects of testing, for example, performance and usability testing. Higher coverage is always better but 100% coverage is generally not achieved, especially when testing large-scale applications [POC93, Mar99].

The following assumption led us to the idea of applying/exploring combinatorial test techniques to select an appropriate T [OMR10]:

The commonalities and variability within an SPL are frequently represented by features. Those features can be interpreted as parameters in the SPL engineering process. Therefore, it seems to be promising to have a look at lessons learned in the field of test case reduction based on parameterization. Combinatorial testing and especially pairwise testing are well-known approaches in that category.

Various approaches exist for applying combinatorial testing to SPLs, which we will discuss within this thesis. However, those approaches either do not describe in detail how to apply it to SPLs or the described method is very complex to handle with regard to calculation time and scalability. According to the best of our knowledge we published the first systematic approach to apply combinatorial testing to SPLs in [OSWo8].

1.2 CONTRIBUTION

This thesis introduces the **Model-based Software Product Line Testing** (MoSo-PoLiTe) concept for SPL testing and provides an implementation of this concept by means of a tool chain. This concept generates a subset of product configurations for testing purposes that meet the previously mentioned T-wise feature interaction coverage criteria. Features from the feature model of the SPL under test are combined in product configurations using a T-wise combinatorial strategy [GOA05, CGMC03, KLK08]. Feature models are frequently used to represent the common and variable features within an SPL, including dependencies and constraints that determine which feature selections are appropriate for a product configuration.

Thus, MoSo-PoLiTe provides a systematic approach for generating a set of configurations covering T-wise feature interaction on the basis of the feature model. Testing this set of configurations is equivalent to T-wise testing of the entire SPL.

Furthermore, the MoSo-PoLiTe concept includes a model-based test approach through a reusable test model of the SPL, which allows for the generation of configuration-specific test cases. The intuition behind our approach is that we use statecharts as a test model for our SPL, whose states and transitions are then mapped to features in feature models. Thus, we are able to interrelate feature model coverage and test model coverage. This is, according to the best of our knowledge, the first contribution to do so.

Figure 1 depicts a schematic overview of the contribution of this thesis—MoSo-PoLiTe. A feature model serves as a basis for configuring the instances of the SPL under test. We will introduce a combinatorial test algorithm which will calculate a set of configurations covering all T-wise feature interactions (left-hand side). Furthermore, the feature model can be used to configure a reusable test model via a mapping between features and test model artifacts (right-hand side). As a

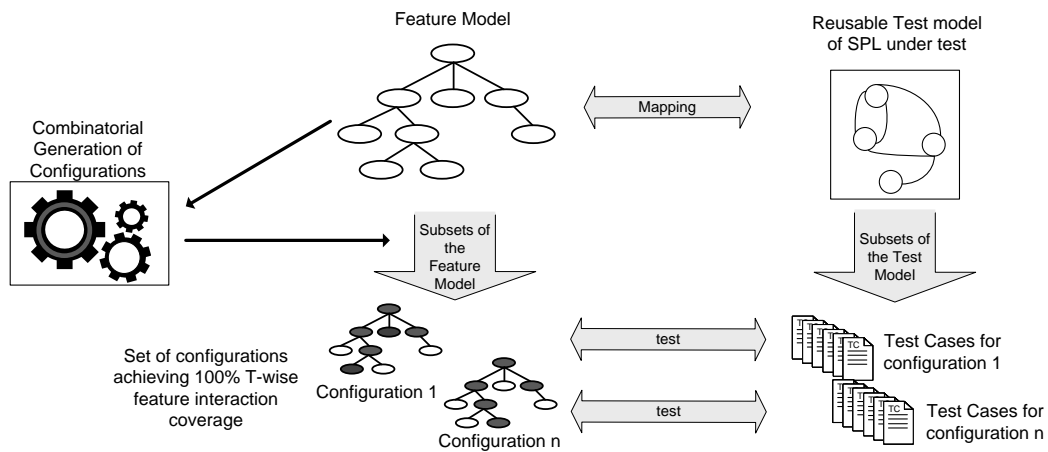


Figure 1: Overview of our contribution

result, MoSo-PoLiTe generates a set of configurations covering all T-wise feature interactions, including test cases for each configuration of the test set. This diagram and also more detailed versions of it will be used throughout this thesis to describe the outline and content of various chapters or to provide a schematic overview of the MoSo-PoLiTe functionalities.

Summarizing this thesis contributes:

- An approach for applying combinatorial testing to SPLs, including the automatic generation of product configurations satisfying T-wise feature interaction coverage.
- A graph transformation-based algorithm to translate a feature model into a binary constraint satisfaction problem (CSP).
- A feature model generator to automatically generate feature models that we use to examine the scalability and efficiency of our algorithm.
- An approach to automatically generate test cases for arbitrary products of an SPL on the basis of model-based coverage criteria.
- A tool chain providing the implementation of our approach that combines T-wise combination and model-based testing.
- Finally, we discuss the results of our industrial case studies and discuss the potentials and limitations of the T-wise testing approach focusing on pairwise feature interaction.

1.2.1 *Benefit for Industry*

The concepts, algorithms, and tools developed within this thesis were applied in several cooperations with various research groups and industrial partners. All those cooperations have shown that industry can reduce the effort for testing SPLs. This gain can be traced back to the step-by-step description of our concepts and the integration into the SPL development process.

There are multiple benefits of our approach: it is lightweight, since it does not require any intervention by programmers; it is tractable, since it uses combinatorial design to shrink the input feature space while maintaining a diversity of feature interactions; and it is scalable, since it can be used on feature models with hundreds of optional features [OMR10]. Furthermore, our approach does not depend on the implementation of the SPL and it does not require any source code or model analyzers.

1.2.2 *Classification*

We classify this contribution to address the following areas of research:

- Software Product Line Engineering
- Software (Product Line) Testing
- Combinatorial Testing
- Feature Modeling
- Model-Based Testing
- Requirement-Based Testing
- Feature Interaction

1.3 OUTLINE

This thesis is structured into five parts plus appendix. This structure intends to support and guide the reader. Figure 2 depicts a schematic overview of the thesis structure.

Part I: The **Introduction** contains the introduction, motivation and a brief summary of the contribution of this thesis. Research questions are used to emphasize the contribution of this thesis. This section answers the question about which domain of research is addressed (**and where**) within this thesis, **what** the thesis is about, **why** the topic addressed in this thesis is relevant in research and industry, and

who can benefit from its results. Furthermore, this section introduces the running example that is used throughout this thesis to explain fundamentals, theory, and concepts, as well as parts of our implementation.

Part II: The part **Background and Related Work** covers the fundamentals relevant for this thesis, beginning with an introduction to SPLs that explains the basic ideas and concepts. Then, software testing in general is described as the method for software quality assurance that we aim to apply to SPLs. First, a general introduction to testing is provided, followed by a detailed description of methods which are used extensively within this contribution, such as combinatorial testing and model-based testing. Subsequently, approaches for SPL testing are summarized and discussed in a chapter on related work.

Part III: The part **Concept and Theory** introduces the basic concepts of our approach. Thus, this part is dedicated to describing how combinatorial testing can be applied to SPLs and how this approach can be extended by model-based testing for test case generation.

Part IV: The part **Implementation & Evaluation** describes the implementation of our approach and introduces our tool chain that can be used to apply our approach in an industrial context. Thus, we are then able to collect data and experiences from our industrial partners, which we summarize in our evaluation.

Part V: This part concludes this thesis and provides an overview of current and future work. The research questions of Part I are discussed and new research questions are summarized to support and motivate further research.

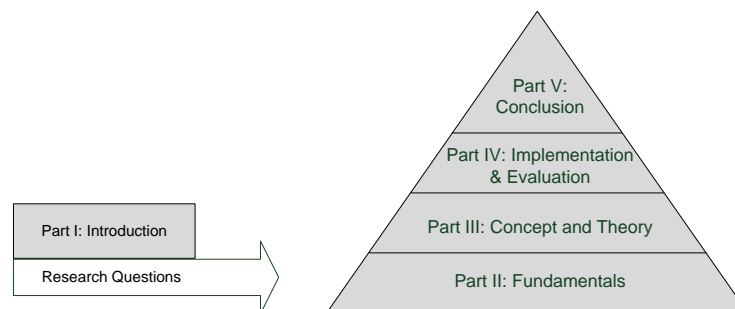


Figure 2: Overview of Parts

A more detailed outline of this thesis is depicted in Figure 3, providing an overview of chapters. The storyline of this thesis is depicted as a bold line on the right-hand side of Figure 3. This storyline starts off with the research questions and is supported by each part's summary. It ends in the conclusion of the thesis, where we discuss the research questions by reflecting on each part. In the following, we briefly describe the structure and content of each chapter:

Chapter 1 describes the general introduction of this thesis, its contribution, and outline. Followed by this detailed overview of chapters, we introduce the thesis's running example. This running example is an SPL from the automotive domain provided by one of our industrial partners.

Chapter 2 introduces the domain of application of our approach: Software Product Lines (SPLs). We provide details about the variability and reuse concepts within SPLs and focus on feature modeling, which is currently the most prominent approach for modeling variability and commonalities. Finally, we briefly summarize some challenges of the SPL development process to provide a critical view of our domain of research.

Chapter 3 is dedicated to software testing. Here, we provide a brief review of terms and techniques and introduce details about model-based testing and combinatorial testing. Furthermore, we describe approaches, including mutations and coverage criteria, which are used to rate the quality of tests.

Chapter 4 summarizes research activities, concepts, and approaches related to our contribution according to the best of our knowledge. This study focuses on SPL testing approaches and pays particular attention to model-based and combinatorial testing approaches for SPLs. The goals of this chapter are to (1) provide an overview of the state-of-the-art in SPL testing, (2) summarize lessons learned from other approaches, and (3) reveal unresolved issues in current approaches that we will address in our approach.

Chapter 5 summarizes the background and related work part of this thesis. There, we reflect on the state-of-the-art in SPL development and testing as well as feature modeling. Furthermore, we examine the impact of our study of related work on our research questions.

Chapter 6 is the first chapter within the concept and theory part. Here, we provide the reasoning about why we use the feature model for SPL testing purposes. We also prepare the ground for the following chapters by providing a formal definition of feature models and feature interactions.

Chapter 7 introduces the concept of our combinatorial test algorithm. Here, we introduce how combinatorial design can be applied to feature models and introduce the concept of graph transformation to translate a feature model, as defined in Chapter 6, into a binary constraint satisfaction problem. Furthermore, we discuss correctness and completeness of this transformation. Pseudocode and activity diagrams are used to describe the internal functionality of our combinatorial algorithm generating a subset of products.

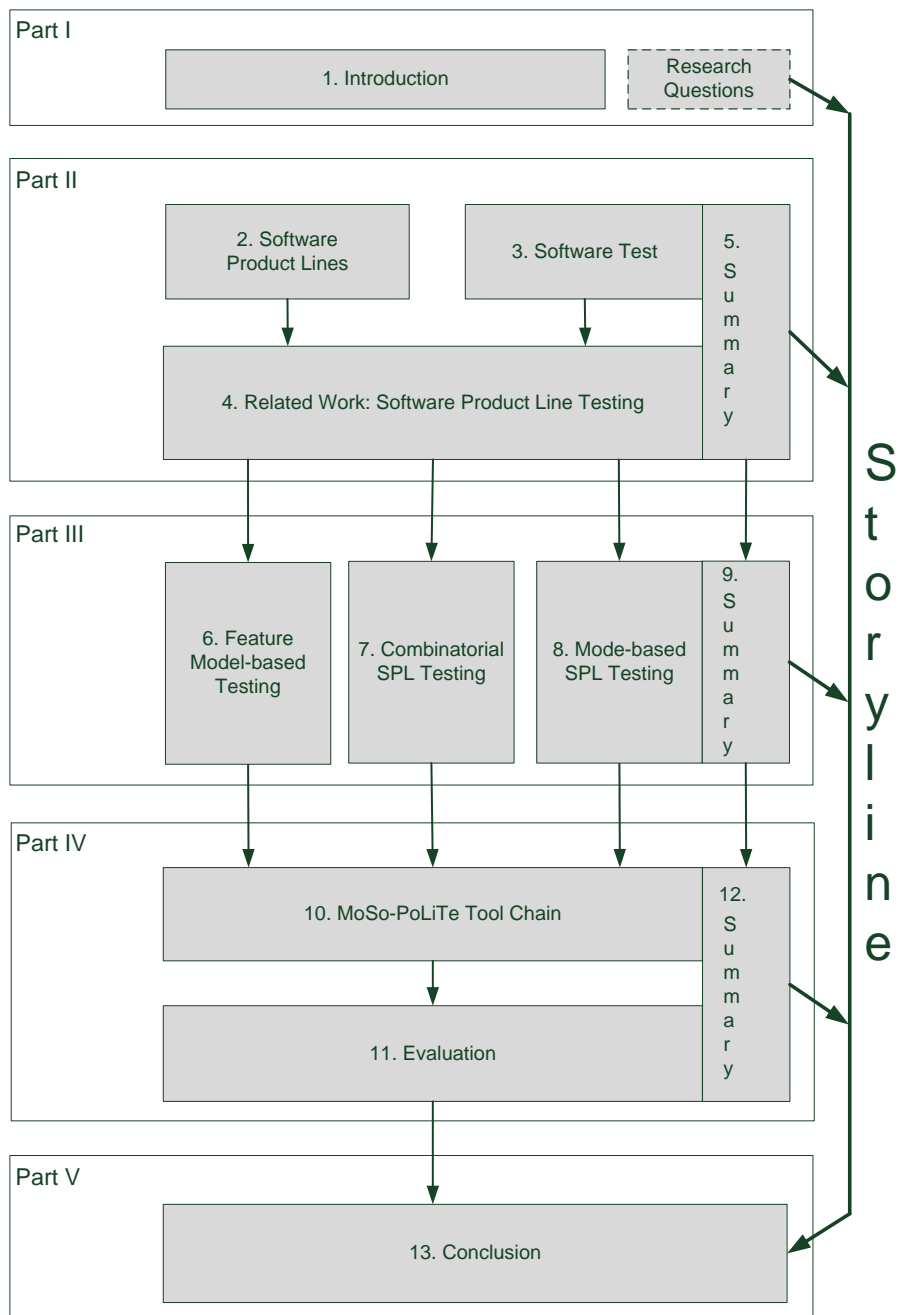


Figure 3: Overview of Chapters

Chapter 8 describes the concept of a reusable test model to generate test cases for an application derived from an SPL. For this purpose, a mapping between the features of the feature model and the elements of the test model is introduced, allowing the configuration of the test model, using the feature model. As a consequence, configurations automatically result in corresponding configuration specific test models which can then be used for test case generation purposes.

Chapter 9 summarizes the concept and theory part of this thesis. It reflects the MoSo-PoLiTe concept and prepares the ground for the implementation.

Chapter 10 describes the implementation of the MoSo-PoLiTe concept. First, the application of pure::variants within the MoSo-PoLiTe framework is described. Subsequently, we describe the implementation for combinatorial testing and model-based testing. Finally, we test our implementation to check whether it aligns with the previously described concept in the Chapters 6, 7, and 8.

Chapter 11 presents the evaluation of the MoSo-PoLiTe concept. Three industrial use cases are used to examine the results of MoSo-PoLiTe. Furthermore, we provide a theoretical discussion of the potentials and limitations of our approach, which can be seen as an extension of our evaluation.

Chapter 12 summarizes the Implementation and Evaluation part of this thesis. There, we recapitulate the implementation of MoSo-PoLiTe and examine to what extent the results of the evaluation affect our research questions.

Chapter 13 finally concludes this thesis and summarizes its contribution. The research questions are recapitulated and answers are provided. New research questions are formulated to motivate future research and discussions.

1.4 RUNNING EXAMPLE: BODY COMFORT SYSTEM

To clearly illustrate the contribution of this thesis, we make use of a sample SPL from the automotive domain, a simplified extract of a *Body Comfort System* (BCS) [MLD⁺09] which we adapted to become an SPL.

A BCS is a combination of several functionalities increasing the driver's comfort. A BCS consists of several ECUs that interact via CAN-Bus to realize functionalities such as:

- central locking control
- power window lift
- rearview mirror adjustment

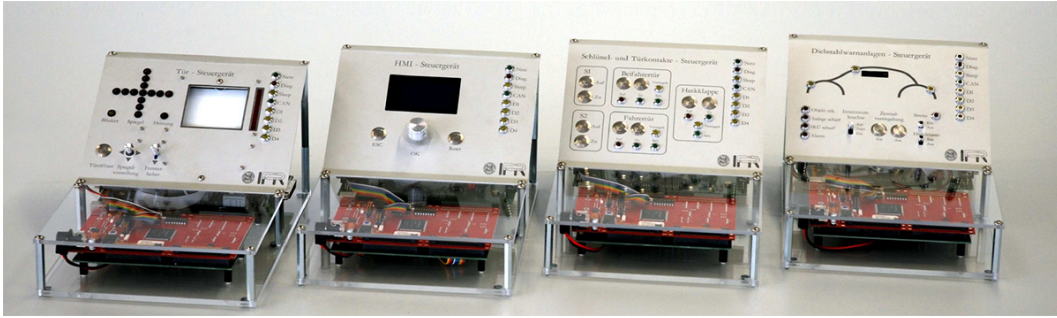


Figure 4: The four ECUs of the BCS [MLD⁺09]

- alarm system
- sideview mirror control
- monitor systems to display the BCS settings

The driver can interact and control these functionalities via a human machine interface (HMI) by means of switches, levers or touch screens. Our sample of the BCS serving as a running example contains an alarm system with an optional interior monitoring, a human machine interface (HMI) offering a passenger control interface, an electrical configurable exterior mirror that may also include a heating function, and a remote control key for the central locking system and the control the alarm system. Furthermore, every BSC includes a power window functionality, providing either a manual or an automatic window closing functionality. The automatic window closing function can be activated using the remote control key.

Figure 4 shows the four **Electronic Control Units** (ECUs) implementing the BCS functionality [MLD⁺09]. From left to right: Door Control ECU, HMI ECU, Key and Door ECU, Alarm ECU.

Part II

BACKGROUND AND RELATED WORK

OVERVIEW PART II

THIS part summarizes the fundamentals relevant for this thesis. Its structure is depicted in Fig. 5. The two basic topics underlying this thesis, namely Software Product Lines and software testing, are described in Chapters 2 and 3 respectively. The Software Product Lines chapter focuses on variability and reuse and how these concepts are managed. A clear understanding of how variability and reuse effects the different configurations of an SPL is vital for our test approach because we aim at identifying a small set of configurations for testing purposes. Afterwards, a chapter dedicated to software testing provides relevant fundamentals about single system testing. Furthermore, this chapter briefly summarizes the concepts of model-based testing and combinatorial testing since those approaches are used extensively within this thesis. A related work section then provides a summary and a discussion of the state-of-the-art approaches and concepts in Software Product Line testing.

Part II

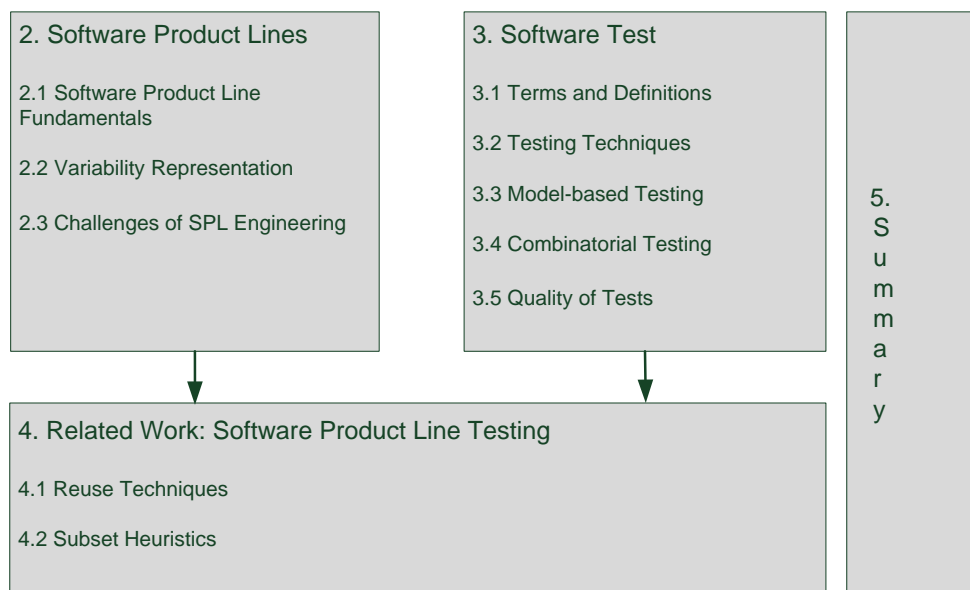


Figure 5: Overview - Part II

Software Product Line (SPL) Engineering is one of the most prominent approaches to improve reusability of software within a range of products sharing a common set of features [Bos00, CN01, PBvdL05]. Clements even elevates SPLs to the dominant software development paradigm of the 21st century [Cle99]. Due to the systematic reuse, the time-to-market, as well as costs for development and maintenance are expected to decrease, while the quality of the individual products is expected to increase. Thus, SPLs are able to support developers to rapidly develop customized products instead of developing new products from scratch repeatedly. The concepts behind the product line paradigm are not new. Domains such as the automotive industry have successfully applied product line development for several years. However, this was generally hardware-based reuse.

The software development industry has recently adopted the idea of product line engineering. Among these, the automotive industry itself focuses on variability and reuse of software due to the fact that software within vehicles now controls the majority of functionalities. Furthermore, various configurations are based on customer choices. Especially when analyzing the development of embedded systems it is evident that the product line paradigm has gained increasing importance in the course of developing products for particular domains, such as control units in the automotive domain [TH02, GKPR08].

The SPL concept is based on the idea of combining the advantages of custom software and off-the-shelf software. Off-the-shelf software is implemented for a wide area of applications or domains used by various types of end users. Typical examples for off-the-shelf software are: Microsoft Windows, MAC OS, Open Office, and Microsoft Office. Those software products are typically used by many end users.

In contrast to the aforementioned concept, custom software is developed for the solution of a certain problem. Thus, developing such a software is more expensive since the market for a specific solution is rather small. Examples for custom products would be a program that can only be installed on a certain platform that is only of interest to a very small community. SPLs intend to develop software products for a certain domain addressing various customers and to provide problem specific products by using variability concepts.

Clements and Northrop [CN01] from the Software Engineering Institute at the Carnegie Mellon University define SPLs as follows:

Definition 1 (Software Product Lines). *A Software Product Line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*

Bosch describes SPLs [Bos01] as:

Definition 2 (Software Product Lines cont'd). *A Software Product Line consists of a product line architecture, a set of reusable components, and a set of products derived from the shared assets.*

These definitions contain much valuable information:

- Members of an SPL share a common set of features and are based on an SPL architecture.
- These commonalities have to be managed explicitly.
- These commonalities define the domain or their market segment in which the SPL is intended to be used.
- An SPL is always limited to a certain domain. Otherwise it is not possible to identify, manage or reuse commonalities.

The term *reuse* is a very important keyword for SPLs. The importance of reuse within Software Engineering was already recognized in 1968 at the NATO conference in Garmisch-Partenkirchen. There, McIlroy stated that software should be reused across different software projects [McI68]. He proposed dividing the market into component developers, component users, and a market place, where the users can purchase the desired components.

This statement is very impressive for two reasons:

- This idea came up for the first time at the same conference where the term Software Engineering was initially mentioned as an engineering process to develop software intensive systems. Thus, the idea of software reuse is as old as the idea of its area of application.
- The idea of reuse seems to be very intuitive. However, it took several years to come up with concepts such as classes, modules, and inheritance focusing on small scale reuse within a certain product development, and then more years to come up with concepts and ideas to implement explicit large scale reuse in Software Engineering covering amongst others software requirements, architectures, implementation, documentation, and tests.

2.1 SOFTWARE PRODUCT LINE FUNDAMENTALS

The development of an SPL affects all activities in the development process. With regard to an SPL development process, we follow the definitions of Pohl et al. [PBvdL05], since their process model is a quasi-standard in the SPL community. In contrast to single system development, the development process is separated into domain engineering and application engineering. The former allows for the development of the common and variable parts of the product line (development for reuse), while the latter allows for the development of an application (also called product) considering the use of individual parts (development with reuse). Thus, an SPL has to align with common and variable requirements [WL99]. Common requirements have to be fulfilled by every product of the SPL and variable requirements describe product individual functionalities. The two levels, domain and application engineering, are again separated into five activities (see Figure 6):

- Product Management
- Requirements Engineering
- Design
- Realization
- Testing

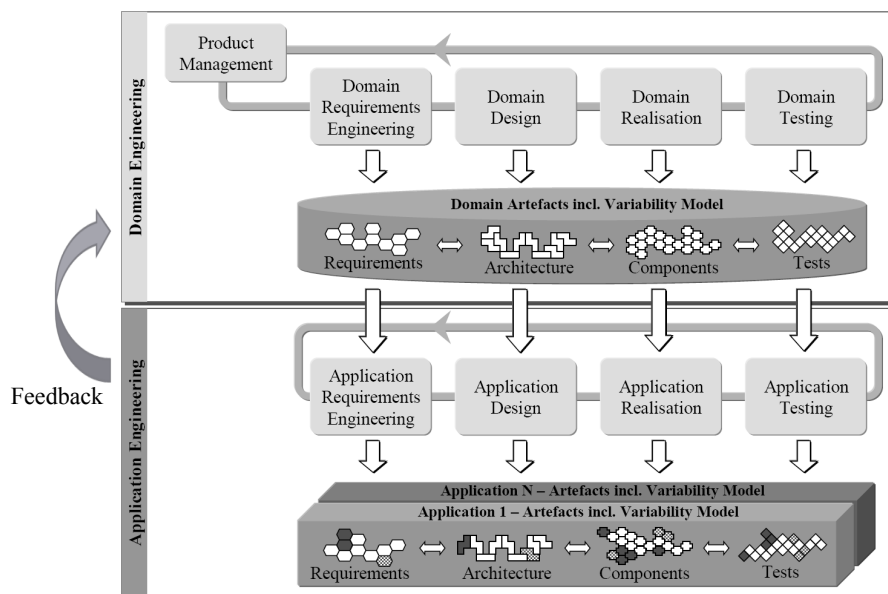


Figure 6: Software Product Line development process by [PBvdL05]

The activity *product management* is a domain engineering-specific activity, supporting the evolution of the entire SPL. The product management controls the other

four activities in domain engineering, starting with the development of the common and variable requirements, their design, realization, and finally testing. Domain testing provides testing of common and variable parts without deriving a product from the SPL. The output of every activity are artifacts including variability.

Definition 3 (Variability). *Variability describes the possibility to define particular artifacts of domain engineering as not necessarily being part of each application of the product line.*

Variability appears within artifacts and is defined by variation points.

Definition 4 (Variation Point). *A Variation Point specifies the type and the location of a variability. Every Variation Point has at least one variant.*

Each activity in application engineering is supported by the corresponding activity in domain engineering. The development artifacts of the domain level are the basis for the development on the application engineering level by deriving the required common and variable parts from the particular domain engineering activity for the corresponding application engineering activity. Deriving refers to the process of binding the variability within the particular artifact to become application specific artifacts, also called variants.

Definition 5 (Variant). *Variants describe the possible assignments for a variation point. A variant can again be a variation point.*

This derivation is performed for every activity in application engineering. The next step, in each application engineering activity, is the development of application individual parts. The common and variable parts are illustrated in Figure 6 by the blank symbols, while the product individual parts are depicted by the filled symbols. The derivation of the common and variable parts for each activity is illustrated by a long unshaded arrow. Further information and variants of the SPL development process can be found in [PBvdL05], [CN01], and [Gom04]. In contrast to the original picture of the SPL development process provided by Pohl, we have added an additional **feedback** link from application engineering to domain engineering. We think that this link is vital for the SPL development process because experiences and requirements that emerged within application engineering might be of interest in the domain engineering phase. Several approaches for variability modeling that model variation points and their corresponding variants exist that will be discussed in Section 2.2.

2.1.1 Variability and Reuse

According to Gilles van Gorp and Jan Bosch in the preface of [vGB03] variability is:

...the ability of a software system or artifact to be changed, customized or configured for use in a particular context. A high degree of variability allows for the use of software in a broader range of contexts, i.e. the software is more reusable.

Variability is one of the key challenges in SPL engineering allowing the derivation of different products. It is, therefore, the main difference compared to single system development. The variability is based on the stakeholder and environmental requirements. With regard to the automotive sector, the choice between a diesel and otto engine is a stakeholder driven variability, whereas the variability of having the steering wheel on the left or the right side of the car is an environmental requirement depending on the country where the car will be driven.

Another typical example for variability is provided by online-car-configurators that are provided by every automotive brand. In our running example, variability is, for example, the choice between **manual power window** and **automatic power window**. A product can either contain a manual or an automatic power window.

According to Pohl et al. we can differentiate between the following different kinds of variability [PBvdL05]:

- **Variability in Space** describes the existence of an object in different shapes at the same time. This is the typical SPL variability because it describes the fact that a certain variation point can bind different variants within different products.
- **Variability in Time** is the existence of an object in different shapes at different times. This variability actually describes the evolution within SPL engineering.

These types of variability can again be categorized as:

- **Internal Variability** is variability within the SPL that is hidden from the customer.
- **External Variability** is visible to the customer and is actually selected by the customer.

However, variability can only have a positive effect if appropriate techniques for reuse are available. Reuse in Software Engineering is generally daily business—but unfortunately this is often interpreted as library (re)use and copy & paste of code fragments, parts of the specification or documentation. Development for reuse is part of the domain engineering phase within the SPL engineering process. To systematically apply variability and reuse, variability mechanisms are required as well as an appropriate method to model and manage variability within the SPL engineering process.

2.1.2 Variability Mechanism and Binding

The selection and integration of the variable artifacts for product derivation is called binding. The binding time specifies the time in the development process at which the variability is resolved.

Generally, the following binding times are used during the life-cycle of a product:

- Product planning: Variability can already be resolved during product planning by choosing certain properties for the planned product.
- Design: Variability within the architecture of the SPL can already be resolved during the design phase.
- Compile time: During compilation variabilities can be resolved using e.g. preprocessors.
- Installation: During the installation or flashing, for example via variability in the parametrization.
- Startup: Variability can be resolved at system startup, for example different program modes can be selected at system startup.
- Run time: Variability can be resolved at run time, for example changing from multi to single player mode within a smartphone game [OWES11].

In the automotive sector, variability within a vehicle is generally resolved as follows [Scho7]:

1. During the product planning and design phases, as mentioned above.
2. Subcontractors develop components that they sell to different OEMs. Thus, those components include variability which is at least partly resolved when delivered to an OEM.
3. On the assembly line variability is resolved by selecting the ECUs to be integrated within a certain car. These ECUs control e.g. certain sensors or actors required for functionalities such as power window.
4. These ECUs are then flashed or parameterized to assure a certain functionality.
5. After a vehicle is assembled, it still includes variability that can be resolved during runtime e.g. turning off the Electronic Stability Control (ESC).

Especially in the automotive sector, bound variants do not necessarily remain static but, in some cases, variability might be rebound. For example, the software on an ECU might get re-flashed or hardware might be exchanged.

The latest time for binding is limited by a so-called variability mechanism describing the method of how the variability is bound. Prominent variability mechanisms are:

- **Code generation from Models:** Models are frequently used to model the behavior of an SPL or parts of it. Thus, these models contain variability concepts that are generally resolved when code or test cases are generated out of those models. Modeling approaches in Matlab Simulink e.g. [Weio8], statecharts e.g. [SVo8], and activity diagrams e.g. [RKPR05] are frequently used.
- **Aspects:** Aspects realize the separation of concerns and map variant code to a certain aspect. During the so-called weaving process, the code of the selected aspects is merged with the common code [Grioo].
- **Preprocessors:** Preprocessors are used to realize conditional compilation depending on which code fragments are required for a certain product. Tools such as pure::variants [pG11] and Gears [Kruo8] use this technique.
- **Inheritance:** A superclass represents all possible variants of an SPL and each variant is then modeled as a class that extends the interface by adding or overriding operations of the superclass. This is, for example, described in the PLUS method [Gom04].
- **Information Hiding:** Variability is distributed across different versions of components. Interfaces then hide the concrete realization [DGP⁺04].
- **Parametrization/Substitution:** Variability is represented by parameters that are resolved to derive a configuration. For example, Bertolino and Gnesi use parameters within use cases [BG03]. Placeholders need to be replaced by concrete parameters. In [RKPR05] the authors use substitution to derive configurations.
- **Frames:** Common and variable code can be separated by using frames as described in [Bas97].
- **Generative Programming:** Czarnecki et al. introduced the generative programming Software Engineering paradigm to model SPLs. Variabilities can be encapsulated into customizable/abstract features [CEoo]. Those abstract features can then be used to generate concrete and specialized components based on customer requirements.

2.2 VARIABILITY REPRESENTATION

Since variability management is one key challenge in SPL engineering and testing, we need to discuss different approaches for modeling variability. Variability representations must be capable of representing variability and commonalities and should be able to describe the effects/consequences of variability.

2.2.1 Feature Models

The Webster's Dictionary of American English defines a Feature as 1) an important part or characteristic and 2) something offered as a special attraction. The Software Engineering community and especially the SPL-community refine and adapt these definitions constantly and the definition of a feature differs in some publications. For our definition, we use the standard in the software test documentation in [ANS83].

Definition 6 (Feature). *A feature is a distinguishing characteristic of a software item (e.g. concerning its performance, portability or functionality).*

Feature models are frequently used to describe the variable and common parts within an SPL. A feature model consists of features, each representing a “logical group of requirements” [Bos00] or, as defined in [CHE05b]: “a system property that is relevant to some stakeholder”. The purposes of feature models are summarized in [HST⁺08] as follows:

1. to describe feature commonalities and variabilities,
2. to graphically represent dependencies and constraints between features, and
3. to specify permitted and forbidden combinations of features.

For this purpose, notations have been defined so that researchers, managers, and clients are usually able to read and interpret feature models easily. Of course, feature models are not able to capture *all the types* of important properties of an SPL; therefore, they are complemented by other development artifacts such as natural language descriptions, function network diagrams, executable models or even code fragments. These artifacts are then mapped to the corresponding features by means of traceability relationships.

Feature models were introduced in [KCH⁺90] as part of FODA (Feature Oriented Domain Analysis), combining a hierarchical decomposition of features into subfeatures with the definition of *mandatory*, *optional*, and *alternative* features using different kinds of relations. A *mandatory* feature is always part of the product if its parent feature is selected. An *optional* feature can be part of a product if its parent is selected. If the parent feature of an *alternative* group is selected, exactly one feature has to be chosen for product derivation. Binary *require* and *exclude* constraints describe further cross-tree constraints between features. The hierarchical structure of feature models, the relations, and constraints determine which feature combinations are allowed to be assembled to products. Furthermore, in FODA, the feature model serves the following purposes: A feature model

- contains all system requirements of the customer and end user,

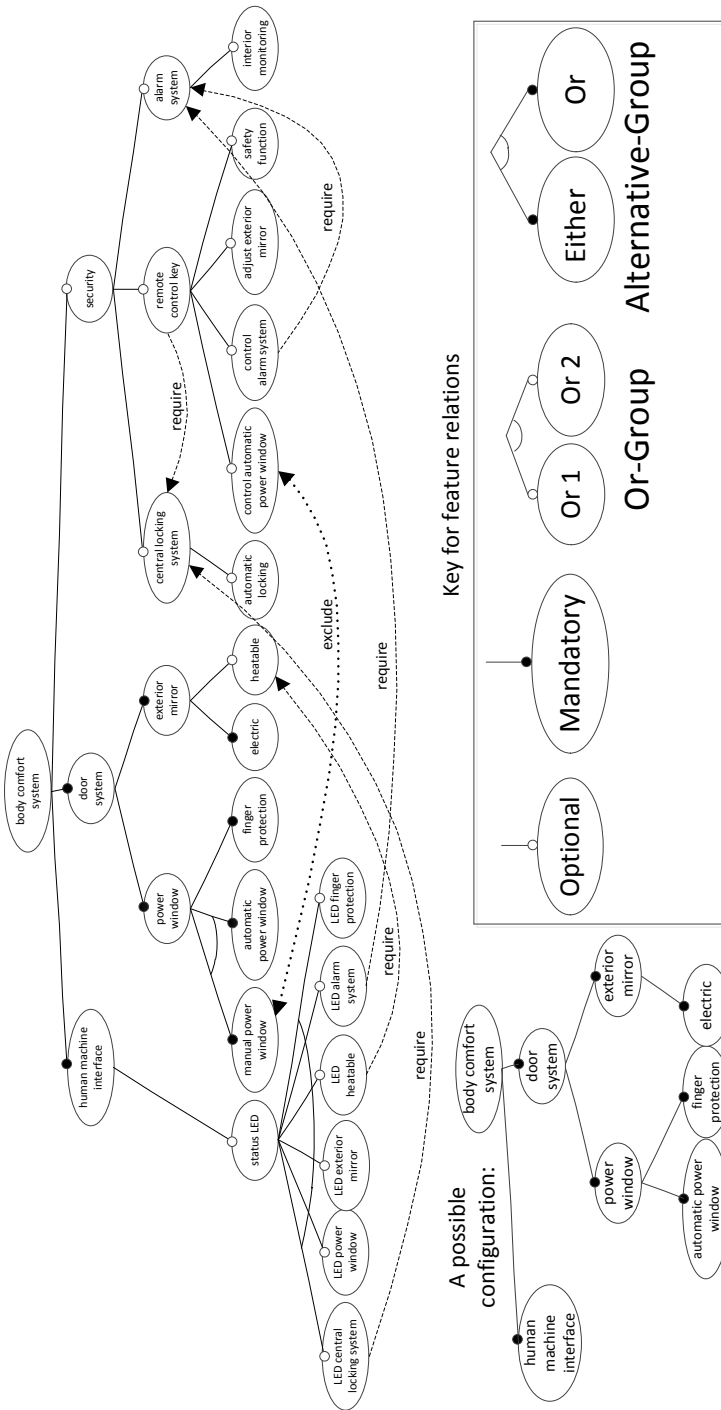


Figure 7: Feature model of the BCS

- supports the communication between users and developers, and
- is used as a starting point for the development of the architecture.

Figure 7 depicts the feature model of the BCS-SPL that serves as a running example throughout this thesis. The four different relations are explained at the bottom right corner. A node marked with a black filled circle represents a *mandatory* feature. A *mandatory* feature such as **door system** is always included within a product of the SPL if its parent node is included. Nodes marked with a white filled circle are *optional* features that can be selected for product derivation; e.g. **central locking system**. Groups of features are marked via a connecting line. A group of features with black-filled circles represents an *alternative* group; e.g. **manual power window, automatic power window**. A group of features with white-filled circles represents an *or* group. For product derivation at least one feature of an *or* group has to be included if the parent feature is included; e.g. **status LED**. Furthermore, *require* and *exclude* constraints restrict the product space of the BCS-SPL. For example, an *exclude* constraint hinders the combination of **manual power window** and **control automatic power window** and a *require* constraint demands that **central locking system** is selected if **remote control** is selected. The BCS feature model allows for deriving 11,616 different product configurations.

Since the initial introduction of feature models within the FODA feasibility study, further extensions have been introduced, improving precision and expressiveness and including, amongst others, cardinalities, probabilities, and weighting of features. Cardinalities can be employed to formulate how many instances of a feature may be integrated within a product [CHE05a]. Probabilities state that a certain feature is more likely to be used than another one [CSWo8]. Weights can be used to represent cost factors of features, thus supporting the engineers in building products appropriate for a certain budget [WDS09]. For a detailed summary of extensions of FODA feature models, see [CHE05b]. The approach in this thesis supports FODA feature models extended by an *or* group. For product derivation, at least one element of an *or* group has to be selected if its parent node is chosen to be part of the product. The following notions should be clarified:

Definition 7 (Configuration). *A configuration is a selection of features for product instantiation. Thus, a configuration only allows feature combinations that could lead to a product and do not violate relations and constraints between features.*

Definition 8 (Product). *A product is a selection of features including their implementation resulting in a product ready to be used.*

2.2.2 *Alternative Representations*

Feature models are the most prominent approach for describing variability within an SPL. In the following, we provide a brief overview of alternative modeling approaches.

2.2.2.1 *OVM*

Our definition of variation points and variants is based on the orthogonal variability model (OVM) proposed by Pohl et al. [PBvdL05]. Variation Points identify locations in development artifacts that contain variability. Furthermore, they support the specification of variability because they are able to provide variable characteristics.

Pohl et al. call their modeling approach orthogonal because variation points capture variability from different abstraction levels such as requirements, design or implementation. As in feature models, constraints between variation points can be modeled using *require* and *exclude* constraints. The OVM supports *mandatory*, *optional*, and *alternative* relations but, according to our knowledge, no *or* group.

Please note that the approach addressed in this thesis can also be based on OVMs, because feature models and OVMs are semantically equivalent [MPH⁺07]. The argument that feature models are only capable of describing variability on the feature level is not valid because various approaches exist which map features to all kinds of development or test artifacts of an SPL [FOS11].

2.2.2.2 *Decision Models*

Decision Models can be used to incrementally derive a specific configuration, where each decision within the decision model represents a variation point. Each decision is a question and each answer is linked to a certain artifact. A decision can influence other decisions. Some of the most prominent representatives of this variability approach are: Synthesis [Bur93], Schmid and John [SJ04], Kobra [ABB⁺02], Dopler [DGR11], and VManage [SG02]. They share very few commonalities and provide very different concepts and formalisms. A survey and discussion of these approaches can be found in [SRG11]. From our point of view, Decision Models can be used to document the development of a feature model. For example, they can provide a textual description of decisions about why a certain feature is within an *alternative* group or why it excludes another feature. However, Decision Models do not seem to be capable of parameterizing an SPL and are thus not suitable for our test approach.

2.2.2.3 *Natural Language*

Thomas von der Maßen discussed the use of natural language to describe variability within his doctoral thesis. He reported that natural language could describe functional and non-functional requirements and could be understood easily by

developers and customers [vdMo7]. However, the major disadvantages that Thomas von der Maßen identified, thus disqualifying this approach for variability modeling, are:

- Natural language cannot be validated automatically in terms of consistency, completeness, and correctness.
- Natural language is not precise enough to describe variability.
- Automatic product derivation is not possible.

In [vdMo7], natural language was only discussed with regard to variability within requirements. Relating this variability representation to other artifacts, such as code fragments and test data, does not seem to be infeasible [vdMo7]. However, we would expect natural languages to compensate these drawbacks if they are restricted to allow some kind of formalization. From our point of view, natural language then would tend to become similar to Decision Models. Still, natural languages seem to be inappropriate for our test approach for the same reason as Decision Models.

2.3 CHALLENGES OF SPL ENGINEERING

In addition to the listed benefits, SPL engineering has several inherent disadvantages that need to be taken into account:

- The initial effort to introduce an SPL development process is very large.
- The internal structure and hierarchies within the company may need to be changed to provide development and management of reusable artifacts and for product derivation.
- The development for reuse does not result in a direct financial benefit. Developing for reuse and for future products takes time to produce income for the company.

These disadvantages prevent many companies from introducing an SPL development process. The reuse concept within SPLs affects requirements, business cases, tools, concepts and processes, architecture, components, tests, documentations, and many more. Changing or preparing these artifacts for reuse requires additional effort. According to [WL99], the return of an investment (ROI) will be reached within the development of the second or third product if all the employees involved are willing to accept the additional start-up effort.

Software Testing is one of several quality assurance techniques in Software Engineering. Software Quality is defined in the IEEE Standard Glossary of Software Engineering Terminology IEEE Standard 729-1983 [ANS83] as:

- *The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, conform to specifications.*
- *The degree to which software possesses a desired combination of attributes.*
- *The degree to which a customer or a user perceives that software meets his or her composite expectations.*
- *The composition of characteristics of software that determine the degree to which software in use will meet the expectations of the customer.*
- *Attributes of software that affect its perceived value, for example, are correctness, reliability, maintainability, and portability.*
- *Software quality includes fitness for purpose, reasonable cost, reliability, ease of use in relation to those who use it, design of maintenance and upgrade characteristics, and compares well against reliable products.*

Testing is one of the most prominent aspects of software quality assurance that can inform the stakeholders about the quality of the system. In this chapter, we first list several definitions and terms related to software testing in general, followed by a brief survey of test techniques. Afterwards, the two test methodologies, model-based testing and combinatorial testing, which play a major role in the remainder of this thesis, are explained. At the end of this chapter, those quality criteria for tests that we intend to use to rate and evaluate our test approach are described.

3.1 TERMS AND DEFINITIONS

Quality assuring methods can be categorized into verification, validation, and testing:

Definition 9 (Verification). *The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Contrast with: validation [IEE90].*

Definition 10 (Validation). *The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Contrast with: verification [IEE90].*

Definition 11 (Testing). *A test is an activity in which a system or component is executed under specific conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component [IEE90].*

Generally, testing is the execution of a certain software system with the intention of (1) finding faults and (2) comparing the implementation with the specified functionality, in order to find possible failures.

Definition 12 (Fault). *A flaw in a component or system that can cause the component or system to fail to perform its required function, e.g. an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system [vV07].*

Definition 13 (Failure). *Deviation of the component or system from its expected delivery, service or result [FP96].*

A fault generally leads to a failure and a fault is often the result of an error.

Definition 14 (Error). *A human action that produces an incorrect result [IEE90].*

With other words, the relationship between these three terms is as follows: An error of a developer or engineer may cause a fault within the implementation that

then leads to a failure of the system. For a list of causes of faults and consequences of failures, we refer to [Wei10]. Testing can be used for verification and validation purposes, depending on its field of application.

- If testing is used to compare the implementation with customer requirements, we can actually use it as a validation technique, because we can check whether the expected system was built.
- If testing is used to compare the implementation with the specified behavior of the system, we can use it as a verification technique, because we can check whether the system behaves as expected.

Nevertheless, testing is only a heuristics and not a formal method that is able to prove anything. One of the most frequently quoted statements in software testing was made by Edsger Dijkstra in [Dij70]:

Program testing can be used to show the presence of bugs, but never to show their absence!

Although this statement is 40 years old it still applies today, because it is still not possible to test the majority of software systems exhaustively. The combinatorial complexity of possible input parameters makes it generally impossible to test every possible combination. For SPLs, this problem is even greater, due to variability resulting in the ability to derive a huge number of different products.

We would like to clarify the following notions, which are used repeatedly in this thesis. These definitions are adopted from the International Software Testing Qualifications Board [vVo7].

Definition 15 (Test). *A set of at least one test case.*

Definition 16 (Test Case). *A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement [IEE98].*

Definition 17 (Test Suite). *A set of test cases that generally can be executed in a certain order.*

Definition 18 (Test Oracle). *A source to determine expected results to compare with the actual result of the software under test.*

3.2 TESTING TECHNIQUES

In software quality assurance different categories of techniques exist. Figure 8 presents a schematic overview according to [Lig09]. The first level is the differentiation between **static** and **dynamic** methods.

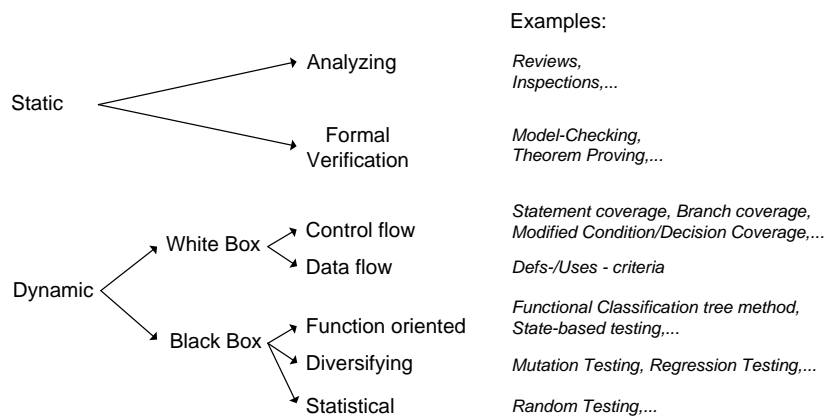


Figure 8: Classification of test techniques

3.2.1 Static and Dynamic Techniques

Static analysis collects information about the system without executing it to find faults within the implementation. Typical methods are: *Audits, Reviews, Walk-throughs*, and *Inspections*, which are executed on the source code of the system under test (SUT).

Various tools support static analysis on source code. Those tools examine [Walo1] e.g.:

- Syntactical Information (Complexity, Dependency Graphs)
- Semantical Information (Anomalies)
- Lexical Information (Length of Procedures)

Thus, static analysis focuses on complexity and semantical faults.

In formal verification, properties are defined on the basis of the system requirements or specification that the system must satisfy. For each property, formal

verification checks whether it holds or not. If the property does not hold, a situation is shown in which this property is violated. If all properties hold, it is proved that the system meets the requirements formulated by the properties. Complete formal verification of a system is rarely performed in Software Engineering, since the extraction of the system properties might be very time consuming. Nevertheless, the industrial importance and awareness of formal methods has increased during the past decade and is growing [Tre99]. Well-known examples of formal verification are Theorem Proving and Model Checking [Lig09].

Dynamic testing of software involves the selection of at least one software testing technique to:

- generate test cases
- execute test cases
- compare the result of the test execution with the expected system behavior defined in the system specification

A tester always starts with the construction of a formal or informal (depending on the test strategy) model that captures certain properties e.g. behavior of the SUT. This model is then used for test case generation, which can then be executed in the SUT.

Within this thesis, we intend to test an SPL without testing each possible product individually. Since our domain of research is more or less the embedded domain, the major advantage of testing is that testing can take Hardware/Software interrelationships into account, including effects on the environment and safety [Lig09]. Since testing is a dynamic technique, we focus on dynamic approaches.

Dynamic Software Testing can be either white box, black box or a combination of both. **White box** approaches take the internal structure of the SUT into account. Thus, the test engineer requires information about the internal behavior, the logic, and the structure. *Structural testing* is an often-used synonym for white box testing in the testing community. This information is typically presented via control flow and data flow graphs of the SUT. The control flow graph represents possible execution paths within the SUT, whereas the data flow graph describes the flow of data and relationships between variables in the SUT. The control flow and the data flow graph can both be used for test case generation [Lig09].

Black box testing techniques test the execution of the SUT against its requirements or specification. Therefore, it is also called *specification based testing*. The test engineer uses only the requirements or the specification for testing, without knowing the internal structure of the SUT. In abstract terms, valid and invalid inputs are used to check whether the expected output has been calculated. Strategies such as classification trees [GG93] and combinatorial testing [Bei90] provide heuristics to reduce the space of input parameters. On the one hand, this technique has the advantage of being close to realistic conditions but, on the other hand, one

important disadvantage is the lack of internal information, which is useful for generating tests and for fault tracing. For example, testing against the specification of a system does not guarantee that every line of code or part of the system is tested adequately.

Cai et al. recommend combining black box and white box techniques, since the correlation between fault detection and code coverage depends on the black box coverage criteria [CL05].

3.2.2 Test Levels

Testing can take place in every phase of the development process. These phases extend from requirement analysis to the implementation. A very prominent model representing the different phases of the development process is the V-Model [dIB97] shown in Figure 9. On the left, the development process is presented, whereas on the right, the location of the corresponding test levels is shown. The development process is described top-down, starting at the very left with the requirements analysis. On the basis of the requirements, the system specification is defined followed by a design phase. Afterwards, the system is split into units that are then implemented. The test levels are described bottom-up, starting with Unit Testing, which tests the different units of the system e.g. classes. The components of the design of the system are then tested via Integration Testing. The entire product is then tested against the system specification during system testing. Finally, the stakeholder requirements are tested via Acceptance Testing.

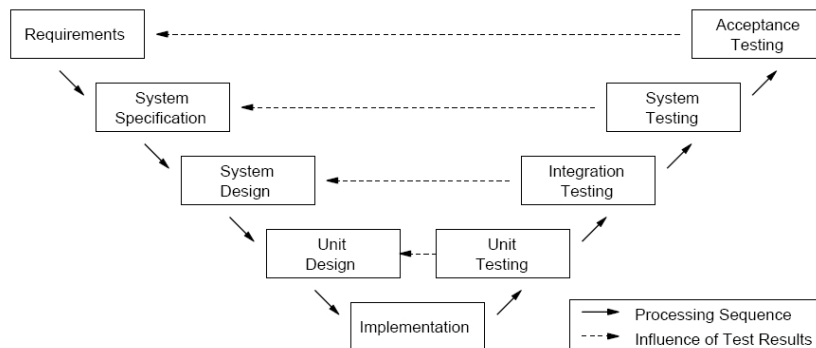


Figure 9: V-Model

3.3 MODEL-BASED TESTING

Model-based testing is derived from the concept of developing software based on models. This means explicitly modeling the structure and behavior of the system to a certain extent by using models on the basis of (semi-)formal modeling approaches.

The difference to non-model-based approaches is achieved by replacing the informal model with an explicit representation [PP04].

Definition 19 (Development Model). *A model represents the behavior and the structure of a system with regard to a certain degree of abstraction and a specific scope.*

Generally, development models are used for code generation and are then called implementation or execution models. For testing purposes, a so-called test model is used.

Definition 20 (Test model). *A test model represents the behavior and the structure of a system under test and is used for test case generation.*

For model-based testing, the test model usually represents system requirements and is used to derive test cases. In Figure 10, the basic idea of model-based development and testing is depicted. At the top-left corner, the informal customer requirements are illustrated as a cloud. From these requirements, three engineering artifacts are derived: first, the development model (top-right), the test model (center) and the test case specification. Both, the development model and the test model describe the behavior of the system under test. The development model is used for manual or automatic code generation for the implementation of the system under test, whereas the test model is used for test case generation, to test the implementation. The test model and the test case specification are used to generate test cases, where the test case specification defines the test procedure or test design to test against the requirements.

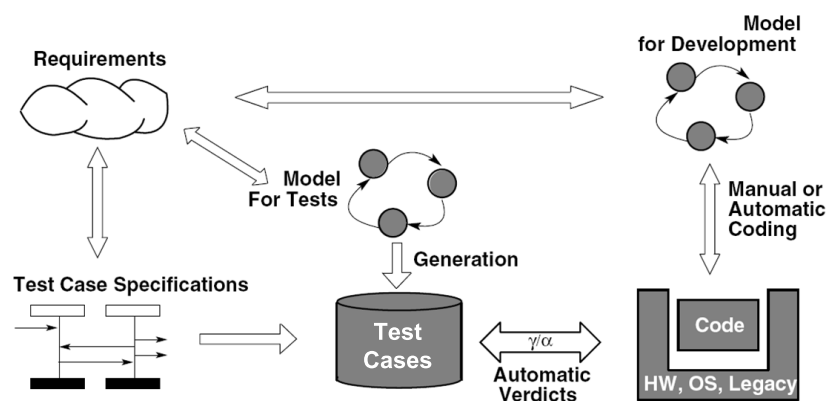


Figure 10: Model-based testing overview [PP04]

Various starting points for the model-based generation of test cases exist. In our example (Figure 10), there are two independent models used to either generate

test cases (test model) or to develop the code (development model). These models are both derived from the informal user requirements. Another version of the model-based testing approach uses only one model to derive both the code and the test cases. This version has a drawback concerning the significance of the test results: Faults in the implementation model are not found, because it is also the basis for the test generation if there is no separate test oracle identifying a wrong result as a fault. In the industrial context, model-based testing is usually executed on the development model, since the development of an additional test model is generally seen as too time-consuming. Further information on model-based testing can be found in [PP04] and [Rob00].

Test case generation within model-based testing is realized via the selection of certain paths through the model [Wei10]. All models serving as test models are kinds of graphs e.g. statecharts, activity diagrams or dataflow programs. The behavior of the system is thus the sum of all possible paths through the graph. A summary of algorithms for path selection is described in [Wei10]. Among those approaches are methods using random selection of paths, model checking, constraint solving, and graph search algorithms.

Model-based testing is a dynamic testing approach, since the generated test cases are then executed on the SUT. Whether model-based testing is white-box or black-box depends on the granularity of the test model. If the test model is simply based on the system requirements/specification, it can generally be interpreted as black-box testing. If the test model is closely related to the implementation or if an implementation model (which is used to generate the code of the implementation) is used to derive test cases, it can be interpreted as white-box testing.

The most crucial question concerning model-based testing is whether the effort for creating additional test models for test case generation can be justified. Furthermore, the challenging part of creating a test model is to keep it as simple as possible, to ease further development/maintenance, and to include sufficient information to support automatic test case generation simultaneous.

3.4 COMBINATORIAL TESTING

Combinatorial testing is a popular black-box testing method that decreases the effort for single system testing by reducing the number of test cases. To prove the correctness of a program, it needs to be tested with all combinations of possible input parameter values [Beig0]. Due to the complexity and size of the majority of products, testing all possible combinations of input parameter values is not feasible. A software with five different input parameters, where each parameter can be initialized in 10 different ways, would require $10^5 = 100,000$ different test cases to be validated. Using combinatorial testing, only certain combinations of parameter values serve as input. One of the best-known applications of combinatorial testing is the pairwise testing approach. This method is based on the assumption that the

majority of faults originate from a single parameter value or are caused by the interaction of two values [SM98]. Success stories related to combinatorial testing can be found in e.g. [DES⁺97], where the authors report that despite an enormous reduction of the number of test cases, block coverage could still be achieved.

The following example demonstrates pairwise testing for a method within the BCS case study, which has three boolean input parameters. Let us assume that these three parameters are **p1**, **p2**, and **p3**, to keep the example as simple as possible. Then, the eight input parameter combinations depicted in the first three columns of Table 1 would need to be tested.

	p1	p2	p3	p1&p2	p2&p3	p1&p3
1	true	true	true	true true	true true	true true
2	true	true	false			
3	true	false	true			
4	true	false	false	true false	false false	true false
5	false	true	true			
6	false	true	false	false true	true false	false false
7	false	false	true	false false	false true	false true
8	false	false	false			

Table 1: All possible combinations of the input parameters **p1**, **p2**, and **p3**

Applying pairwise testing, only pairwise combinations of the parameter values need to be covered. Thus, each value combination of **p1&p2**, **p2&p3**, and **p1&p3** needs to be tested with the value combinations (true, true), (true, false), (false, false), and (false, true). The value combinations 1, 4, 6, and 7 fulfill this requirement and thus these four combinations could be chosen for pairwise testing purposes. This set of value combinations is generally called **covering array** [CDS06]. We refer to each row within this covering array as a test set.

The difficulty for pairwise testing algorithms and, in general, for all combinatorial algorithms, is to cover a pair of input parameters exactly once to achieve a small test set of input values [LT98].

Grindal et al. provide a summary and categorization of combinatorial testing strategies in [GOA05]. Figure 11 depicts the categorization introduced in [GOA05], including additional categories that we have introduced as a consequence of recent publications.

Non-deterministic approaches include a certain factor which leads to the situation that, if the algorithm is executed twice, different results may be obtained. Deterministic approaches always result in the same covering array.

Examples of non-deterministic approaches are algorithms that use some kind of heuristics such as greedy or artificial life-based combination strategies, such as

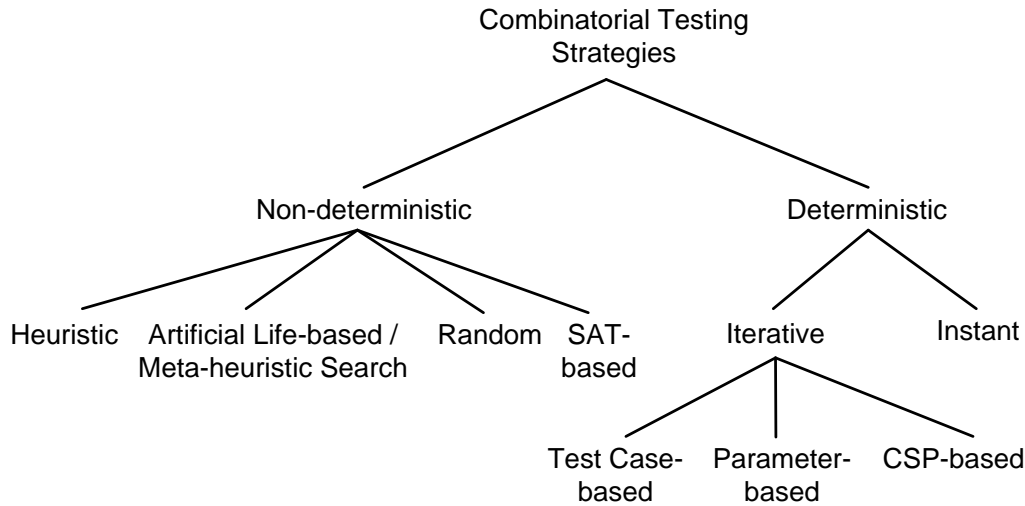


Figure 11: Categories of combinatorial testing strategies

genetic algorithms, or meta-heuristics such as simulated annealing [CDS07]. Algorithms that use some kind of randomizing component are also non-deterministic. At this point we like to add a so-called SAT-based category, which represents the approach introduced in [PSK⁺10]. Perrouin et al. apply T-wise combination based on an Alloy representation and apply SAT solvers to calculate the covering array. We will discuss this approach in more detail in the related work section.

Deterministic algorithms can be categorized into iterative approaches, where the covering array is calculated iteratively or via instant approaches, where the covering array is calculated at once. The iterative approaches can again be categorized into: Test-case-based, Parameter-based, and CSP-based (CSP = Constraint Satisfaction Problem) strategies. The CSP-based category is a new one that we introduce, since the algorithm that we introduce within this thesis is a deterministic, and CSP-based approach. The most prominent deterministic algorithm is the parameter-based algorithm IPO (In Parametric Order) which calculates the covering array in parallel [LT98]. When the situation occurs that a certain pair of values cannot be added to the current set within the covering array, a new test set is generated. Instead of discussing each category in detail, we focus on approaches within those categories that are capable of handling constraints between parameter values.

The most popular combinatorial tool that is capable of handling constraints is AETG (**Automatic Efficient Test Generator System**) [CDKP94]. The AETG is a commercial tool realizing T-wise coverage. T signifies that this algorithm supports, pairwise testing ($T = 2$), threewise testing ($T = 3$), up to n-wise testing, where T is equal to the number of parameters within the program under test.

AETG is a heuristic-based non-deterministic algorithm for combinatorial testing. It is based on the greedy algorithm and incrementally sets up the covering array until it fulfills the selected T-wise coverage. For each iteration, various solutions

for combining parameter values are generated. A weighting function is then used to select the best solution to continue with. The details of the algorithm are not available, since AETG is a commercial tool used in industry. We refer to [CDKP94] for further details.

Other than the AETG approach, very few existing standard combinatorial algorithms are capable of processing dependencies or constraints of any kind between parameters or their values. A summary of these approaches can be found in [CDS07]. Table 2 presents an overview of those algorithms that support some kind of constraint solving.

Tool	Citation	Constraint Handling	Category
AETG	[CDKP94]	Remodel	non-deterministic, heuristic
DDA	[CCT04]	Soft Only	non-deterministic, heuristic
Whitch	[Har05]	Simple/Expand	non-deterministic, heuristic
TestCover	[She11]	Remodel	deterministic, iterative, Test case-based
Simulated Annealing	[CCL03]	Soft Only	non-deterministic Meta-heuristic
PICT	[Cze06]	Full	non-deterministic, heuristic
mAETG_SAT	[CDS07]	Full	non-deterministic, heuristic
SA_SAT	[CDS07]	Full	non-deterministic, Meta-heuristic
Alloy	[PSK ⁺ 10]	Full	non-deterministic, Alloy-based

Table 2: Comparison of combinatorial algorithms able to handle constraints

Those algorithms realize different methodologies for constraint handling. Cohen et al. categorized these methodologies as follows:

- Remodel: AETG and TestCover are both commercial products, thus, the internal constraint handling cannot be examined. Both, require that the user re-models the input data into separate unconstrained inputs which are then combined at the end of processing.
- Expand: Whitch demands that the user expands the input by providing a list of all forbidden combinations.
- Soft Only: The deterministic density algorithm (DDA) and Simulated Annealing support only soft constraints. This means that the algorithm should avoid combinations violating those constraints but there is not a guarantee.

- Full: This category is the only category providing support for any arity of constraints.

Only a few approaches exist that provide full constraint support within their combinatorial testing algorithm. The PICT approach is commercial and it is not possible to obtain further details. mAETG_SAT, SA_SAT, and the Alloy-based approach all use SAT techniques to calculate the covering array considering constraints. However, we plan to apply combinatorial testing to feature models. Because feature models mainly make use of binary constraints, using a CSP seems to be the natural choice [WW09, Beno4] for a constraint-solving component within the combinatorial testing. Thus, we will introduce a CSP-based combinatorial algorithm that is capable of processing binary and, to some extent, n-ary constraints within this thesis. Furthermore, we will compare our CSP-approach with the Alloy-based SAT approach in the evaluation chapter of this thesis.

3.5 QUALITY OF TESTS

In this thesis, testing is used as a quality assuring technique for SPLs. These tests are meant to check the SPL for faults and failures. However, the quality of these tests also has to be checked. In the following, we introduce two methods that are frequently used to evaluate the quality of tests in the Software Engineering community.

3.5.1 Coverage Criteria

Coverage criteria are heuristics, just like tests themselves. They estimate the quality of test suites according to certain criteria, which state what the tests have covered with regard to e.g. the implementation, the requirements or models describing the behavior of the system. Test engineers use coverage criteria to assess whether the test activities are sufficient or not. In other words, coverage criteria can be used as test end criteria.

Achieving greater test coverage is correlated with the probability of detecting more defects [POC93] and increasing confidence in software [CLW96, MLBK02]. Test coverage alone may not be a strong indicator of software quality [KBP01, page 181]. However, it is generally agreed in the Software Engineering community that achieving greater test coverage is desirable for achieving software reliability [POC93, CLW96].

With regard to combinatorial testing (cf. Section 3.4), the selection of a certain degree of input parameter combinations (e.g. pairwise, threewise,...) is a suitable coverage criterion. For example, pairwise coverage of input parameters is a frequently used coverage criterion. The following coverage criteria are used in [GOA05]. Some of those criteria can be taken into account if illegal values are ac-

cepted. Such false values are used to check whether the system behaves as expected e.g. an error or an exception is thrown [GG93].

- Each-used: 100% **each-used** coverage is defined as including each value of every parameter in the test suite at least once.
- Pairwise: 100 % **pairwise** coverage is defined as including every pair of values of any two parameters in the test suite.
- T-wise: 100 % **T-wise** coverage is defined as including all value combinations of any T parameters in the test suite.
- Variable strength: is an extension of the T-wise coverage criteria introduced by [CGMC03]. Here, the T may vary for different subsets of parameters.
- Valid: is an extension of the variable strength criteria demanding that only valid value combinations are built and that only valid values are included within a test suite.
- N-wise: 100 % **N-wise** coverage is achieved if all possible combinations of all parameters and their values are included in the test suite.
- Base choice: 100 % **Base choice** coverage is achieved by combining the most frequently used values of each parameter within a test suite, where the rest of the values are base values as well.
- Single error: 100% **Single Error** coverage is achieved if there is exactly one test case for each error value. All other values within each test case need to be valid.

With regard to model-based testing (cf. Section 3.3), structural coverage criteria involving the behavior of the system under test are suitable coverage criteria [Wei10]. According to Weißleder, the following structural coverage criteria exist.

- Transition-Based coverage criteria focus on transitions within the system. The most prominent ones are *All-States*, *All-Transitions*, and *All-Paths*. *All-States* and *All-Transitions* demand the covering of all states or transitions, respectively. The coverage criteria *All-Paths* is a very stringent criterion demanding to cover every possible route through code or model.
- Control-Flow-Based coverage criteria are generally applied to a control flow graph extracted from code or model. These criteria focus on the control-flow within the system. One of the most prominent ones is the so-called MC/DC (Modified Condition/Decision Coverage) criterion that is proposed by the RTCA in DO-178B for Software Considerations in Airborne Systems and Equipment Certification [fAR82]. MC/DC demands that:

- every point of entry and exit in the program has been invoked at least once,
 - every condition in a decision has taken all possible outcomes at least once,
 - every decision in the program has taken all possible outcomes at least once,
 - and each condition in a decision has been shown to independently affect that decision's outcome.
- Data-Flow-Based coverage criteria is a variation of the *All-Paths* coverage criteria and considers routes within the system between the assignment of variables and references of the variable.

3.5.2 Mutation Testing

Coverage criteria, such as the ones we have listed for black and white box testing, measure the quality of the test suite. Due to the fact that testing generally does not cover the complete SUT, these coverage criteria can be used to decide when to stop testing. Besides the well-known black- and white-box coverage criteria, mutation testing is another frequently used criterion to measure the quality of testing activities. In general, mutation testing injects faults into the SUT, which should then be found by the test suites. The quality of the test suite depends on the number of mutations found. The faults are injected using so-called *mutation operators* and the injected faults are called *mutants*.

When a test case fails, the mutant is *killed*, because the fault was detected and the mutant can be considered *dead*. The mutation testing process can be divided into three parts:

1. Creating mutants of a test program
2. Executing the faulty programs
3. Analyzing the output and marking the equivalent mutants

After creating mutants and executing all tests, the results have to be compared with the original output. This leads to a killed mutant in the case of fault detection or to a remaining mutant. A remaining mutant is placed in one of the two categories: The mutant is killable: means that the set of present test cases is not good enough to detect the fault, but it is possible to do so by extending the test cases. Or, the mutant is equivalent: means that the mutant is functionally equivalent to the original program and will not be detected at all.

THIS chapter introduces the state-of-the-art in SPL testing approaches. Our scope is related work addressing the selection of configurations for testing purposes and test case generation for SPLs. We refer to Engström et al. [ER11] for a summary of the state-of-the-art approaches categorized in:

- Test organization and process,
- Test management,
- Testability,
- System and acceptance testing,
- Integration testing,
- Unit testing, and
- Test automation

Every concept of development is as sufficient and reliable as the support it receives from concepts for testing. In single system engineering, testing often consumes 25% to 50% of the development costs [LL05]. Due to the variability within an SPL, the testing of SPLs is more challenging than single system testing. If these challenges are met by adequate approaches, the benefits outweigh the higher complexity and effort of testing activities. The challenges of testing an SPL are caused by the product line variability and the systematic reuse. For example, the testing of a component that is to be reused in different products illustrates one challenge in this context. The component must be tested accurately, as a fault would be conveyed to every product that include it. It is not sufficient to test this component only once in an arbitrary configuration, because the behavior of the component varies, depending on the corresponding product. Identifying and solving the SPL specific test challenges is important to achieve the benefits the product line paradigm provides. A suitable example for the importance of testing reused artifacts can be found in the Ariane V accident [Lig09]. On the first test flight of the expendable launch system Ariane V, a malfunction in the control software forced the ground crew to activate the self-destruction 37 seconds after launch. A data conversion from a 64-bit floating point value to a 16-bit signed integer representing the horizontal bias caused the malfunction. This software component was developed for the Ariane IV and was reused in Ariane V. In the

Ariane IV, this component worked properly but its integration in the Ariane V was not tested sufficiently. The reuse of this component caused a serious and very expensive malfunction.

This chapter summarizes the current approaches and research results in SPL testing. It is focused on transferring the concept of reuseability within SPL development to testing. Two concepts of reuse can be differentiated:

- Test artifacts such as test cases, test descriptions, test scripts, and test models are reused to decrease the testing effort. Instead of generating tests over and over again, these artifacts are reused.
- The second kind of reuse is the reuse of test results. Here, the test artifacts are not only reused, but a certain test is not re-executed, since the test result is expected to be the same as that of a previous test.

Both concepts include some risk. When reusing test artifacts, the test engineer needs to ensure that the test artifact is still appropriate for the test. When reusing a test result, the test engineer needs to ensure that the part of the SUT has not changed to an extent that would make the result non-representative.

Concerning test generation for SPLs, McGregor [McGo1] and Tevanlinna [TTK04] propose a well-structured overview of the main challenges for testing product lines. Studying related work focusing on SPL-Testing, we have identified three practices which are state-of-the-art:

- **Contra-SPL-philosophies** do not take into account commonalities and variabilities within the SPL. These approaches contradict the SPL-reuse-philosophies. Each product is tested individually and independently. In [TTK04] the authors refer to this approach as product-by-product testing. However, considering the number of derivable products of today's SPLs, this approach is no longer feasible. Thus, this approach is beyond the scope of this thesis.
- **Reuse-Techniques** are test approaches focusing on reusing test artifacts or test results to decrease the testing effort. Here, requirement-based test approaches and especially model-based test approaches are frequently used. Another test approach within this category is regression testing techniques, which test all products incrementally [ESRo8].
- **Subset-Heuristics** is the category which aims to generate and test a subset of all possible products, trying to ensure a certain coverage for the entire SPL. Thus, this approach is also based on reusing test results. Instead of testing every possible product, only the subset of products is tested. The results are expected to be the same for the remaining products.

In the following, the approaches related to the contribution of this thesis are summarized and discussed. These are **Reuse-Techniques** and **Subset-Heuristics**.

Anticipating the subsequent part of this thesis, both concepts will be integrated, forming a new approach for testing SPLs.

4.1 REUSE-TECHNIQUES

Methods of this concept utilize reuse-techniques to reduce the test effort. These approaches either make use of regression testing techniques to incrementally test products or reuse domain tests during application testing.

Regression testing is a technique to check whether a certain product behaves as expected after it was changed e.g. revision. IEEE 1990 defines regression testing as:

Definition 21 (Regression Testing). *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 [IEE90].*

and as

- testing that is performed after making a functional improvement or repair to the program. Its purpose is to determine if the change has regressed other aspects of the program [Mye79].

Instead of verifying that a modified product still behaves as expected, regression testing is used to incrementally test product after product, interpreting the difference between two products as manipulation/revision. Thus, every product is incrementally tested [ESR08, ER10, ERS10, ERW10]. The challenging part of this approach is to find an appropriate product to start with and to define to which extent every product is retested. Although Engström et al. are currently evaluating their approach with real world SPLs, it is not possible to rate the potentials of this approach yet.

Reusing domain tests created during domain engineering for product tests is a very popular approach, especially in the requirement-based testing community. Bertolino et al. consider parametrization and use equivalence classes for requirement-based SPL testing [BG03, BFGL06]. They introduced the Product Line Use case Test Optimization (PLUTO); within PLUTO, tests are generated on the basis of use cases of an SPL (Product Line Use Cases - PLUC) [BG03, BFGL06]. For this purpose, the authors adapt the normal category partitioning method [OB88] to select a certain combination of use case scenarios as test case specifications for a selected product. Scenarios are represented as textual use cases and variability is described by using special tags within those use cases. Then, test cases are derived for each product of the SPL instead of intertwining SPL and test case definition activities.

Nebut et al. developed a test approach starting with the definition of customizable use cases for an entire SPL and deriving product specific use case for testing purposes [NFLTJ04].

Apart from use case oriented processes, model-based testing is a popular approach to reuse test cases across different products. In SPL model-based testing, statecharts, activity diagrams, and sequence diagrams are frequently used to specify the behavior of software systems.

CADeT [Olio8], ScenTED [RKPR05] and Hartmann et al. [HVR04] utilize activity diagrams as reusable test models that are created during domain engineering. Test cases are generated on the basis of the paths within the activity diagrams. The test cases are then adapted for each individual product during application engineering for system testing.

Weißleder et al. [WSSo8] use a single state machine as the test model that describes the functionality of an entire SPL, and the tool ParTeG for automatic test case generation. Together with the approach for model-based product design verification in [KNo4], those two approaches are the only ones using statecharts as test models for SPLs, according to the best of our knowledge. Existing approaches for applying general criteria of data and control coverage to statecharts are mainly based on reachability trees [MMB94, SMFM00], flow graphs [HKC⁺00, BHo8], and automata variants [BHS99].

A summary and comparison of model based testing approaches for SPLs can be found in [OWES11]. There, we located two major challenges when using model-based testing for SPLs: the representation of variability within test models and how to map model artifacts to a variability representation such as feature models. Due to the fact that model-based testing is an important component of this thesis, we will discuss the related work and research in model-based testing for SPLs in detail.

4.1.1 *Model-based Testing of SPLs*

In [OWES11], we summarized and compared the various approaches for model-based SPL testing, namely CADeT, ScenTED, the Hartmann et al. approach (in the following referred to as Hartmann), the Kishi et al. approach (in the following referred to as Kishi), and the Weißleder approach (in the following referred to as Weißleder). We used the following criteria for comparison purposes:

- **Input:** What kind of input is required or can be processed.
- **Output:** We usually expect test cases or a description of them to be generated. Regarding the generation process, we additionally examine the degree of automation, for example, whether the test case generation is executed automatically or semi-automatically. Another important approach for evaluating model-based testing approaches is to measure the coverage of their tests. Therefore, we include the existence and type of coverage criteria belonging to a certain approach as a criterion, to compare the different approaches.
- **Test Levels:** We examine whether a test approach covers a specific test level, for example, unit, integration or system testing.

- **Traceability:** An important property of SPL testing approaches is the mapping between test cases and requirements, architecture, and features of the feature model. It offers the possibility to trace the faults found back to the responsible component and even back to the requirements specification. In addition, if a requirement is changed, we know exactly which test must be adapted. Another point with regard to traceability is the derivation of concrete product specific test cases. By choosing requirements for a concrete product, the corresponding test cases should be determined by tracing from the selected variants of the requirements to the test cases covering this requirement. Therefore, we check if traceability is supported and to what extent.
- **Development Process:** the question arises whether the testing approach can be integrated within the development process of SPLs. A key attribute is the differentiation between domain and application engineering. If an approach differentiates between domain and application engineering we must examine how variability within the different activities is modeled and handled. Then, we determine how reuse and variability interact for testing purposes.
- **Application:** This criterion addresses the question of whether a particular approach can be integrated into a company's development process. In order to do this, a step-by-step description is important.

We introduce a conceptional process model comprising all model-based test procedures for SPLs, to visualize the commonalities and varieties of the different approaches. This model is depicted in Figure 12 and forms a superset of all test approaches that will be discussed. The individual characteristics and properties of each model-based approach are captured by specializing this process model.

Figure 12 is based on the following two principles.

1. According to the development process for SPLs depicted in Figure 6, the testing process is subdivided into application and domain engineering as well.
2. Each phase can be subdivided according to the levels of the V-model introduced in [DWoo]. For each step in the development process (left-hand side), a corresponding test level exists (right-hand side). Therefore, we can visualize the different levels of testing: Unit tests, Integration tests, and System tests. Each test phase contains a test model and a test case specification to generate test cases.

Additional vertical edges connecting domain and application engineering indicate that artifacts developed in domain engineering are reused in application engineering. Using this process model, we can discuss:

- whether the approaches differentiate between domain testing and application testing

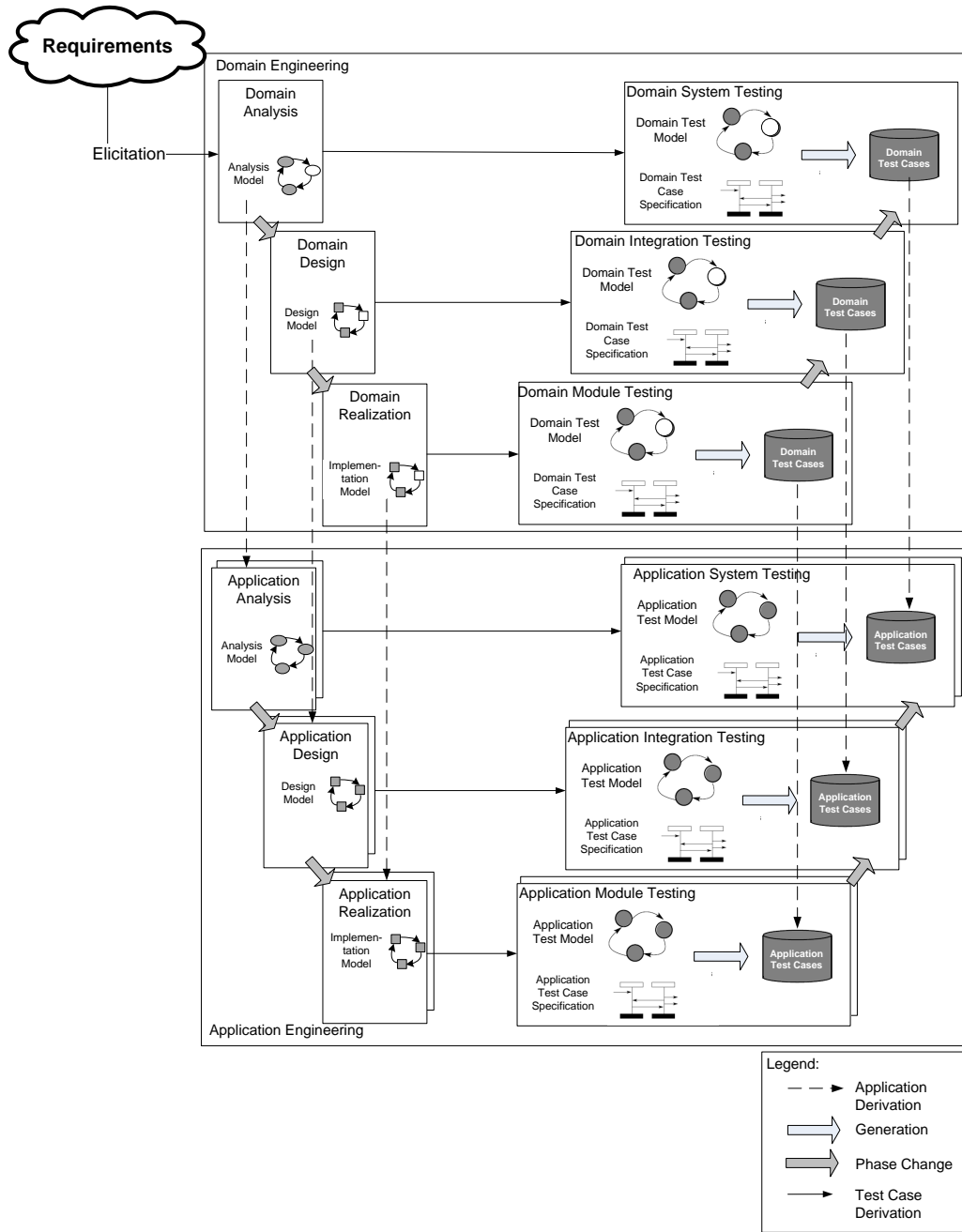


Figure 12: Conceptual process model for Software Product Line testing [OWES11]

- whether they consider unit, integration, and system testing, and
- how domain or product specific test cases are generated.

CADeT

CADeT (Customizable Activity Diagrams, Decision Tables, and Test specifications) is a model-based test approach introduced by Olimpiew in [Olio8]. CADeT provides a detailed description of how to apply it to an SPL project. It aims at the system test level and defines use cases for requirements modeling based on Gomaa's PLUS (Product Line UML based Software engineering) method [Gomo4]. Figure 13 shows the CADeT approach according to the conceptional process model. In Domain Analysis, textual use cases are manually created according to the PLUS approach, on the basis of the SPL requirements. Furthermore, a feature model representing the different use cases and providing an overview of the commonality and variability in the SPL under test is also created during Domain Analysis. In Domain System Testing, activity diagrams that include variability are manually derived from the use cases. The test cases are then derived automatically, based on activity diagrams. To map test cases to the different use cases, a decision table is used, linking the features of the feature model representing the different use cases to corresponding test cases. By choosing a particular feature configuration, product

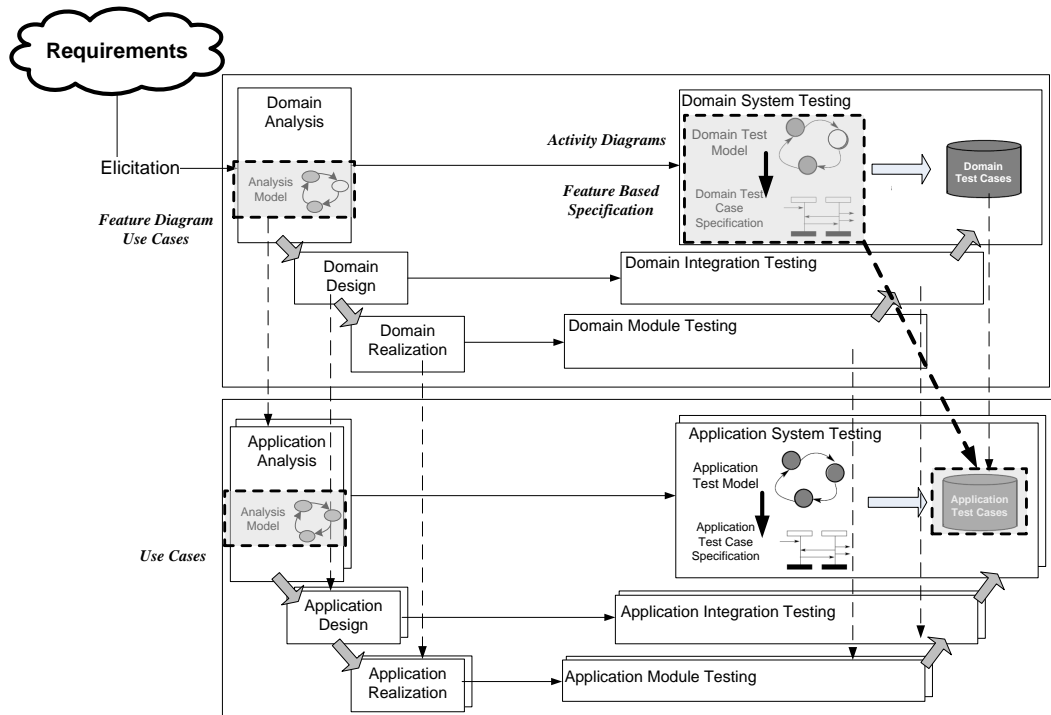


Figure 13: CADeT process model [OWES11]

specific test cases can be derived from the domain test cases. CADeT offers different feature coverage criteria to test example product configurations. Test coverage, from the classical point of view, is not supported by the approach. In addition, test data definition in domain engineering is not supported by CADeT.

ScenTED

ScenTED (Scenario-based Test case Derivation) is a model-based test procedure introduced by Reuys et al. [RKPR05]. Figure 14 depicts those parts of our conceptual process model that are covered by the ScenTED approach. ScenTED preserves variability within the entire process of deriving domain and application test cases for system testing. In Domain Analysis, ScenTED provides domain use cases including variability. For Domain System Testing, so-called domain activity diagrams are manually created and serve as test models including variability. Variability is described using alternative paths within the activity diagrams. Domain test cases are generated on the basis of the activity diagrams and branch coverage criteria are used, ensuring that every branch is covered by at least one test case. Test cases for different products can be derived easily from the domain test cases. At the time an application is derived, variability is resolved and product specific test cases are derived on the basis of the domain test cases. Thus, the domain test scenarios for system testing can be reused for various applications.

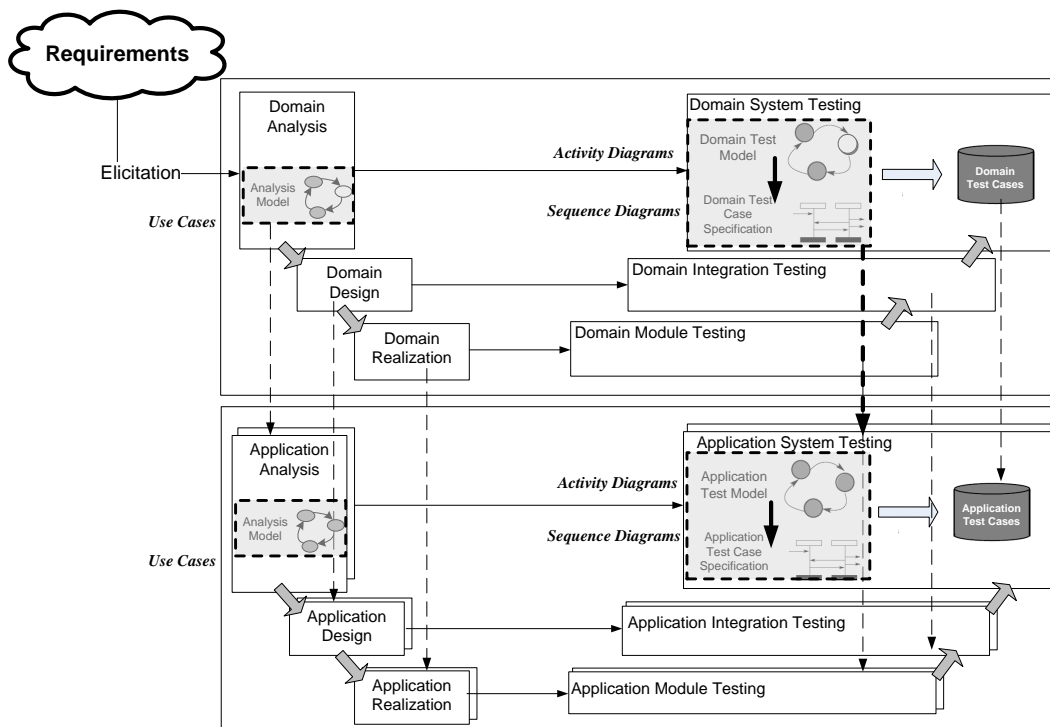


Figure 14: ScenTED process model [OWES11]

As in CADeT, Integration tests and Unit tests are also not considered in ScenTED. A major drawback of the ScenTED approach is the fact that it is not possible to interrelate different variation points representing variability in the test artifacts to describe further dependencies.

Hartmann

Hartmann et al. provide a UML-based approach to generate test cases for system testing. Activity diagrams represent the system specification in Domain Analysis and serve as a test model in Domain System testing. As only one representation is used for both the system specification and the test model, this approach suffers from the drawback that the common bias prevents independent detection of faults. The test model is used to derive product specific test cases. The approach of Hartmann et al. does not explicitly model variability within activity diagrams. They use a tree structure of activity diagrams to build up the appropriate activity diagram for the selected product. As the selection criteria for the subactivity diagrams depend on concrete product variants, the approach does not apply to new product variants, i.e. those that do not already exist in the model. This means that all possible or desired product variants must be known or anticipated from the beginning. Furthermore, the approach does not describe in detail how to generate test cases. From our point of view, the test case generation is not a product line specific problem, because the generation works on activity diagrams without variability. Finally, the approach does not provide detailed information about coverage and test data and the additional stereotypes complicate the automation of this approach. Figure 15 shows the approach according to the conceptual process model.

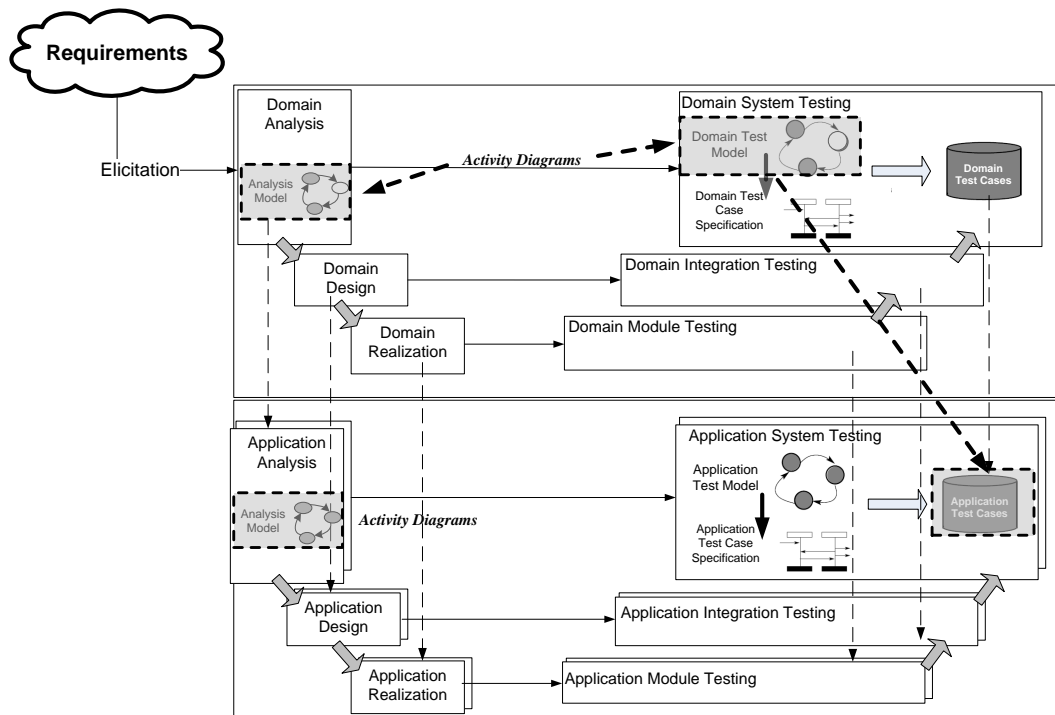


Figure 15: Hartmann process model [OWES11]

Kishi

In contrast to the approaches described above, Kishi et al. focus on the fact that the majority of expensive faults in Software Engineering originates during requirements analysis and design phases [KNo4]. The work of Kishi et al. introduces an approach to test the design of a product by using model checking and reusable state machines. The authors expect to test the design more exhaustively than when other, more traditional, approaches are applied, such as reviewing. The complete verification of a product is, however, not achieved. Rather than applying model checking on individual state machines of products of an SPL, a reusable *environment* model emulating event sequences is introduced, using a single *target* model representing the behavior of the SPL. Their approach is located in application engineering only. However, a well-known problem for model checking is state explosion [CGDA99]. This problem also arises in Kishi's approach. Figure 16 depicts those parts of our conceptual process model that are covered by the Model-Checking approach.

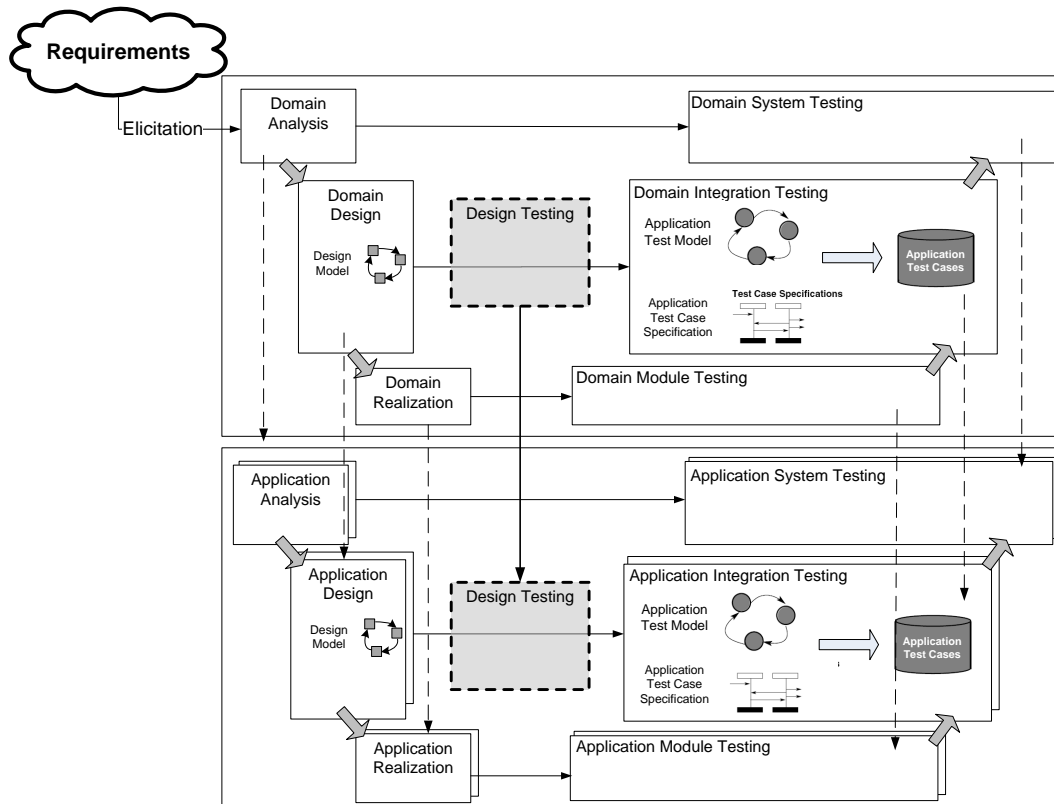


Figure 16: Model checking process model [OWES11]

Weißleder

Weißleder et al. propose reusing test models for model-based testing of SPLs by reusing state machines for context classes [Weio9]. A state machine that represents the entire SPL has to be implemented. One single state machine is used as test

model describing the behavior of the entire SPL in Domain System testing. Test models for the individual applications are represented by subclasses in Application System testing. The test case specifications are described using OCL constraints. Application test cases are derived using ParTeG [Weio9].

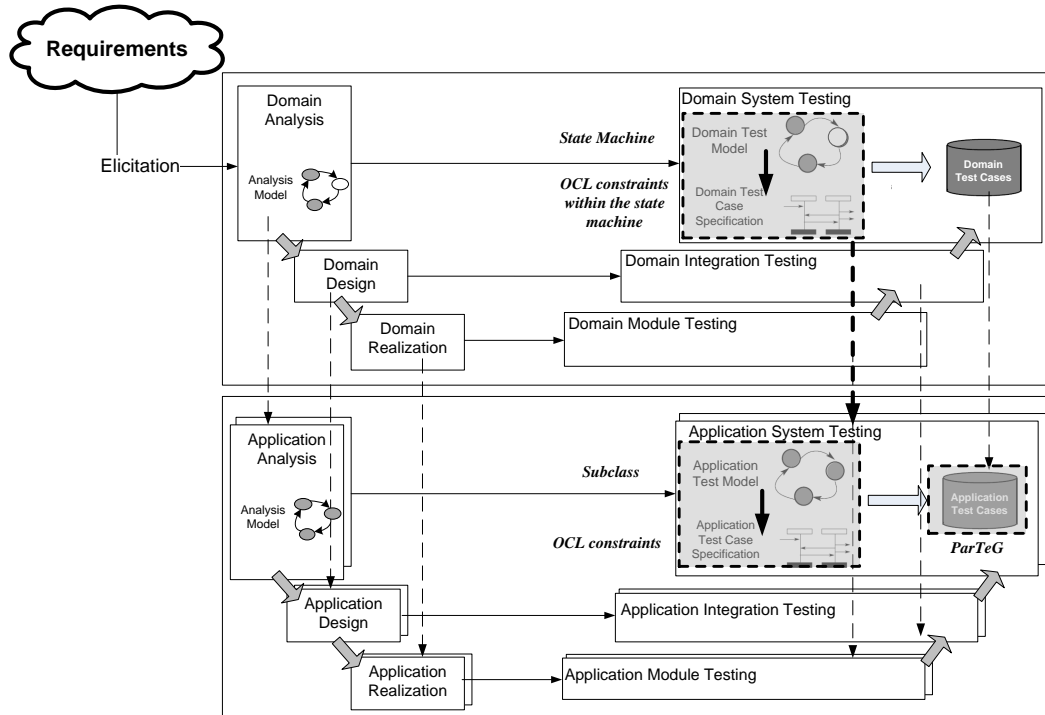


Figure 17: Reusing state machines process model [OWES11]

In contrast to the other approaches, this method preserves variability within a single test model. It is not necessary to generate individual models including variability for each feature, feature combination or use case. However, there are still unanswered questions that must be discussed. For instance, one important question is how to derive class hierarchies from product lines automatically. The authors assume that aspect-oriented approaches within the development process may offer a solution. Each feature could be represented by a certain aspect. Then a hierarchical structure of aspects can be set up. Figure 17 depicts those parts of our conceptional process model that are covered by the Reusing State Machine approach.

Table 3 shows how the various approaches fulfill the criteria for comparison. The approaches define the important elements of a product line testing approach, such as variability definition, test case generation, and product derivation. All of these elements encourage the SPL developers to achieve the SPL-specific advantages such as reduction of costs or a higher product quality. In addition, a detailed examination of the comparison reveals some interesting facts about the compared approaches. Every approach concentrates on particular elements. For example, the CADeT

approach provides traceability from features to test case specifications by using decision models. The ScenTED approach integrates test and variability coverage for activity diagrams. All but one of the approaches concentrate on the system test level and none of the approaches considers test data in domain engineering.

It becomes clear that some elements of SPL testing remain uncovered, for example Unit Testing. Based on the comparison, the following research objectives can be formulated:

- Integrate all Test Levels.
- Automate more steps of the test suite generation process.
- Improve the output including concrete test data.
- Improve traceability between variability representation, requirements, and test artifacts.
- Provide detailed instructions about how to apply the test approach in the industrial context.

4.1.2 *Variability and Mapping*

Typically, model-based testing for SPLs requires two additional properties compared to single system model-based testing:

- variability in the test model
- mapping to a variability description

In the following, we will discuss related work regarding these two properties.

In [SVo8], the authors introduce an approach that presents the behavior of a configuration as a combination of the statecharts that implement all the corresponding features. The behavior of a configuration is built by a stepwise refinement, starting with the statechart of the root feature of the feature model. This stepwise generation of a configuration does not permit inconsistency within the statecharts during the refinement. However, the labeling is kept very simple and Szasz et al. do not provide additional dependencies within the feature model representing the interaction of features [SVo8]. Since this approach is not meant to be used for model-based testing, there is no support to generate test cases.

In [GLo8], the authors use one single statechart to model the behavior of the SPL. The statechart consists of a common part and optional fragments. The common part is a classical statechart describing the commonalities of the SPL, whereas the optional fragments describe the variability within the model. The features of the feature model are mapped to optional states and transitions that are, unfortunately, the only variable artifacts within this approach.

	CADeT	ScenTED	Hartmann	Kishi	Weißleder
Input - Test Model	Activity Diagram	Activity Diagram	Activity Diagram	State Machine	State Machine
Input - Dev. Model	Use cases	Use cases	Activity Diagrams	Design Model	not mentioned
Output - TC - Component	Test Steps pre-, post-condition	Test Steps, pre-, and post-cond. input data expected output data	Test Steps	Test Steps input data	Test Steps input data pre- and post-cond.
Output - Automation	(o)	(-)	(-)	(o)	(o) / (+)
Output - TC - Coverage	not mentioned	branch coverage with variability	not mentioned	not mentioned	boundary based coverage
Test Levels	System Test	System Test	System Test	Design Test	System Test
Traceability	yes	no	no	no	yes
Dev. Process	Domain & Application Eng.	Application Eng.	Application Eng.	Domain & Application Eng.	Application Eng.
Dev. Process Integration	(+)	(+)	(-)	(o)	(o)
Dev. Process - Variability	mandatory optional	mandatory, optional, or alternative	optional	mandatory, optional, or alternative	mandatory, optional, or alternative
Reuse of artifacts	yes	yes	yes	yes	yes
Application - Instructions	(+)	(o)	(o)	(o)	(o)

Table 3: Comparison of model-based test approaches for SPL [OWES11]

One of the biggest challenges in model-based approaches for SPL development, regardless of whether the model serves for test case generation or not, is to make the model configurable. To interrelate the artifacts of the test model and the elements of the feature model, a mapping is required.

There are various approaches for mapping features of a feature model to elements of the SPL architecture. The approach in [SLKH07] identifies traceability between feature models and architecture models, using Formal Concept Analysis. Wasowski [Was04] describes an approach to automatically generate variants of behavioral models by means of statecharts, using a form of partial evaluation and slicing based on specified restrictions. Sochos et al. [SRP06] propose Feature Architecture Mapping (FARm) for mappings between features and architectural components, utilizing progressive transformations. Czarnecki [CHE05b] annotates a template design model with logical, feature-based presence conditions. In [Olio8], Olimpiew uses decision tables to map features to use cases in order to generate test cases for products derived from an SPL.

4.1.3 *Lessons Learned - Reuse-Techniques*

Although all of the aforementioned reuse-techniques benefit from reducing the effort for testing, each product has to be tested individually. Furthermore, the reuse-techniques lack traceability between a variation model, use cases, and test specifications. The next category focuses on identifying a subset of products to approximate a complete SPL-test according to various coverage criteria.

4.2 SUBSET-HEURISTICS

This approach aims at reducing the effort for testing by extracting a subset of feature combinations or products. Instead of testing every product of the SPL, a subset for testing is created. We identified two different methodologies:

- methods generating a subset of products that are representative for testing purposes of the entire SPL, e.g. risk-driven or requirement-based,
- approaches using combinatorial design to generate a subset of configurations covering certain combinations of features, and
- an approach generating a set of test cases covering the entire SPL. To execute the tests, a suitable set of configurations/products needs to be generated as well.

4.2.1 *Representative Sets*

Kolb introduced a method proposing a set of product configurations for testing purposes based on a risk analysis [Kolo3]. Unfortunately, this approach could not be compared with other approaches, due to the lack of implementation.

In [Scho7], the author introduces an approach generating an optimal representative set of products, so that all requirements of the SPL are covered. Scheidemann uses a greedy algorithm for the product selection, based on the assumption that the verification result for each requirement is independent of the architecture of those elements which are not included in the product. The major disadvantages of this approach are (1) that it does not scale with real-world SPLs and (2) that the effort to set up the representative set is enormous.

4.2.2 *Combinatorial Interaction Testing*

Another possibility is to generate a subset of configurations, using combinatorial design [McGo1]. The commonalities and variability within an SPL are frequently represented by features that can be interpreted as parameters in the SPL engineering process. One of the best-known applications of combinatorial testing is the pairwise testing approach. This method is based on the assumption that the majority of faults originate from a single parameter value or are caused by the interaction of two values [SM98]. This idea can be transferred to features. Combinatorial testing can be used to cover a certain T-wise feature interaction.

McGregor initially introduced combinatorial testing to SPLs in [McGo1]. However, he neither describes how combinatorial testing may be applied to SPLs nor how variability models, such as feature models or OVMs, can be mapped onto an appropriate representation to apply existing combinatorial testing algorithms.

Gustafsson introduced an algorithm generating a subset of products so that every single feature is covered at least once [Gus07]. The disadvantages of such an approach realizing 1-wise combination are obvious. There is no guarantee that a feature functions as expected on its own nor does this approach take feature interactions into account.

Cohen et al. use the OVM approach to model the variable and common parts of the SPL, which are mapped onto a relational model. This relational model serves as a semantical basis for defining coverage criteria for the SPL under test [CDS06]. Furthermore, Cohen et al. describe the development of combinatorial interaction testing (CIT) that achieves a desired level of coverage. In [CDS07], the authors use the CIT approach to systematically select products that should be tested.

The approach described in [PSK⁺10] is similar to the one of Cohen et al.. The significant difference is that Perrouin et al. utilize SAT-solvers and do not use a relational model. Nevertheless, the output of the presented algorithm is not deterministic, due to random components that we will explain in the evaluation

chapter of this thesis. The number of found products may vary strongly, due to this random component.

These aforementioned methods provide feature oriented approaches for generating a representative set of configurations for an SPL. However, none of the approaches actually provides a test case generation process for the generated set of configurations. Furthermore, the application of these methods is rather complicated and not supported by a tool chain combining variability modeling and combinatorial configuration selection. In this thesis, this will be considered as well.

4.2.3 *Subset of Test Cases*

Instead of generating a set of products for testing purposes, a set of test cases that covers the entire SPL seems to be a natural alternative. One approach moving in this direction is the FMT (Feature Model for Testing) language [OMS09, SOM09]. There, a new methodology, together with an integrated modeling language, called FMT, that combines the capabilities of “classical” feature modeling and classification-tree-based test case description languages. The integration of feature models and classification trees [GG93] is based on the following ideas:

- Classification tree-based testing approaches deal with single product instances only and thus are not capable of processing variability.
- The integration with feature models is vital to handle the complexity caused by variability and to survey the entire SPL.

The FMT language is still under development, as well as the accompanying new SPL modeling and testing process. A first version of an FMT tool prototype has been implemented using the metamodeling tool MOFLON [AKRS06]. The FMT approach has the following advantages, compared to the state-of-the-art, where the selection of a set of product instances as SUTs and the selection of test cases for each SUT are separate activities:

- Heuristics creating test parameter values can easily be adapted to the new task of generating a set of SPL instances as SUTs [SOM09].
- Integrated algorithms can be developed to generate SPL specific black-box test cases.
- Manual activities that are currently performed for each selected SUT in a product-by-product SPL testing approach can be reduced to a minimum [SOM09].

Another approach addressing the generation of a representative set of products on the basis of a representative set of test cases can be found in [KBK11]. Kim et al. introduced a framework realizing a test-based coverage criterion to generate a

	Result	Selection Criteria	Comment
Kolb et al. [Kolo3]	products	risk analysis	no implementation available
Scheidemann [Scho7]	products	all requirements covered	time consuming and does not scale for real-world SPLs
McGregor [McGo1]	products	Combinatorial Design	no description of how to apply combinatorial testing to an SPL
Gustafsson [Gus07]	products	single feature	There is no guarantee that a feature functions as expected on its own nor does this approach takes feature interactions into account.
Cohen et al. [CDS06]	products	Combinatorial Design	no test cases for the configurations
Perrouin et al. [PSK ⁺ 10]	products	Combinatorial Design	no test cases for the configurations
Oster et al. [OMS09]	test cases	Black box heuristics	Not yet implemented and only one small industrial case study [Pet11].

Table 4: Comparison of the different subset-heuristics

set of configurations for SPL testing purposes. For a given set of test cases Kim et al. can determine which features need to be bound for it to be executed. Thus, they are able to determine a set of products to execute all test cases for the entire SPL. A major drawback is that the test cases have to be known in advance and that the authors do not present a methodology to generate such a set of test cases. Furthermore, the set of configurations heavily depends on the quality/coverage of the tests.

4.2.4 Lessons Learned - Subset-Heuristics

In summary, all subset-heuristics intend to generate a representative subset of configurations/products of the SPL with regard to a certain selection/coverage criterion. Table 4 provides an overview of the previously discussed subset-heuristics. Combinatorial interaction testing is a frequently used approach to generate a representative subset. But only Perrouin et al. [PSK⁺10] provide an approach to directly apply combinatorial testing to SPLs. However, this approach does not provide an appropriate test case generation concept to test the resulting subset. Additionally, as we will discuss in the evaluation chapter, this approach does not generate significant small subsets (sometimes even twice as large as our corresponding subset) and does not scale for feature models including more than 80 features.

SUMMARY PART II

THIS part has summarized preliminaries vital for this contribution. The basic concepts and ideas of SPL engineering, along with the associated challenges and benefits, are introduced. A brief summary of the field of software testing paves the way for discussing SPL testing approaches. Three different categories for SPL testing exist: contra-SPL-philosophies, reuse-techniques, and subset-heuristics. We focus on reuse-techniques and subset-heuristics, since these two categories address the reduction of the testing effort for SPLs. After summarizing the different approaches within these two categories we explained that:

- approaches of the subset-heuristic category do not provide a systematic approach to apply their algorithms to an SPL and do not provide mechanisms for generating test cases for each product of the subset.
- approaches relying on reuse-techniques still have to test each individual product.

The next part, **Concept and Theory**, addresses these shortcomings by combining the advantages of the two categories:

- **Subset-Heuristics:** Generating a subset of configurations for testing purposes to achieve 100% T-wise feature interaction coverage in the entire SPL, on the basis of the corresponding feature model.
- **Reuse-Techniques:** A reusable test model serves as a basis to generate tests for each configuration of the subset.

With regard to the model-based test approach, we intend to apply lessons learned from the review of the state-of-the-art. Thus we:

- will use statecharts as reusable test models, to benefit from the various tools for test case generation based on statecharts.
- will provide a detailed mapping between features of the feature model and elements of the test model, to allow automatic test case generation.
- will provide a detailed description of our procedure, to apply our test approach in the industrial context.

Part III

CONCEPT AND THEORY

THIS part provides a stepwise introduction to our Model-driven Software Product Line Testing (MoSo-PoLiTe) approach that addresses the shortcomings in current SPL testing approaches. This approach realizes a combination of the two SPL testing categories: reuse-techniques and subset-heuristics. MoSo-PoLiTe is driven by the following three concepts:

- Feature model-based Testing
- Combinatorial Testing
- Model-based Testing

MoSo-PoLiTe combines these concepts for SPL testing—subset-heuristics by means of combinatorial testing, reuse-techniques by means of model-based testing and integrates both by using a feature model as a central component. The feature model is used to link the combinatorial test approach and the model-based test approach and, thus, enables the combination of both. Furthermore, the feature model eases the integration of MoSo-PoLiTe into existing SPL development processes, since the feature model is an SPL-typical development artifact of the requirements analysis phase.

Figure 18 provides a compact graphical representation of the MoSo-PoLiTe development process. The central component in MoSo-PoLiTe is the feature model depicted in the center of Fig. 18. It is created on the basis of the SPL requirements during domain engineering. The feature model provides a hierarchical structure of the SPL requirements and represents the common and variable parts of an SPL. Selecting features according to the dependencies and constraints results in configurations that can be interpreted as subtrees of the original feature model (cf. Section 2.2.1 in Part II). Typically, features of a feature model are additionally linked to development artifacts such as code fragments or behavioral models. Then, the selection of a configuration leads to a product. Chapter 6 describes the usage of feature models within MoSo-PoLiTe.

In MoSo-PoLiTe, the configuration selection is performed by a specific selection algorithm based on combinatorial testing (cf. Figure 18 left-hand side). Our algorithm generates a minimal set of configurations containing all T-wise (e.g. pairwise) feature interactions. Therefore, testing this subset of products is equivalent to T-wise testing the entire SPL. This algorithm is introduced in Chapter 7.

For a model-based test case derivation the feature model is additionally mapped to a reusable test model representing the behavior of the entire SPL. We will provide a mapping approach ensuring that if a valid configuration is derived from

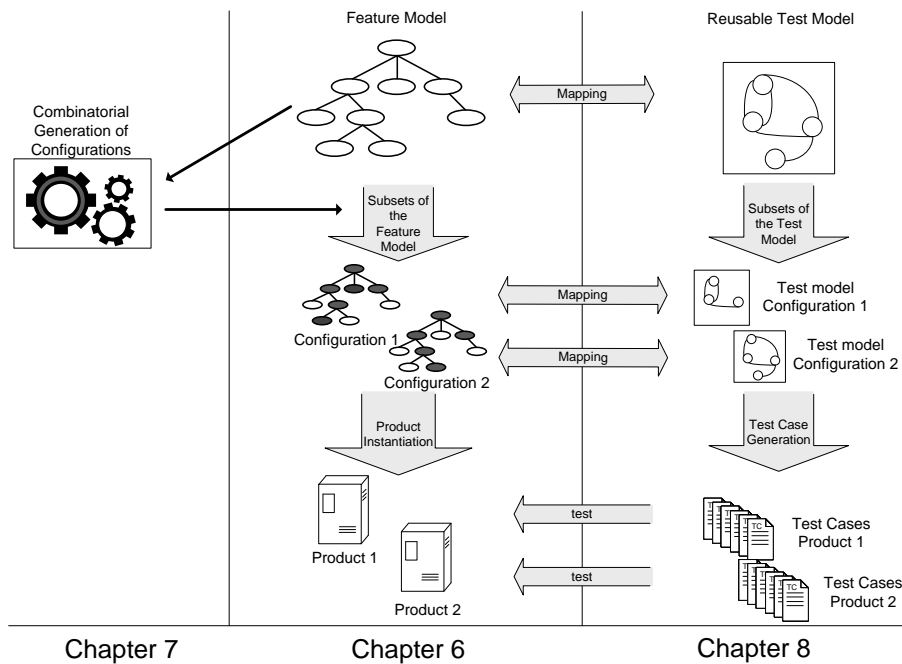


Figure 18: Overview - Part III

the feature model, a corresponding test model representing the behavior of this configuration is generated, too. Thus, for every configuration of our combinatorial subset, a corresponding test model can be generated. This test model can then be used for generating test cases for this configuration. Please note that for test case generation, this thesis refers to existing approaches. Reinventing or improving model-based test case generation is beyond the scope of this thesis. The model-based testing component of MoSo-PoLiTe is discussed in Chapter 8.

In the following, we provide an argumentation for why MoSo-PoLiTe is a combination of the aforementioned concepts and also provide a detailed description of each of these concepts.

THE feature model is the central component within the MoSo-PoLiTe development process. In this chapter, we provide arguments for why we have chosen the feature modeling concept for our testing approach. Furthermore, we will introduce a formal definition of feature models, to prepare the ground for a precise definition of the combinatorial subset selection procedure and for an explicit mapping to our test model. Three lines of reasoning have led us to the idea to base our approach upon feature modeling:

- The first reason is very intuitive. Feature models provide a hierarchical and structured representation of the SPL requirements. Commercial tools such as pure::variants [pG11], Gears [Kru08], and preevision [aG11c] interrelate feature models to requirement specifications. Especially pure::variants and Gears pay particular attention to mapping the features to requirement engineering tools such as Doors®, CaliberRM™ or MKS Integrity Requirements. Thus, it seems to be promising to benefit from such a representation of SPL requirements for testing purposes. Algorithms ensuring a certain degree of requirements coverage within product derivation can take advantage of the feature model.
- The feature oriented software development community (FOSD) [FOS11] uses the features of a feature model as fundamental artifacts within the development process of variant-rich systems, such as SPLs. There, the features are linked to various artifacts such as code fragments, behavioral models, requirements, specifications, documentation, and tests. Thus, it seems to be a natural choice to use the feature model as a basis to generate a representative set of configurations, including test cases.
- Commercial tools such as pure::variants, Gears, and preevision use feature models to parameterize the derivable products and provide tool integrations between their feature model editor and development tools such as MATLAB®/ Simulink®, Rational Rhapsody, OpenArchitectureWare, and Enterprise Architect. Thus, using the feature model to parameterize corresponding test cases to the derivable products seems to be reasonable [MSM04].

A test approach based on feature models can be interpreted as validation or verification, depending on the kind of data that is linked to the feature model. In single system engineering validation can be understood as answering the question:

are we building the right product based on the stakeholder requirements. Verification, on the other hand, seeks to answer the question: are we building the product correct on the basis of the system specification. Mapping these definitions to SPL engineering, we can state:

- Validation: we can say that an SPL is validated if we can answer the question: do the right products result from the SPL? This question can be answered according to the stakeholder requirements.
- Verification: we can say that an SPL is verified if we can answer the question: are all derivable products of the SPL verified? This question can be answered according to the system specification.

As already mentioned, a feature model only provides a hierarchical structural representation of the requirements and/or specifications of the SPL. To derive products or tests using a feature model, the features have to be linked to development artifacts or test artifacts. The scope of this thesis is testing, thus, a mapping approach to test artifacts is required. For this purpose, we will introduce a formal definition of feature models to provide an accurate mapping between features and test artifacts (in Chapter 8). Furthermore, we also discuss feature interactions, because interacting features is a foundation of a fault model for SPL, where faults are likely to be revealed at execution points at which features exchange information with other features or influence one another [Bin99, page 557].

We will answer the following questions:

- What is feature interaction?
- What kind of feature interactions exist?
- How do we efficiently address feature interaction in testing activities?

6.1 FORMAL DEFINITION OF FEATURE MODELS

In the following, we provide a formal definition of the feature model introduced in the previous part.

Definition 22 (Feature Model). *A feature model $FM = \langle F, \mathcal{R}, \mathcal{C} \rangle$ is defined over a finite set of features $F = \{f_1, \dots, f_n\}$ equipped with a collection of relations \mathcal{R} and constraints \mathcal{C} .*

We write \mathcal{FM}_F to refer to the set of all feature models $FM \in \mathcal{FM}_F$ over features F . To specify the variability relationships among features, four different relations are used:

- $R_{\text{man}} \subseteq F \times F$ – if $(f, f') \in R_{\text{man}}$, then feature f' is a mandatory variability of feature f ,

- $R_{\text{opt}} \subseteq F \times F$ – if $(f, f') \in R_{\text{opt}}$, then feature f' is an optional variability of feature f ,
- $R_{\text{alt}} \subseteq F \times \mathcal{P}(F)$ – if $(f, F') \in R_{\text{alt}}$, features $F' \subseteq F$ constitute an alternative group for feature f , i.e., exactly one feature $f' \in F'$ is a mandatory variability of feature f , and
- $R_{\text{or}} \subseteq F \times \mathcal{P}(F)$ – if $(f, F') \in R_{\text{or}}$, features $F' \subseteq F$ constitute an or group for feature f , i.e., arbitrary feature sets $F'' \setminus \{\emptyset\} \subseteq F'$ are variabilities of feature f .

These relations are pairwise disjoint and the union: $\mathcal{R} = R_{\text{man}} \cup R_{\text{opt}} \cup R_{\text{alt}} \cup R_{\text{or}}$ results in the tree structure of the feature model in which:

- each feature F represents one node
- the relations \mathcal{R} represent the edges
- two features $f, f' \in F$ are related by the child relation $\prec_c \subseteq F \times F$ if f' is a direct child of f , i.e., $f \prec_c f'$ iff $(f, f') \in \mathcal{R}$ or $(f, F') \in \mathcal{R}$ with $f' \in F'$.

We then call f the parent or variation point and f' the child or variant.

- the root node of the feature model is a special feature f_r representing the concept of the SPL. The node f_r is the only node within a feature model without a parent node: $\forall f \in F: \neg f \prec_c f_r$.
- (F, \prec_c) is a rooted tree i.e. $f' \prec_c f \wedge f'' \prec_c f \rightarrow f' = f''$

The constraints \mathcal{C} introduce further restrictions on feature combinations and can be interpreted as cross-tree constraints in the feature model:

- $C_{\text{req}} \subseteq F \times F$ – if $(f, f') \in C_{\text{req}}$, then feature f requires feature f' in every product, and
- $C_{\text{exc}} \subseteq F \times F$ – if $(f, f') \in C_{\text{exc}}$, then feature f excludes feature f' in every product.

Note that the exclude constraint is symmetric, i.e., from $(f, f') \in C_{\text{exc}}$ it follows that $(f', f) \in C_{\text{exc}}$.

A feature model is used to configure the products of an SPL. Such a so-called **configuration** is a subtree of the feature model and includes all features which are included in the product.

Definition 23 (Product Configuration). A product configuration $F_p = \{f_1 \dots f_n\}$ is a valid subtree of F . The consistency rules for valid product configurations are defined as follows:

- the root feature $f_r \in F$ is a special feature denoting the concept/name of the entire SPL and is, therefore, part of every product: $f_r \in F_p$
- for every feature $f \in F_p$ the following conditions hold:
 - if $(f, f') \in R_{\text{man}}$, then $f' \in F_p$,
 - if $(f, F') \in R_{\text{alt}}$, then $\exists! f' \in F' \cap F_p$,
 - if $(f, F') \in R_{\text{or}}$, then $\exists f' \in F' \cap F_p$,
 - if $(f, f') \in C_{\text{req}}$, then $f' \in F_p$,
 - if $(f, f') \in C_{\text{exc}}$, then $f' \notin F_p$, and
 - if $f' \prec_c f$, then $f' \in F_p$.

An $\text{FM} \in \mathcal{FM}_F$ defines the set of valid product configurations by means of feature combinations $F_p \subseteq F$ permitted in the SPL. We write products $: \text{FM} \rightarrow \mathcal{P}(\mathcal{P}(F))$ to refer to that product space shaped by FM.

At this point we like to remind the reader that the hierarchy, the relations, and the constraints within the feature model provide valid information for testing purposes, since they represent relations between the various functionalities of the SPL. Apart from these relationships, features can interact with each other [JZ98, Meto4].

6.2 FEATURE INTERACTION COVERAGE

According to Binder, faults within any software system are likely to occur at execution points, where features exchange information with other features or influence one another [Bin99, page 557]. We define this situation as feature interaction and align with the definition by Ferber et al. of [FHS02]:

Definition 24 (Feature Interaction). *A feature f_i interacts with a feature f_j if the behavior of f_i depends on whether f_j is present or absent.*

A very detailed but outdated summary about feature interaction is provided by Calder et al. [CKMRM03]. There, the authors categorize feature interaction into three different domains of research: Software Engineering, formal methods, and on-line techniques. Ferber et al. [FHS02] identify five different kinds of feature interactions and dependencies on the Basis of an Electronic Control Unit (ECU) case study. Those are: Intentional Interaction, Resource-Usage Interaction, Environment Induced Interaction, Usage Dependency, and Excluded Dependency. The dependencies and interactions are not directly added to the feature model but modeled within an additional view. However, this approach focuses on the management

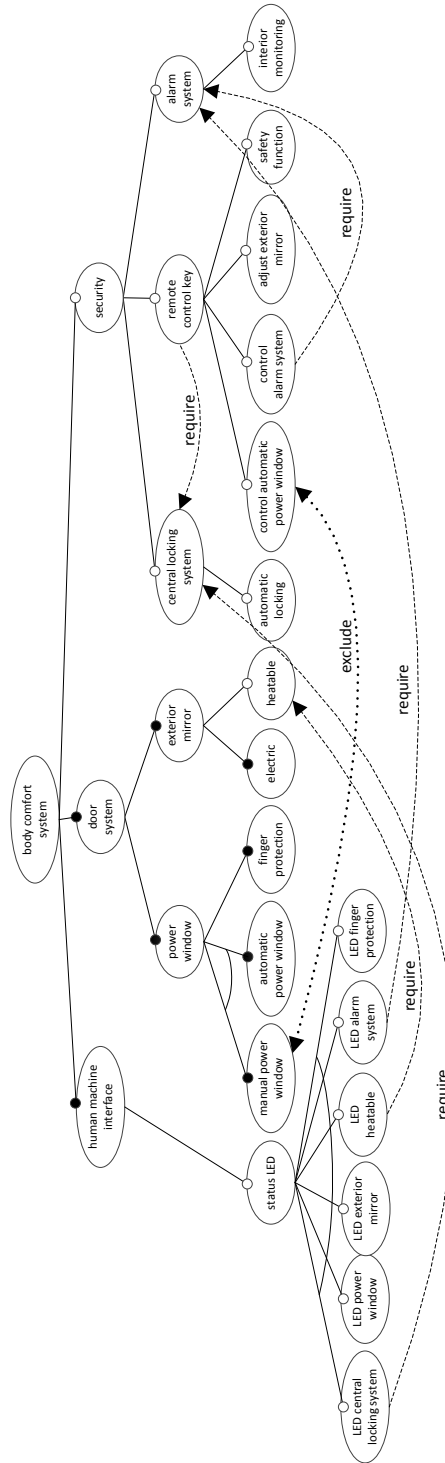


Figure 19: Feature model of our running example

of feature interactions, omitting techniques to actually identify them. Metzger et al. describes an algorithm for feature interaction detection in embedded systems in [Met04]; in [MBLP05] the authors describe the adaptation of this algorithm to apply it to the SPL context. The detection of feature interactions is based on a metamodel of the development products describing the development artifacts and the relations between them. The information about feature interactions are extracted from the requirements. In [JDDJ08] the authors define feature interaction on the basis of the set of actuators used by the features. This results in a set of properties in temporal logic detecting feature interactions by using model checking. Due to the application of model checking, this approach faces the problem of state space explosions when working on rather big models. Lochau et al. introduces feature interaction detection on the basis of reachability trees that can be generated on the basis of statecharts describing the behavior of a system or SPL [LG10].

Potential feature interactions occur when features interrelate. In SPL engineering we distinguish two different ways of feature interrelation:

1. via relations and constraints defined in the feature model and
2. via interactions which originate from the implementation of the SPL and are not captured in the feature model.

First, we will consider **case 1**) of feature interrelations originating from the feature model itself: We distinguish three types of *binary* dependencies for every feature pair $\{f_i, f_j\} \subseteq F$ organized in a feature model $FM(F)$. For each type we present an example using the feature model of our running example (cf. Figure 19).

Definition 25 (Implication). f_i *implies* f_j , written $f_i \Rightarrow f_j$: In a feature model the presence of feature f_i in a configuration *implies* the presence of feature f_j in that configuration:

$f_i \Rightarrow f_j \Leftrightarrow \forall F : f_i \in F \rightarrow f_j \in F$. This relation is in general per definition reflexive, non-symmetrical, and transitive.

For example, the transitivity of this relation can be proved easily:

$$f_i \Rightarrow f_j \wedge f_j \Rightarrow f_k$$

then

$$\forall F : (f_i \in F \rightarrow f_j \in F) \wedge (f_j \in F \rightarrow f_k \in F)$$

holds. Thus, it follows:

$$\forall F : f_i \in F \rightarrow f_k \in F$$

and thus

$$f_i \Rightarrow f_k$$

In a feature model this relation occurs as follows:

- Features always imply their parent feature: if $(f_j, f_i) \in C$ then $f_i \Rightarrow f_j$ (e.g. *status LED* implies *human machine interface*),
- Features always imply their *mandatory* child features: if $(f_i, f_j) \in R_{\text{man}}$ then $f_i \Rightarrow f_j$ (e.g. *door system* implies *power window*), and
- Features always imply features they require: if $(f_i, f_j) \in C_{\text{req}}$ then $f_i \Rightarrow f_j$ (e.g. *remote control key* implies *central locking system*).

If two features require each other, for example, if f_i implies f_j and f_j implies f_i , we write $f_i \Leftrightarrow f_j$ for short.

Definition 26 (Exclusion). f_i *excludes* f_j , written $f_i \not\sqsubseteq f_j$: The presence of a feature f_i in a configuration implies the absence of feature f_j in that configuration.

$$f_i \not\sqsubseteq f_j :\Leftrightarrow \forall F : \neg f_i \in F \vee \neg f_j \in F$$

Thus, $\not\sqsubseteq \subseteq F \times F$ is in general per definition irreflexive, symmetrical, and non-transitive.

In a feature model the exclude relation occurs as follows:

- alternative features exclude each other: if $(f, F_i) \in R_{\text{alt}}$, then $f_j \not\sqsubseteq f_k$ for each pair $\{f_j, f_k\} \subseteq F_i$ (e.g. *manual power window* excludes *automatic power window*), and
- excluded features: if $(f_i, f_j) \in C_{\text{exc}}$, then $f_i \not\sqsubseteq f_j$, and $f_j \not\sqsubseteq f_i$ e.g. *manual power window* excludes *control automatic power window*).

Definition 27 (Independence). f_i and f_j are *independent*, written $f_i \perp f_j$. In all other cases—if two features do not exclude or imply each other, those features are defined as independent, written $f_i \perp f_j$, where:

$$\perp = F \times F \setminus \{\Rightarrow \cup \not\sqsubseteq\}$$

Independence means that their presence/absence within the same product configuration is mutually independent. The relation \perp is irreflexive, symmetrical, and non-transitive.

	positive	negative
intended	feature cooperation	feature vetoing
unintended	undesired interference	required but missing

Table 5: Categories of feature interactions

Please note that we only address binary feature interactions caused by our definition of feature models. A feature model containing n-ary constraints, e.g. a feature requires the combination of several others, would result in n-ary interactions which are defined analogous to the definitions above. The reason for this restriction is that we focus to examine and evaluate MoSo-PoLiTe on the basis of pairwise feature interaction coverage.

With regard to **case 2**, those feature interactions are not easy to identify or to localize. Various related publications address those kind of feature interactions. Feature interactions can only be revealed by analyzing varying feature combinations by means of different configurations. There, we run into the same problem as defined in the beginning. Intuitively, to guarantee a 100% coverage of feature interactions, all possible configurations would need to be tested.

To further discuss feature interaction coverage we introduce the categories depicted in Table 5. Feature interactions can be either *intended*, i.e., being an integral part of the product line requirements or they are *unintended*, therefore potentially leading to failures. We assume that all intended feature interactions are included in the feature model of the SPL.

Interactions might arise at different levels of abstraction, e.g., by shared resource accesses and even environmental influences [FHS02]. We focus on the functional/-logical level detectable on the basis of test model specifications.

We further distinguish positive and negative interactions. A positive interaction is defined by the cooperation of features, e.g. if two features complete one another to realize some functionality. The counterpart is defined by negative interactions, where features hinder one another.

According to these definitions we obtain the following four cases for feature interaction:

- **Feature cooperation** is the positive and intended feature interaction, where features cooperate with each other to implement a certain functionality. Generally, we assume that these interactions can be characterized in the feature model. This would require a dependency analysis of the implementation or architecture.
- **Undesired interference** is the positive and unintended feature interaction, where features cooperate leading to undesired influences and even failures.
- **Feature vetoing** is the negative and intended feature interaction, where features hinder each other of being executed. This interaction is vital for e.g.

safety critical reasons. With regard to our running example *central locking system* vetoes *automatic power window* to (re-)open the window while being locked. This information is not depicted within the feature model.

- **Required but missing** is the negative and unintended feature interaction, where features should cooperate but cannot be caused by missing features.

Generally, dependencies and interaction interrelations among features are incomparable, i.e., dependencies do not imply interactions, and interactions do not imply dependencies. Therefore, we further distinguish *mandatory* and *optional* feature interactions of non-excluding interacting features f_i and f_j :

- if $f_i \Rightarrow f_j$ then f_i *mandatorily interacts* with f_j , else f_i *optionally interacts* with f_j .
- if $f_i \Leftrightarrow f_j$, then f_i and f_j *mandatorily interact*, and if $f_i \perp f_j$, then f_i and f_j *optionally interact*.

According to this definition, every feature *mandatorily interacts* with all its parent features in the feature model by *varying* them.

Our combinatorial SPL testing approach for SPLs selects feature combinations under some criterion to test their *intended* interactions to be correct, and to rule out *unintended* interactions. Potential interactions then imply test coverage obligations, e.g., by introducing interaction fragments like coordination patterns, behavioral influences etc. to be covered in the test model. The feature interactions, which are not characterized in the feature model, may include more than two features thus leading to an n-ary feature interaction. Within our case study and evaluation we will examine the occurrence of n-wise feature interactions and the capability of MoSo-PoLiTe to cover those when only using pairwise feature combination.

6.3 FEATURE MODEL AS PROPOSITIONAL FORMULA

According to Czarnecki et al. [CW07] feature models can be expressed as logical formulas. We will use propositional formulas in the remainder of this part to check the semantical equivalence between different feature models.

The relations: *mandatory*, *optional*, *alternative*, and *or* are defined as:

- **Mandatory:** The *mandatory* feature *Human Machine Interface* is always selected if its parent feature (*Body Comfort System*) is selected. Furthermore, the selection of *Human Machine Interface* implies that the parent is selected as well.
- **Optional:** The selection of the *optional* feature *Safety Function* indicated that its parent is also selected.
- **Alternative:** Within an *alternative* group exactly one element has to be selected if its parent feature is selected. Thus, *Electric* implies either *Automatic Power*

Name	Notation	Propositional Formula
Mandatory		$(B \rightarrow H) \wedge (H \rightarrow B)$
Optional		$S \rightarrow H$
Alternative		$E \rightarrow alt(A, M, ..n) \wedge$ $(A \rightarrow E) \wedge (M \rightarrow E) \wedge$ $(..n \rightarrow E)$
Or		$s \rightarrow (Lc \vee Lp \vee ..n) \wedge$ $(Lc \rightarrow s) \wedge (Lp \rightarrow s) \wedge$ $(..n \rightarrow s)$

Figure 20: Relations in logic

Window or *Manual Power Window*. Additionally, the selection of one element of the *alternative* group always implies its parent feature (*Electric*).

- **Or:** Within an *or* group at least one element has to be selected. Thus, *status LED* implies *LED central locking system* or *LED power window*. Since the parent needs to be selected if an element of the *or* group is selected, both, *LED central locking system* and *LED power window* imply *status LED*.

The cross-tree constraints *require* and *exclude* are defined as depicted in Figure 21.

A feature model can be interpreted as a combination of those propositional formulas. If Φ_i is the set of propositional formulas for i features/feature groups, then a feature model can be interpreted as a propositional formula consisting of: $\bigwedge_{n=1,..,i} \Phi_n$. A valid configuration of an SPL is an assignment of the feature model where the logical expression turns to true — in other words: all valid assignments of the different relationships and constraints have to hold. With regard to feature interactions mandatory interaction can be identified by translating the feature

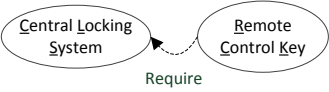
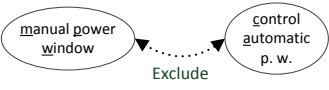
Name	Notation	Propositional Formula
Require		$(RCK \rightarrow CLS)$
Exclude		$(\neg mpw \vee \neg ca)$

Figure 21: Cross-tree constraints in logic

model into propositional formulas. The implies (\rightarrow) function directly indicates when two features mandatorily interact.

DUE to the enormous number of possible products of industrial SPLs, it is not feasible to check each individual product against its requirements. The number of possible products depends on the number of varying features parameterizing the SPL. Also for ordinary single systems, an exhaustive test including all possible combinations of parameters and their values is not feasible [Ber91]. Testing an SPL, therefore, has to cope with similar trade-offs like testing a single system with its input parameters.

A logical consequence is to take lessons learned into account from the field of software testing reducing the test effort with regard to parametrization. Combinatorial and especially pairwise testing are widely spread approaches to reduce the testing effort [GOA05, CGMC03, KLK08]. This is why we consult a method such as combinatorial testing for SPL validation and verification.

We will treat features of the feature model as parameters. Instead of generating combinatorial sets of input parameter values, we generate combinatorial sets of configurations. This set is then significantly smaller than the total number of possible products. A second reason for applying combinatorial testing is to cover feature interactions. Generating configurations with T-wise combination of features results in a T-wise feature interaction coverage.

As we have already stated in our related work section, this idea is not new and former contributions have already proposed this idea [McG01, Olio8]. However, those approaches do not provide a systematic method of how to apply combinatorial testing to feature models or similar variability representations. Furthermore, those approaches neglect feature interactions and are not able to consider constraints between features.

Applying combinatorial testing to feature models is challenging because the hierarchical structure, the different relations, and constraints of the feature model are not processable by standard combinatorial test algorithms. Configuring a product in SPL engineering can be considered as a Constraint Satisfaction Problem (CSP) and each possible configuration is a possible solution to that CSP [SDD⁺09]. Due to the fact that the feature model, as defined in Section 6 Definition 22, includes almost entirely binary constraints it can be translated into a binary CSP [OMR10]. Thus, the combinatorial algorithm needs to include standard constraint solving techniques such as Forward Checking [HE79, BG95], which is well-suited for handling binary constraints between variables [Ben04, WW09, SDD⁺09]. Current methods providing combinatorial testing considering constraints are either commercial so that the internal structure cannot be analyzed or are based on SAT-solving. However, since

feature models contain more or less only binary constraints, Forward Checking seems to be the natural choice [Beno4, WW09].

To apply combinatorial testing to feature models we:

1. transform the feature model to get rid of the hierarchical structure resulting in a binary CSP, followed
2. by the implementation and application of a combinatorial algorithm capable of solving constraints.

In the following, we will refer to the transformation of the feature model as **feature model to CSP transformation**. This **transformation** translates the feature model into a feature model-based representation of parameters and their corresponding values. The restrictions based on the relations and the hierarchical structure are then replaced by additional constraints. The feature model CSP will only contain parameters (features) and their corresponding values (the possible feature allocations within configurations).

To the combinatorial algorithm we will refer to as **subset extractor** because the algorithm generates a set of configurations that is a subset of all possible configurations. A formal definition and details about the feature model to CSP transformation and the subset extractor will follow within this chapter.

7.1 CONSTRAINT SATISFACTION PROBLEM

A constraint satisfaction problem (CSP) is a high-level description of a problem based on constraints. The framework of a CSP is given by a set of variables and a set of values. The actual problem is given by a set of constraints specifying relations between the variables. The search for possible values so that all variables fulfill the set of constraints is called constraint satisfaction or solving.

Popular applications of CSPs in real world situations are for example:

- wiring harness to find the optimal cabling within a car.
- stand allocations for airports
- circuit layout computation in electrical engineering

A CSP is called a binary CSP if the constraints only address pairs of variables and the corresponding values. A binary CSP is defined as:

Definition 28. *A binary CSP is a triple $\langle V, D, C \rangle$ with*

- $V = \{v_1, \dots, v_n\}$ is a finite set of variables.
- $D = \{D_1 \cup \dots \cup D_n\}$ the set of values for each variable, where $D_i = \{d_{i_1} \dots d_{i_q}\}$ is a set of possible values for v_i .

- $C = \{c_1, \dots, c_k\}$ is a set of constraints. Each constraint is a triple $\langle x, R, y \rangle$ with $x, y \in V$ are pairs of variables
 R is a set of binary relations $R \subseteq D \times D$ restricting the possible values for the combination of the variables x and y .

An ordinary CSP can be defined analogously. A **solution** of the binary CSP is an assignment that satisfies every constraint. A solution is a function that assigns to every variable v_i a value $f(v_i) \in D_i$, such that

$$\forall \langle x, R, y \rangle \in C : (f(x), f(y)) \in R$$

A CSP is satisfied if it has at least one solution.

With regard to our feature model, this definition has to be slightly adapted:

- Variables V within the feature model are the features themselves
- The domain of possible values D needs to be extracted from the hierarchical structure and the relations. For example, the *optional* feature security within the BCS feature model can be part of a configuration or not. Thus, security has the possible values: security and \neg security, where \neg security represents the situation, where security is not selected within the configuration.
- The cross-tree constraints *require* and *exclude* that additionally restrict feature combinations are the constraints within the binary CSP.

The extraction of the binary CSP consists of the following steps:

1. Resolve the hierarchical structure of the feature model. This step results in a flat feature model, where each feature represents a parameter (variable) of the CSP.
2. Then, the values are extracted on the basis of the relations. This step defines the domain of values for the binary CSP.
3. Finally, the binary CSP is extracted from the feature model representation.

On the basis of the binary CSP the problem that needs to be solved can be formulated. The problem is to find a minimal set consisting of configurations covering all valid pairs (or all T-wise combinations) of features that are permitted combinations within a configuration. To generate configurations, our algorithm covers corresponding assignments of feature values to variables that fulfills all constraints. Such a representative set of products that covers all pairs of features is called a *hitting set* [Scho7]. Actually, the optimization problem of finding the *minimal* subset of products is the minimum cardinality hitting set problem that is, in the general case, NP-hard [Scho7]. Our pairwise configuration selection algorithm

is a heuristics that always finds a hitting set, but it does not guarantee to find the minimal set of configurations.

In the next section we introduce the rules to translate the feature model into a binary CSP [OMR10]. Afterwards, the algorithm that generates a set of valid configurations containing all valid pairs of features is introduced. Testing this set of products is equivalent to pairwise testing the entire SPL. The developed pairwise algorithm can handle *constraints* between features and guarantees the generation of valid configurations containing all valid pairs of features.

7.2 FEATURE MODEL TO CSP TRANSFORMATION

CSP algorithms as well as ordinary pairwise algorithms require variables/parameters and corresponding values to operate on. Please note that the term **variable** is settled in the CSP community and the term **parameter** is a general term in the combinatorial testing community. Due to the fact that we combine both approaches, we use both terms as synonyms. To ease the understanding and improve the readability of this thesis we want to prevent to switch between those two terms repeatedly. Thus, we stick with the term **parameter** in the remainder of this thesis.

In our case, parameters and their values need to be extracted of a feature model to transform a feature model into a CSP. As stated in the beginning of this part of the thesis, features are the parameters of an SPL. Extracting parameters and values is a two step procedure:

1. **Flattening:** Every feature with its associated relations and constraints is iteratively pulled up until it is placed directly beneath the root feature. Every feature then serves as a parameter. We refer to this feature model as **flat feature model** (FFM).
2. **Value Extraction:** The algorithm assigns every parameter its correspondent parameter values. The values of a feature are defined by its relationship to its former parent feature. E.g. an *optional* feature has the values true and false. We refer to this feature model as **feature model CSP** (FMCSP).

Figure 22 depicts the feature model to CSP transformation process. We will first explain the different flattening rules followed by a description of how we extract parameter values. Then, we will demonstrate both steps of our algorithm by using our running example.

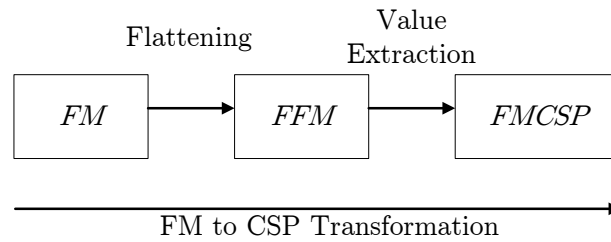


Figure 22: Feature model to CSP transformation

7.2.1 Flattening

Several model transformation rules control the flattening process.

Each rule is iteratively applied to a subtree of a feature model. This subtree always consists of three levels: the grandparent feature, the parent feature, and the child feature. Figure 23 depicts an example of such a subtree.

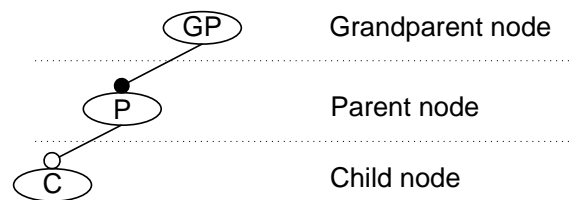


Figure 23: Subtree for flattening

Different rules are required for the flattening process depending on the relations between the involved features. Our feature model supports four different relations (*optional*, *mandatory*, *alternative*, and *or*) and for every possible combination of parent and child relationship a separate transformation rule is required. Therefore, we need $4 \times 4 = 16$ rules. Those 16 rules are sufficient to ensure that all feature models according to the definition in Chapter 6 (a feature model with *optional*, *mandatory*, *alternative*, and *or* relations and binary *require* and *exclude* constraints) can be transformed. This allows us to claim that our transformation algorithm assures completeness. Each rule can be shown to preserve the semantical equivalence by translating the original and the resulting subtree into propositional formulas and to check whether these rules form a tautology. In the following, all 16 transformation rules are introduced and for the first four, the semantical equivalence will be proved. Proving the semantical equivalence preservation property for all rules confirms the consistency of the flattening algorithm. A summary of all transformation rules including the proof of semantical equivalence can be found in the appendix of this

thesis. The transformation rules can be checked automatically using a SAT-solver and check whether:

(prop. formula of the leftside \leftrightarrow prop. formula of the rightside)

turns to true or false. If both implications result in true, both sides are semantically equivalent.

In the following, we present all 16 transformation rules using a domain specific notation (feature subtrees) to support the readability. As we will discuss in the Implementation and Evaluation part of this thesis, those rules can be translated easily into precisely defined graph transformation rules as well as into Java code. Please note that the flattening algorithm is not reversible. It is not possible to generate the original feature model using the flat feature model in the general case.

Child Optional:

Figure 24 depicts the transformation rule pulling up an *optional* child C located beneath a *mandatory* parent P .



Figure 24: Transformation rule pulling up an *optional* child beneath a *mandatory* parent

The former child feature is pulled aside its parent feature directly beneath the grandparent feature. To prove that this transformation preserves the semantical equivalence we translate both sides into propositional formulas and show that it is a tautology.

$$\begin{array}{ll}
 GP \wedge (P \rightarrow GP) \wedge (C \rightarrow P) \wedge & GP \wedge (P \rightarrow GP) \wedge (C \rightarrow GP) \wedge \\
 (GP \rightarrow P) & (GP \rightarrow P) \\
 = GP \wedge P & = GP \wedge P
 \end{array}$$

Figure 25 depicts the transformation to flatten an *optional* child/*optional* parent subtree. Again, the C feature is simply pulled up. To ensure that C cannot be selected without its former parent (P) being selected, an additional *require* constraint is required. Again, the translation into propositional logic helps to prove the semantical equivalence.

$$\begin{array}{ll}
 GP \wedge (P \rightarrow GP) \wedge (C \rightarrow P) & GP \wedge (P \rightarrow GP) \wedge (C \rightarrow GP) \wedge (C \rightarrow P) \\
 = GP \wedge (C \rightarrow P) & = GP \wedge (C \rightarrow P)
 \end{array}$$



Figure 25: Transformation rule pulling up an *optional* child beneath an *optional* parent

Figure 26 depicts the transformation of an *optional* child / *alternative* parent subtree.



Figure 26: Transformation rule pulling up an *optional* child beneath an *alternative* parent

In accordance with the previous transformation rule, C is pulled up and requires its former parent. Please note that the dots and additional lines between P_1 and P_n symbolize that the *alternative* group may consist of arbitrary elements.

$$\begin{aligned} & GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge (C \rightarrow P_1) \\ & \wedge (GP \rightarrow \text{alt}(P_1, P_n)) \\ = & GP \wedge (C \rightarrow P_1) \wedge \text{alt}(P_1, P_n) \end{aligned}$$

$$\begin{aligned} & GP \wedge (C \rightarrow GP) (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \\ & \wedge (GP \rightarrow \text{alt}(P_1, P_n)) \wedge (C \rightarrow P_1) \\ = & GP \wedge (C \rightarrow P_1) \wedge \text{alt}(P_1, P_n) \end{aligned}$$

Figure 27 depicts the transformation triggered by a subtree including an *optional* child beneath a parent within an *or* group. Again, the child feature C is pulled up and an additional *require* constraint is added to its former parent. Again, please note that the dots and additional lines between P_1 and P_n symbolize that the *or* group may consist of arbitrary elements.



Figure 27: Transformation rule pulling up an *optional* child beneath an *or* parent

$$\begin{aligned} & GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \\ & \wedge (GP \rightarrow P_1 \vee P_n) \wedge (C \rightarrow P_1) \\ = & GP \wedge (P_1 \vee P_n) \wedge (\neg C \vee P_1) \end{aligned}$$

$$\begin{aligned} & GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \\ & \wedge (GP \rightarrow P_1 \vee P_n) \wedge (P_1 \rightarrow GP) \wedge (C \rightarrow P_1) \\ = & GP \wedge (P_1 \vee P_n) \wedge (\neg C \vee P_1) \end{aligned}$$

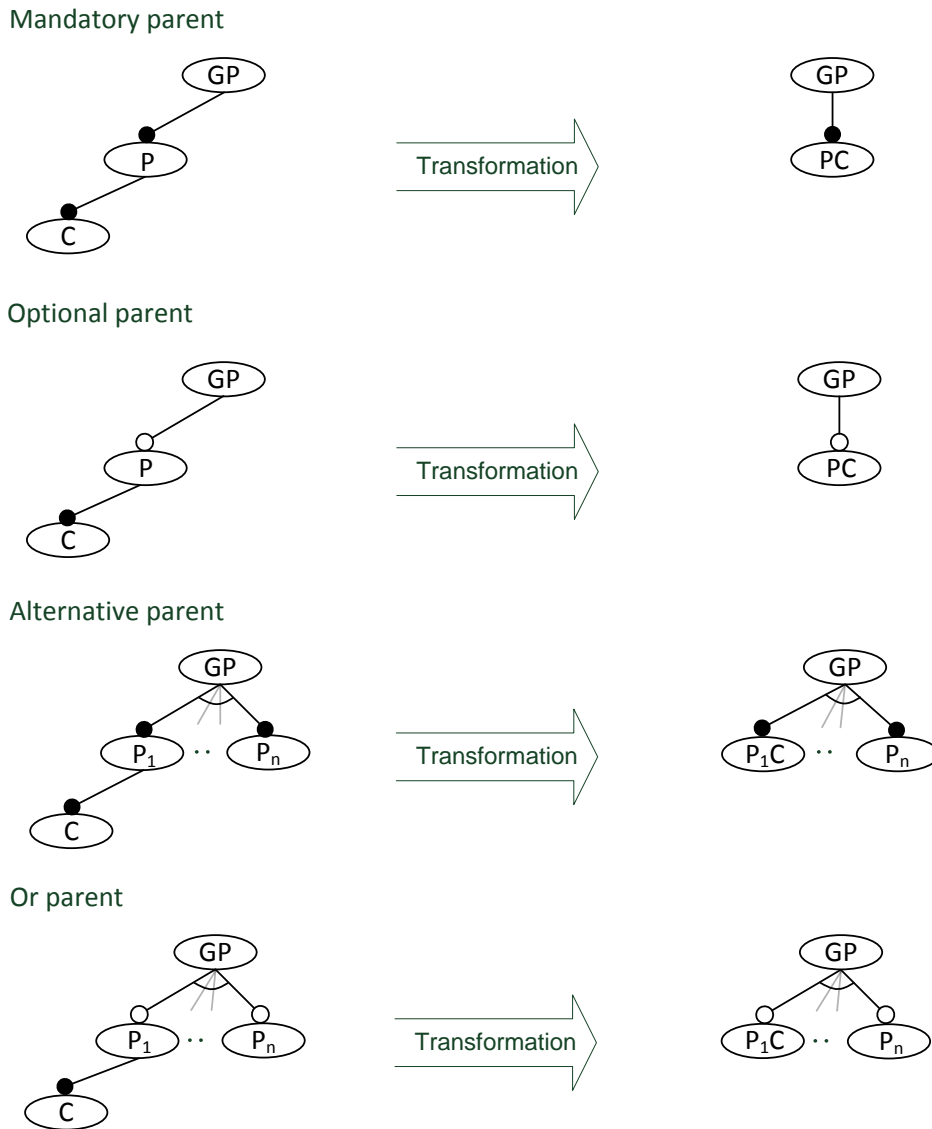


Figure 28: Transformation rule pulling up a *mandatory* child beneath arbitrary parents

Child Mandatory

Pulling up a *mandatory* child beneath a *mandatory* parent results in a combined feature including the child and the former parent feature. Both features *P* and *C* can be combined since it is not possible to include one of them without the other one within any configuration. Combining features means to merge them into one single feature. The name of the new feature is a concatenation of the feature names merged within this feature. Beside this characteristics, the rules are similar to the ones considering *optional* children.

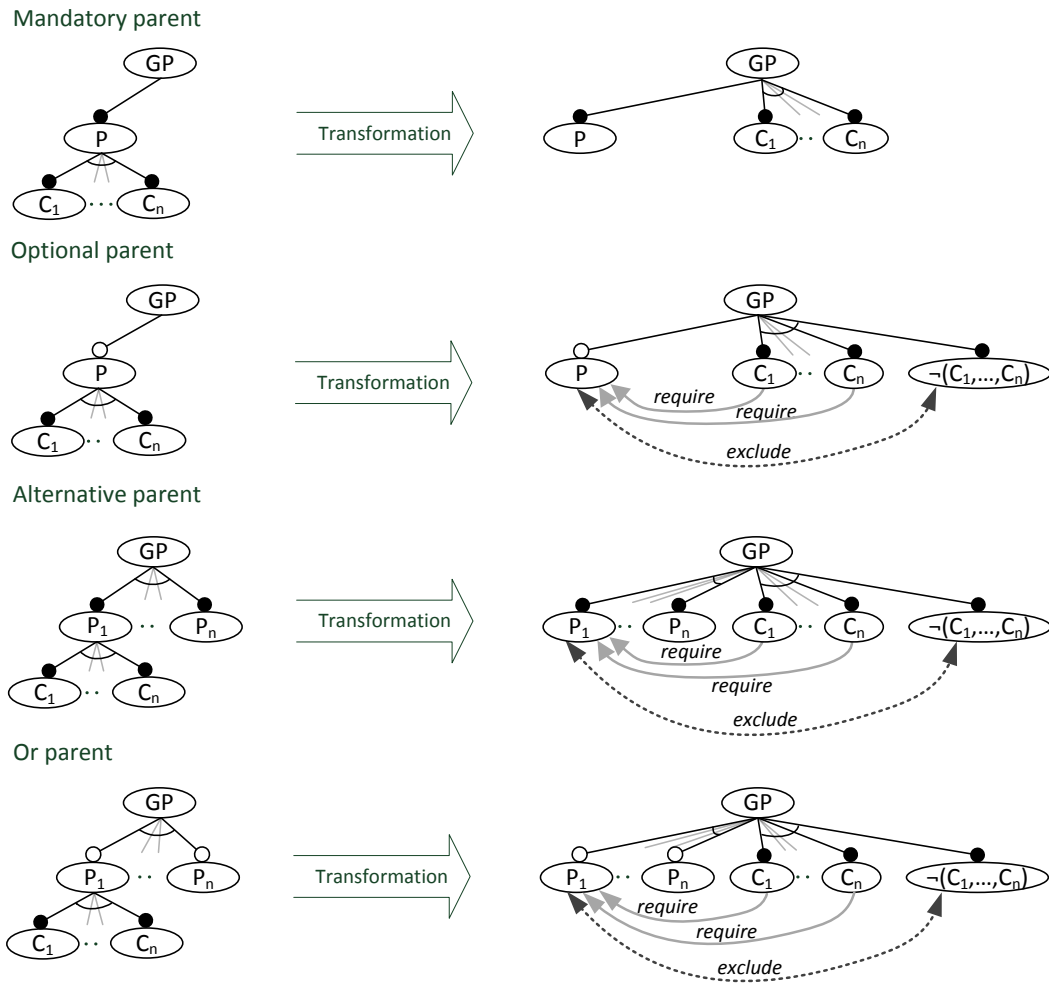


Figure 29: Transformation rule pulling up an *alternative* child beneath arbitrary parents

Child Alternative:

An *alternative* child group is pulled up aside the former parent feature directly beneath the grandparent feature. Figure 29 depicts the four rules pulling up an *alternative* child group.

The first rule considers a *mandatory* parent. The *alternative* group is simply pulled up directly beneath the grandparent feature. Considering the other constellations, additional *require* and *exclude* constraints have to be added. For all three cases, an additional negation-feature (\neg) is added to the *alternative* group representing the case when no element of the *alternative* group is chosen. It simply represents the negation of the elements of the *alternative* group. This is very important due to the fact, that the *alternative* group must not be selected at all if the former parent feature is not selected.

- if the parent P is *optional* and not selected, the underlying *alternative* group is not selected as well. To keep this semantics when pulling up the *alternative* group the $\neg(C_1, C_n)$ -feature is added. Both former elements of the *alternative* group, namely C_1 and C_n require the former parent feature P . P and $\neg(C_1, C_n)$ exclude each other since $\neg(C_1, C_n)$ may only be active if P is deselected.
- if the parent is within an *alternative* group, the $\neg(C_1, C_n)$ feature excludes the former parent feature P_1 of the *alternative* group and C_1 and C_n require its former parent P_1 .
- if the parent is within an *or* group, the $\neg(C_1, C_n)$ feature excludes the former parent feature P_1 of the *alternative* group and C_1 and C_2 require its former parent P_1 .

Please note that at the first glance, the rules considering the parent of being within an *alternative* or *or* group can be changed so that the $\neg(C_1, C_n)$ feature requires feature P_n . This is true if the *alternative* and the *or* group only consists of two features. With increasing elements of these groups too many *require* constraints would result. Thus, we recommend that $\neg(C_1, C_n)$ excludes P_1 .

Child Or:

Flattening a subtree with child features within an *or* group is similar to the rules considering an *alternative* child group. Figure 30 depicts the four corresponding transformation rules.

Applying the flattening rules to a feature model removes its hierarchical structure. The 16 transformation rules transform the feature model into a flat feature model (FFM).

The transformation rules considering *alternative* and *or* child groups might raise the impression that they are only capable of processing two child features. The dots and the additional lines between P_1 and P_n and between C_1 and C_n indicate that an arbitrary number of additional features may be placed within the *alternative* and *or* groups.

To foster the understanding of the flattening algorithm, we apply the flattening algorithm to our running example. We use a subset of the BCS feature model, depicted in Figure 31. In the following, we refer to this feature model as BCS-small. BCS-small allows for the derivation of 40 different configurations.

We will only present some steps of the **flattening** algorithm. Figure 32 depicts the flattening, starting at the left side of the feature model. First, **LED central locking system** and **LED power window** are pulled aside **status LED**. Each of the elements of the *or* group need a *require* constraint towards their former parent feature (**status LED**). Then, **LED central locking system**, **LED power window**, and **status LED**

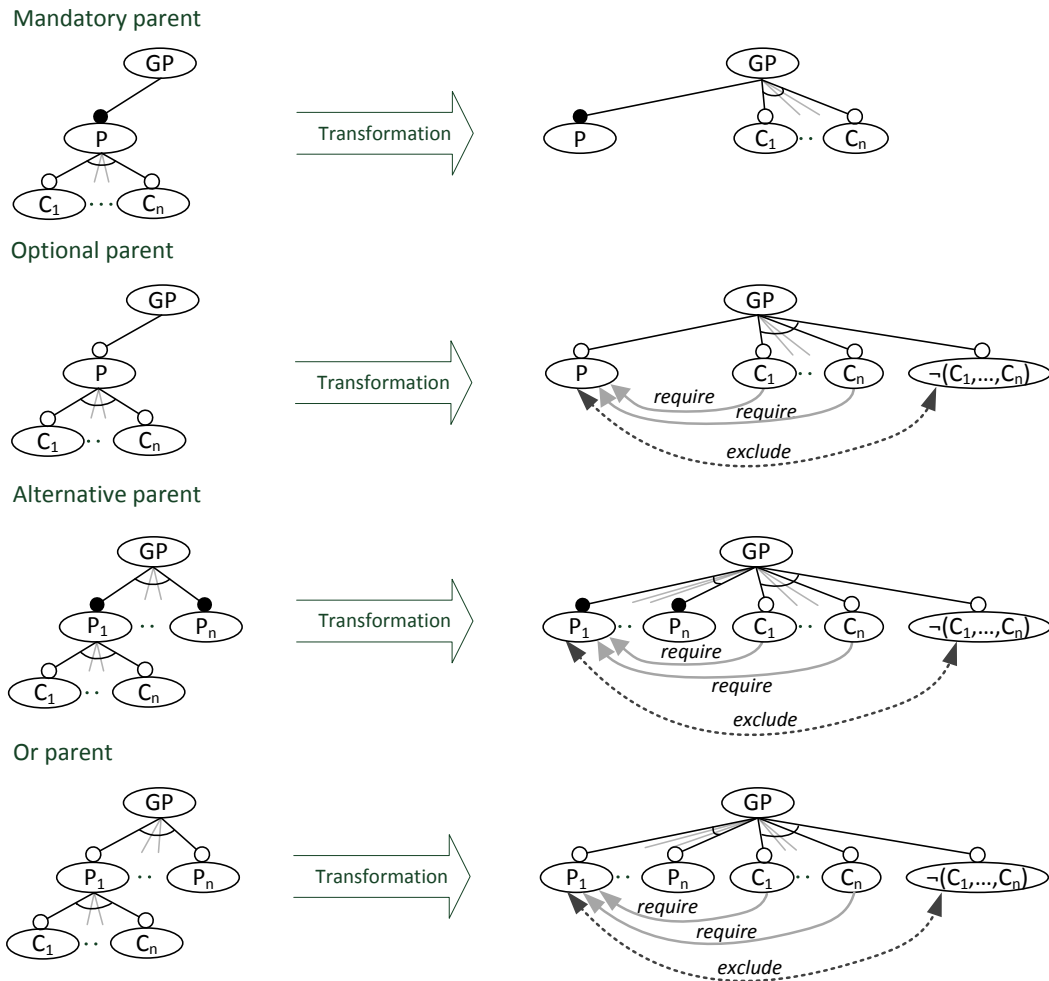


Figure 30: Transformation rule pulling up an *or* child beneath arbitrary parents

are pulled up aside **human machine interface**. Since **human machine interface** is a *mandatory* feature, no additional constraint is required. The *require* constraint starting at **LED central locking system** towards **central locking system** persists.

Figure 33 depicts the result of the flattening of the subtree beneath **door system**. First, the *alternative* group consisting of **manual power window** and **automatic power window** is pulled up aside **power window**. Since **power window** is a *mandatory* feature, no additional constraint is required. Then, **power window**, **manual power window**, and **automatic power window** are pulled up aside **door system**. Again, no additional constraint is required because **door system** is a *mandatory* feature.

Finally, 34 depict the FFM of our subset of the BCS feature model. In the last step, all features beneath **security** are pulled up. Since all features are *optional*, several additional *require* constraints are necessary.

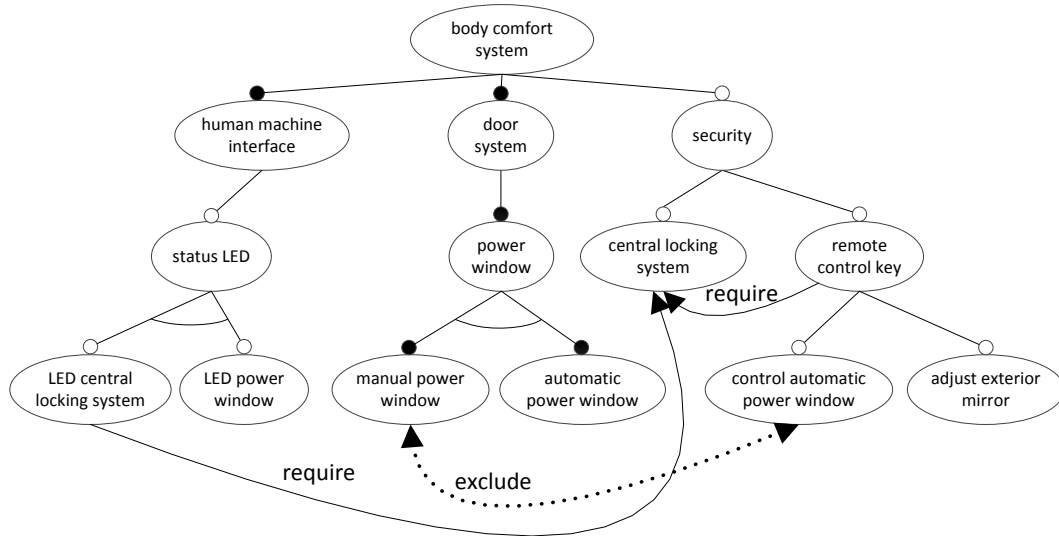


Figure 31: Subset of the BCS feature model

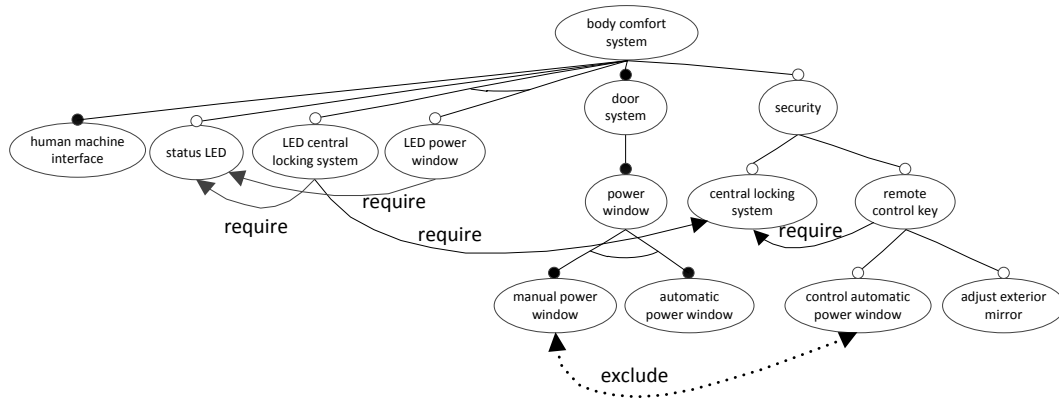


Figure 32: Step 1 of the flattening

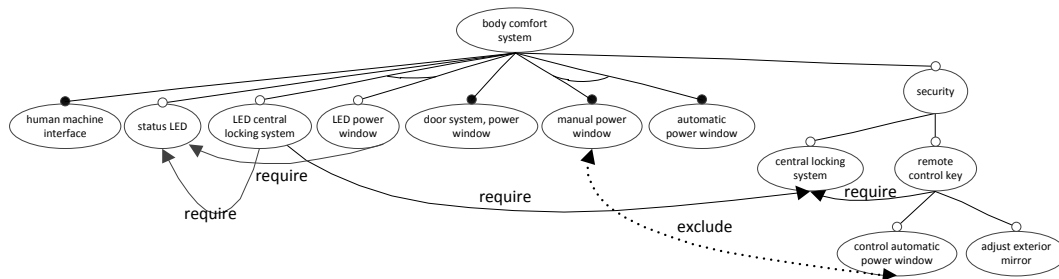


Figure 33: Step 2 of the flattening

Again, we like to emphasize that all cross-tree constraints within BCS-small are preserved during the flattening process. All cross-tree constraints remain with the

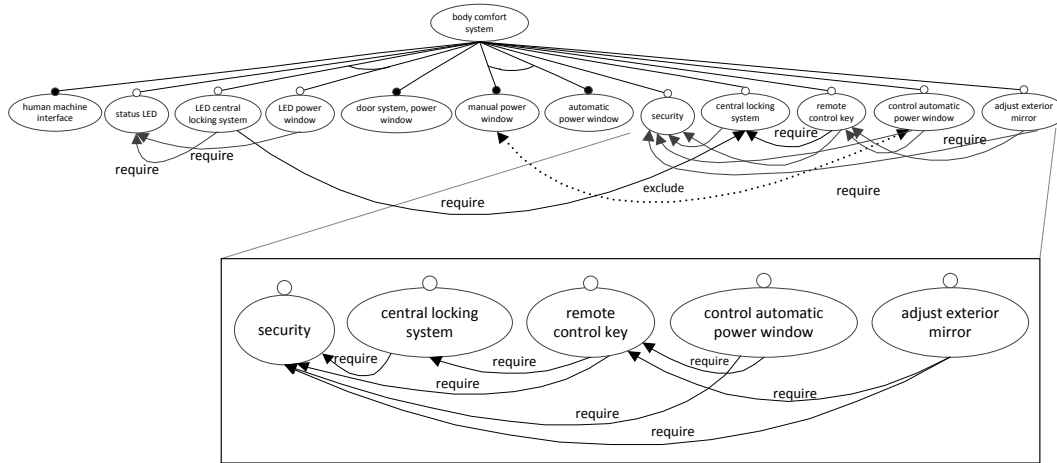


Figure 34: Step 3 of the flattening

corresponding features. Since no feature is deleted no constraint will get lost. A merged feature inherits the constraints of the included features.

7.2.1.1 Completeness and Consistency

All 16 transformation rules ensure completeness and consistency of the subtree. As already mentioned, the 16 rules cover all possible combinations of the different relationships (*mandatory*, *optional*, *or*, and *alternative*) for the three level subtree. Thus, we can state that the 16 transformation rules guarantee completeness. Using the translation into propositional formulas, we prove the consistency of each transformation rule (each transformation preserves the semantical equivalence).

With regard to applying the transformation rules to an entire feature model, completeness and consistency need to be assured as well. Due to the fact that we restrict ourselves to a feature model using the syntax and semantics introduced in Chapter 6 Definition 22, we can guarantee that the feature model is either already flat or contains at least one three-level subtree as previously described. Thus, we can state that the transformation rules with regard to operating on an entire feature model are complete as well, since our algorithm can process every feature model that aligns with the definition in Chapter 6.

To prove that the transformation rules assure that the flat feature model is semantical equivalent to the original feature model (consistency), we exploit the fact that all 16 rules preserve the semantics. Figure 35 depicts our proof sketch, where FM_1 represents the feature model that will be flattened to become FM_2 .

In our algorithm, FM_1 is analyzed bottom up and each three-level-subtree is matched to the left-hand side L iteratively. The matching algorithm is injective to ensure that each feature within the subtree is processed individually. Each child within an *alternative* or *or* group is processed as previously described (cf. Figure 30 and 29). It is also possible to process subtrees with mixed notations. Then, the

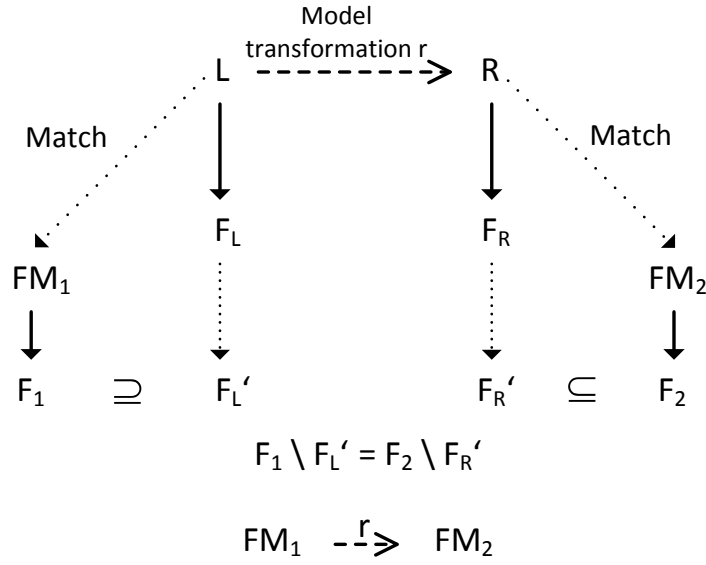


Figure 35: Procedure to proof the semantical equivalence

different notations are pulled up sequentially. The matched three-level subtree can be translated into a propositional formula F_1 . This translation is depicted as a bold arrow.

At the top of Figure 35 the model transformation (r) from left (L) to right (R) represents the 16 transformation rules. As we have already mentioned, both sides can be translated into propositional formulas. This step is depicted by two arrows from L to F_L and R to F_R respectively. The logical expression is represented by F_L for the left-hand side and by F_R for the right-hand side.

In the next step, the feature names within F_L and F_R are replaced by the concrete names used in the feature model ($F_L \rightarrow F_L'$ and $F_R \rightarrow F_R'$). This step is again an injective mapping. New feature names in R that did not exist in L (e.g. merged features) are mapped on new names in $FM_2(F_2, F_R')$ which are not used in $FM_1(F_1, F_L')$

Then, $F_1 \supseteq F_L'$ holds, since otherwise the subtree within the feature model wouldn't have matched the one in the transformation. In other words, F_1 implies F_L' but not vice versa.

On the right-hand side the same relations hold. The resulting feature model FM_2 consists of subtrees e.g. F_2 , where $F_2 \supseteq F_R'$ with F_R' being the resulting flat subtree of the transformation rule with the correct feature names.

We already proved that for all 16 transformation rules $F_L \Leftrightarrow F_R$ holds. Thus, we can also state that $F_L' \Leftrightarrow F_R'$. Furthermore, we know that $F_1 \setminus F_L' = F_L' \setminus F_R'$ then $F_2 = F_1 \setminus F_L' \cup F_R'$ i.e. F_2 can be described by removing F_L' from F_1 and then adding F_R' . Thus, we need to prove that F_1 is sufficient to generate F_2 to prove that the transformation rules preserve the semantical equivalence.

Notation: $F \Leftrightarrow F'$ complies with: $F \vdash F'$ and $F' \vdash F$. Every formula in F' can be generated using F and vice versa. Thus, we need to show: $F_1 \vdash F_2$ and $F_2 \vdash F_1$

$$F_2 = F_1 \setminus F'_L \cup F'_R$$

$F_1 \vdash F_1 \setminus F'_L$ holds per definition

$F_1 \vdash F'_R$ holds, since $F_1 \supseteq F'_L \vdash F'_R$

due to the fact that we have shown that $F_L \Leftrightarrow F_R$

Thus, $F_1 \vdash F_2$

$$F_1 = F_2 \setminus F'_R \cup F'_L$$

$F_2 \vdash F_2 \setminus F'_R$ holds per definition

$F_2 \vdash F'_L$ holds, since $F_2 \supseteq F'_R \vdash F'_L$

due to the fact that we have shown that $F_L \Leftrightarrow F_R$

Thus, $F_2 \vdash F_1$

Thus, we have shown:

- each transformation complies with a transformation of a set of propositional formulas.
- the transformations of the propositional formulas align with the well-known laws for propositional formulas e.g. commutative and associative law.
- the flat feature model is semantically equivalent to the original feature model.

7.2.2 Value Extraction

After the first step of the feature model to CSP transformation, all features are placed directly beneath the grandparent feature serving as parameters (cf. Figure 34). In the next step, the algorithm extracts the corresponding values. Again, different rules are applied to extract the values of the features—one for each relation. Figure 36 depicts the four value extraction rules.

- **mandatory:** *Mandatory* features (cf. Figure 36 1) stay *mandatory* and obtain an additional child feature representing the value of this feature. The value has the same notation and name (e.g. *HMI*). All cross-tree constraints in which the *mandatory* feature was involved are propagated to the value.

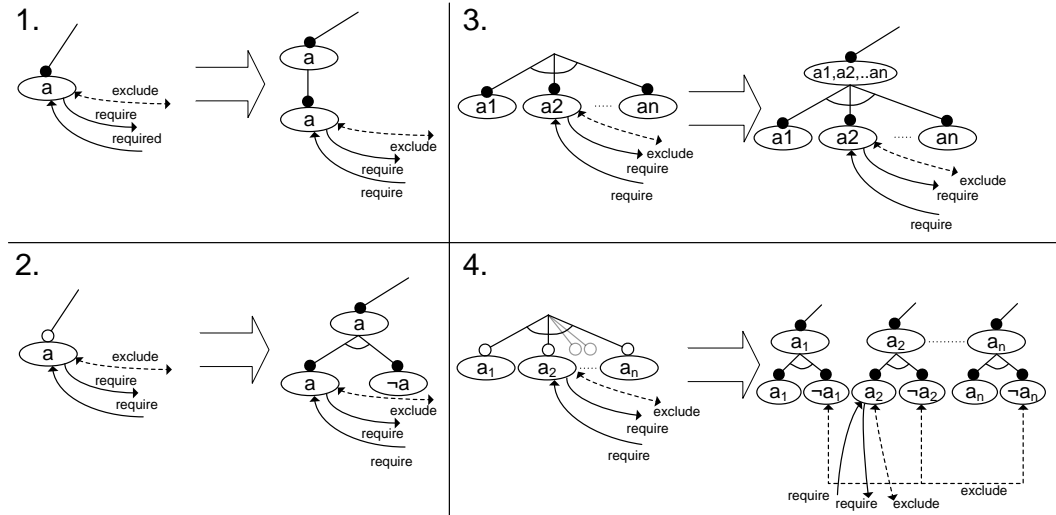


Figure 36: Rules for value extraction for *mandatory*(1), *optional*(2), *alternative*(3), and *or*(4) features

- optional:** An *optional* feature (cf. Figure 36 2) is changed to a *mandatory* feature with two child features. The *optional* feature *CLS* turns into a *mandatory* feature with an *alternative* child group containing a feature *CLS* and \neg *CLS*. For product instantiation the feature *CLS* is selected and one element of the *alternative* group has to be chosen as well. Therefore, either the feature *CLS* or the feature \neg *CLS* is selected. All cross-tree constraints in which the *optional* feature was involved are propagated to the positive value.
- alternative:** An *alternative* group (cf. Figure 36 3) stays unchanged but we add an additional *mandatory* feature representing the parameter of the *alternative* group. Again, all features of the *alternative* group are summarized within this parameter feature. The *alternative* group itself represents the possible values. With regard to the BCS case study, the two features **ManPW** and **AutPW** form an *alternative* group within the FFM. To extract a parameter and corresponding values, an additional feature representing the parameter of the *alternative* group is added. The name of this parameter is built from the concatenation of the individual members of the *alternative* group and thus: **ManPW**, **AutPW**. The *alternative* group then is added beneath this parameters representing the situation that for the parameter **ManPW**, **AutPW** either **ManPW** or **AutPW** can be selected. All cross-tree constraints in which a feature within the *alternative* group was involved are propagated to the corresponding value.
- or:** Flattening an *or* group (cf. Figure 36 4), transforms every element of the *or* group into an *optional* feature. To prevent that no feature of the *or* group is selected, we add an n-ary *exclude* constraint between all negative

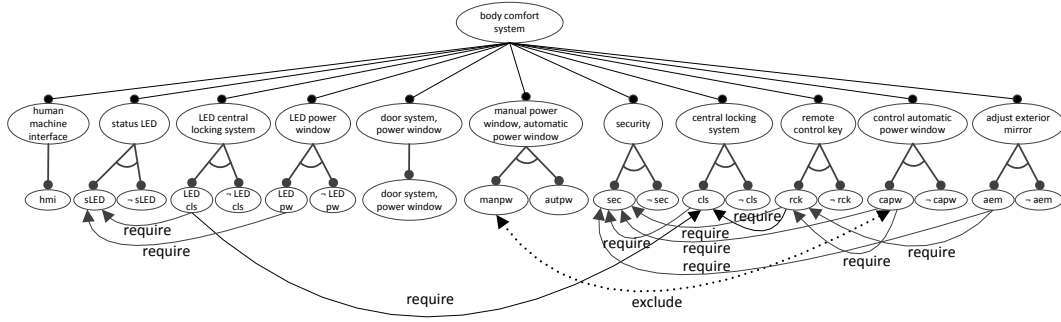


Figure 37: The FMCSF of the BCS subset

values. Again, the incoming and outgoing constraints are propagated to the corresponding value.

Figure 37 shows the flat feature model of BCS-small including the parameter values. This step ensures that every possible feature configuration is considered for pairwise combination.

The flattening process results in additional *require* and *exclude* constraints between features. Furthermore, we also have to consider existing cross-tree constraints within the feature model to ensure the semantical equivalence between the original and the flat feature model. All *require* and *exclude* constraints are transferred to the values of the features. There are no longer any constraints on the parameter-level.

7.2.3 CSP Extraction

After extracting parameters and corresponding values of the feature model, the binary CSP for our problem can be specified as shown in Definition 29. At a first glance features are the appropriate candidates to represent the parameters V within our FMCSF. However, we have to consider the fact that during flattening and value extraction new features are generated e.g. merged features and additional parameter features that are generated during the value extraction of alternative groups. Therefore, the parameters of the CSP are the features of our flat feature model.

Definition 29 (FMCSF). *The mathematical representation of our feature model-based binary CSP is a triple $\langle V, D, C \rangle$*

- $V = F \in \text{FFM}$ The parameters of the CSP are the features of the flat feature model.
- $D = D_1 \cup D_2 \dots \cup D_n$ set of values for each parameter, where $D_i = d_{i_1}, d_{i_2}, \dots, d_{i_q}$ is a set of possible values for v_i . D is generated on the basis of f as previously described.

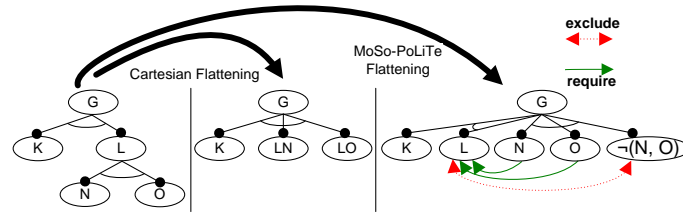


Figure 38: Comparison of the flattening approaches of an *alternative* parent with *alternative* child elements

- C is a set of constraints restricting the combination of certain values for pairs of features. Each $c \in C$ is a triple $\langle x, R, y \rangle$ with $x, y \in F$ and $R : \{D_{\text{req}} \uplus D_{\text{exc}}\} \subseteq D \times D$ is a set of the require and exclude constraints.

A solution to this CSP is a function $c : V \rightarrow D$ that assigns every feature f_{i_j} in $v_i = (f_{i_1}, f_{i_2}, \dots, f_{i_k}) \in V$ a value $d_i = (d_{i_1}, d_{i_2}, \dots, d_{i_k}) \in D$, where $c(v_i) = d_i$. For constraints C , i.e. triples $\langle v_i, R, v_j \rangle$, where $v_i = (f_{i_1}, f_{i_2}, \dots, f_{i_k})$ and $v_j = (f_{j_1}, f_{j_2}, \dots, f_{j_l})$ we differentiate two different types of constraints R :

- a *require* constraint between two values d_{i_m} and d_{j_n} demands that if $v_i = d_{i_m}$ then $v_j = d_{j_n}$ (but not vice versa)
- an *exclude* constraint between two values d_{i_m} and d_{j_n} demands that $v_i \neq d_{i_m} \vee v_j \neq d_{j_n}$

7.2.4 Cartesian Flattening

At this point, we like to discuss related work considering our flattening algorithm. In [WDS09] the authors realize a cartesian flattening of feature models which is a similar to our flattening algorithm. There, the motivation is to translate the feature model into a knapsack problem, which is then used to generate highly optimal architectural variants/products of the SPL. There are some significant differences to our flattening approaches; amongst others cardinality groups (or groups in our approach) are translated into an XOR (*alternative* group in our approach) with a maximum number boundary in [WDS09]. With this boundary an exponential explosion of all possible feature combinations is prohibited. For testing purposes all valid feature combinations need to be identified and we would lose the semantical equivalence between the original feature model and the flat feature model if we would use a boundary, limiting the maximum number of combinations. In [WDS09] a different rule for flattening an *alternative* group beneath an *alternative* parent feature is presented. Fig. 38 shows an abstract example used in [WDS09].

In the Cartesian flattening approach the features \mathbf{N} and \mathbf{O} are merged with its parent feature. Let us now assume that the parent feature \mathbf{L} is required by some other feature \mathbf{X} . The feature \mathbf{X} would then require \mathbf{L}, \mathbf{N} xor \mathbf{L}, \mathbf{O} . These constraints cannot be captured using binary constraints such as the ones we support in our subset extraction algorithm.

Due to the different field of application White et al. apply different transformation rules to prepare the feature model for their algorithms. However, this approach offers an additional evidence that it is possible to change the structure of the feature model to apply well-known algorithms such as knapsack or binary constraint solving. At this point we like to emphasize that our transformation approach was published earlier in [OSWo8].

7.3 SUBSET EXTRACTION

The subset extraction realizes the heuristics to find a set of solutions for the binary CSP. A solution of our CSP describes a product configuration c .

The problem statement can be defined as follows:

1. The subset of all possible products/configurations can be described as $C = \{c_1, \dots, c_k\}$
2. A pairwise subset is a subset $PWC = \{c_1, \dots, c_l\}$, where each **valid** combination of present/absent values for feature pairs $\{f_i, f_j\} \subseteq F$ is covered by at least one configuration $c_i \in C$.
3. A minimal pairwise subset PWC_{\min} is a pairwise subset and for every other PWC' , $|PWC_{\min}| \leq |PWC'|$ holds.

The Subset Extraction algorithm generates all valid pairwise combinations of features regarding cross-tree constraints. A valid pair is a combination of features not violating cross-tree constraints, the hierarchical structure, and the different feature notations in the feature model. Then, the algorithm incrementally combines those pairs of features to create valid configurations.

Definition 30 (Feature Pairs). *For a feature pair $\{f_i, f_j\}$ the following combinations need to be addressed on the assumption that the listed pairs are permitted combinations within a configuration:*

- if $f_i \Rightarrow f_j$, then (f_i, f_j) , $(\neg f_i, \neg f_j)$, and $(\neg f_i, f_j)$ are valid pairs, i.e., f_j is to be tested in the presence as well as in the absence of f_i ,
- if $f_i \Leftrightarrow f_j$, then (f_i, f_j) and $(\neg f_i, \neg f_j)$ are valid pairs, i.e., f_i is only testable in the presence of f_j ,

	positive	negative
intended	feature cooperation	feature vetoing
unintended	undesired interference	required but missing

Table 6: Categories of feature interactions

- if $f_i \perp f_j$ then (f_i, f_j) , $(\neg f_i, \neg f_j)$, $(\neg f_i, f_j)$, and $(f_i, \neg f_j)$ are valid pairs, i.e., f_i is to be tested in the presence and in the absence of f_j and vice versa, and
- if $f_i \not\perp f_j$, then $(\neg f_i, f_j)$, $(\neg f_i, \neg f_j)$, and $(f_i, \neg f_j)$ are valid pairs, i.e., no interaction is to be tested.

Please note that this assumption is rather naive and might not hold for certain configurations. For example, if $f_i \Leftrightarrow f_j$ and f_i is a *mandatory* feature and always selected for product configuration then we are not able to test the combination $(\neg f_i, \neg f_j)$.

However, pairwise combinations of features suffices to cover all interactions according to Table 6, which we have introduced in Chapter 6.

If two features $\{f_i, f_j\}$ interact pairwise combination covers:

- *intended positive/negative* interactions are tested to correctly cooperate/veto,
- *unintended negative* interactions arise, if the interaction is missing/faulty or
- *unintended positive* interactions by means of behavioral influences is tested.

Furthermore, for *optional* interactions,

1. the correctness of *intended* interactions in the presence of both features is tested, and
2. the conceptional independence of both features is tested by isolating them from each other.

Beside the fact that combinatorial testing of SPLs does not result in test cases directly but in test configurations, the following difference to ordinary combinatorial testing should be emphasized:

- Combining pairs of features we ensure to include each possible pairwise feature interaction. Thus, we are able to test (1) if the two features together within a configuration result in an unexpected behavior and (2) we are also able to check whether these two features really interact.
- The goal of generating a set of valid configurations containing all possible feature interactions adds an additional characteristics: instead of including pairs, we automatically obtain some threewise, fourwise, etc., and finally N-wise combination of features.

- Furthermore, testing complete products instead of feature combinations has the following two advantages: (1) the commonalities of the SPL are repeatedly tested leading to a solid basis of the SPL and (2) testing individual features or feature combinations is often not possible since some features are simply not executable/testable on their own [McGo1].

Our Subset Extraction algorithm combines combinatorial design and constraint solving. With regard to constraint solving, Subset Extraction uses Backtracking and Forward Checking to allow its combinatorial component to handle constraints between values of the different parameters.

Backtracking search is the basic algorithm for CSPs realizing a depth-first search for CSPs with single parameter assignments. Forward Checking then keeps track of remaining legal values for unassigned parameters. It terminates its search when any parameter has no legal values left.

7.3.1 *Subset Selection Pseudocode*

In the following we describe the subset extraction algorithm using pseudocode and activity diagrams. Our algorithm is able to generate a subset of configurations satisfying T-wise feature interaction coverage. However, we describe the internal functionality by means of pairwise feature interaction coverage.

Our algorithm requires the following data to calculate the combinatorial set of configurations:

- P is the data domain for parameters and V is the data domain for values. Those parameters and values correspond to the parameters and values within our FM – CSP (cf. Definition 29)
- $\text{ParVal} = \text{DataPair}\langle P, V \rangle$ is the data domain to assign a value v to a parameter p .
- $\text{Config} = \text{Array}\langle \text{ParVal} \rangle$ represents a configuration as a list of parameter assignments. In each configuration the parameters occur in compliance with the FMCSP.
- $\text{Subset} = \text{Set}\langle \text{Config} \rangle$ is the data domain of the combinatorial set of configurations.
- $\text{parameters} = \text{SortedHashMap}\langle P, \text{Set}\langle V \rangle \rangle$ is the data domain to describe a sorted list of parameters and associated values. parameters is sorted in the same order than Config .
- $\text{pairsToCover} = \text{SortedHashMap}\langle \text{ParVal}, \text{List}\langle \text{ParVal} \rangle \rangle$ contains all pairs of values with their corresponding parameters to be covered. Each element of the list $\text{List}\langle \text{ParVal} \rangle$ together with its key of the HashMap ParVal forms

Key	Value
(human machine interface, hmi)	((sLED, sLED), (sLED, ¬ sLED))
...	(aem, aem), (aem, ¬ aem))

Table 7: Example for the pairsToCover HashMap

such a pair that needs to be covered by the Subset. For example considering the BCS-small running example pairsToCover would look as depicted in Table7:

- `DataPair<ParVal, ParVal> singlePair` is the data domain of a single pair of parameter assignments to be covered.

Using these data domains our algorithm will proceed as follows: Our algorithm starts with the `subsetGeneration()` function (cf. Listing 1) that requires parameters and pairsToCover to generate Subset, the combinatorial subset of configurations (the covering array). pairsToCover is generated beforehand based on *require* and *exclude* constraints between the parameter values of our CSP without regarding transitive influence. The advantage of using pairsToCover is that each pair of values knows its corresponding parameters.

The `subsetGeneration()` function starts with the initialization of an empty Subset that will be filled with Configs (line 3). As long as there are uncovered pairs within pairsToCover, the algorithm will continue to build configurations and to add those to the subset (lines 5-12). For this purpose, an arbitrary pair (singlePair) out of pairsToCover is selected (line 6) and removed from pairsToCover (line 7). Directly removing singlePair from pairsToCover has two reasons:

- The algorithm will start building a configuration (Config) with singlePair as its parameter values. Thus, this pair is covered and does not need to be covered again.
- If the algorithm fails in finding a valid configuration including singlePair, then singlePair is an invalid pair based on transitive influence that the algorithm identifies using Backtracking with Forward Checking. Thus, this pair needs to be removed to avoid that the algorithm will try to add this pair within another configuration repeatedly.

Figure 39 depicts the corresponding activity diagram.

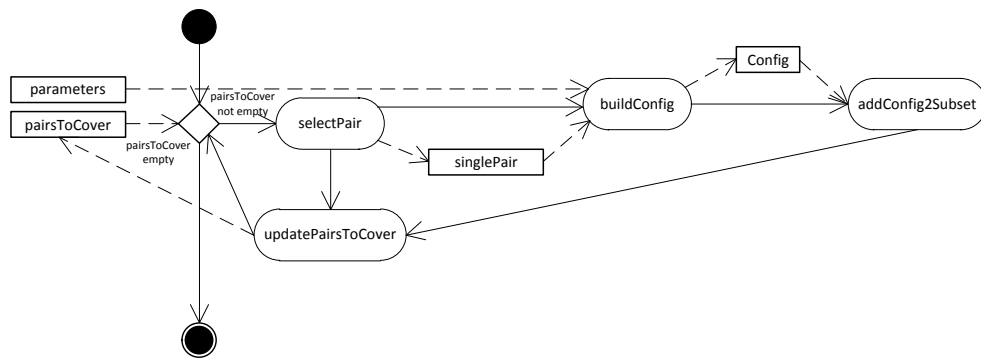


Figure 39: Activity diagram subsetExtraction()

Listing 1: Subset extraction pseudocode - I

```

1  /*Input: parameters, pairsToCover
   Output: subset*/
3  function subsetGeneration(parameters, pairsToCover)
   Subset ← ∅
5  while(pairsToCover ≠ ∅)
   singlePair ∈ pairsToCover
7  pairsToCover ← pairsToCover \ {singlePair}
   Config ← buildConfiguration(parameters, singlePair)
9   if Config ≠ null then Subset ← Subset ∪ Config
   pairsToCover ← { pairsToCover / covered by Config}
11  end if
   end while
13  return subset

```

Then, the **buildConfiguration()** function (cf. Listing 2) is called to actually add **singlePair** to a configuration. The **buildConfiguration()** builds the initial configuration that is then filled incrementally with other **ParVal** to result in a complete configuration not violating any constraints. If the **buildConfiguration()** function returns a configuration **Config**, this configuration is then added to **Subset** (lines 9). Afterwards, **pairsToCover** is updated by removing all pairs that are covered by **Config** (line 10).

Listing 2 depicts the pseudocode of the **buildConfiguration()** function. The **buildConfiguration()** function operates as follows. First, an empty configuration is initialized (line 4). Then, the values within **singlePair** are used to assign the parameter/value pairs in this **Config** (line 5). After adding the values within **singlePair** to the corresponding parameters of the configuration, the **fillConfiguration()** function is called to add further values to the **Config**. If the function **fillConfiguration()** succeeds, **buildConfiguration()** will return this configuration (line 7) which is then added to the subset in **subsetGeneration()**.

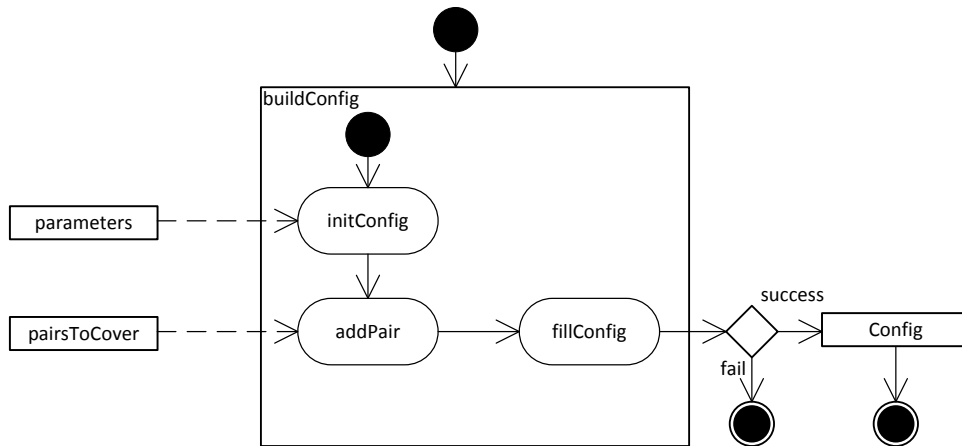


Figure 40: Activity diagram of buildConfiguration()

Listing 2: Subset extraction pseudocode - II

```

1  /*Input: parameters, singlePair
   Output: Config*/
3  function buildConfiguration(parameters, singlePair)
   Config ← ∅
5   Config[pi] = vi and Config[pj] = vj , where {(pi, vi), (pj, vj)} ==
   singlePair
   if(fillConfiguration(Config, parameters, 0)) then
7     return Config
   else
9     return null
   end if

```

Figure 40 depicts the corresponding activity diagram. The selected pair is added to the `Config` and `fillConfiguration()` is used to fill the `Config`.

`fillConfiguration()` (cf. Listing 3) requires the frequently initialized configuration, the list of parameters, and an index for the parameter selection p (line 3). The `fillConfiguration()` function starts with checking whether the `Config` is already complete (line 4). The function `fillConfiguration()` is called with $p = 0$ to start with the first parameter. If this parameter is already allocated with a value, the `fillConfiguration()` function is called recursively to continue with the next parameter (line 7-8). If the parameter p is not yet allocated, the values in `pairsToCover` belonging to this parameter are sorted by using the priority function (line 10). The priority function calculates the priority of each value within the `ParVal` key of the `pairsToCover` `HashMap` (lines 29-33). The priority is calculated by comparing the size of the list of `ParVal` representing the required pairs for a certain parameter/value combination. The parameter/value combination for a certain parameter, which has the most re-

quired combinations, obtains the highest priority. This functionality can be adapted to take other criteria into account, for example safety critical feature combinations. Those configurations would then occur more often within the resulting subset and, thus, are tested more often.

Then, Forward Checking is used to check whether the selected value violates constraints within the configuration. For that purpose, the algorithm checks (forward check) whether adding a certain value v for the parameter p results in a valid configuration with regard to the existing `Config` (lines 11-12). Please note that Forward Checking takes only parameter/values into account that share constraining relations with the recently added value. Then, the current list of parameters is saved as `parametersTemp` (line 13). For each unassigned parameter within `Config` (line 15) the algorithm removes values that violate the current configuration (lines 15-16). Afterwards, the algorithm checks whether there is a parameter in `parametersTemp` without any possible values (lines 18-20). If this is the case, the function `fillConfiguration()` returns false and Backtracking is used to undo the previous parameter allocation and to choose a different value. Backtracking is implemented by using the recursive call of the `fillConfiguration()` function as described in [Knu98].

If certain pairs in `pairsToCover` are invalid due to transitive influence, Forward Checking ensures that this pair is not combined within a certain configuration. Then, those pairs will remain in the list until the `buildConfiguration()` function selects those to be the `singlePair` and the pair will be removed from `pairsToCover`. Our algorithm will then try to generate a configuration on the basis of this pair and will fail. Thus, the resulting subset solely contains valid configurations. Furthermore, we can guarantee that the subset extraction algorithm terminates because the list of pairs to cover is decreased by at least one pair within every iteration.

Listing 3: Subset extraction pseudocode - III

```

  /*Input: Config, parameters, index for parameter
  2 Output: valid Config*/
  function fillConfiguration(Config, parameters, p)
  4   if sizeOf(Config) ≤ p then
      return true
  6   end if
      if Config[p] ≠ null then
  8     return fillConfiguration(Config, parameters, p+1)
      end if
  10  for each value v in parameters(p) ordered by priority(parameters(p))
      if v is consistent with Config
  12    Config[p] ← v
      parametersTemp ← parameters //Save current state
  14  for each parameter x in Config where x has no value
      remove ParVal from parametersTemp which lead to inconsistent
      configuration due to constraints
  16  end for

```

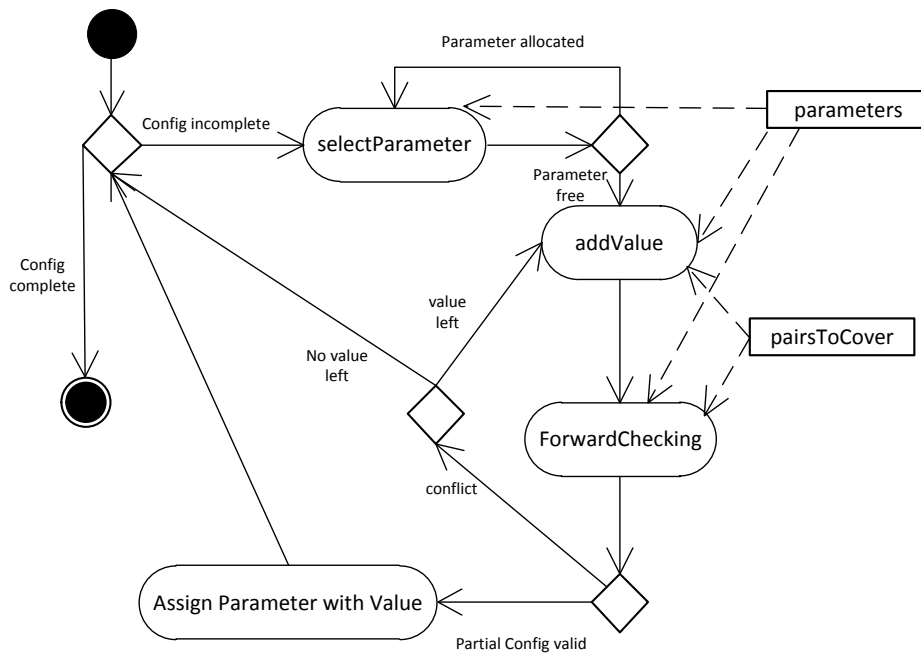


Figure 41: Activity diagram of fillConfiguration()

```

18     if
20         for all parm in parametersTemp
22             parm ≠ ∅
24         end for
26     then
28         result = fillConfiguration(Config, parametersTemp, p+1)
30         if result == true
32             return true
34         else //Continue with next v
36         end if
38     end if
40 end for
42 return false

function priority(parameter)
    sorts the values of the parameter in descending order
    according to their occurrence in pairsToCover

```

Figure 41 depicts the corresponding activity diagram. The recursion realizing the Backtracking-idea is clearly visible.

In comparison to algorithms where the configurations are built in parallel, our algorithm configures the subsets incrementally. This is a consequence of using

Backtracking. Due to the fact that we use Backtracking, a configuration can be changed. Algorithms such as IPO [LT98] would create a new configuration if a certain value cannot be added. But, using Backtracking, it may happen that the configuration is changed and the value that was recently rejected within a certain configuration could suddenly be added. The new configuration is then obsolete.

7.3.2 *Subset Extraction for BCS*

After introducing the pseudocode of the subset extraction, we use the FMCSP of BCS-small to depict the result of applying this algorithm.

On the basis of the BCS-small feature model 40 different configurations can be derived, which are depicted in Table 8. Underscores symbolize features that are merged due to the transformation rules and empty columns represent the selection of a \neg feature. The subset extraction algorithm generates nine configurations for pairwise feature interaction coverage. Those nine configurations are: C_1 , C_2 , C_6 , C_{10} , C_{21} , C_{32} , C_{34} , C_{35} , and C_{40} .

C1-C40 are the configurations; HMI = human machine interface; LED = status LED; LED_CLS = LED central locking system; LED_PW = LED power window; DS_PW = door system, power window; ManPW = manual power window; AutPW = automatic power window; sec = security; CLS = central locking system; RCK = remote control key; AEM = adjust exterior mirror; CAP = control automatic power window.

C1:	HMI				DS_PW	AutPW	sec	CLS	RCK	AEM	CAP
C2:	HMI	LED	LED_CLS	LED_PW	DS_PW	AutPW	sec	CLS	RCK	AEM	CAP
C3:	HMI	LED		LED_PW	DS_PW	AutPW	sec	CLS	RCK	AEM	CAP
C4:	HMI	LED	LED_CLS		DS_PW	AutPW	sec	CLS	RCK	AEM	CAP
C5:	HMI				DS_PW	AutPW	sec	CLS	RCK		CAP
C6:	HMI	LED	LED_CLS	LED_PW	DS_PW	AutPW	sec	CLS	RCK		CAP
C7:	HMI	LED		LED_PW	DS_PW	AutPW	sec	CLS	RCK		CAP
C8:	HMI	LED	LED_CLS		DS_PW	AutPW	sec	CLS	RCK		CAP
C9:	HMI				DS_PW	ManPW	sec	CLS	RCK	AEM	
C10:	HMI	LED	LED_CLS	LED_PW	DS_PW	ManPW	sec	CLS	RCK	AEM	
C11:	HMI	LED		LED_PW	DS_PW	ManPW	sec	CLS	RCK	AEM	
C12:	HMI	LED	LED_CLS		DS_PW	ManPW	sec	CLS	RCK	AEM	
C13:	HMI				DS_PW	AutPW	sec	CLS	RCK	AEM	
C14:	HMI	LED	LED_CLS	LED_PW	DS_PW	AutPW	sec	CLS	RCK	AEM	
C15:	HMI	LED		LED_PW	DS_PW	AutPW	sec	CLS	RCK	AEM	
C16:	HMI	LED	LED_CLS		DS_PW	AutPW	sec	CLS	RCK	AEM	
C17:	HMI				DS_PW	ManPW	sec	CLS	RCK		
C18:	HMI	LED	LED_CLS	LED_PW	DS_PW	ManPW	sec	CLS	RCK		
C19:	HMI	LED		LED_PW	DS_PW	ManPW	sec	CLS	RCK		
C20:	HMI	LED	LED_CLS		DS_PW	ManPW	sec	CLS	RCK		
C21:	HMI				DS_PW	AutPW	sec	CLS	RCK		
C22:	HMI	LED	LED_CLS	LED_PW	DS_PW	AutPW	sec	CLS	RCK		
C23:	HMI	LED		LED_PW	DS_PW	AutPW	sec	CLS	RCK		
C24:	HMI	LED	LED_CLS		DS_PW	AutPW	sec	CLS	RCK		
C25:	HMI				DS_PW	ManPW	sec	CLS			
C26:	HMI	LED	LED_CLS	LED_PW	DS_PW	ManPW	sec	CLS			
C27:	HMI	LED		LED_PW	DS_PW	ManPW	sec	CLS			
C28:	HMI	LED	LED_CLS		DS_PW	ManPW	sec	CLS			
C29:	HMI				DS_PW	AutPW	sec	CLS			
C30:	HMI	LED	LED_CLS	LED_PW	DS_PW	AutPW	sec	CLS			
C31:	HMI	LED		LED_PW	DS_PW	AutPW	sec	CLS			
C32:	HMI	LED	LED_CLS		DS_PW	AutPW	sec	CLS			
C33:	HMI				DS_PW	ManPW	sec				
C34:	HMI				DS_PW	ManPW					
C35:	HMI	LED		LED_PW	DS_PW	ManPW	sec				
C36:	HMI	LED		LED_PW	DS_PW	ManPW					
C37:	HMI				DS_PW	AutPW	sec				
C38:	HMI				DS_PW	AutPW					
C39:	HMI	LED		LED_PW	DS_PW	AutPW	sec				
C40:	HMI	LED		LED_PW	DS_PW	AutPW					

Table 8: Possible configurations of the BCS-small feature model

UNTIL now, this part covered the feature modeling and the combinatorial testing concepts of the MoSo-PoLiTe approach. These two components realize the generation of a subset of all possible configurations for testing purposes. This set of configurations needs to be tested. We assume that these configurations are representative for the entire SPL with regard to fault detection, where the faults originate from T-wise feature interaction. To reuse test artifacts when testing the combinatorial set of configurations, we take model-based testing into account as discussed in Chapter 4.

Generally, the MoSo-PoLiTe configurations can be tested like ordinary software systems to achieve 100% T-wise feature interaction coverage because the variability is resolved in each configuration. However, with T increasing, the number of configurations for a certain SPL increases as well. Hence, a test approach reusing test artifacts seems to be vital. The two main reasons for using model-based testing are:

- According to our former studies in [OWES11], model-based testing is suitable for SPLs since models can be easily reused and there are various tools supporting a (semi-) automatic test case generation from test models.
- The approaches of the category reuse-techniques that we have introduced in the related work chapter apply model-based testing to allow a systematic reuse of test artifacts.

Model-based techniques are frequently used to implement SPLs in various domains. We refer to [Pen06] for further details of model-based SPL development. There, the author summarizes and evaluates different model-based frameworks for SPL development.

Model-based testing [UL07] aims at the automation of design and application steps of testing activities for detecting faults in software system implementations. Our scope is to generate test cases with oracles from a behavioral and reusable test model providing a behavioral specification that relates system inputs to expected outputs. Due to the fact that we aim to test embedded systems e.g. in the automotive sector, we focus on the generation of test cases with oracles from a behavioral model, which is capable of describing the behavior of systems with potentially infinitely many executions. Model-based testing techniques are then able to generate a finite set of behavioral test cases by selecting representative executions of the model as test cases until certain coverage criteria are fulfilled.

For testing embedded systems, which constantly interact with the environment, statecharts are an appropriate candidate to serve as a test model [Lig09]. Statecharts are nowadays widely used and adopted, e.g., as UML state machines. For the development and implementation of reactive/embedded control systems, statecharts constitute an industrial de-facto standard, e.g., underlying the MATLAB/Simulink/Stateflow tool set. Concerning testing, this idea is supported by the fact that various tools exist providing an automatic test case generation on the basis of a statechart. For embedded systems in particular, the emulation of environmental sensor or user stimuli sequences serves as test inputs, and the oracles define corresponding output signals expected for the actuator components.

The MoSo-PoLiTe approach aims at using existing approaches for model-based testing. We assume that the existing test case generation approaches for statecharts are appropriate for SPLs as well [OWES11]. However, with regard to [OWES11] comparing different approaches for model-based testing in the SPL context, we have the following requirements:

- using standard algorithms for automatic test case generation including tool support
- supporting standard coverage criteria
- mapping between features and elements of the statechart

None of the state-of-the-art approaches for model-based testing for SPLs support all of our requirements. Only ParTeG provides a similar approach. However, ParTeG does not support feature modeling at all. Thus, we will introduce a very simple approach to:

- set up a reusable test model (a so-called 150% test model) by means of statecharts
- mapping elements of the test model to features

As a result, we obtain a model-based testing framework that allows us to automatically generate test cases for each derivable configuration of the SPL using standard coverage criteria and standard test case generation algorithms and tools.

In the following, the construction of a reusable test model is described followed by a description of the mapping between the feature model and the test model. We refer to the reusable test model describing the behavior of the entire SPL as 150% model. Then, the derivation of configuration specific test models is described that are then used to generate configuration specific test cases. A configuration specific test model describing the behavior of a specific configuration is referred to as 100% model. Last but not least, we will discuss potentials and limitations of this 150% test model philosophy.

8.1 150% TEST MODEL

Features in a feature model are abstract entities that describe certain characteristics of the SPL and determine whether these entities are common within all products or may vary. To equip features with a certain behavior or functionality, a mapping to appropriate artifacts such as code fragments and models is required. For testing purposes features need to be mapped to test artifacts. In MoSo-PoLiTe, we use statecharts as reusable test models. The mapping specifies common and variable artifacts within the statechart-based test model. A statechart [Har87] is an Extended Finite State Machine (EFSM) enriched with several additional concepts for specifying complex system behavior: Based on well-defined operational semantics [HK01], statechart specifications can be used for simulation and (automated) implementation derivation as well as for static analysis of properties such as deadlocks, reachability, and validity of execution sequences [MMB94]. Fig. 42 shows a sample statechart, the *Manual Power Window* submachine from the BCS SPL. The basic states encode vertical window positions up, down, and pending, and transitions are labeled with `trigger[guard]/actions`. The user triggers vertical window movements via buttons that release according input events, and, depending on the internal state, fire corresponding output events for controlling the movement actuators. Additional transition guards possibly hinder window movements, for example, via status flags for the **finger protection** (fp) feature.

We follow the idea of [GKPR08, DW09] to use a 150% test model including variability. The 150% model contains the behavior of all features, no matter if those exclude each other within a configuration of the SPL. A model including features that exclude each other is inconsistent. Thus, a 150% model is generally inconsistent or overspecified and is only of virtual nature.

We simply regard a test model to consist of a finite collection of modeling artifacts. These artifacts are of arbitrary internal structure and their interrelations and compositions depend on the corresponding modeling formalism. A coverage criterion applied to the test model then selects certain artifacts to be covered, i.e., traversed by test case executions.

For the creation of a 150% test model we recommend that the root feature should be mapped to the initial state within the test model. *Mandatory* features are supposed to map to an entire statemachine or a submachine realizing major functionalities common to all products. Varying features extend this core model with:

- additional states and transitions,
- additional hierarchical states to concretize/refine functionality, and
- additional submachines for further functionalities.

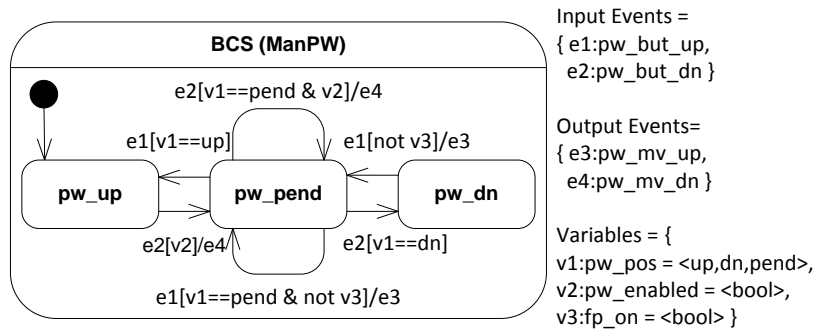


Figure 42: Statechart test model of *ManPW*

8.2 100% MODELS AND TEST CASE GENERATION

We map the feature of the feature model to our reusable test model enabling us to interrelate feature selection for product instantiation and test case generation. As a result, we are then able to automatically derive test cases for arbitrary products of the SPL. Furthermore, the mapping allows us to investigate

1. what is covered within the test model when a set of features is selected, and
2. which test cases are necessary to cover a certain feature combination.

Features $F = \{f_1, f_2, \dots, f_n\}$ of an SPL organized in a feature model $FM(F) \subseteq \mathcal{P}(F)$ are abstract entities that describe characteristics of product variants from the user’s point of view. To interrelate features and the test model a *mapping* between the features and the 150% test model allowing to configure the test model according to feature selection is required. We restrict our discussions to a simple mapping function:

- States and transitions within the test model can be mapped to propositional formulas over features. Those artifacts then become variable or more precisely configurable by means of feature selection.
- Variable test model artifacts are selected to become part of a 100% test model, if the propositional formula over features turns to true.
- Test model artifacts that are not mapped to a feature belong to the so-called core assets of the test model and, thus, belong to every possible test model instance.

Figure 43 depicts an example of our mapping approach with regard to the alarm system functionality within the BCS running example. At the top level, the 150% test model is depicted including mapping annotations. Three transitions are mapped to the **control alarm system** feature and one transition is mapped to the

150 % Test model of the BCS alarm system

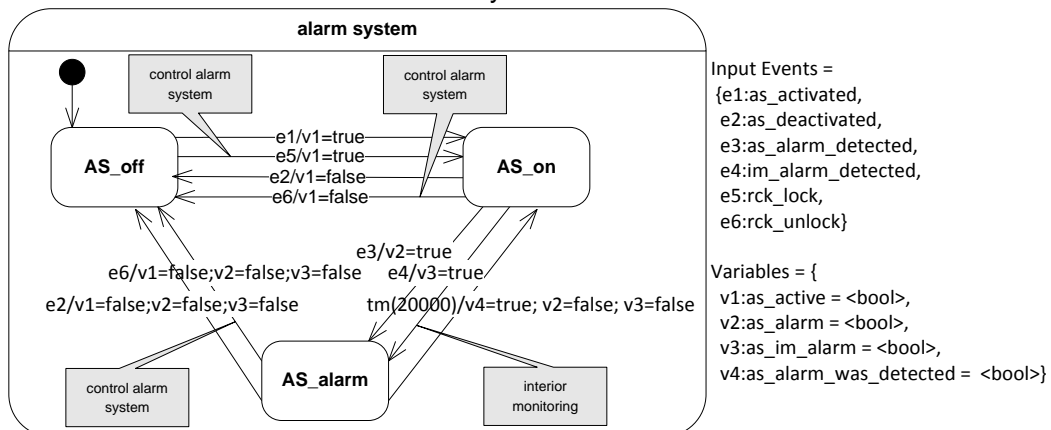
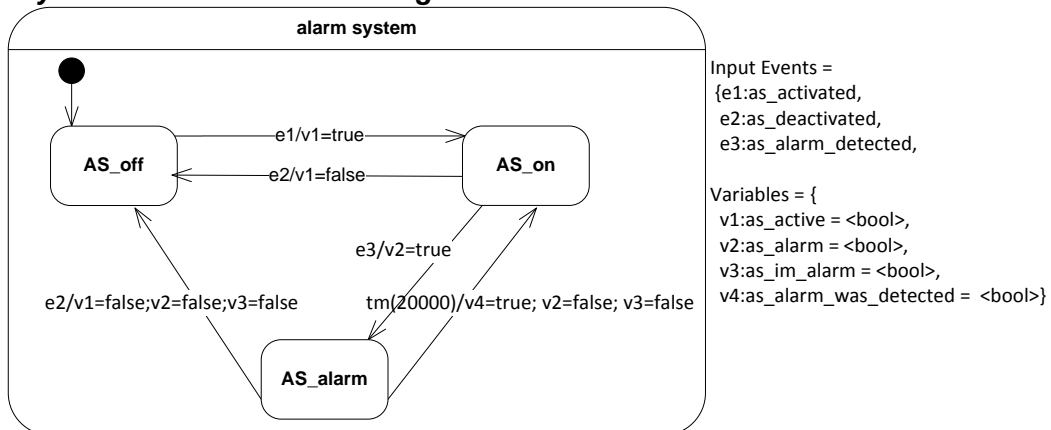
100 % Configuration not including control alarm system and interior monitoring

Figure 43: Mapping example by means of the BCS alarm system functionality

interior monitoring feature. If a configuration does not include those two features, the corresponding transitions will not be included within that 100% test model. To result in a suitable mapping, we consider the following three restrictions:

1. We have to assure that the logical expressions mapped to transitions and states conform to the semantics of the corresponding feature model. That means for example that a certain artifact within the test model cannot be mapped to the feature combination $(a \wedge b)$ if those two features (a and b) are not allowed to be part of the same product e.g. exclude each other.
2. Each transition needs a state to start and a state to end.

3. Every state within the test model should be reachable from the initial state or is the initial state itself.

As a result, each 100% test model which is configured via a selection of features in the feature model is syntactical correct with regard to the aforementioned conditions: Those restrictions have to be assured manually within the test model.

- The mapping does not violate the semantics of the feature model.
- Each transition has a starting state and an ending state.
- Each state within a 100% test model can be reached.

We refer to a test model specification of a full variant, i.e., a valid product configuration $PC \in FM(F)$ assembled from all fragments of the variant's features as its 100% test model in the following. The 100% test model $TM_{PC100\%} \subseteq TM_{150\%}$ for a product configuration $PC \in FM(F)$

The 100% test models can be used for test case generation. There are three categories of test specifications for model-based test case generation according to Pretschner [Pre03]:

- functional — which are based on use cases derived from the system requirements or specification
- structural — which are based on the structure of the test model
- stochastic — where test cases are derived randomly

Within this thesis, we align with the structural category and use the structure of the test model for test case generation. State-, Transition-, and MCDC-Coverage are taken into account. Statecharts are also established to serve as test models, where test case generation is supported by various tools such as Rhapsody/ATG and ParTeG [WSS08]. Considering statecharts as test models TM_{SC} , modeling artifacts $E \in TM_{SC}$ refer to (composite and basic) states, transitions, labels, and so forth. Corresponding, statechart coverage criteria are mainly structural control/data flow oriented, e.g., *all-states*, *all-transitions*, *all-transition-pairs* [UL07].

8.3 DISCUSSION OF THE 150% TEST MODEL PHILOSOPHY

As we have already discussed in the beginning of this thesis, model-based testing gains in popularity within the SPL community. Apart from the “usual” advantages of model-based testing, the concept of a 150% test model allows for describing the behavior of the entire SPL and to derive appropriate test cases according to well-known coverage criteria. However, there are several shortcomings that one need to be aware of:

- The creation and maintenance of a “large” 150% test model is difficult and time consuming. Questions like: Who is in charge of the 150% test model? Who ensures that there is a systematic reuse philosophy behind the components of the reusable test model? arise.
- The test model is generally not feature-oriented. That means that the behavior of a certain feature may be a crosscut through the 150% test model. Thus, we assume that when a feature should be changed, identifying its representatives within the test model is a complex task but feasible due to the mapping.
- As a logical consequence, checking consistency within a 150% test model is a complicate task as well.

SUMMARY PART III

WITHIN this part we have introduced the MoSo-PoLiTe approach. MoSo-PoLiTe is based on the integration of three approaches: feature model-based testing, combinatorial testing, and model-based testing. Figure 44 depicts a detailed overview of the MoSo-PoLiTe concept.

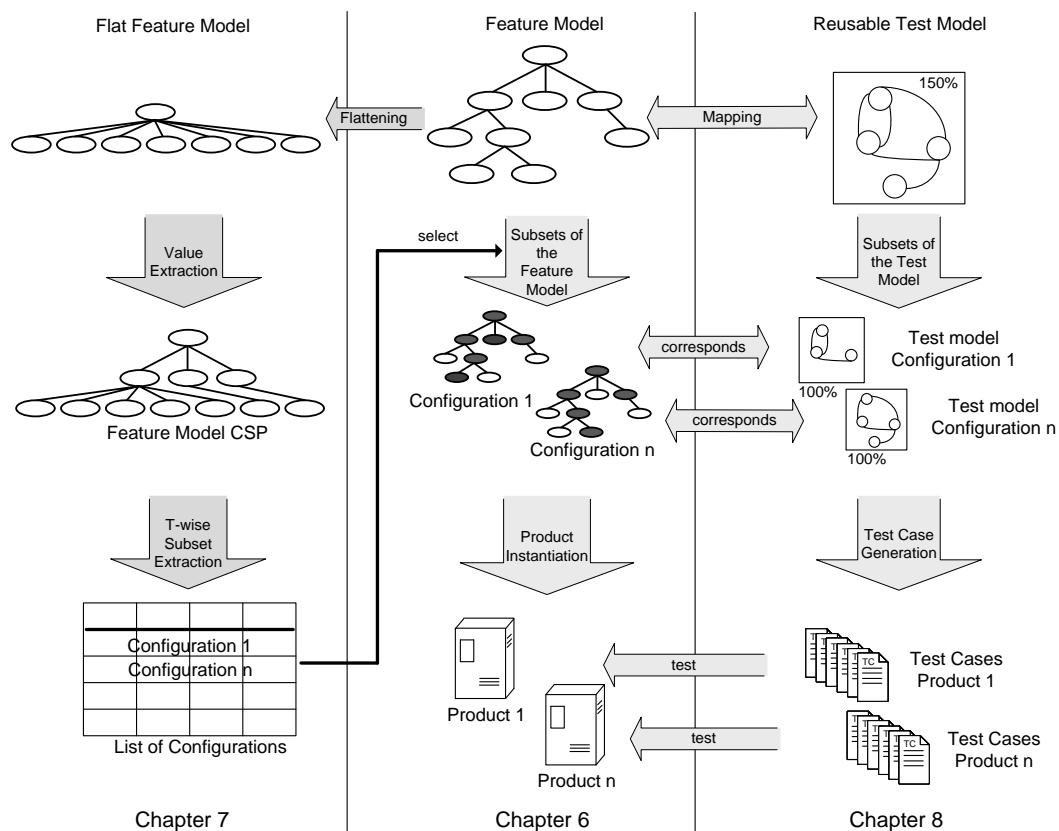


Figure 44: Detailed overview of the MoSo-PoLiTe approach

In the middle column, the feature model, which we have introduced in the beginning of this part in Chapter 6, serves as a central component. The BCS running example was used to introduce the basics to feature modeling. Furthermore, feature interaction was defined and we have discussed how those interactions influence testing purposes.

From there on, the feature model is transformed into a binary CSP (cf. left-hand column) followed by the subset extractor that generates the combinatorial set of

configurations (cf. left-hand column third row). Chapter 7 introduced the feature model-to-CSP transformation algorithm and the combinatorial subset extractor.

Chapter 8 then introduced our reusable test model depicted on the right-hand side of Figure 44. The features of the feature model are mapped to states and transitions of the reusable test model. Due to that mapping, we are then able to use the feature model to configure the 150% test model. For each configuration of the combinatorial set of configurations, a corresponding 100% test model is created. Additional conditions ensure that a valid configuration results in a syntactical correct 100% test model. The test models for each configuration can then be used for test case generation.

The combination of the combinatorial subset selection and the reusable test model via a feature model results in the ability to generate a subset of configurations achieving 100% pairwise feature interaction coverage. For each of these configurations test cases can be generated on the basis of the reusable 150% test model.

After introducing the concept and theory of our approach, we describe the implementation of MoSo-PoLiTe. To support its application in the industrial context, our scope in the next part is to provide a tool chain allowing the industry to apply our approach. Thus, a summary of existing tools for feature modeling is necessary. Furthermore, the next part present the results of applying MoSo-PoLiTe tool chain to the BCS-SPL and some results of applying the combinatorial subset selection to two industrial SPLs.

Part IV

IMPLEMENTATION AND EVALUATION

THIS part provides a detailed description of the implementation and evaluation of MoSo-PoLiTe. First, Chapter 10 describes the implementation of a tool chain realizing the MoSo-PoLiTe approach. Different tools are integrated to a tool chain that is capable of being applied in the industrial context. This tool chain consists of commercial and self-implemented tools. For all commercial components we provide a recommendation for open source alternatives. Please note that with regard to the combinatorial testing part of MoSo-PoLiTe, we focus on pairwise feature interaction coverage and to some extent on threewise feature interaction coverage.

Afterwards, an evaluation chapter summarizes the results of a systematic analysis and evaluation of MoSo-PoLiTe. The evaluation phase addresses three different topics:

1. to apply the MoSo-PoLiTe tool chain to our industrial case study to show the applicability of our approach.
2. to examine the impact of the developed approach on real world SPLs by means of two additional industrial experiments.
3. to examine and discuss the abilities and limitations of pairwise testing within the SPL development.

Figure 45 depicts the structural overview of Part IV according to our MoSo-PoLiTe development process. Chapter 10 describes the implementation of the MoSo-PoLiTe approach. The structure of this chapter is described according to the MoSo-PoLiTe-big-picture.

- Section 10.1 focuses on feature modeling, including configuration and product generation.
- Section 10.2 describes the implementation of the combinatorial testing approach.
- Section 10.3 then realizes our model-based testing approach on the basis of an existing modeling tool set.
- Section 10.4 is dedicated to an extensive testing of our tool chain to ensure that the concept is implemented as specified.

Chapter 11 is dedicated to the evaluation using two industrial SPLs and our automotive running example.

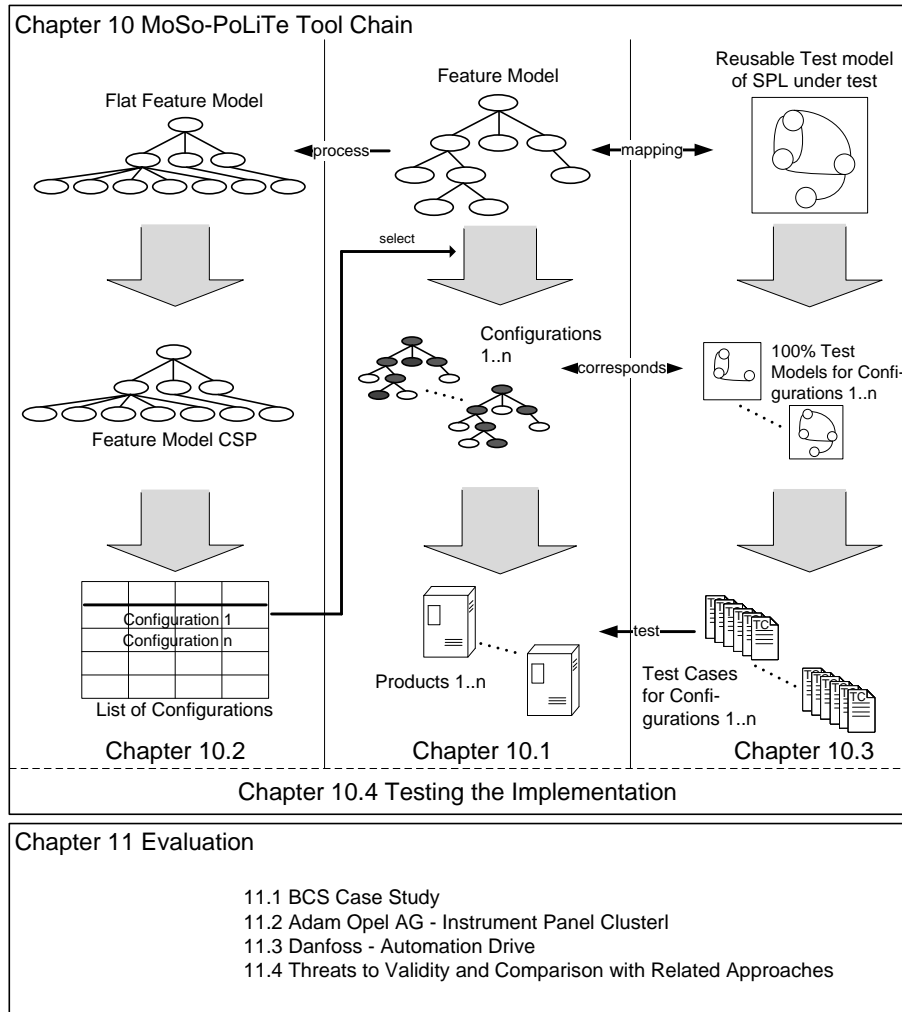


Figure 45: Overview - Part IV

- Section 11.1 describes the application of the MoSo-PoLiTe tool chain to our BCS running example and a discussion of the potentials and limitations of the MoSo-PoLiTe approach.
- Section 11.2 describes and discusses the application of the MoSo-PoLiTe tool chain to an automotive SPL provided by Opel [AG11a].
- Section 11.3 describes and discusses the application of the MoSo-PoLiTe tool chain to an automation SPL provided by Danfoss [Dan11].
- Section 11.4 provides a a discussion of threats to validity of our evaluation and a detailed comparison to related approaches.

IN this chapter we thoroughly describe the implementation of the MoSo-PoLiTe concept and introduce a tool chain allowing its application in the industrial context. Please note that for the combinatorial subset selection, we have chosen pairwise feature interaction coverage to be the most appropriate for demonstration purposes since the number of generated configurations is rather small. However, our implementation allows T-wise feature interaction coverage. We will consider pairwise feature interaction for describing the implementation and for evaluation purposes. Furthermore, we provide a systematic procedure to test whether our implementation aligns with the described MoSo-PoLiTe concept.

Our tool chain includes the following three components:

- Feature modeling
- Combinatorial subset selection
- Model-based test case generation

The feature modeling and the model-based test case generation are realized using existing tools. Also the mapping between features and test model artifacts (states and transitions in the test model) is implemented by `pure::variants`, a commercial variability management tool. However, MoSo-PoLiTe is the first project using the `pure::variants`/Rational Rhapsody integration for model-based testing purposes. The combinatorial subset selection is implemented as an eclipse plugin open for extensions and can generally be used in combination with any existing variant management tool and model-based testing tool.

In the following, we will describe the implementation of the tool chain by means of our running example. The tool chain consists of two commercial tools and our combinatorial subset selection plugin. Figure 46 provides an abstract overview of the MoSo-PoLiTe tool chain. The feature modeling is done using `pure::variants`. For model-based testing we utilize Rational Rhapsody [IBM11] and the Rhapsody addon ATG [AG11b] for model-based test case generation. Please note that the different tools within this tool chain are exchangeable and could be replaced by other tools.

The central component of our approach is the eclipse based combinatorial subset selection component. It imports the `pure::variants` feature model and exports the combinatorial subset of configurations. Those configurations are re-imported in `pure::variants` and then used to configure the 150% test model in Rational Rhapsody. Subsequently, ATG can be used to generate test cases and test suites

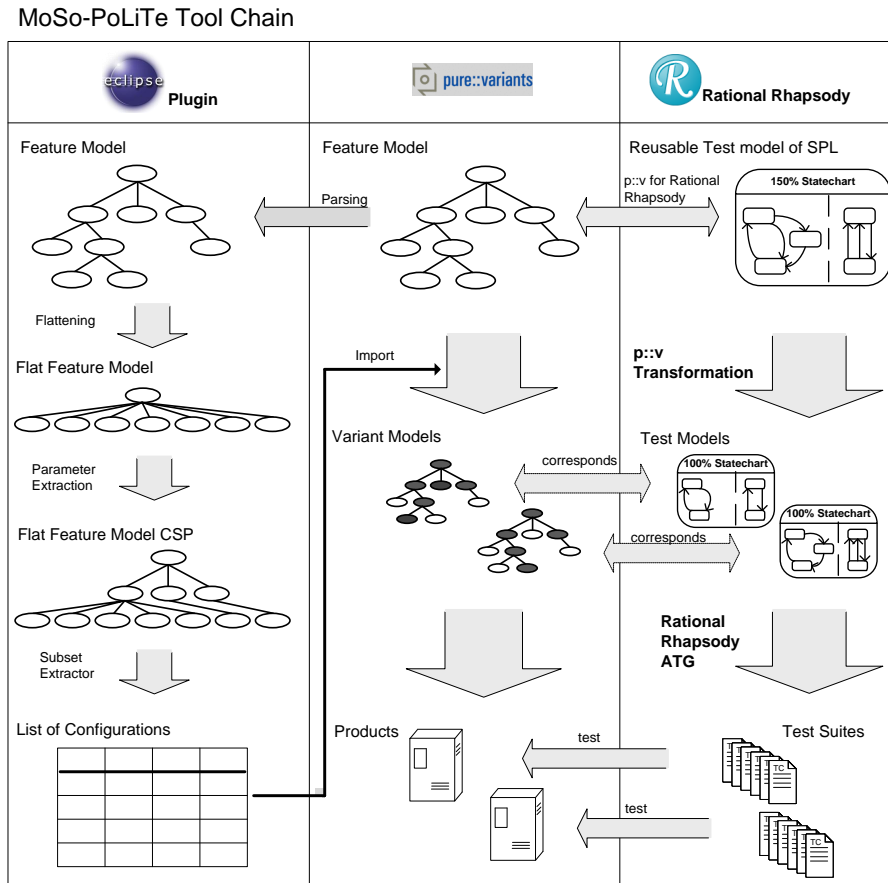


Figure 46: Overview of the MoSo-PoLiTe tool chain

satisfying different coverage criteria. This setup enables us to automatically generate configurations with corresponding test suites. In the dedicated sections, we will also discuss possible alternative tools to replace the initially selected tool chain.

10.1 FEATURE MODELING

The basis of the MoSo-PoLiTe concept is the feature model; the feature model interrelates the combinatorial subset selection and the reusable test model. Due to the positive response from industry, our goal was to implement the MoSo-PoLiTe concept within an applicable tool chain. Thus, we decided to select an existing variant management tool including feature modeling.

10.1.1 Feature Modeling Tool Comparison

The most prominent tools for feature modeling are:

- GEARS from Biglever [Kru08],

- PREEVision from Aquintos [aG11c],
- pure::variants from pure-systems [pG11],
- FeatureIDE [LAMS05] , and
- Feature Modeling Plug-in [ACo4].

The first three are commercial tools. GEARS is a widespread variant management tool including a feature modeling user interface. Its area of application is mainly based in North America and it has some success stories in Asia and Europe.

The second commercial tool is PREEVision, a variant management tool focusing on automotive specific toolsets. In comparison to its competitors, this tool mainly focus on product lifecycle management (PLM) mapping features to software and hardware artifacts within the development process.

pure::variants is the most prominent feature modeling tool in Germany and very popular in the SPL research domain. As being developed within eclipse, pure::variants can be extended by additional functionalities easily.

The two other tools, namely FeatureIDE and Feature Modeling Plug-in are open source and lack of integrations towards other tools such as modeling tools. They focus on configuring source code via feature selection.

To chose a specific tool for our tool chain, we considered the following requirements:

- Which tool is currently in use in industry? This requirement is important for us to consider since we want MoSo-PoLiTe to be applied in the industrial context.
- The tool needs to support the feature modeling relationships used by our algorithm. Thus, we check whether the tools cover the same relations (*optional*, *mandatory*, *alternative*, and *or*) and constraints (*require* and *exclude*).
- The tool needs to support the derivation of configurations. Otherwise, we would not be able to generate our combinatorial subset consisting of configurations.
- The tool needs to support a mapping between the feature model and code e.g. Java or C/C++ to support the derivation of products.
- The tool should allow mapping to a behavioral model e.g. statecharts to apply model-based testing techniques.

Table 9 summarizes how the aforementioned tools align with our requirements. Gears and pure::variants obtain the best results within our simple comparison that does not claim of being complete. Ignoring the technical details of both tools, the main difference is extendability. Actually, we expect GEARS to be as extendable as pure::variants, however, we are not aware of any research-driven extensions for

GEARS, whereas there are plenty of those for pure::variants [pG11]. Furthermore, GEARs does not support *or* groups.

Criteria	GEARS	p::v	prevision	FeatureIDE	Feature Plug-in
Popularity	+	+	o	o	o
Relations	+/o	+	+	+	+
Constraints	+	+	+	+	+
Configuration	+	+	+	+	+
Code	+	+	+	+	-
Model	+	+	+	-	+
Extendability	o	+	-	+	+

Table 9: Feature model tool comparison

For our prototype implementation, we selected pure::variants for the following reasons:

- one of the most prominent variant management tools in the industrial domain as well as in the SPL and Feature Oriented Software Development (FOSD) community [FOS11].
- pure::variants was already set at some of our industrial partners that supported us in the evaluation process.
- pure::variants can be extended since it is based on eclipse.

However, we also plan to integrate MoSo-PoLiTe with GEARs and since MoSo-PoLiTe is implemented as an eclipse plugin, it can also be integrated with FeatureIDE. But, since FeatureIDE does not yet support the configuration of models, it can not support the entire MoSo-PoLiTe framework including model-based testing.

10.1.2 *pure::variants Overview*

Figure 47 depicts the conceptual overview of pure::variants [pG11]. On the left-hand side the problem space consists of the feature model and the variant model. The variant model is a configuration in which the selection for a certain product was made. The solution space is modeled using a family model and a variant realization. The feature model represents requirements, properties and relations, whereas the family model represents the implementation artifacts of the SPL such as code fragments, components, and documentations. To generate the variant realization representing a product the configuration in the variant model is used to configure the family model. Thus, the code fragments or other artifacts are selected to be part of the product.

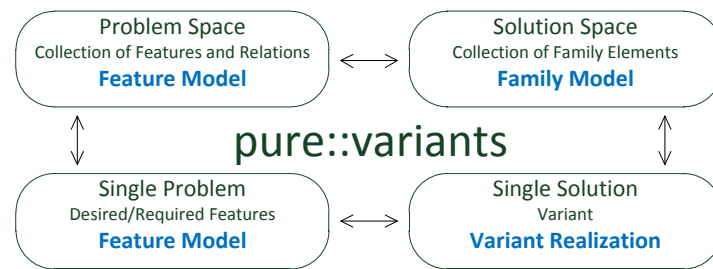


Figure 47: pure::variants overview [pG11]

Hence, a variant management project in pure::variants includes three different types of models. The feature model itself (*.xfm), a family model (*.ccfm) representing e.g. the data structure of the code implementing the SPL, and the variant model (*.vdm), where the features are selected for configuration/product derivation.

Beside the ability of generating products based on programming languages such as C/C++ or Java pure::variants can configure artifacts in different tools. To name some, the current pure::variants integrations address Doors, Rhapsody, Enterprise Architect, and Matlab/Simulink. For our contribution we use pure::variants for Rhapsody allowing to map states and transitions within a statechart in Rational Rhapsody to features in pure::variants.

Figure 48 depicts the feature model of our running example in the pure::variants' typical explorer view. Exclamation marks represent *mandatory* features, question marks indicate *optional* features, crosses represent *or* groups, and the double arrow represents *alternative* groups. *Require* and *exclude* constraints are represented via green and red arrows. Please note that we cannot describe the pure::variants data structure due to non disclosure agreements.

10.2 COMBINATORIAL SUBSET SELECTION

The implementation of the combinatorial subset selection is a two step procedure: First, our flattening algorithm transforms the feature model into a binary CSP. Then, our subset extraction algorithm generated the combinatorial subset of configurations.

We have implemented both steps as an eclipse plugin ready for extension and implemented an import and export towards pure::variants to use this tool for:

- feature modeling
- map features to statechart artifacts within Rational Rhapsody to which pure::variants provides a tool integration.

Figure 49 depicts an abstract schema of our tool chain. To import the feature model from pure::variants, we have written an xml parser to parse the variant model into

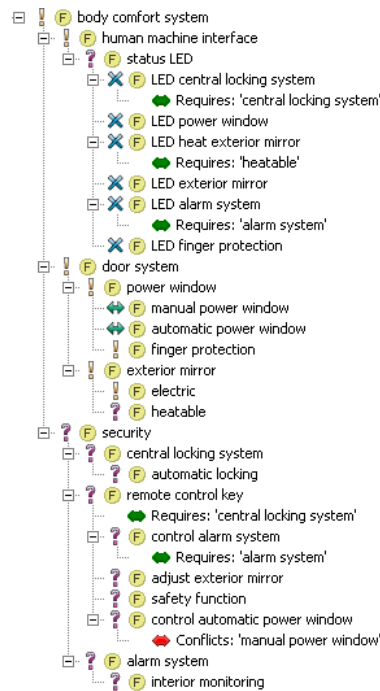


Figure 48: BCS in pure::variants

our data structure of our eclipse plugin. It would also be possible to parse the feature model itself, however, the following arguments let us to the idea to parse the variant model instead:

- all information included in the feature model are also available within the variant model
- one additional requirement towards our algorithm is to handle pre-selected configurations/features as well. For this purpose variant models are required.

The pure::variants feature model is stored within our own data structure and our combinatorial subset selection algorithms calculate the subset of configurations. Figure 50 depicts the data structure of our eclipse plugin.

A **feature** has a type (*optional*, *mandatory*, *alternative*, and *or*), a name, an ID, and a priority. An additional class **FeatureGroup** allows for describing groups of features as necessary for *or* and *alternative* features. Furthermore, this data structure provides *require* (**requires**) and *exclude* (**excludes**) constraints.

The child dependency is used to describe the hierarchical structure of the feature model. The additional **IConstraints** provide one possible extension point—an additional interface to support other types of constraints. Until know, this interface is not used.

Besides the Java-based implementation, we additionally implement the flattening with Fujaba/MOFLON to demonstrate that this part of our algorithm can be realized using well-known model transformation techniques.

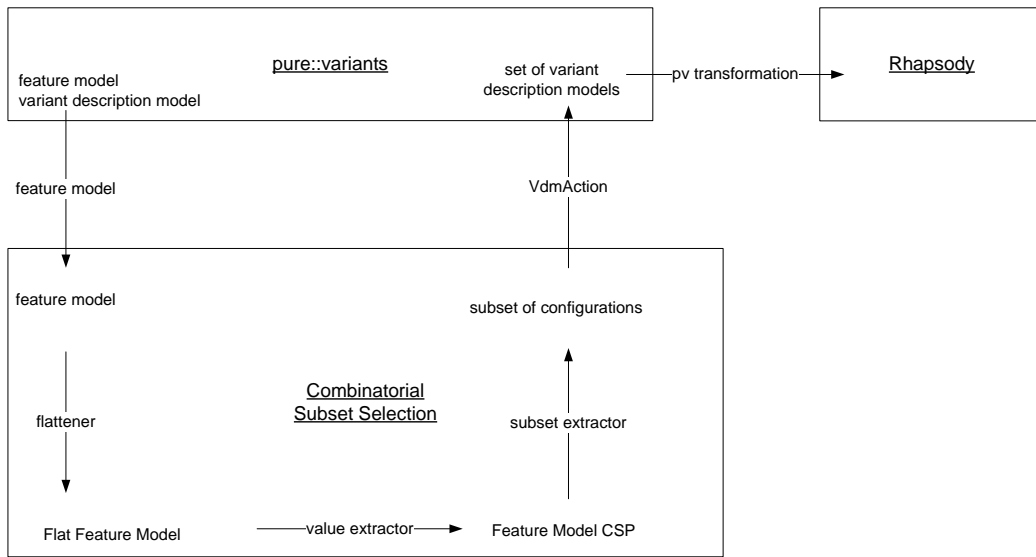


Figure 49: Tool chain dataflow

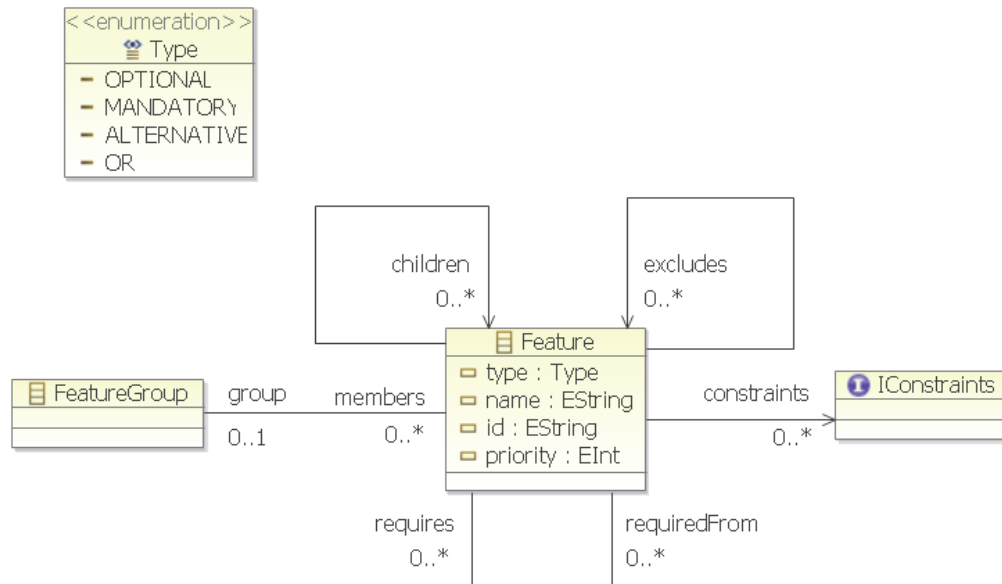


Figure 50: Data structure of the eclipse plugin

10.2.1 CSP Extraction

The following pseudocode in listing 4 describes the implementation of the flattening as described in the Concept and Theory part on page 85. The algorithm starts bottom-up by searching for a three-level subtree within the feature model and incrementally manipulates the entire feature model until there is no more three-

level subtree left (line 1–2). This is exactly the case when all features are pulled up directly beneath the root node. In the following, we use the child, parent, and grandparent notion as described in the **Concept and Theory** part on page 85 to ease the understanding—the algorithm, however, only uses the child/member relation.

The first block (line 5–9) handles subtrees with *mandatory* child features. In this case, all child features are merged with the parent feature and all constraints starting or ending at those child features are moved to the new merged feature.

The second block (10–15) handles *optional* child features. If the parent node is not *mandatory*, a *require* constraint is added between the child and the parent. Then, the child is pulled aside of its former parent node by setting the former grandparent as its new parent.

The third block (16–27) covers child features within an *alternative* or an *or* group. For each child a *require* constraint is added pointing to the parent node if the parent node is not *mandatory*. Then, each child is pulled aside its parent node by setting the former grandparent as its new parent. Finally, if the parent is not *mandatory*, an additional negation feature is added to the group of child features. This features represents the negation of all child features.

Listing 4: Flattening pseudocode

```

begin
2   for every three-level subtree do begin
   for every child of subtree do begin
4     switch(.TypeOfChild)
     case(mandatory)
6       combine child with its parent forming the new parent feature
       propagate all dependencies to new parent node
8       delete child nodes
       break
10    case(optional)
       if parent is not mandatory
12      add require dependency between child and its former parent to
        ensure that the former child node requires its former parent
       end if
14    child node is pulled aside its former parent node
       break
16    case(alternative/or)
       for every child of group do begin
18       if parent is not mandatory
        insert require edge between child and its parent
20       end if
        child node is pulled aside its former parent node
22      end
       if parent is not mandatory

```

```

24         add one negation feature representing the negation of all
           former child features to group
           insert exclude edge between negation feature and the former
           parent
26         end if
           break
28     end for
   end for
30 end

```

The flattening results in the flat feature model (FFM). To transform the FFM into a feature model CSP (FM CSP), parameters and corresponding values need to be extracted. For this purpose the aforementioned (cf. Part III Chapter 7) value extraction is applied. Here, we only provide a brief description and refer to page 95 for further details.

- **optional:** An *optional* feature is changed to a *mandatory* feature with two child features. One presenting the presence of this feature and one representing its absence.
- **mandatory:** *Mandatory* features stay *mandatory* and obtain an additional child feature representing the value of this feature.
- **or:** Extracting the parameter values of an *or* group is the most complex rule. Each element of the *or* group turns into an *optional* feature with two corresponding values. An additional n-ary *exclude* constraint is added to exclude the deselection of all features at a time.
- **alternative:** An *alternative* group stays unchanged but a single placeholder feature is added in between the *alternative* group and the root node representing the parameter. The *alternative* group itself represents the possible values.

This computation is done for each feature/group of features and a list of features with its corresponding values is created. Figure 51 depicts the data structure of the FM CSP. Each parameter knows its possible values and each value knows its

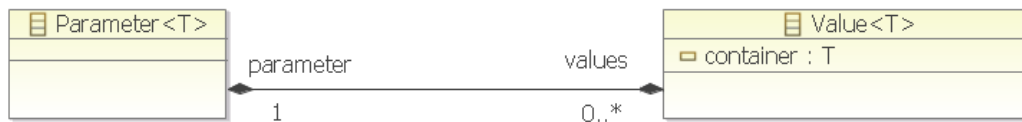


Figure 51: Data structure of the FM CSP

corresponding parameter. Furthermore, each value has a container including the constraints: *require* and *exclude*.

10.2.2 MOFLON - SDM

It is a natural choice to implement the graph transformation rules to translate a feature model into a binary CSP with a graph transformation language. One possible framework to implement these rules is provided by MOFLON/Fujaba [AKRS03]. The graph transformation language provided by Fujaba is a combination of graph transformation rules and UML activity diagrams and is called **story-driven modeling** (SDM). SDM realizes in-place model transformation specified by graph transformation rules. Activity diagrams are applied to specify the application and control flow of the graph transformation rules.

A graph transformation rule describe the modification of a certain graphical structure. Thus, this structure is the precondition of a certain transformation rule. The postcondition is the resulting modified structure of the graph. The structure of a graph that matches a certain precondition is referred to as left-hand side (LHS). The resulting graph is called right-hand side (RHS).

The modifications realized by a graph transformation rule includes:

- creation of new parts which are not part of the LHS
- deletion of parts which are then not part of the RHS

In SDM, LHS and RHS and the modifications are described within one single visual representation. Colors are used to depict the parts of the LHS (black and red) and the RHS (black and green). Additionally, stereotypes indicate if a part of the LHS is removed (stereotype «destroy») or a part is added (stereotype «create»).

For our SDM implementation, we use the recently released eMoflon plugin for Enterprise Architect [ALPS11]. Using this plugin Enterprise Architect can be used to create SDMs and to generate EMF based Java code. Figure 52 depicts the SDM rule for arbitrary parents and *optional* children (cf. page 86). The similarity between the four rules for *optional* child nodes allows for handling them with one single rule. The SDM rule includes the following four steps:

1. Check whether the precondition is satisfied: child optional.
2. Pulling up the child aside the parent feature.
3. If the parent feature is *mandatory* the transformation is complete.
4. Otherwise, if the parent in *optional* or within an *alternative* or *or* group an additional *require* constraint is added pointing from the child to the parent.

Figure 53 depicts the underlying SDM transformations realizing the four steps to process subtrees with *optional* child features. In the first step a question mark at the parent feature indicates that this relation is not of interest in this step. The relation of the parent comes into play in step 3 and 4.

All in all, three SDM diagrams are required to describe the behavior of all 16 transformation rules, which we list in the appendix of this thesis. One for *mandatory* child features, one for *optional* child features and one for child features within an *or/alternative* group. Furthermore, we present the SDM-based implementation for iterating through the feature model.

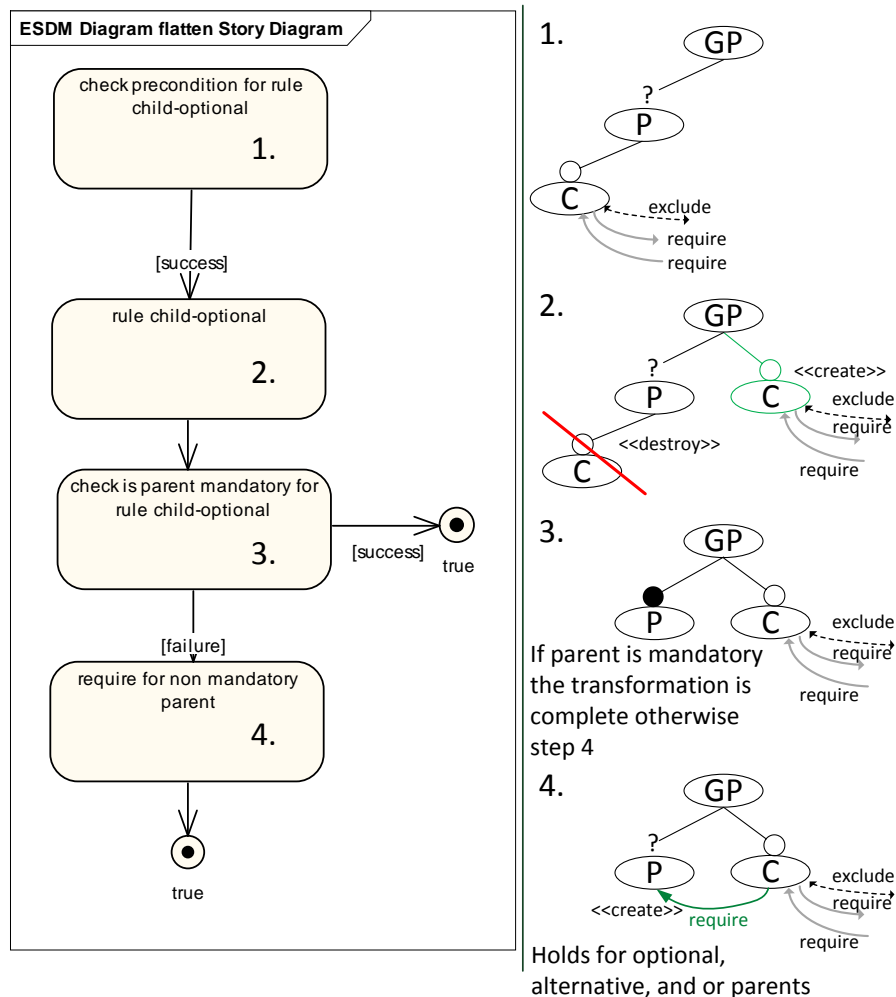
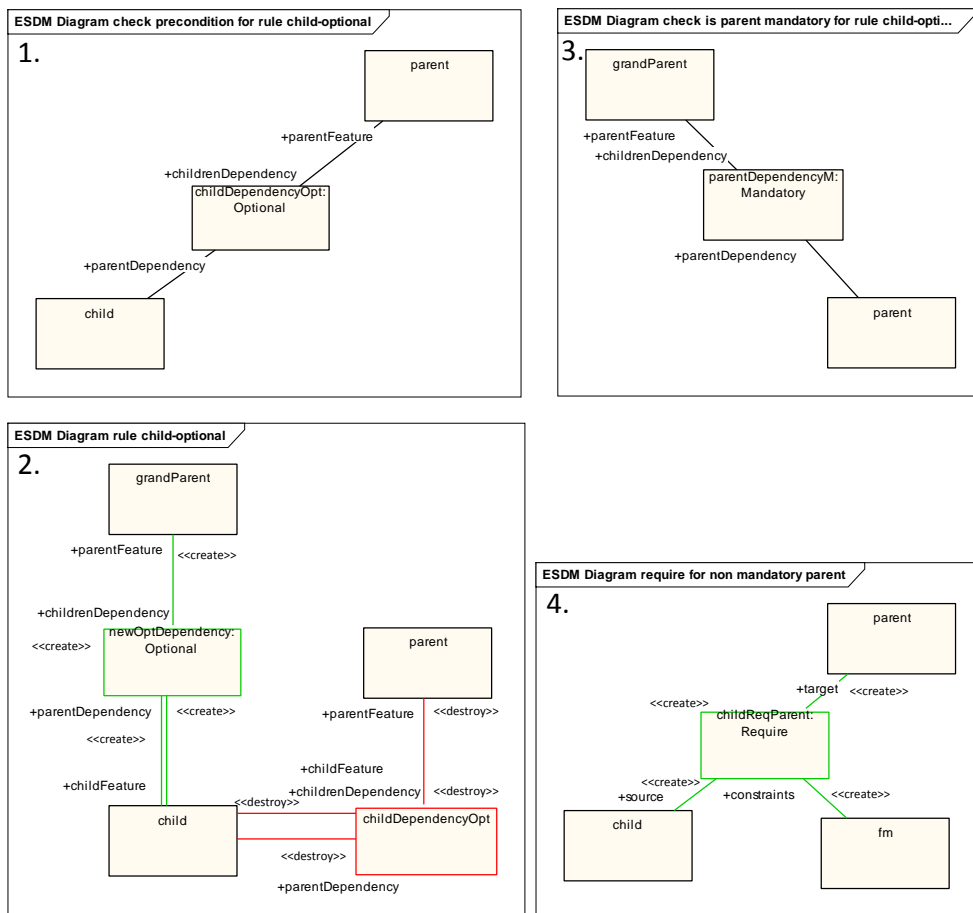


Figure 52: Steps of the SDM-based flattening handling a *optional* child node

The SDM-based graph transformations are then used for code generation. The resulting code contained much more lines of code than our Java implementation. Reasons for that are:

- The generated code always contains some additional information and checks as we will show with regard to the generated code of one of the transformation rules.

Figure 53: Detailed view of the SDM transformation processing an *optional* child

- we do not claim that we implemented the SDM flattening rules the possibly best way.

To compare the manually written code with the generated code, we select the generated code fragment implementing step 4 of the transformation and compare it to our manual implementation. Listing 5 depicts the generated code.

Listing 5: Generated Source Code for *child optional* step 4

```

try {
2           fujaba__Success = false;

4           // check object child is really bound
           JavaSDM.ensure(child != null);
6           // check object fm is really bound
           JavaSDM.ensure(fm != null);
8           // check object parent is really bound

```

```

10     JavaSDM.ensure(parent != null);
        // check isomorphic binding between objects parent and
        // child
        JavaSDM.ensure(!parent.equals(child));

12

14     // create object childReqParent
        childReqParent = FeatureModelLanguageFactory.eINSTANCE
            .createRequire();
        // create link
        childReqParent.setSource(child);

16

18     // create link
        org.moflon.util.eMoflonEMFUtil.addOppositeReference(
            childReqParent, parent, "target");
        // create link
        fm.getConstraints().add(childReqParent);

20

22     fujaba__Success = true;

24     }

```

The manual implementation processing optional child features is much shorter and depicted in Listing 6. Line 6 and the lines 16–18 implements step 4 of the transformation. With regard to lines of code (LoC), the generated code for optional child features has 142 LoC and the manual implementation around 19 LoC. With regard to step 4 the generated code requires 24 LoC and the manual implementation 4 LoC (including the method `requireFeatures()`). We recognized that the generated code includes more LoC because of additional checks ensuring that the child, the parent and the entire subtree is initialized and not null. The generated code executes these checks for every step during each transformation.

Listing 6: Manually written source code for child *optional*

```

List<Feature> featuresToAddToGrandParent = new ArrayList<Feature>();
2   ...
   case OPTIONAL: {
4       featuresToAddToGrandParent.add(child);
        if(newparent.getType() != Type.MANDATORY){
6           Featuremodel.requireFeatures(child, newparent);
        }
8       break;
   }
10  ...
    grandParent.getChildren().addAll(featuresToAddToGrandParent);

12

14  /** define require dependency
    * @param requires
    * @param required
16  */

```

```

    public static void requireFeatures(Feature requires, Feature required)
    {
18         requires.getRequires().add(required);
           required.getRequiredFrom().add(requires);
20     }

```

To further compare the manually written Java implementation with the generated code, we implemented a runtime analysis. For this purpose we implemented a feature model generator that we will explain in detail in Section 10.4 of this chapter. We used the feature model generator to generate 1650 feature models with a varying number of features (100-200,000) and the following distribution of relationships: *alternative* (21,7%), *or* (24,1%), *optional* (23,0%), and *mandatory* (28,1 %) according to the lessons learned summarized in [SR11]. For all of these feature models, we compared the execution time of both flattening implementations. The results of this comparison are depicted in Figure 54.

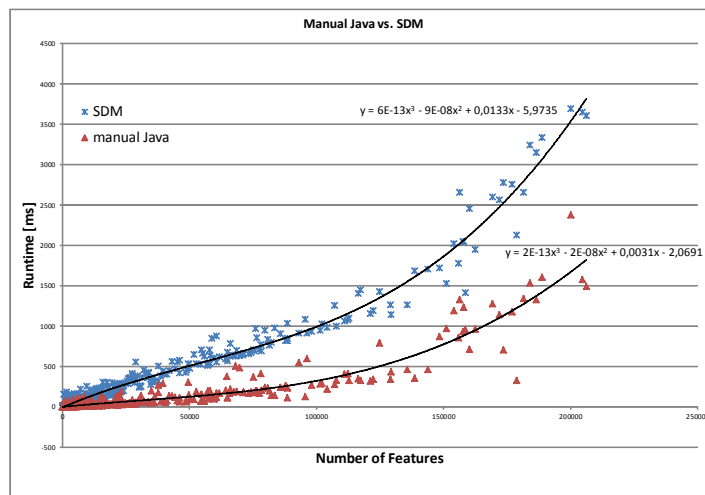


Figure 54: Runtime Flattening for manual Java implementation and SDM-based

Initially, the manually written Java code is 10 times faster than the generated code. However, the performance of the manually written code seems to decrease with growing feature models. This can be visualized by plotting the difference of the manual Java runtime and the SDM-based runtime depicted in Figure 55. The decreasing difference between manually written code and generated code seems to be logarithmic indicating that the difference will decrease even further. We are currently busy to precisely compare the generated code and our Java implementation with regard to further optimizations and to find out details about the varying differences in the runtime performance. However, this is beyond the scope of this thesis. Within our tool chain, we stick with the Java implementation of the flattening algorithm since we are not aware of feature models containing more than 6000 features. Currently, the feature model of the linux kernel (6000 features)

and the ECOS feature model (1000 features) are the biggest feature models known in the SPL community [Lab]. Unfortunately, we are currently not able to use these feature models for evaluation purposes since we are not able to process the data structures of these feature models.

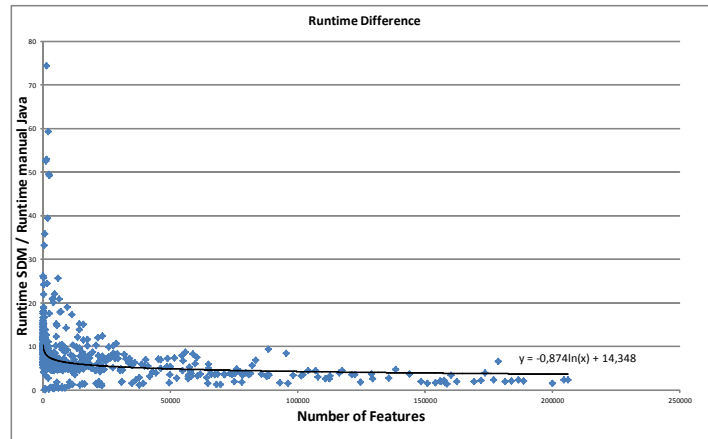


Figure 55: Visualization of the runtime delta

10.2.3 Subset Extraction

The pseudocode of the subset extraction that we have introduced within our Concept and Theory part is implemented as an eclipse plugin that is capable of processing feature models created with pure::variants. Figure 56 depicts the generated set of configurations exported as variant models in pure::variants.

The different configurations include varying numbers of features. Figure 57 shows 3 of the 17 configurations: 1, 7, and 16. Configuration 7 has very few features, whereas configuration 16 has many features. Configuration 1 is of medium size.

Pre-Selection Functionality

In the general case, companies that switch to SPL engineering already have a certain set of products that are already sold on the market. We assume that these products have been tested extensively before they are passed on to the customers. Thus, we implemented a so-called pre-selection functionality that is capable of taking an arbitrary set of products into account when calculating the combinatorial subset of configurations. Those products need to be added as configurations into pure::variants. Our algorithm then calculates the pairs (or T-tuples) of features that are already covered by those products and calculates additional configurations covering the remaining pairs (or T-tuples). The resulting set of products then contains the pre-selected products along with further configurations to cover all valid pairs (or T-tuples) of features. The benefit of this approach is the fact that only the additional products have to be tested.

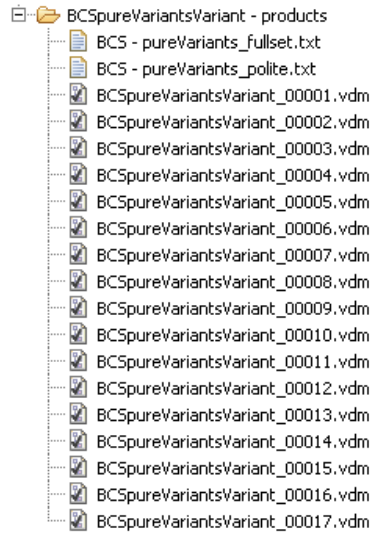


Figure 56: Subset of configurations in pure::variants

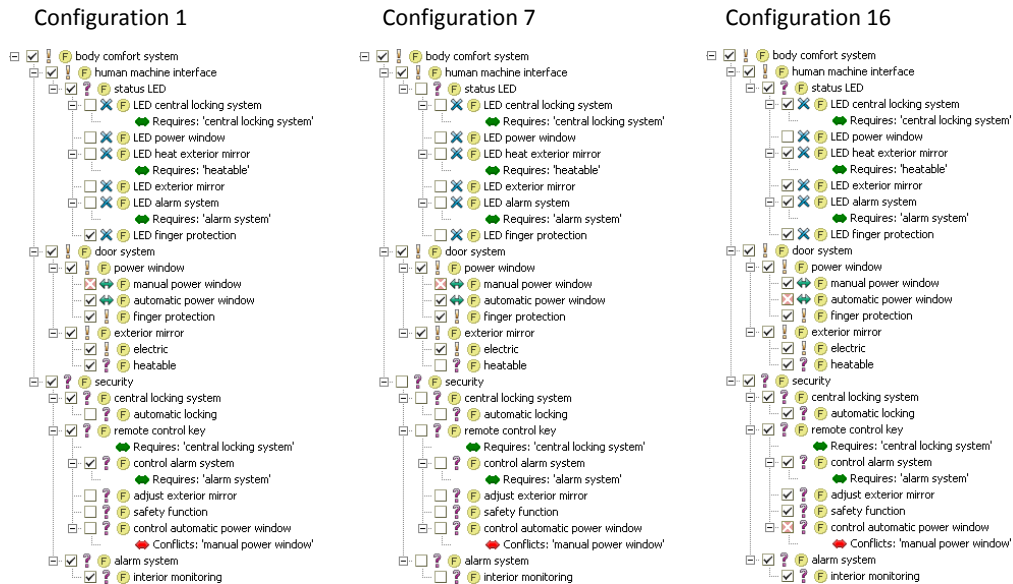


Figure 57: Three of 17 configurations of the combinatorial subset in pure::variants

Please note that a feature model is still required for the pre-selection functionality. Steven She introduced a so-called feature model mining algorithm to compute a feature model out of a set of configurations/products [Sheo8]. His algorithm can be used to generate an initial feature model that can then be extended under consideration of stack holder requirements.

10.3 MODEL-BASED TEST CASE GENERATION

Each configuration/product of the combinatorial subset requires test cases. Model-based testing is one possibility to generate test cases for product configurations.

In this thesis, we describe the application of Rhapsody ATG [AG11b] to set up a 150% test model and to map it to features of the feature model in pure::variants. Rhapsody is the quasi standard UML-oriented CASE tool for model-driven engineering in the automotive domain [IBM11]. Rhapsody provides different quality assuring techniques such as automated test case generation with ATG on the basis of code, statecharts, and sequence diagrams.

pure::variants already provides a mapping between features and artifacts of models in Rhapsody. This mapping is able to be used exactly the way we have specified it in the Concept and Theory part: A feature can be either mapped to states or transitions of a statechart. Then, configuration specific 100% test models can be generated and ATG can be used to derive test cases satisfying certain structural coverage criteria.

The integration of pure::variants and Rational Rhapsody has already been implemented by pure-systems. However, the idea of building a reusable test model and to use ATG to automatically generate test cases for each derivable application is our contribution. According to the best of our knowledge we have provided the first contribution towards model-based testing using a variant management tool such as pure::variants in combination with a commercial modeling or testing tool such as Rhapsody and ATG. In conclusion we can state that we do not reinvent or improve model-based testing concepts in general; but we brought together a well-known variant management tool and a model-based testing tool chain.

10.3.1 Rational Rhapsody

Rational Rhapsody [IBM11] is a UML-compliant model-driven Software Engineering tool specialized on modeling real-time systems and embedded systems. It was initially developed by I-Logix in 1998 and bought by IBM in 2008 and integrated in the Rational product line.

Figure 58 depicts a simple Rhapsody statechart diagram including all different notations that will be used within this thesis.

The statechart consists of two parallel substatecharts separated by a dotted line. Both substatecharts are executed in parallel if **LED_PW** is activated. The transitions are annotated with **Trigger[Guard]/Actions**. **Trigger** and **Guard** control whether a transition is fired or not. The **Guard** defines the precondition of a transition to be fired and the **Trigger** defines an event that needs to occur before the transition can fire. Both, **Trigger** and **Guard** are optional within a statechart.

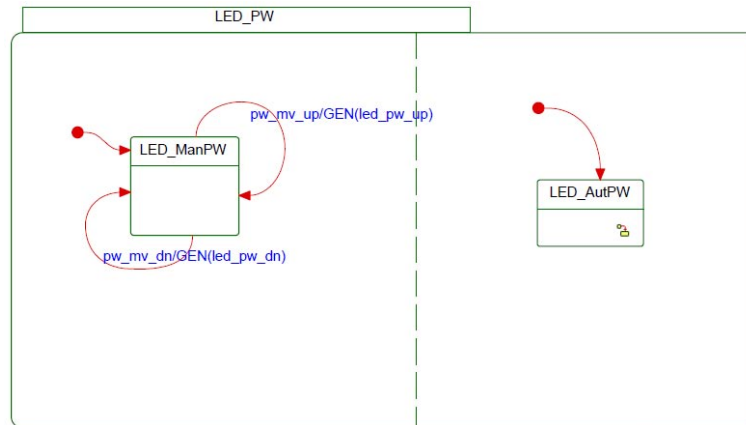


Figure 58: BCS statechart example in Rhapsody

The result of a transition is defined as **Action**. An **Action** can be a value assignment of a certain variable or to throw an event. Those events are either outputs like messages or the activation of a **Trigger** starting another Transition.

LED_AutPW is a so-called hierarchical state including a statechart. This is depicted as a little yellow colored icon at the bottom right side in **LED_AutPW**. If **LED_AutPW** is activated the underlying statechart is executed.

Within MoSo-PoLiTe, we use Rhapsody to model the 150% test model representing the entire SPL under test. To configure the 150% test model using pure::variants, the pure::variants-Rhapsody integration is used. Technically, this is solved by adding additional constraints within the Rhapsody models including a stereotype *pv restriction*. Those constraints mark a certain artifact as variable and allow the model to be configured.

Within the constraints, terms can be defined consisting of feature names and compositions *NOT*, *AND*, and *OR*. Those constraints allow to configure the 150% model by selecting features. A model artifact is selected if:

- it has no constraint - then it generally belongs to the commonalities of the SPL
- if the logical expression within the constraint turns to true with regard to the feature selection in the variant model.

Thus, the constraints realize the mapping between pure::variants and the test model in Rhapsody. With regard to statecharts, *states* and *transitions* can be mapped to features within pure::variants.

The different types of mapping possibilities are depicted in Figure 59. Please note that we use an abstract example to depict the different types of annotation because our BCS implementation does not use all of those mapping strategies.

The examples 1 and 2 in Figure 59 show the mapping of *feature0* to a state or a transition respectively. It is not possible to configure the captions of a transition

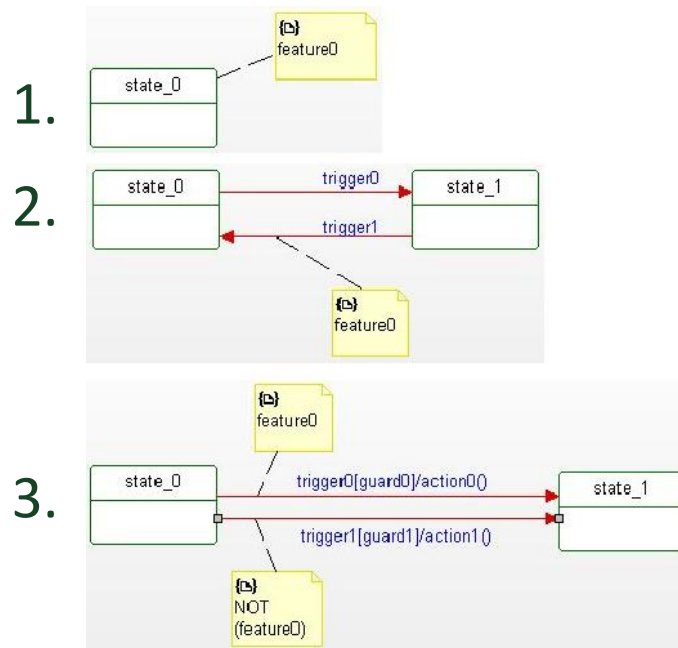


Figure 59: Examples for mapping features to statechart artifacts in Rhapsody

such as **Triggers**, **Guards** or **Actions**. However, this drawback can be compensated by adding a copy of the transition, adapt the **Trigger**, **Guard** or **Action** and add a constraint including a different feature dependency. This is shown in example 3 of Figure 59. The transition with `action0()` is mapped to `feature0` and is selected if `feature0` is part of the configuration in `pure::variants`. The transition with `action1()` is mapped to `NOT feature0`. Thus, this transition is selected if `feature0` is not within the corresponding configuration in `pure::variants`.

When a Rhapsody project is selected to be configured within a `pure::variants` project then the entire 150% model is generated when a configuration is derived. Afterwards, the *pv restrictions* within the Rhapsody project are resolved using the feature selection. States and transitions that are mapped to features that are not selected within the variant model are deleted within the 150% model resulting in a 100% model.

Figure 60 depicts a 150% test model and a 100% test model representing the behavior of the **Finger Protection** feature. The left-hand side shows the 150% implementation. The statechart includes the transition specific for **Automatic Power Window (AutPW)** and **Manual Power Window (ManPW)**. The right-hand side shows the statechart of the **Finger Protection** of configuration number 7 that includes the **Finger Protection** features for as well as the **AutPW** feature. The transition mapped to **ManPW** was simply deleted.

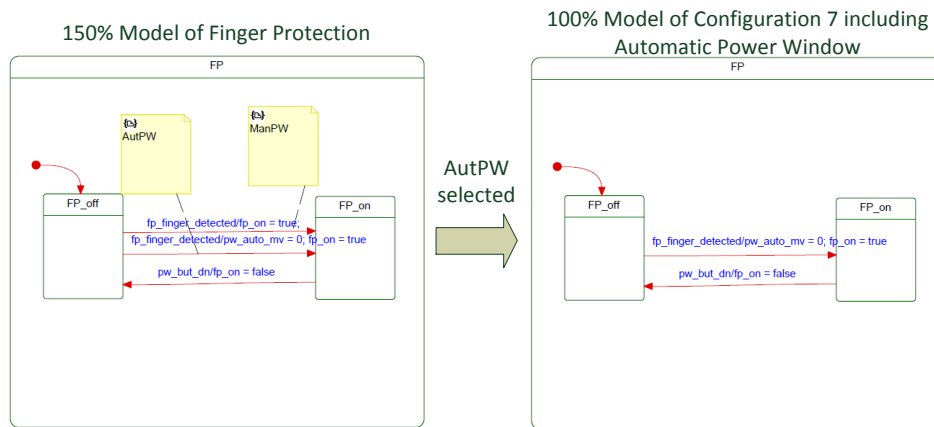


Figure 60: Example for deriving a 100% test model out of a 150% test model

10.3.2 ATG

ATG (Automatic Test Generation) is an Add-on for Rational Rhapsody for model-based test case generation [AG11b]. We refer to [AG11b] with regard to the technical details of how to use ATG in combination with Rhapsody. ATG generates test cases with regard to certain coverage criteria. Two different coverage criteria can be selected.

- Model Element Coverage which includes state and transition coverage.
- Model Code Coverage which complies with MC/DC coverage as introduced in the Background and Related Work part of this thesis.

Additionally, a time out specifying a certain period of time for test case generation can be defined. Thus, ATG either generates test cases until the coverage criteria are satisfied or the test case generation exceeds the time-out. In ATG a test case consists of a set of input signals that need to be fired to reach the different test goals to fulfill a certain coverage criteria. For a detailed description of how the test cases are generated and how those can be transformed into JUnit Tests, we refer to [Zin11].

10.3.3 Alternative Tooling

An alternative to Rhapsody/ATG is ParTeG [WSSo8]. A prototypical tool chain is currently under development. According to the ParTeG strategy, a context class is used to map the features of the feature model to elements of the test model. ParTeG uses two diagrams for describing test models: state machines and class diagrams. For the behavioral description, the state machine references elements from its context class such as attributes, constants or operations. The behavior of

the state machine depends on the definition of the corresponding context class and the behavior of a product variant can be adapted by adapting or exchanging the context class.

Again, a 150%-state machine is used to describe the behavior of all possible products of the SPL. Until now, the first mapping approach between features and the context class are restricted to a mapping from one feature to constant attribute values. For a first prototype implementation this is sufficient since many product variant descriptions are configured via simple boolean compilers. Thus, the interface of feature models and system models consists of the defined context class elements in the system model. By setting the constant attribute values, the behavior of the 150%-state machine can be adapted. Together with FeatureIDE, ParTeG seems to be an appropriate candidate to provide an open source tool chain for MoSo-PoLiTe.

10.4 TESTING THE IMPLEMENTATION

To systematically test the implementation we need to check the following characteristics:

- Semantical equivalence: All transformation rules of the flattening preserve the semantics of the feature model.
- Consistency: we only generate valid products.
- Completeness: the generated products cover all valid pairs of features.

We have used some of the feature models listed on [SR11] to test our implementation. Furthermore, we have implemented a feature model generator similar to SPLOT [SR11] generating feature models in our data structure. This feature model generator creates feature models with varying relations, constraints, and varying size, depth, and width. The feature model generator compensates the lack of a wide variety of different feature models of industrial SPLs. We apply our approach to various generic feature models and determine the outcomes. To examine the coverage of our approach according to pairwise interaction coverage we utilize a SAT-Solver scenario to prove that all valid pairs of features of the SPL under test are covered using our approach.

10.4.1 *Systematic Validation*

For validation purposes, we have implemented a feature model generator (FMG) that creates random feature models considering certain input parameters controlling the size of the feature models. A root node is created, which obtains a random number of features restricted by an adjustable maximum. The FMG sets the relationships of the features beneath this root node. When generating a feature model

two possible configurations are possible: Every node receives a random relation or the distribution of *mandatory*, *optional*, *or*, and *alternative* features can be configured. The percentage of every relation, *mandatory*, *optional*, *or*, and *alternative* can be adjusted by parameters. For our test runs the FMG uses the following distribution of relations: *alternative* (21,7%), *or* (24,1%), *optional* (23,0%), and *mandatory* (28,1 %) according to [SR11]. Every child is handled as a root node and obtains its own children in an analogous manner. Thus, the algorithm creates a complete feature model iteratively, aborted by a defined maximum depth. To generate asymmetric feature models, the random number of added children can be zero, too. In order to approximate real SPL feature models, the generator also creates constraints. To satisfy this functionality, a valid random pair of features is selected and a constraint (either *require* or *exclude*) is set. The number of inserted constraints depends on the total number of nodes and a parameter configures the percentage of pairs of nodes involved within constraints.

10.4.2 Semantical Equivalence

Two feature models are semantically equivalent if they describe the same set of products with respect to a given set of features. We validate the flattening algorithm by comparing the propositional formulas of the original and the flat feature model as proposed by [TBK09]. With regard to our running example, the following logical expression (in conjunctive normal form) is calculated for both FMs:

$$\begin{aligned}
& BCS \wedge (\neg AS \vee security) \wedge HMI \wedge (\neg RCK \vee security) \\
& \wedge (\neg RCK \vee CLS) \wedge (\neg CLS \vee security) \wedge PW \\
& \wedge (\neg IM \vee AS) \wedge (\neg AL \vee CLS) \wedge (\neg CAS \vee AS) \\
& \wedge (\neg AEM \vee RCK) \wedge (\neg SF \vee RCK) \wedge fp \\
& \wedge electric \wedge (\neg ManPW \vee \neg CAPW) \wedge (\neg LEDcls \vee CLS) \\
& \wedge (\neg LEDheatable \vee heatable) \wedge (\neg LEDas \vee AS) \\
& \wedge DS \wedge EM \wedge (\neg CAPW \vee RCK) \wedge \\
& (\neg LEDcls \vee StatusLED) \wedge (\neg LEDpw \vee StatusLED) \\
& \wedge (\neg LEDem \vee StatusLED) \wedge (\neg LEDheatable \vee StatusLED) \\
& \wedge (\neg LEDas \vee StatusLED) \wedge (\neg LEDfp \vee StatusLED) \\
& \wedge (\neg StatusLED \vee LEDcls \vee LEDpw \vee LEDem \vee \\
& LEDheatable \vee LEDas \vee LEDfp) \wedge (AutPW \vee AutPW) \\
& \wedge (\neg AutPW \vee \neg ManPW)
\end{aligned}$$

We used the FMG to create additional feature models and apply the flattening algorithm. For all feature models, our implementation preserves the semantic equivalence.

10.4.3 Consistency

The consistency can be checked automatically within `pure::variants`. `Pure::variants` has an integrated solver to check for inconsistency and whether a feature configuration is a valid product or not. We rely on the fact that the internal solver works correctly.

10.4.4 Completeness

To further evaluate the implementation of our subset derivation algorithm we need to examine whether we cover all valid pairs of features with our set of products. To achieve 100% pairwise feature interaction coverage, we need to assure that all valid pairs of features are covered. The implementation of the subset derivation algorithm is complete if all products that contain an uncovered feature pair are inconsistent. Fig. 61 depicts this assumption.

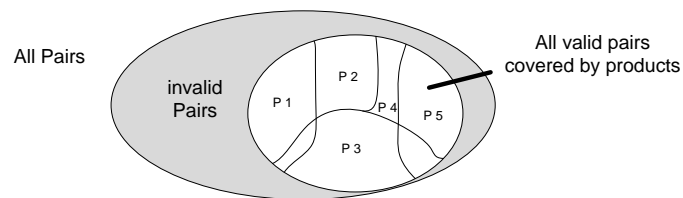


Figure 61: Valid and invalid pairs of features

We apply a SAT-Solver to prove that all uncovered pairs cannot be part of a valid product in order to validate the implementation of our algorithm.

1. we generate all pairs of features ignoring *require* and *exclude* constraints within the FM and write them into a list
2. we remove all pairs that are covered from the list
3. we check if all remaining pairs are invalid

For our experiments we use MiniSAT [ESo4] as SAT solver and solve the following equation:

$$\text{formula}(\text{feature model}) \wedge \left(\bigvee_{i=1}^{i=n} \text{invalid pair}_i \right) \quad (10.1)$$

where, n is the number of invalid pairs

If equation (10.1) is unsatisfiable, all pairs must have been invalid with respect to the feature model. Therefore, the products that are generated by our algorithm, must cover all possible pairs of features. We applied MiniSAT to the BCS feature

model combined with all pairs that are not part of any generated product instance and, therefore, must be invalid. We found 417 potentially invalid pairs and reviewed them with one run of MiniSAT. This proves that all pairs not covered by any of the generated product instances are invalid pairs like shown in Fig. 61. MiniSAT took less than 1 second and 2 MB of memory to find that the expression is unsatisfiable. We used an ordinary desktop computer without assuring that any other program, thread or service influences MiniSAT. With regard to T-wise testing, the test for completeness can be done in the same manner replacing pairs with T-tuples.

10.4.5 Evaluation of Efficiency

To test the efficiency of our algorithm, we applied it to some human-made feature models listed on [SR11]. The results are listed in Table 10. The last two columns contain the number of products realizing pairwise coverage and the runtime of the algorithm in milliseconds on a 2 Ghz Single Core machine with 2 GB RAM.

Feature Model	Features	# of Config.	2wise Config.	Runtime [ms]	3wise Config.	Runtime [ms]
AndroidSPL	27	36240	24	26	105	116
Smart Home	35	1,048,576	23	33	61	208
Inventory	37	2,028,096	19	33	93	317
Sienna	35	2,520	25	38	87	122
Web Portal	38	2,120,800	26	41	160	869
Doc_Generation	44	$5.57 \cdot 10^7$	23	39	132	812
Arcade Game	61	$3.3 \cdot 10^9$	38	76	254	5939
Model_Transf.	88	$1.65 \cdot 10^{13}$	65	108	643	29386
Coche ecologico	94	$2.32 \cdot 10^7$	108	93	898	5916

Table 10: Feature models from SPLOT research [SR11]

To further test our approach, we additionally generate a set of 1023 random feature models automatically. The probability of the feature relations is selected with: *alternative* (21,7%), *or* (24,1%), *optional* (23,0%), and *mandatory* (28,1 %) according to [SR11]. We set the maximum depth to 5 and the maximal number of children per node to 4. Therefore, the maximum possible number of features is 256. The generated 1023 feature models have a mean number of 35 features with a standard deviation of 28 features. Please note that those generated feature models do not include *require* and *exclude* constraints. Thus, our algorithm just has to run on the constraints that are generated during the feature model to CSP transformation. We abandon this option within this test to have a significant different input compared to the SPLOT feature models.

Fig. 62 A) shows the relation between the number of features in the feature model and the number of generated configurations. We apply linear regression to the set

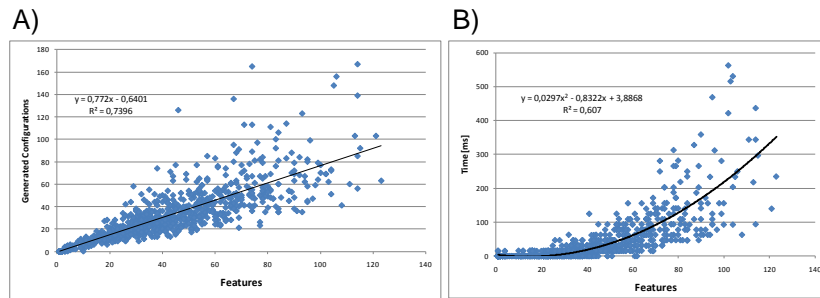


Figure 62: Statistics - Pairwise

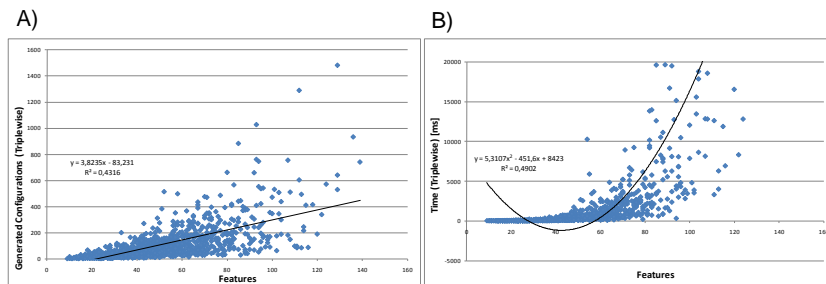


Figure 63: Statistics - Threewise

of values and found a slope of 0.772 and a y-intercept of -0.6401 . The square of the correlation coefficient is 0.7396 , which shows a strong linear dependence of the two values. Therefore, the average of the number of generated configurations will increase by 77.2% when the number of features in the feature model is doubled.

The relation between the number of features in the feature model and the corresponding calculation time is depicted in Fig. 62 B). The calculation time increases quadratic with an increasing number of features.

Figure 63 depicts the same diagrams considering threewise feature interaction coverage. In Fig. 63 A) threewise indicates that the number of generated configurations will increase by 382% when the number of features in the feature model is doubled. As indicated in 63 B) the calculation time drastically increases for the threewise case. For around 100 features, up to 20000 ms are required to calculate the result.

Fig. 64 shows the results for seven automatically generated feature models. Each feature model has its individual color. The left-hand side depicts the increasing percentage of pairwise coverage with every additional configuration in the subset. On the right-hand side, the same feature models are used for threewise feature interaction coverage. Except of the dark blue curve that requires an outstanding number of additional configurations to cover the last 10% of feature interaction coverage, there seems to be no obvious similarities between pairwise and threewise. We did not often experience cases similar to the dark blue curve indicating that a lot of different configurations are required to cover the last few pairs or triples of

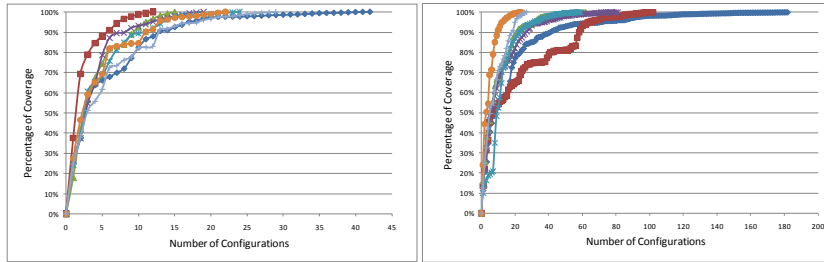


Figure 64: Comparison of pairwise and threewise feature interaction coverage for seven randomly generated feature models

features. In that particular case, the alternative groups contained several elements that had to be combined in various configurations to assure pairwise feature interaction coverage.

All experiments with regard to efficiency were executed on an ordinary desktop computer running Windows XP equipped with a dual core CPU with 2.66 GHz and 4 GB RAM: The experiments were done without taking into account that other programs, services or tasks might have a negative impact on our results. The execution time results are calculated by repeating our experiments 10 times and by taking the average value. The deviations between the different runtime results were around $\pm 10\%$. We measured the time of execution by adding counters within our algorithm. The results were then written in a logfile.

EVALUATION

THIS chapter summarizes the results of applying MoSo-PoLiTe to our BCS case study. On that basis, we provide a theoretical discussion about the potentials and limitations of pairwise feature interaction testing. Furthermore, we present the results of applying MoSo-PoLiTe to two different industrial SPLs. Last but not least, we discuss threats to validity and compare MoSo-PoLiTe with related approaches and compare their performance and results.

Hence, the evaluation can be divided into three applications:

- First, we apply the entire MoSo-PoLiTe tool chain to the BCS case study generating a combinatorial set of configurations with corresponding test cases generated on the basis of the reusable test model. Afterwards, we apply mutation analysis to the source code of the BCS to analyze the degree of fault coverage (Section 11.1) achieved by testing the combinatorial set of configurations. We also use the BCS case study as a basis to provide a theoretical discussion of the potentials and limitations of pairwise testing within SPLs in general.
- Then, we apply the combinatorial SPL testing component of the MoSo-PoLiTe tool chain to two industrial SPLs provided by Danfoss [Dan11] and the Adam Opel AG [AG11a]. Please note that we solely focus on the combinatorial configuration selection algorithm within this part of the evaluation. There, the test cases that are used to test the combinatorial set of configurations are either created manually or already existed (Section 11.2 and 11.3).
- We conclude this Section by discussing threats to validity and by comparing MoSo-PoLiTe with related approaches (Section 11.4).

Figure 65 depicts the schematic representation of this chapter's outline.

11.1 BCS CASE STUDY

Applying MoSo-PoLiTe to the BCS-small feature model, we obtain **9** configurations out of **40** possible configurations. The entire BCS SPL consists of **27** features and the constraints limit the product space to **11,616** valid product configurations. Applying MoSo-PoLiTe to the BCS SPL we obtain **17** configurations listed in Table 11 achieving 100% pairwise interaction coverage.

	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇	P ₈	P ₉	P ₁₀	P ₁₁	P ₁₂	P ₁₃	P ₁₄	P ₁₅	P ₁₆	P ₁₇
security	x	x	x	x	x	x			x		x	x	x	x	x		
status LED	x		x		x	x		x	x	x	x		x			x	
LED central locking system			x			x			x		x		x			x	
LED power window			x		x	x		x	x		x						
LED heatable			x		x	x		x	x				x			x	
LED exterior mirror			x		x	x		x	x							x	
LED alarm system			x		x	x			x		x					x	
LED finger protection	x				x	x		x	x	x	x		x				x
manual power window			x		x	x				x						x	
automatic power window	x	x		x			x	x	x			x		x	x		
heatable	x		x		x	x		x	x			x	x	x	x		x
central locking system	x	x	x	x	x	x			x		x	x	x	x	x		
remote control key	x	x	x	x					x		x	x	x	x	x		
alarm system	x	x	x	x	x	x			x		x			x	x		
automatic locking		x			x						x	x	x	x	x		
control alarm system	x	x	x						x		x			x	x		
safety function		x			x				x		x	x	x	x		x	
control automatic power window		x			x				x			x		x	x		
interior monitoring	x			x	x	x			x		x			x	x		
Number of variable Features	10	9	13	9	11	14	1	7	17	4	14	8	11	11	10	16	2

Table 11: Overview of the configurations generated by MoSo-PoLiTe

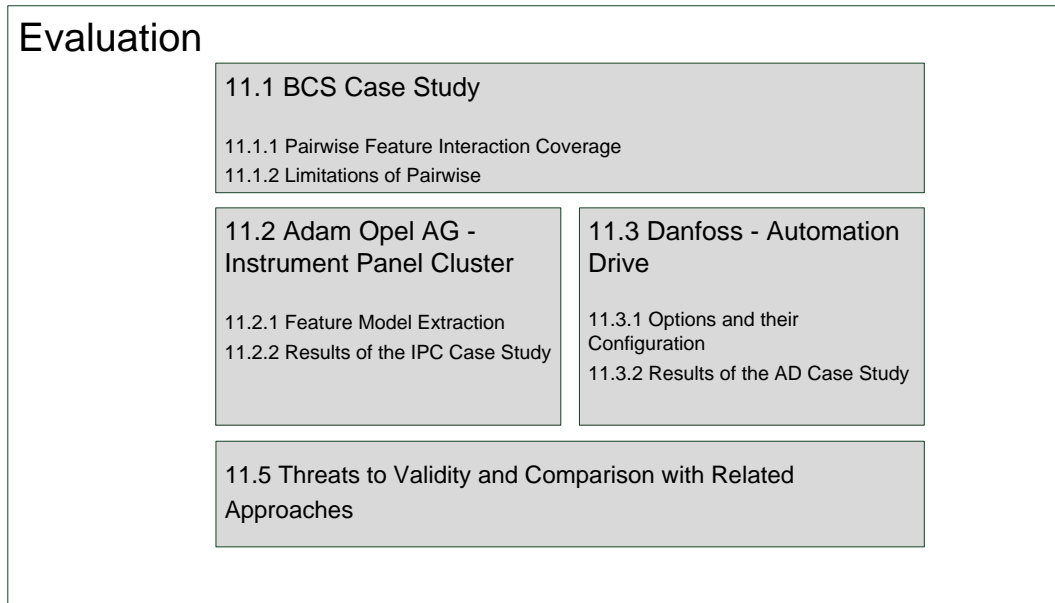


Figure 65: Outline of the evaluation chapter

Figure 66 depicts the generation progress of the combinatorial subset for pairwise and threewise interaction coverage. For pairwise interaction coverage, the number of generated configurations increases fast until five configurations and slowly saturates to 100%, afterwards. For threewise interaction coverage, much more configurations are required (more than four times as much) to satisfy the requested degree of coverage. Here, the interaction coverage increases fast until 16 configurations are generated (80% threewise interaction coverage achieved) and slowly saturates to 100% afterwards. To achieve threewise interaction coverage 77 configurations are required. Thus the testing effort for a guaranteed 100% threewise interaction coverage is much higher than for pairwise interaction coverage.

The BCS reusable statechart test model $TM_{SC150\%}$ consists of 93 states (20 composite states, 22 concurrent submachines, and 51 basic states) and 107 transitions [Zin11]. The mapping function interrelates artifacts of the test model with the features of the BCS feature model as follows:

- 12 features have a single mapping to statechart artifacts,
- 15 pairs of features are mapped in combination to statechart artifacts,
- 3 triples of features are mapped in combination to statechart artifacts, and
- 9 features have no mapping.

For the 9 features without a mapping, we assume a mandatory 0-wise mapping onto a *core* statechart model, which belongs to the commonalities of the SPL and is included within every model describing the behavior of a configuration.

#	CLS	AutPW	RCK
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Table 12: Triples of feature interaction for (1) $\{CLS, AutPW, RCK\}$

#	CLS	ManPW	RCK
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Table 13: Triples of feature interaction for (1) $\{CLS, ManPW, RCK\}$

interaction if none of the features is present. Thus, 17 different triples should be tested to cover all possible threewise interactions. MoSo-PoLiTe generates 17 representative configurations-under-test $PC_{UT} = S(FM(F))$ for our pairwise subset selection heuristics S .

Table 15 summarizes the coverage of $TM_{SC150\%}$ achieved under pairwise. For the 15 pairwise interactions, all 37 valid pairs are covered in PC_{UT} . Considering threewise interactions, just 14 of the 17 valid triples are covered in PC_{UT} . Together (pairwise and threewise), MoSo-PoLiTe covers 95.5 % of feature interactions.

For the test case generation and conduction on PC_{UT} , Rhapsody/ATG generates 62 test cases on average per product using *Model Element Coverage* and *Model Code Coverage* (MC/DC [ULo7]) criteria. For evaluating the feasibility of the testing approach and the quality of the tests generated, we applied several *mutation operators* to the products-under-test implementation code. In the majority of cases in which mutation detection fails, the test cases generated by ATG were not sufficient to cover them, whereas only a small number of mutations were missed because

#	CLS	AutPW	RCK
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

Table 14: Triples of feature interaction for $\{CLS, AutPW, LED\}$

	k=1	k=2	k=3	k>3	sum
# FI in SC	12	37	17	-	66
# covered	12	37	14	-	63
% covered	100	100	82,4	-	95,5

Table 15: Feature interaction coverage under pairwise in the statechart

they arise in particular, uncovered feature combinations. For instance, a faulty *LED* activation remained undetected because the unintended interaction only appears in the triple $(LED, AutPW, \neg CLS)$, which is not covered by the subset under pairwise. We refer to [Zin11] for further details about the BCS evaluation including a detailed description of the mutation analysis.

Since we generate complete configurations for testing purposes, the test activities intend to realize system testing. We assume that it would be possible to realize integration testing if feature combinations are selected to generate test cases. However, this is beyond the scope of our contribution.

Furthermore, we like to emphasize the fact that it depends on the granularity and degree of abstraction if the test model tends to support black-box or white-box testing. In the following, we discuss the potentials and limitations of the MoSo-PoLiTe approach with regard to the pairwise subset selection.

11.1.1 Pairwise Feature Interaction Coverage

The reliability of detecting, testing and covering feature interactions heavily depends on the appropriateness of the test model. We assume the test model $TM_{150\%}$ to specify intended feature interactions at a given level of abstraction. The mapping function interrelates features to fragments of the test model that define/implement

the intended cooperation and/or vetoing behavior between features in a product configuration. For our pairwise test approach, we make the following assumptions.

According to [McGo1] features $f \in F$ and feature combinations $F_i \subseteq F$ are not testable in isolation. Thus, features and feature combinations need to be assembled into some *valid* product configuration $PC \in FM(F)$ *under test*. Our subset-heuristics S selects product configurations under test $S(FM(F)) \subseteq FM(F)$ that fulfill a given (combinatorial) coverage criterion e.g. pairwise feature interaction coverage. As a consequence, interactions among features $F_i \subseteq F$ are only testable, if the criterion matches this F_i to be selected.

As already discussed in the Concept and Theory part of this thesis, the *one-wise* criterion ensures every feature $f \in F$ to be assembled to at least one product under test, and the *N-wise* criterion enforces every valid combination of features to be covered. Accordingly, T-wise criteria, $1 < T < N$ realize a reasonable trade off, especially $T = 2$, i.e., *pairwise* feature combination coverage. Concerning potential interactions between feature pairs as characterized above, our pairwise subset-heuristics is designed such that:

1. all *valid* combinations of feature pairs $\{f_i, f_j\} \subseteq F$, are covered. Thus, we do not only cover *intended* interactions but also (potentially) *unintended* pairwise feature interactions, and
2. all valid pairwise presence/absence combinations are covered, thus covering all ways of *optional* pairwise interactions.

Therefore, pairwise SPL testing suffices to support test cases for all (pairwise) interactions as defined in Section 6.2.

If there is an *intended* interaction, then

1. either *intended positive/negative* interactions are tested to correctly cooperate/veto,
2. *unintended negative* interactions arise, if the interaction is missing/faulty or
3. *unintended positive* interactions by means of behavioral influences is tested

Furthermore, for *optional* interactions

1. the correctness of *intended* interactions in the presence of both features is tested and
2. the conceptional independence of both features is tested by isolating them from each other.

Those interaction errors are only reliably discoverable if the underlying test models, the test case generation tools, and coverage criteria applied are adequate.

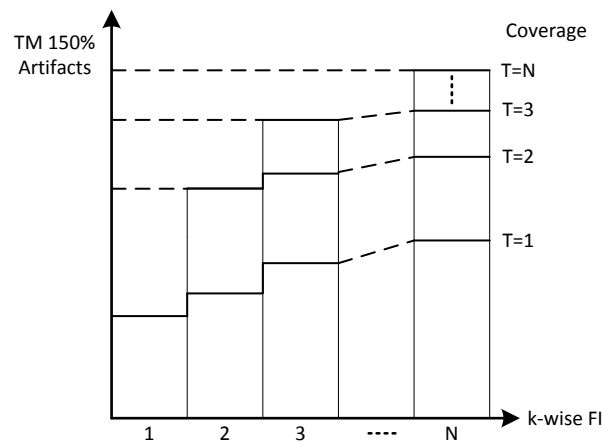


Figure 67: T-wise coverage of k-wise interactions

11.1.2 Limitations of Pairwise

Pairwise combinatorial subset selection ensures every valid combination of feature pairs $\{f_i, f_j\} \subseteq F$ to be covered by at least one configuration. In contrast, e.g., for an interaction involving $\{f_i, f_j, f_k\}$, no general coverage statement can be given.

We can visualize this by generalizing the problem to a k -wise interaction specified in a test model $TM_{150\%}$. Using a T -wise combinatorial coverage criterion with $T < k$ the coverage degree depends on

1. the (in-)dependencies between features in F_i . Independent features bare the risk to include optional interactions. For each optional interaction, potentially unintended/erroneous feature interactions might occur that need to be covered.
2. the subset selection heuristics algorithm used. Thus, for *fully dependent* features, i.e., *k-wise mandatory* interactions, $T = 1$ suffices to cover them. For fully independent features, i.e., *k-wise optional* interactions, $T = k$ is required and in any other case of *partial* dependencies, some $1 < T < k$ is satisfactory.

This relationship between k and T is illustrated in Fig. 67: T -wise suffices to cover all k -wise interactions, $k \leq T$, and some further interactions $k > T$, that are “accidentally” covered by some product-under-test. For example pairwise feature interaction is covered by using T -wise testing with $T = 2$ and additional threewise and fourwise interactions are automatically covered due to the fact that valid configurations are generated. In our BCS case study we covered 82,4% of threewise feature interactions with pairwise testing.

11.2 ADAM OPEL AG - INSTRUMENT PANEL CLUSTER

In the following, we summarize the results of the MoSo-PoLiTe industrial evaluation at the Adam Opel AG. For a detailed description and a complete overview of the results, we refer to [Kar11]. The Industrial Panel Cluster (IPC) served as a case study. IPC is a control panel placed in front of the driver controlling the operation of the vehicle and visualizing vehicle information.

11.2.1 Feature Model Extraction

At Opel and its parent company General Motors (GM), so-called code-rules are used for variant management. For this purpose, the **Vehicle Description Summary (VDS)**, an Engineering document, identifies models, option families, and option availability for General Motors' products. Furthermore, it identifies the models and option codes available within each year and product line. Models and option codes represent the functionalities and properties of a car. A model typically includes information about marketing division, vehicle line, series, and body style, whereas options focus on equipment, parts or information used for customer orders and assembly build. Each option code is assigned an option type of **Regular Production Option (RPO)**, which is a three-position code describing equipment, part and additional assembly information. Thus, these codes control the "feature selection" for a certain configuration.

Furthermore, XML configurations are used to configure ECUs for different vehicle variants and the corresponding configuration parameters are included in XML files. Actually, an XML configuration file contains all parameters that shall be configured for an ECU in a specific carline and model year.

IPC functionality is implemented only by the IPC ECU, which means that, in order to be tested in bench, only the XML file of IPC ECU needs to be flashed. The XML file of IPC for a specific Carline and Model Year is used, in order to extract the list of RPOs. The feature model of IPC was built directly on the basis of the corresponding RPOs, which provide different configurations for it. Furthermore, the VDS was used to extract further *require* and *exclude* constraints. The fact that only the IPC ECU needs to be configured for various configurations of this subsystem allows us to collect the corresponding RPOs straight from its XML file and build the feature model.

The configurations computed by MoSo-PoLiTe can then be directly tested in bench or in Rest Bus Simulation. The IPC feature model consists of 43 features allowing to configure 19.680 different configurations. MoSo-PoLiTe computed 22 configurations to be tested achieving pairwise interaction coverage. In order to execute testing in a reasonable period of time, we selected only a restricted number of RPOs to build a smaller feature model for the IPC. Figure 68 depicts this IPC feature model.

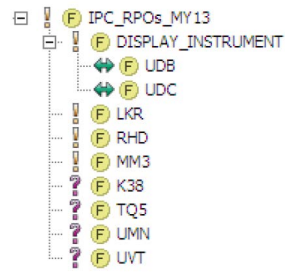


Figure 68: Feature model of the Opel SPL case study

Test Cases	RPO	P1	P2	P3	P4	P5	P6
TC1	& RHD& UMN	+	+	+	-	-	-
TC2	& RHD& UMN	-	-	-	+	+	+
TC3	LKR	+	+	+	+	+	+
TC3	MM3	+	+	+	+	+	+
TC3	TQ5	+	+	-	-	+	+
TC3	UVT	-	+	-	-	+	-
TC4	UDC	+	+	-	+	+	-
TC5	RHD	+	+	+	+	+	+
TC6	& LKR& MM3	+	+	+	+	+	+
TC7	K38	-	+	-	+	-	-
TC8	(By default)	+	+	+	+	+	+
TC9	(By default)	+	+	+	+	+	+
TC10	(By default)	+	+	+	+	+	+

Table 16: BCS statechart coverage under pairwise

11.2.2 Results of the IPC Case Study

Using the IPC feature model depicted in Figure 68, 16 valid configurations could be derived and MoSo-PoLiTe computed six configurations for testing purposes. We then used the so-called Rest Bus Simulation in order to test the MoSo-PoLiTe configurations. Exhaustive testing of an ECU during its development is often not possible because other ECUs that interact with the ECU under test are not available. In Rest Bus Simulation those missing ECUs are simulated allowing to program and test the interaction between different ECUs.

The six configurations were tested in the Rest Bus Simulation by using a repository of test cases that are generally used to test all possible IPCs. Test results obtained by test execution are given in Table 16. The symbols + and - imply the presence and absence of specific RPOs correspondingly, except for “By default” test cases, which should be +, independent from the specific configuration.

We have noticed that all test cases available at Opel for the RPOs included in our feature model can be executed using the MoSo-PoLiTe configurations. This results in 100% coverage of the current test requirements and further allows the expectation that at least faults already found by current testing at Adam Opel AG would have also been found by our MoSo-PoLiTe configurations. In addition, each test case is executed in at least two MoSo-PoLiTe configurations, meaning that an RPO can be tested in at least two different product configurations. Consequently, not only its absence or presence can be tested, but also its interaction with several different groups of RPOs, which is actually the most usual cause of faults in overall system operation. Applying MoSo-PoLiTe implies time-saving, since it does not require testing of each possible configuration, in order to execute all test cases.

11.3 DANFOSS - AUTOMATION DRIVE

The second industrial evaluation was done in cooperation with Danfoss Power Electronics. Danfoss Power Electronics is one of the leading producers of frequency converters in the world and has already introduced SPL principles successfully to handle the increasing number of variants [JDB07]. Frequency converters are electronic power conversion devices able to control shaft speed or torque of a three-phase induction engine to satisfy the needs of a given application scenario. Danfoss gave us the opportunity to use some of their subsystems to evaluate MoSo-PoLiTe in the industrial context. In the following, we provide a short introduction of one of these industrial case studies and then summarize the results of the evaluation. For a detailed description we refer to [Ste11]. Our case study covers parts of the so-called **Automation Drive** consisting of the following components:

- **Application** features represent optional customer requirements such as optional features and hardware options.
- **Auxiliary Functions** represent the mandatory features of a drive
- **Control Core** functions control the device. For this purpose, various control principles exist.
- **Quality** functionalities aim to ensure the quality of the drive, e.g. by steering tests.

Figure 69 depicts the feature model of the **Automation Drive** including cross-tree constraints. To obtain valuable results, a subset of this case study was chosen to compare the test results of pairwise testing with n-wise (product-by-product) testing. For this purpose the options that are part of the **Application** features were chosen. Those can be used to evaluate MoSo-PoLiTe with regard to hardware as well as software configurations.

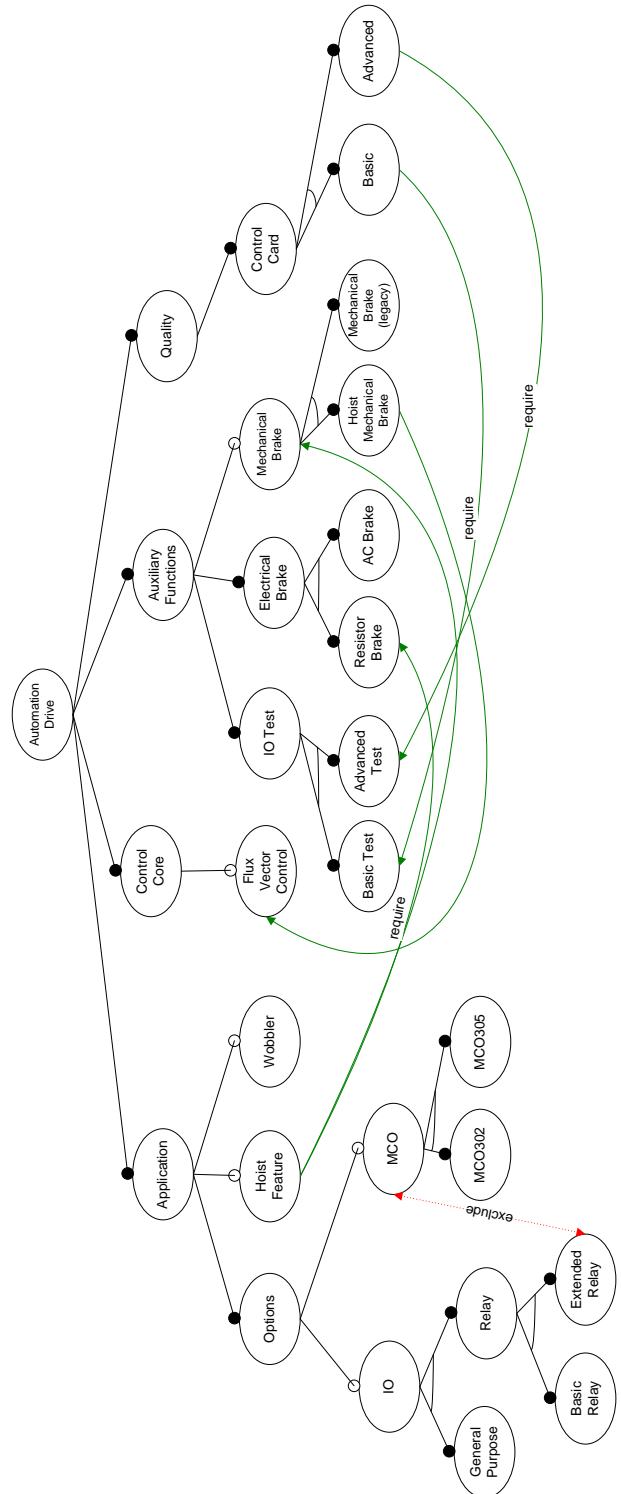


Figure 69: Feature model of the Danfoss use case

11.3.1 Options and their Configuration

Options belong to the heterogeneous hardware/software part of the Danfoss drive SPL. Options are customer specific features that can be changed by the customer by changing plugin modules. The **Automation Drive** consists of up to 4 option slots (named A, B, C₀, C₁), where **Communication Options** and **I/O Options** can be inserted in different combinations. We focused on the slots A, B and C₁. In C₁ an additional adapter is installed (B-in-C-adapter) so that C₁ splits up into two slots: E₀ and E₁. Figure 70 depicts the feature model of the Options including *exclude* constraints representing invalid combinations of options. It is not permitted to install two identical Options in a single drive at the same time. Communication options are only allowed to be installed in slot A and I/O options may be installed in B, E₀ or E₁.

Based on this example, we apply two different evaluation strategies for MoSo-PoLiTe:

- A user of the Automation Drive could accidentally install invalid combinations of options. Thus, we have to ensure that the user obtains the expected error messages or warnings when doing so. We call this test **hardware focused** and delete all *exclude* constraints within the Options feature model allowing us to configure invalid combinations of Options. We then apply MoSo-PoLiTe to this feature model and compare whether the resulting configurations cover all invalid combinations of Options.
- The second category is **software focused**, where we check whether the valid combinations of Options behave as specified. Thus, we again compare the faults found by testing every possible combination of Options with the configurations obtained by applying MoSo-PoLiTe.

11.3.2 Results of the Automation Drive Case Study

With regard to **hardware focused** testing, 432 different combinations of options were tested and two different types of faults occurred.

- Black screen: Instead of a warning for identical options, a black screen is shown.
- No warning for unsupported modules: Instead of a warning for modules that should not be supported in a slot, the modules are detected as expected.

13.89% of configurations contain the fault black screen and 19.44% the fault no warning for unsupported modules. 41.67% of the faults are black screen faults and 58.33% are no warning for unsupported modules faults. MoSo-PoLiTe calculates 57 configurations and both fault types were detected. This means, that according to this test scenario, sufficient coverage is ensured.

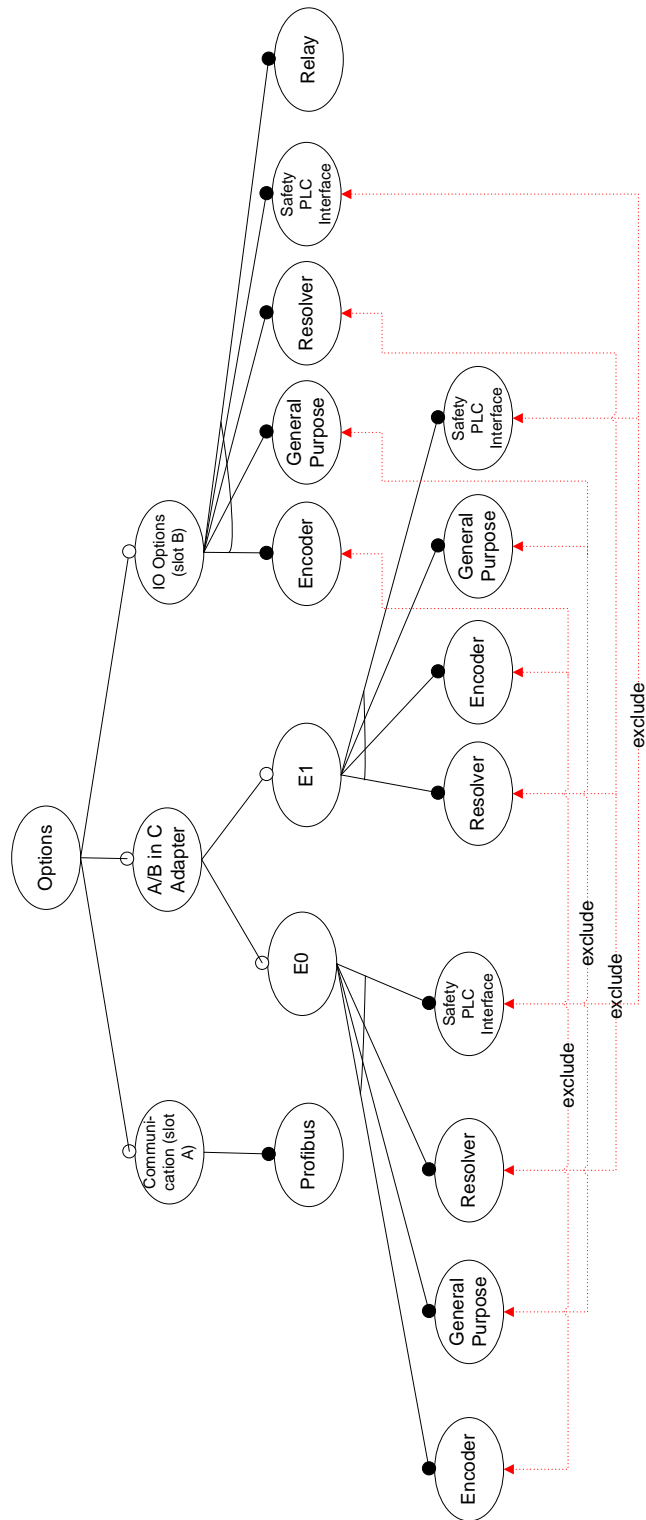


Figure 70: Feature model describing the Options-FM

Test Scenario	Coverage in %	Testing effort referred to all combinations in %
Hardware Testing	100	13.19
Mutated Resolver module	97.48	20.10
Mutated GPIO module	98.53	20.10
Mutated Resolver and covered GPIO module (small scenario)	100	25
Updated adapter software	100	20.10

Table 17: Correlation between test coverage and testing effort by using MoSo- PoLiTe

The **software focused** test provides more significant results by comparing product-by-product testing with pairwise testing. Here, we applied mutation techniques to seed faults within the software that determines the functionality of the options. We then compared how many faults were found by testing the MoSo-PoLiTe configurations with how many faults would be found by testing each possible configuration individually. MoSo-PoLiTe computed 40 configurations to be tested out of the 199 possible configurations. Again, we refer to [Ste11] for a detailed overview and discussion of this evaluation, whereas we only summarize the final results of this experiments. Table 17 shows a summary of the different experiments including the aforementioned hardware focused testing. The term coverage does not refer to the ordinary coverage criteria term. Here, coverage describes the percentage of faults seeded in the software that is found by testing the MoSo-PoLiTe configurations. The faults that MoSo-PoLiTe was not able to cover within the Mutated Resolver Test and the Mutated GPIO test are based on threewise interaction. We discussed that issue already with regard to the BCS case study.

11.4 THREATS TO VALIDITY AND COMPARISON WITH RELATED APPROACHES

We already discussed the potentials and limitations of the pairwise test approach for SPLs in the beginning of this chapter with regard to the BCS case study. Furthermore, we have to consider the following threats to validity with regard to our evaluation at Opel and Danfoss.

- Regarding the industrial case studies, the feature models are rather small. Actually, our experiments would need to be executed on feature models including more features. An ideal candidate would be the feature models for the linux kernel (6000 features) and ECOS (1000 features). However, for our evaluation we require a feature model and a test model or a test repository containing tests for the entire SPL. These artifacts were not available, neither

for the linux kernel nor for ECOS. Actually, an SPL with a complete test suite can hardly be found in the real world. The reason for this situation are:

1. very few “real” SPLs including some kind of variability representations such as feature models exist and if they exist it is nearly impossible to obtain details due to nondisclosure issues.
 2. to find an SPL with a consistent and complete SPL test suite is even harder since the product-by-product test methodology dominates and in many cases there are no systematic test approaches.
- Another threat to validity is the fact that the tests used within our industrial case studies at Danfoss and Opel are not based on certain coverage criteria. Thus, we are not able to interrelate a certain coverage criteria for the SPL with our feature (interaction)-based coverage criteria.

After discussing the threats to validity, we continue with comparing our approach with related work.

In [Olio8] the author uses the pairwise algorithm *Jenny* to generate a set of products realizing pairwise feature interaction. A Banking SPL is used to demonstrate her approach. Using the same example with the MoSo-PoLiTe algorithm, the result of our product set was smaller. Figure 71 shows the 13 products generated by Jenny and the 12 products generated by our Subset Generator. Unfortunately, Jenny does not provide a methodology to compute feature models directly and Olimpiew does not describe how this issue is solved within her thesis. Furthermore, we could not find any indication that Jenny is capable of processing constraints between features.

	ATM Kernel	Language	Expired card action	Call police action	Phone branch action	Alarm action	Pin format [3..10]	Pin attempts [1..5]	Greeting
T1	TRUE	English	Confiscate action	TRUE	FALSE	FALSE	3	1	Enhanced
T2	TRUE	French	Eject action	FALSE	TRUE	TRUE	4	3	Standard
T3	TRUE	Spanish	Eject action	FALSE	FALSE	TRUE	10	5	Enhanced
T4	TRUE	French	Confiscate action	TRUE	TRUE	FALSE	10	5	Standard
T5	TRUE	Spanish	Eject action	FALSE	TRUE	TRUE	3	1	Standard
T6	TRUE	Spanish	Confiscate action	TRUE	FALSE	FALSE	4	3	Enhanced
T7	TRUE	English	Confiscate action	FALSE	FALSE	FALSE	4	5	Standard
T8	TRUE	French	Confiscate action	TRUE	TRUE	TRUE	3	5	Enhanced
T9	TRUE	Spanish	Eject action	TRUE	TRUE	FALSE	3	1	Standard
T10	TRUE	English	Confiscate action	TRUE	FALSE	FALSE	10	3	Standard
T11	TRUE	French	Confiscate action	FALSE	FALSE	FALSE	10	1	Standard
T12	TRUE	French	Eject action	FALSE	TRUE	TRUE	4	1	Enhanced
T13	TRUE	English	Confiscate action	FALSE	TRUE	TRUE	3	3	Enhanced
P1	TRUE	French	Confiscate action	FALSE	FALSE	TRUE	3	3	Standard
P2	TRUE	French	Eject action	FALSE	FALSE	FALSE	10	1	Enhanced
P3	TRUE	French	Confiscate action	TRUE	TRUE	FALSE	4	5	Standard
P4	TRUE	Spanish	Eject action	FALSE	FALSE	FALSE	10	3	Enhanced
P5	TRUE	English	Eject action	TRUE	TRUE	TRUE	10	1	Enhanced
P6	TRUE	English	Confiscate action	FALSE	FALSE	FALSE	10	5	Standard
P7	TRUE	Spanish	Eject action	FALSE	FALSE	FALSE	3	1	Standard
P8	TRUE	Spanish	Confiscate action	TRUE	TRUE	TRUE	4	1	Enhanced
P9	TRUE	English	Eject action	FALSE	FALSE	FALSE	4	3	Enhanced
P10	TRUE	English	Eject action	TRUE	TRUE	TRUE	3	3	Enhanced
P11	TRUE	Spanish	Eject action	FALSE	TRUE	TRUE	3	5	Enhanced
P12	TRUE	English	Confiscate action	TRUE	FALSE	TRUE	3	3	Standard

Figure 71: Comparison to Jenny

Perrouin et al. provided the most similar approach to our MoSo-PoLiTe concept. However, they only focus on the generation of configurations for testing purposes based on combinatorial testing. Test case generation is out of scope within their contribution. [POS⁺₁₁] is dedicated to an extensive comparison of MoSo-PoLiTe

and the Alloy-based approach introduced by Perrouin et al. In the following, we limit ourselves to present the facts and numbers that result from our comparison. The Alloy-based approach by Perrouin et al. consists of two different approaches realizing a divide and compose technique, namely, Binary Split and Incremental Growth.

	CP	SH	AG	MT	ES
Features	19	35	61	88	287
Possible Products	61	1048576	$3.3 * 10^9$	$1.65 * 10^{13}$	$2.26 * 10^{49}$
MoSo-PoLiTe (ms)	26	33	76	108	2586
BinarySplit (ms)	11812	11457	33954	> 32400000	> 32400000
IncGrowth (ms)	56494	1372094	13847835	> 32400000	> 32400000
MoSo-PoLiTe (number)	8	40	46	92	215
BinarySplit (number)	12-20	92	514	time out	time out
IncGrowth (number)	15-18	28	74	time out	time out

Table 18: Execution Times for pairwise generation on feature models. Key: CP = Cell Phone, SH = Smart Home, MT= Model Transformation, ES= Electronic Shopping

Table 18 summarizes the execution times for some of the feature models of the SPLOT homepage. With regard to the MoSo-PoLiTe results, the execution times are calculated by taking the average of 10 execution runs. The deviation between the different results was always less than 10%.

Numbers such as > 32,400,000 mean that the experiments were stopped after running more than 9 hours. MoSo-PoLiTe seems to be at least 1000 times faster than any of the Alloy-based strategies. Furthermore, the MoSo-PoLiTe execution times are growing gently with the feature model complexity, whereas the Alloy-based strategies execution times follows a steeper increasing curve. However, this seems to be the natural result of decomposing the problems in hundreds or thousands of solving steps in Alloy.

Unfortunately, it was not possible to compare the algorithms by executing them on the same test hardware. MoSo-PoLiTe ran on an ordinary desktop computer running Windows XP equipped with a dual core CPU with 2.66GHz and 4GB RAM. For each of the SPLOT feature models, the tests were executed 10 times and the average result is depicted in Table 18. BinarySplit and IncGrow ran on a Mac Book Pro 2.8GHZ Core duo 4GB RAM. For each of the SPLOT feature models, the tests were executed 10 times and the best result is depicted in Table 18.

Please note that at the time this comparison was made, the AndroidSPL was significant smaller than the current version, which was used in the analysis of efficiency in the previous chapter. Finally, in Table 19 we summarize the functionalities and characteristics of both approaches. With regard to efficiency and generation time, MoSo-PoLiTe outranges the Alloy-based approach. However, the Alloy-based

approach supports cardinalities and n-ary constraints. We are currently busy to extend our eclipse plugin to be capable of handling n-ary constraints, too.

	MoSo-PoLiTe	Alloy-based
#Number of Products	+	-
Generation Time	+	-
T-wise support	+	+
Cardinalities	-	+
Binary constraints	+	+
N-ary constraints	-	+

Table 19: Test generation characteristics

Last but not least, we like to compare MoSo-PoLiTe with the model-based testing approaches for SPLs that we have summarized in Chapter 4. In general, all model-based test approaches for SPLs can be applied in combination with our combinatorial subset selection. Table 20 summarizes the differences between MoSo-PoLiTe including our model-based test approach and the related approaches. Two significant differences between MoSo-PoLiTe and the related approaches exist.

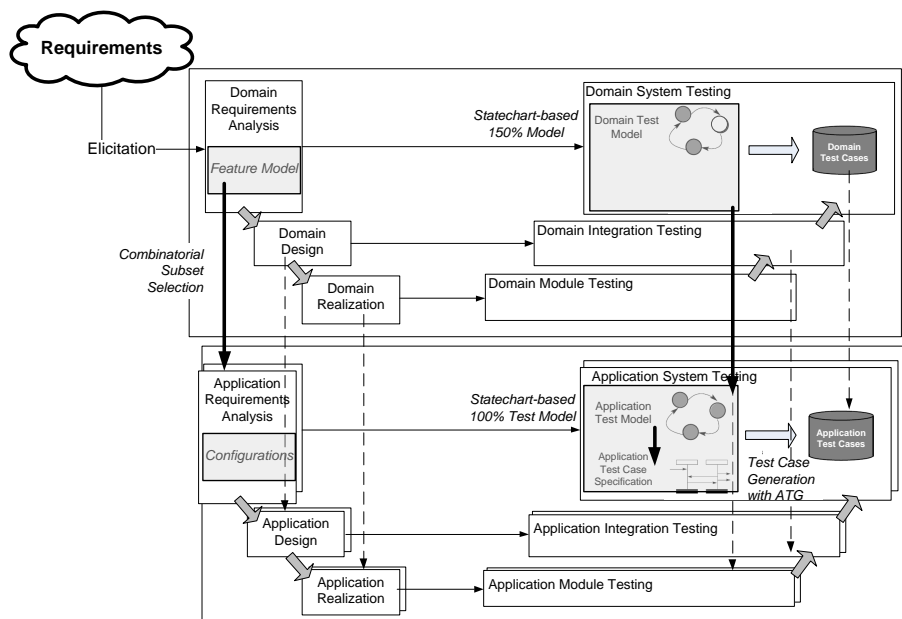


Figure 72: MoSo-PoLiTe process model

First, MoSo-PoLiTe does not only provide a mechanism to generate test cases but also provides a representative subset of configurations according to T-wise feature interaction coverage. Furthermore, due to the fact that MoSo-PoLiTe is completely supported by a tool chain, a full automation for configuration selection and test

case generation is assured. Figure 72 depicts the MoSo-PoLiTe process model in accordance to the conceptional process model introduced in Chapter 4. During domain engineering, MoSo-PoLiTe requires a feature model and a 150% test model. The relation between features and the reusable test model is realized via a mapping strategy based on the pure::variants/Rhapsody integration. During application engineering, configurations are derived using the combinatorial subset selection. For each configuration a corresponding 100% test model can be derived, which can be used to derive configuration specific test cases.

	CADeT	ScenTED	Hartmann	Kishi	Weiglleder	Moso-PoliTe
Input - Test Model	Activity Diagram	Activity Diagram	Activity Diagram	State Machine	State Machine	Feature Model Statechart
Input - Dev. Model	Use cases	Use cases	Activity Diagrams	Design Model	not mentioned	Statechart
Output - TC - Component	Test Steps pre-, post-condition	Test Steps, pre-, and post-cond. input data expected output data	Test Steps	Test Steps input data	Test Steps input data pre- and post-cond.	Subset of Configurations executable Test Cases
Output - Automation	(o)	(-)	(-)	(o)	(o) / (++)	(+)
Output - TC - Coverage	not mentioned	branch coverage with variability	not mentioned	not mentioned	boundary based coverage	various e.g. MCDG
Test Levels	System Test	System Test	System Test	Design Test	System Test	System Test
Traceability	yes	no	no	no	yes	yes
Dev. Process	Domain & Application Eng.	Application Eng.	Application Eng.	Domain & Application Eng.	Application Eng.	Domain & Application Eng.
Dev. Process Integration	(+)	(+)	(-)	(o)	(o)	(+)
Dev. Process - Variability	mandatory optional	mandatory, optional, or, alternative	optional	mandatory, optional, or, alternative	mandatory, optional, or, alternative	mandatory, optional, or, alternative
Reuse of artifacts	yes	yes	yes	yes	yes	yes
Application - Instructions	(+)	(o)	(o)	(o)	(o)	(+)

Table 20: Comparison of model-based test approaches for SPL [OWES11] including MoSo-PoliTe

SUMMARY PART IV

IN this part of this thesis we have described the implementation and the evaluation of MoSo-PoLiTe. MoSo-PoLiTe is implemented by means of an eclipse plugin and integrated with pure::variants for the feature modeling and Rhapsody for model-based testing purposes. Beside the tool chain implementation, we have implemented the flattening algorithm to convert the feature model into a binary CSP using graph transformations by means of SDM. To test whether our implementation aligns with MoSo-PoLiTe concepts, we have implemented a test framework to:

- test whether the flattening implementation preserves the semantical equivalence of the feature model.
- check whether all pairs or T-wise combinations of features are included within our set of configurations.
- check whether each configuration in our set is a valid configuration.

For this purpose, we have used some feature models listed on the SPLOT website [SR11] and feature models generated by our own feature model generator. To evaluate the entire MoSo-PoLiTe concept, we use the BCS case study. For evaluating the feasibility of the testing approach and the quality of the tests generated, we applied several *mutation operators* to the products-under-test implementation code. In the majority of cases, in which mutation detection fails, the test cases generated by ATG were not sufficient to cover them, whereas only a small number of mutations were missed because they arise in particular, uncovered feature combinations.

Furthermore, we applied the combinatorial testing component of MoSo-PoLiTe to two additional industrial SPLs. In the automotive case study, the MoSo-PoLiTe subset of configurations was capable of executing all test cases available for the SPL with the exact same results. Thus, we assume that testing only the MoSo-PoLiTe configurations would have been sufficient instead of testing all possible configurations. In the automation case study, MoSo-PoLiTe covered around 95% of the mutations. The other mutations could not be found since they depend on threewise feature interaction.

According to the evaluation, the MoSo-PoLiTe tool chain ensures every valid combination of feature pairs to be covered by at least one configuration but for an interaction between more than two features, no general coverage statement can be given.

Part V

CONCLUSIONS

CONCLUSIONS

THE development of effective and efficient new SPL testing approaches is of vital importance in many engineering domains due to the rapidly increasing complexity of SPLs and, especially, embedded system SPLs. Given the enormous number of product instances that can be created from even a modest SPL, it is important to reduce the testing space while preserving the bug-finding power of testing. In this thesis, we introduce MoSo-PoLiTe, which combines feature model-based testing, combinatorial testing, and model-based testing, to reduce the effort for testing SPLs. Figure 73 depicts the MoSo-PoLiTe concept.

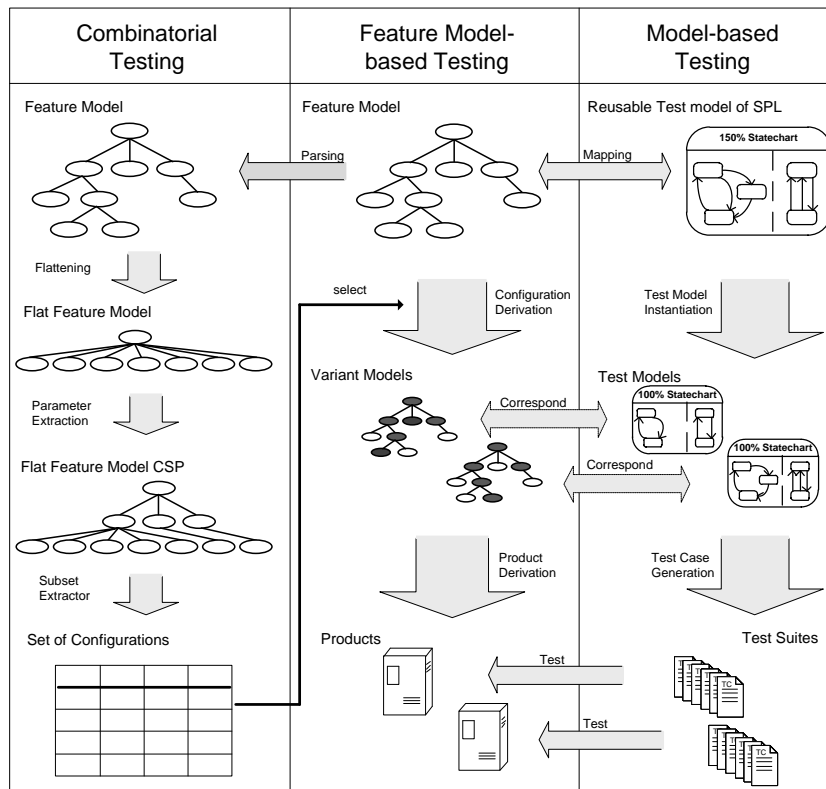


Figure 73: Comparison - MoSo-PoLiTe and related model-based SPL testing approaches

The central component in MoSo-PoLiTe is the feature model. It is created on the basis of the SPL requirements during domain engineering. The feature model provides a hierarchical structure of the SPL requirements and represents the common and variable parts of an SPL. The selection of features according to the dependen-

cies and constraints results in configurations that can be interpreted as subtrees of the original feature model. Typically, features of a feature model are additionally linked to development artifacts such as code fragments or behavioral models. Thus, the selection of a configuration can lead to a product.

In MoSo-PoLiTe, the configuration selection is performed by a specific selection algorithm based on combinatorial testing (cf. Figure 73 left-hand side). First, a so-called flattening algorithm removes the hierarchy within a feature model. Then, a so-called value extraction transforms the feature model into a binary Constraint Satisfaction Problem (CSP) consisting of parameters, values, and constraints. Our subset extraction algorithm then generates a minimal set of configurations containing all T-wise feature interactions e.g. pairwise feature interaction on the basis of this CSP. Therefore, testing this subset of products is equivalent to testing the entire SPL on the basis of T-wise feature interaction.

For a model-based test case derivation, the feature model is additionally mapped to a test model representing the behavior of the entire SPL. A mapping approach ensures that if a valid configuration is derived from the feature model, a corresponding test model representing the behavior of this configuration is generated. Thus, for every configuration of our combinatorial subset, a corresponding test model can be generated. This test model is then capable of generating test cases for each configuration of the representative subset.

The MoSo-PoLiTe tool chain is based on the variant management tool `pure::variants` [pG11] and the CASE tool Rhapsody [IBM11]. Our pairwise subset selection heuristics is implemented on top of the feature model editor of `pure::variants` by means of an eclipse plugin, and the mapping capabilities are applied for tracing to corresponding statechart fragments built in Rhapsody. The Rhapsody ATG component provides automated model-based test case generation.

Hence, MoSo-PoLiTe combines feature model-based testing, combinatorial testing, and model-based testing.

We applied our MoSo-PoLiTe tool chain to the BCS-SPL which served as a running example within this thesis. Additionally, we applied the pairwise SPL testing component to two industrial SPLs. There, the test case generation was out-of-scope. The results suggest that, with our approach, higher coverage of feature interactions is achieved at a fraction of cost, when compared with a baseline approach of testing all feature interactions.

MoSo-PoLiTe has the following advantages compared to the state-of-the-art SPL testing approaches as introduced in Chapter 4:

- reuse of test artifacts: The model-based test component within our approach realizes the reuse of test artifacts. The 150% test model includes all artifacts describing the behavior of the entire SPL. Subsets of this test model are reused for each product.

- reuse of test results: On the basis of combinatorial testing, we generate a representative set of configurations for testing purposes. We intend to reuse the test results of this set of configurations for all other products of the SPL.

13.1 DISCUSSION

In the motivation section (cf. Section 1.1), we introduced several research questions which were discussed throughout this thesis in each part's summary. Here, we again summarize and discuss all research questions, to recapitulate the contribution of this thesis.

1. Can we test an entire SPL without testing each possible product? Yes, to a certain extent. Testing is always a heuristics and thus there is no guarantee of covering or finding all possible faults within a system or SPL. Pairwise subset selection testing is, by its nature, limited to covering pairwise interactions among independent features. For instance, if three independent features f_1 , f_2 , and f_3 are responsible for a faulty interaction, and this fault, for some reason does not appear when testing any pair of this triple, then it is possible that this triple is not necessarily within our subset of products. However, these situations may also arise in methods beyond pairwise, i.e., T-wise, since there might always be an $T + 1$ -ary set of (independent) features, resulting in a fault in that exact combination. To overcome such situations, T needs to be increased properly to cover all interactions. In the worst case, this ends up at N-wise, thus testing every possible product of an SPL, as already stated in the introduction, is not feasible.
2. How can we apply lessons learned from the software testing community to decrease the test effort for SPLs? We have chosen combinatorial testing for adoption in SPL testing, due to its positive impact in single system testing [SM98]. To apply combinatorial testing to feature models, we transformed the feature model into a binary CSP. This is then solved using a combination of combinatorial testing and constraint solving algorithm, including Forward Checking and Backtracking.
3. How should we systematically select a subset of possible products for testing purposes with regard to feature interaction? For this purpose, we selected T-wise testing to cover T-ary feature interaction. Our algorithm generates a set of configurations fulfilling this T-ary feature interaction coverage.
4. What is the effect of testing T-wise feature interaction in the SPL context? As already discussed with regard to RQ1, T-wise and especially pairwise testing is capable of dramatically decreasing the testing effort for SPLs. However, with regard to feature interaction between more than two features, this approach cannot guarantee any degree of fault coverage. Thus, the test engineer has to

decide which T is appropriate for testing the SPL. Furthermore, we strongly depend on how complex it is to identify a certain interaction in the test model. The probability of covering a certain interaction is proportional to how obvious this interaction is in the test model. In addition, the adequacy of the approach for revealing unintended interactions further depends on the actual product test strategy and corresponding coverage criteria applied.

5. How can we reuse test artifacts to test the product of an SPL? Using combinatorial testing, we reused test results by selecting a subset of configurations and implicitly relied on the probability that other configurations would end up with the same test results. To reuse test artifacts, we found model-based testing to be the most suitable methodology [OWES11]. We followed the concept of implementing a 150% test model and have developed an approach to automatically generate test cases for each derivable configuration.
6. What is necessary to support industry with a suitable tool chain with regard to the RQs presented above? To support the industry sector, we have implemented our tool chain by using commercial tools. Both pure::variants and Rhapsody are frequently used in the automation and automotive sector. Since our algorithm is implemented in Java and realized as an eclipse plugin, its range of application can be quite extensive. Furthermore, our algorithm could be ported to many other programming languages or be integrated with many other tools. With regard to our model-based component in MoSo-PoLiTe, this technique can be replaced by any other test case generation process.

Furthermore, there are multiple benefits of our approach:

- it is lightweight, since it does not require any intervention by programmers,
- the combinatorial testing simply works on the basis of the feature model, which is a quasi standard within SPL engineering,
- it is tractable since it uses combinatorial testing to shrink the number of configurations for testing purposes thus ensuring a certain T -wise e.g. pairwise feature interaction coverage, and
- it is scalable, since it can be used on feature models with hundreds of optional features.

Summarizing, our combination of feature model-based testing, combinatorial testing, and model-based testing paves the way for exhaustively testing the entire SPL. But the model-based testing component within MoSo-PoLiTe can be simply replaced by other testing techniques, as shown in the evaluation. Thus, MoSo-PoLiTe is not only limited for use in a model-based testing environment.

In the next section, we will summarize and discuss new research questions and plans for future work.

13.2 FUTURE WORK

T-wise testing techniques for SPLs can provide a 100% coverage if T is equal to the number of all features within the feature model. However, this would result in testing every possible product of the SPL under test, which is not feasible. Decreasing the T results in a corresponding reduction of the size of our test set of configurations. With regard to SPLs with more than 300 features our algorithm will probably compute more than 50 products to test every pairwise feature interaction. Increasing the T has the consequence that the number of products exceeds the number of products that are testable within a reasonable time. Thus, in our future work, we will adapt our algorithm to combine specified sets of features in an arbitrary K-wise manner, where K is the number of features involved in interactions. In [RSM⁺10], the authors have shown that better coverage can be achieved by selecting input data objects that contain diverse values for configuration variables. This procedure requires a feature interaction analysis of the SPL under test, which is the scope of our future work and initially discussed in [LG10].

The idea for realizing a so-called *K-wise* feature interaction approach seems to be promising. However, even if we are able to deduce *K-wise* interactions from the test model and conduct an appropriate subset selection, several uncertainties remain:

- is every feature interaction captured in the test model?
- is every feature interaction reliably detectable in the test model?
- how can one keep the testing effort for increasing K manageable?

In addition, we are currently working on concepts to support versioning of the feature model and the test models within our approach. Urgent questions in this field are: How does the evolution of the feature model affect the results of MoSo-PoLiTe. How can these changes be managed and maintained?

With regard to our tool chain, we plan to use other tools for the model-based testing component, for example ParTeG, to provide an open source solution, and Matlab Simulink, which is one of the most prominent tools in the automation and automotive sector.

Furthermore, we are currently examining a new model-based coverage-criteria-driven approach for SPL testing. A set of configurations is *representative* for all configurations of an SPL with regard to a chosen coverage criterion if, in a set of test cases, that is necessary to satisfy the chosen coverage criterion on every single product, each test case is at least executable on one of these products in this representative set. A feature model is used to configure a 150% test model representing the behavior of the entire SPL. In contrast to MoSo-PoLiTe, this 150% test model is then used to derive a set of valid test cases that guarantees a complete test model coverage for all possible configurations of the SPL. These test cases are then used to find a minimal representative set of configurations required to execute

each individual test case. In our future work, we will implement a tool chain for this approach and combine it with MoSo-PoLiTe.

Additionally, we are currently examining whether our approaches for generating a representative set of configurations could be used in combination with regression testing techniques [ERS10]. Specifically, we intend to find out if (1) regression testing could be used to incrementally test the various configurations within our representative subset and (2) if our representative set can be used as an initial starting point for regression testing, to test other products of the SPL.

Finally, we will continue optimizing the SDM-based flattening algorithm as well as the manual implementation. We think that our flattening transformations provide a good and representative case study to compare generated code with a manual implementation with regard to performance, complexity, and clarity.

BIBLIOGRAPHY

- [ABB⁺02] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Peach, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [ACo4] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature Modeling Plug-In for Eclipse. In *The 2004 OOPSLA Workshop on Eclipse Technology eXchange - Eclipse '04*, pages 67 – 72, Vancouver, British Columbia, Canada, 2004. ACM Press.
- [AG11a] Adam Opel AG. www.opel.de, 2011.
- [AG11b] BTC Embedded Systems AG. Automatic Test Generation (ATG). website, visited last August 2011.
- [aG11c] aquintos GmbH, last visited August 2011.
- [AKRS03] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. Adapting FUJABA for Building a Meta Modelling Framework. In H. Giese and A. Zündorf, editors, *Proceedings of the 1st International Fujaba Days*, volume tr-ri-04-247, pages 29–34, 10 2003.
- [AKRS06] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, 2006.
- [ALPS11] A. Anjorin, M. Lauder, S. Patzina, and A. Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In *3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011)*, Lecture Notes in Informatics, Bonn, 2011. Gesellschaft für Informatik. accepted for publication.
- [ANS83] ANSI/IEEE Standard 729-1983, New York. *IEEE Standard Glossary of Software Engineering Terminology*, 1983.
- [Bas97] Paul G. Bassett. *Framing Software Reuse: Lessons from the Real World*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
- [Bei90] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

- [Beno4] H. Bennaceur. A Comparison between SAT and CSP Techniques. *Constraints*, 9(2):123–138, 2004.
- [Ber91] G. Bernot. Testing against Formal Specifications: a Theoretical View. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD)*, volume 2, pages 99–119, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
- [BFGLo6] A. Bertolino, A. Fantechi, S. Gnesi, and G. Lami. Product Line Use Cases: Scenario-Based Specification and Testing of Requirements. In T. Käkölä and J. C. Ducnas, editors, *Software Product Lines: Research Issues in Engineering and Requirements*, pages 425–445. Springer Verlag, 2006.
- [BG95] F. Bacchus and A. J. Grove. On the Forward Checking Algorithm. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 292–308, London, UK, 1995. Springer-Verlag.
- [BG03] A. Bertolino and S. Gnesi. Use Case-based Testing of Product Lines. *SIGSOFT Software Engineering Notes*, 28(5):355–358, 2003.
- [BH08] F. Belli and A. Hollmann. Test Generation and Minimization with "Basic" Statecharts. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 718–723, New York, NY, USA, 2008. ACM Press.
- [BHS99] K. Bogdanov, M. Holcombe, and H. Singh. Automated Test Set Generation for Statecharts. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*, pages 107–121, London, UK, 1999. Springer-Verlag.
- [Bin99] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [Bos00] J. Bosch. *Design and Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley Longman, Amsterdam, 2000.
- [Bos01] J. Bosch. Software Product Lines: Organizational Alternatives. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE '01*, pages 91–100, Washington, DC, USA, 2001. IEEE Computer Society.

- [Bos05] J. Bosch. Software Product Families in Nokia. In J. Obbink and K. Pohl, editors, *Proceedings of the 9th International Conference on Software Product Line Engineering (SPLC2009)*, pages 2–6, 2005.
- [Bur93] N. Burkhard. Reuse-Driven Software Processes. Version 02.00.03 SPC-92019-CMC, Software Productivity Consortium, November 1993.
- [CCL03] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Augmenting Simulated Annealing to Build Interaction Test Suites. In *Proceedings of the 14th International Symposium on Software Reliability Engineering, IS-SRE '03*, pages 394–406, Washington, DC, USA, 2003. IEEE Computer Society.
- [CCT04] C. J. Colbourn, M. B. Cohen, and R. Turban. A Deterministic Density Algorithm for Pairwise Interaction Coverage. In *Proceedings of the 26th International Conference on Software Engineering*, pages 345–352, 2004.
- [CDKP94] D. M. Cohen, S. R. Dalal, A. Kajla, and G. C. Patton. The Automatic Efficient Tests Generator. *Fifth IEEE International Symposium on Software Reliability Engineering*, IEEE:303–309, 1994.
- [CDS06] M.B. Cohen, M.B. Dwyer, and J. Shi. Coverage and Adequacy in Software Product Line Testing. In *Proceedings of the ISSTA 2006 workshop ROSATEA '06*, pages 53–63, New York, NY, USA, 2006. ACM.
- [CDS07] M.B. Cohen, M.B. Dwyer, and J. Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, pages 129–139, 2007.
- [CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.
- [CGDA99] E. M. Clarke, O. Grumberg, and Peled D. A. *Model Checking*. MIT Press, 1999.
- [CGMC03] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing Test Suites for Interaction Testing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 38–48, 2003.
- [CHE05a] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. In *Software Process: Improvement and Practice*, pages 7–29, 2005.

- [CHE05b] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.
- [CKMRM03] M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. *Computer Networks*, 41(1):115–141, 2003.
- [CL05] X. Cai and M.R. Lyu. The Effect of Code Coverage on Fault Detection Under Different Testing Profiles. In *Proceedings of the ICSE 2005 Workshop on Advances in Model-Based Software Testing, A-MOST*, pages 8–15, St. Louis, Missouri, USA, 2005.
- [Cle99] P. Clements. Software Product Lines: A New Paradigm for the New Century. In *Crosstalk: The Journal of Defense Software Engineering*, pages 21–23, 1999.
- [CLW96] M.-H. Chen, M.R. Lyu, and W.E. Wong. An Empirical Study of the Correlation between Code Coverage and Reliability Estimation. In *Proceedings of the 3rd IEEE International Software Metric Symposium*, pages 133–141, 1996.
- [CN01] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001.
- [CSW08] K. Czarnecki, S. She, and A. Wasowski. Sample Spaces and Feature Models: There and Back Again. In *Proceedings of the 12th International Software Product Line Conference*, pages 22–31, Washington, DC, USA, 2008. IEEE Computer Society.
- [CW07] K. Czarnecki and A. Wasowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the 11th International Conference on Software Product Lines (SPLC2007)*, pages 23–34. IEEE Computer Society, 2007.
- [Cze06] J. Czerwonka. Pairwise Testing in Real World. In *Proceedings of 24th Pacific Northwest Software Quality Conference*, pages 419–430, 2006.
- [Dan11] Danfoss A/S, Devision Power Electronics. <http://www.danfoss.com/>, 2011.
- [DES⁺97] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Ianino. Applying Design of Experiments to Software Testing: Experience Report. In *Proceedings of the 19th international conference on Software engineering, ICSE '97*, pages 205–215, New York, NY, USA, 1997. ACM.

- [DGP⁺04] M. Deubler, J. Grunbauer, G. Popp, G. Wimmel, and C. Salzmann. Tool Supported Development of Service-Based Systems. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference, APSEC '04*, pages 99–108, Washington, DC, USA, 2004. IEEE Computer Society.
- [DGR11] D. Dhungana, P. Grünbacher, and R. Rabiser. The DOPLER Meta-Tool for Decision-oriented Variability Modeling: a Multiple Case Study. *Automated Software Engineering*, 18:77–114, March 2011.
- [dIB97] Rat der IT-Beauftragten. V-Modell, 1997.
- [Dij70] E. W. Dijkstra. Notes on Structured Programming. Circulated privately, April 1970.
- [DWoo] W. Dröschel and M. Wiemers. *Das V-Modell 97*. Wien, 2000.
- [DW09] C. Dziobek and J. Weiland. Variantenmodellierung und -konfiguration eingebetteter automotive Software mit Simulink. In *MBEES*, pages 36–45, 2009.
- [ER10] E. Engström and P. Runeson. A Qualitative Survey of Regression Testing Practices. In Muhammad Ali Babar, Matias Vierimaa, and Markku Oivo, editors, *Proceedings of the 11th International Conference of Product-Focused Software Process Improvement, PROFES 2010, Limerick, Ireland, June 21-23*, Lecture Notes in Business Information Processing, pages 3–16. Springer, 2010.
- [ER11] E. Engström and P. Runeson. Software Product Line Testing - A Systematic Mapping Study. *Information and Software Technology*, 50:1098–1113, January 2011.
- [ERS10] E. Engström, P. Runeson, and M. Skoglund. A Systematic Review on Regression Test Selection Techniques. *Information & Software Technology*, 52(1):14–30, 2010.
- [ERW10] E. Engström, P. Runeson, and G. Wikstrand. An Empirical Evaluation of Regression Testing Based on Fix-Cache Recommendations. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9*, pages 75–78. IEEE Computer Society, 2010.
- [ES04] N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

- [ESRo8] E. Engström, M. Skoglund, and P. Runeson. Empirical Evaluations of Regression Test Selection Techniques: a Systematic Review. In H. Dieter Rombach, Sebastian G. Elbaum, and Jürgen Münch, editors, *Proceedings of the Second International Symposium on Empirical Software Engineering and Measurement, ESEM 2008, October 9-10, Kaiserslautern, Germany*, pages 22–31, 2008.
- [fAR82] Radio Technical Commission for Aeronautics (RTCA). DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1982.
- [FHS02] S. Ferber, J. Haag, and J Savolainen. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In *Proceedings of the 2nd International Software Product Line Conference*, pages 235–256, London, UK, 2002. Springer-Verlag.
- [FOS11] FOSD-Community. Feature-Oriented Software Development Research, 2011.
- [FP96] N. E. Fenton and S. L. Pfleeger. *Software Metrics: a Rigorous and Practical Approach*. International Thompson computer, 2nd edition, 1996.
- [GG93] M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. *Software Testing, Verification and Reliability*, 3:63–82, 1993.
- [GKPR08] H. Grönniger, H. Krahn, C. Pinkernell, and B. Rumpe. Modeling Variants of Automotive Systems using Views. In *Tagungsband Modellierungs-Workshop MBEFF: Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, volume Informatik-Bericht 2008-01. CFG-Fakultät, TU Braunschweig, 2008.
- [GL08] A. Gonzalez and C. Luna. Behavior Specification of Product Lines via Feature Models and UML Statecharts with Variabilities. In *Proceedings of the Internatioanl Conference of the Chilean Computer Science Society*, pages 32–41, Washington, DC, USA, 2008. IEEE Computer Society.
- [GLRW04] B. Geppert, J. Jenny Li, F. Rößler, and D. M. Weiss. Towards Generating Acceptance Tests for Product Lines. In *Proceedings of the 8th International Conference on Software Reuse, ICSR 2004, Madrid, Spain, July 5-9*, pages 35–48, 2004.
- [GOA05] M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.

- [Gom04] H. Gomma. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.
- [Gri00] M. L. Griss. Implementing Product-Line Features by Composing Aspects. In *SPLC*, pages 271–288, Norwell, MA, USA, 2000. Kluwer Academic Publishers.
- [Gus07] T. Gustafsson. An Approach for Selecting Software Product Line Instances for Testing. In *International Workshop on Software Product Line Testing (SPLiT)*, 2007.
- [Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
- [Har05] A. Hartman. Software and Hardware Testing Using Combinatorial Covering Suites. In Martin Charles Golumbic and Irith Ben-Arroyo Hartman, editors, *Graph Theory, Combinatorics and Algorithms*, volume 34 of *Operations Research/Computer Science Interfaces Series*, pages 237–266. Springer US, 2005.
- [HE79] R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. In *Proceedings of the 6th International joint Conference on Artificial intelligence*, volume 1, pages 356–364, 1979.
- [HK01] D. Harel and H. Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In *LNCS 3147*, pages 325–354. Springer, 2001.
- [HKC⁺00] H. S. Hong, Y. G. Kim, S. D. Cha, D. H. Bae, and H. Ural. A Test Sequence Selection Method for Statecharts. *Software Testing, Verification and Reliability*, 10(4):203–227, 2000.
- [HST⁺08] P. Heymans, P. Y. Schobbens, J. C. Trigaux, Y. Bontemps, R. Matelevicius, and A. Classen. Evaluating Formal Properties of Feature Diagram Languages. *Software, IET*, 2(3):281–302, 2008.
- [HVR04] J. Hartmann, M. Vieira, and A. Ruder. A UML-based Approach for Validating Product Lines. In B. Geppert, C. Krueger, and J. Li, editors, *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004)*, pages 58–65, 2004.
- [IBM11] IBM. Rational Rhapsody. Website, 2011.
- [IEE90] IEEE - The Institute of Electrical and Eletronics Engineers. IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard, September 1990.

- [IEE98] IEEE. IEEE standard for software test documentation ieee std 829-1998, 1998.
- [JDB07] H. P. Jepsen, J. G. Dall, and D. Beuche. Minimally Invasive Migration to Software Product Lines. In *Proceedings of the 11th International Software Product Line Conference*, pages 203–211, Washington, DC, USA, 2007. IEEE Computer Society.
- [JDD]08] A. L. Juarez-Dominguez, N. A. Day, and J. J. Joyce. Modelling Feature Interactions in the Automotive Domain. In *Proceedings of the International Workshop on Models in Software Engineering, MiSE '08*, pages 45–50, New York, NY, USA, 2008. ACM.
- [JZ98] M. Jackson and P. Zave. Distributed Feature Composition: A Virtual Architecture for Telecommunications Services. *IEEE Transactions on Software Engineering*, 24:831–847, 1998.
- [Kan96] C. Kaner. Software Negligence & Testing Coverage. In *Proceedings of the International Conference on Software Testing Analysis & Review*, Jacksonville, FL, USA, 1996.
- [Kar11] K. Karytsioti. Industrial Evaluation of Combinatorial Testing in Software Product Lines. Master thesis, Technische Universität Darmstadt Department of Electrical Engineering and Information Technology Real-Time Systems Lab, 2011. Referee: Prof. Dr. rer. nat. Andy Schürr, Advisor: Dipl.-Inform. Sebastian Oster.
- [KBK11] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proceeding of the 10th International Conference on Aspect-Oriented Software Development*, pages 57–68. ACM, 2011.
- [KBP01] C. Kaner, J. Bach, and B. Pettichord. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2001.
- [KCH⁺90] 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, 1990.
- [Kim03] Y. W. Kim. Efficient use of Code Coverage in Large-Scale Software Development. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research*, pages 145–155. IBM Press, 2003.
- [KLK08] R. Kuhn, Y. Lei, and R. Kacker. Practical Combinatorial Testing: Beyond Pairwise. *IT Professional*, 10(3):19–23, 2008.

- [KN04] T. Kishi and N. Noda. Design Testing for Product Line Development based on Test Scenarios. In B. Geppert, C. Krueger, and J. Li, editors, *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004)*, pages 19–26, 2004.
- [Knu98] D. E. Knuth. *The Art of Computer Programming (2nd ed.)*, volume 4A - Enumeration and Backtracking (chapter 7 part 1). Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
- [Kolo3] R. 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.
- [Kru08] C. W. Krueger. The BigLever Software Gears Unified Software Product Line Engineering Framework. In *Proceedings of the 12th International Software Product Line Conference*, pages 353–353, Washington, DC, USA, 2008. IEEE Computer Society.
- [Lab] Generative Software Development Lab.
<http://gsd.uwaterloo.ca/feature-models-in-the-wild>, last visit august 2011.
- [LAMSo5] T. Leich, S. Apel, L. Marnitz, and G. Saake. Tool Support for Feature-oriented Software Development: FeatureIDE: an Eclipse-based Approach. In *Proceedings of the OOPSLA workshop on Eclipse technology eXchange, eclipse '05*, pages 55–59, New York, NY, USA, 2005. ACM.
- [LG10] M. Lochau and U. Goltz. Feature Interaction Aware Test Case Generation for Embedded Control Systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 264:37–52, 2010.
- [Lig09] P. Liggesmeyer. *Software-Qualität - Testen, Analysieren und Verifizieren von Software (2. Aufl.)*. Spektrum Akademischer Verlag, 2009.
- [LL05] J. Ludewig and H. Lichter. *Software Engineering*. Dpunkt.Verlag GmbH, 2005.
- [LT98] Y. Lei and K. C. Tai. In-Parameter-Order: a Test Generation Strategy for Pairwise Testing. In *IEEE High Assurance Systems Engineering Symposium*, pages 254–261, 1998.
- [Mar99] B. Marick. How to Misuse Code Coverage. In *Proceedings of the 16th International Conference on Testing Computer Software*, pages 16–18, 1999.

- [MBLP05] A. Metzger, S. Bühne, K. Lauenroth, and K. Pohl. Considering Feature Interactions in Product Lines: Towards the Automatic Derivation of Dependencies between Product Variants. In Stephan Reiff-Marganiec and Mark Ryan, editors, *Feature Interactions in Telecommunications and Software Systems VIII*, pages 198–216, Leicester, UK, 2005. IOS Press.
- [McGo1] J. D. McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon, Software Engineering Institute, 2001.
- [McI68] M. D. McIlroy. Mass-Produced Software Components. *Proceeding of the NATO Conference on Software Engineering, Garmisch, Germany, 1968*.
- [Met04] A. Metzger. Feature Interactions in Embedded Control Systems. *Computer Networks*, 45(5), 2004.
- [MLBK02] Y. K. Malaiya, M. N. Li, J. M. Bieman, and R. Karcich. Software Reliability Growth With Test Coverage. *IEEE Transactions on Reliability*, 51:420–426, 2002.
- [MLD⁺09] T. Müller, M. Lochau, S. Detering, F. Saust, H. Garbers, L. Martin, T. Form, and U. Goltz. A comprehensive Description of a Model-based, continuous Development Process for AUTOSAR Systems with integrated Quality Assurance. Technical Report 2009-06, TU Braunschweig, 2009.
- [MMB94] P. C. Masiero, J. C. Maldonado, and I. G. Boaventura. A Reachability Tree for Statecharts and Analysis of some Properties. *Information and Software Technology*, 36(10):615 – 624, 1994.
- [MPH⁺07] A. Metzger, K. Pohl, P. Heymans, P. Schobbens, and G. Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE2007)*, pages 243–253, 2007.
- [MSMo4] J. D. McGregor, P. Sodhani, and S. Madhavapeddi. Testing Variability in a Software Product Line. In *Proceedings of the Software Product Line Testing Workshop (SPLiT)*, pages 45–50, Boston, MA, 2004. Avaya Labs.
- [Mye79] G. J. Myers. *The Art of Software Testing*. Wiley, New York :, 1979.
- [NA09] A. S. Namin and J. H. Andrews. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 57–68, New York, NY, USA, 2009. ACM.

- [NFLTJ04] C. Nebut, F. Fleurey, Y. Le Traon, and J.M. Jézéquel. A Requirement-Based Approach to Test Product Families. In *Proceedings of the International Workshop on Product Family Engineering*, pages 198–210. Springer, 2004.
- [OB88] T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676–686, 1988.
- [Olio8] E. M. Olimpiew. *Model-Based Testing for Software Product Lines*. PhD thesis, George Mason University, 2008.
- [OMR10] S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proceedings of the 14th International Software Product Line Conference*, pages 196–210, 2010.
- [OMS09] S. Oster, F. Markert, and A. Schürr. Integrated Modeling of Software Product Lines with Feature Models and Classification Trees. In *Proceedings of the 13th International Software Product Line Conference (SPLC09)*, MAPLE 2009 Workshop Proceedings, pages 75–82. Springer Verlag, 2009.
- [Ost11] S. Oster. A Semantic Preserving Feature Model to CSP Transformation. Technical Report 11, Technische Universität Braunschweig, 2011.
- [OSW08] S. Oster, A. Schürr, and I. Weisemöller. Towards Software Product Line Testing using Story Driven Modelling. In U. Aßmann, J. Johannes, and A. Zündorf, editors, *Proceedings of the 6th International Fajaba Days*, pages 48–51. Technische Universität Dresden, September 2008.
- [OWES11] 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*, pages 339–381. CRC Press/Taylor&Francis, 2011.
- [PBvdL05] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
- [Pen06] B. Penzenstadler. Evaluation and Testing of Approaches to Support Variants in Model-based Design for Software Product Lines. Master’s thesis, Universität Passau, 2006.
- [Pet11] P. Pettakou. Feature Model-based Development and Testing in the Industrial Context. Master’s thesis, Technische Universität Darmstadt Department of Electrical Engineering and Information Technology

Real-Time Systems Lab, 2011. Referee: Prof. Dr. rer. nat. Andy Schürr, Advisor: Dipl.-Inform. Sebastian Oster.

- [pG11] pure::systems GmbH, last visited Feb. 2011.
- [POC93] P. Piwowarski, M. Ohba, and J. Caruso. Coverage Measurement Experience During Function Test. In *Proceedings of the 15th International Conference on Software Engineering*, pages 287–301, Los Alamitos, CA, USA, 1993. IEEE CS.
- [POS⁺11] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal - Special issue on Quality Engineering for Software Product Lines*, 2011. accepted for publication.
- [PP04] A. Pretschner and J. Philipps. Methodological Issues in Model-Based Testing. In *Model-Based Testing of Reactive Systems*, pages 281–291, 2004.
- [Pre03] A. Pretschner. *Zum modellbasierten funktionalen Test reaktiver Systeme*. PhD thesis, Technische Universität München, 2003.
- [PSK⁺10] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. L. Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *Third International Conference on Software Testing, Verification and Validation*, 2010.
- [RKPR05] A. Reuys, E. Kamsties, K. Pohl, and S. Reis. Model-based System Testing of Software Product Families. In *Proceedings of the 17th International Conferenc on Advanced Information Systems Engineering*, pages 519–534, 2005.
- [Rob00] H. Robinson. Intelligent Test Automation. *Software Testing & Quality Engineering*, September/October:24–32, 2000.
- [RSM⁺10] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 445–454, 2010.
- [Scho7] K. Scheidemann. *Verifying Families of System Configurations*. PhD thesis, Technical University of Munich, 2007.
- [SDD⁺09] C. Salinesi, D. Diaz, O. Djebbi, R. Mazo, and C. Rolland. Exploiting the Versatility of Constraint Programming over Finite Domains to Integrate Product Line Models. In *Proceedings of the 17th IEEE International Requirements Engineering Conference, RE*, pages 375–376, Washington, DC, USA, 2009. IEEE Computer Society.

- [SG02] European Software Institute Spain and IKV++ Technologies AG Germany. MASTER: Model-driven Architecture inSTRumentation, Enhancement and Refinement, IST-2001-34600 MASTER-2002-D1.1-V1-PUBLIC2002.
- [Sheo8] S. She. Feature Model Mining. Master's thesis, University of Waterloo, 2008.
- [She11] G. Sherwood. TestCover. web, last visited August 2011.
- [SJ04] K. Schmid and I. John. A Customizable Approach to Full Lifecycle Variability Management. *Science of Computer Programming*, 53:259–284, December 2004.
- [SLKH07] T. K. Satyananda, D. Lee, S. Kang, and S. I. Hashmi. Identifying Traceability between Feature Model and Software Architecture in Software Product Line using Formal Concept Analysis. In *Proceedings of the International Conference Computational Science and its Applications*, pages 380–388, Washington, DC, USA, 2007. IEEE Computer Society.
- [SM98] B. Stevens and E. Mendelsohn. Efficient Software Testing Protocols. In *Conference of the Centre for Advanced Studies on Collaborative research*, pages 22–36. IBM Press, 1998.
- [SMFM00] S. Souza, J. Maldonado, S. Fabbri, and P. Masiero. Statecharts Specifications: A Family of Coverage Testing Criteria. In *Proceedings of the Latin-American Conference of Informatics*, pages 167–185, 2000.
- [SOM09] A. Schürr, S. Oster, and F. Markert. Model-Driven Software Product Line Testing: An Integrated Approach. In *Proceedings of the 36th International Conference on Current Trends in Theory and Practice of Computer Science*, Lecture Notes in Computer Science (LNCS), pages 112–131, 2009.
- [SR11] SPLOT-Research. www.splot-research.org, last visit August 2011.
- [SRG11] K. Schmid, R. Rabiser, and P. Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2011)*, ACM ICPS, pages 119–126, Namur, Belgium, 2011. ACM Press.
- [SRP06] P. Sochos, M. Riebisch, and I. Philippow. The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based*

- Systems*, pages 308–318, Washington, DC, USA, 2006. IEEE Computer Society.
- [Ste11] M. Steffens. Industrial Evaluation of Feature Model Based Development and Combinatorial Testing for Software Product Lines. Master’s thesis, Technische Universität Darmstadt Department of Electrical Engineering and Information Technology Real-Time Systems Lab, 2011. Referee: Prof. Dr. rer. nat. Andy Schürr, Advisor: Dipl.-Inform. Sebastian Oster.
- [SV08] N. Szasz and P. Vilanova. Statecharts and Variabilities. In *Proceedings of 2nd International Workshop on Variability Modelling of Software-intensive Systems*, pages 131–140, 2008.
- [TBK09] T. Thum, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the IEEE 31st International Conference on Software Engineering*, pages 254–264, Washington, DC, USA, 2009. IEEE Computer Society.
- [TH02] S. Thiel and A. Hein. Modeling and Using Product Line Variability in Automotive Systems. *IEEE Software*, 19(4):66–72, 2002.
- [Tre99] J. Tretmans. Testing Concurrent Systems: A Formal Approach. In *Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65, London, UK, 1999. Springer-Verlag.
- [Treo4] T. Trew. What Design Policies Must Testers Demand from Product Line Architects? In *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004)*, pages 51–57, 2004.
- [TTK04] A. Tevanlinna, J. Taina, and R. Kauppinen. Product Family Testing: A Survey. *ACM SIGSOFT Software Engineering Notes*, 29:12–18, 2004.
- [UL07] M. Utting and B. Legeard. *Practical Model-Based Testing. A Tools Approach*. M. Kaufmann, 2007.
- [vdMo7] T. von der Maßen. *Feature-basierte Modellierung und Analyse von Variabilität in Produktlinienanforderungen*. PhD thesis, Rheinisch-Westfälische Technische Hochschule Aachen, 2007.
- [vGB03] J. van Gurp and J. Bosch, editors. *IEEE Workshop on Software Variability Management (SVM’03)*, Portland, Oregon, May 2003. IEEE.
- [vVo7] E. van Veenendaal, editor. *ISTQB-Glossary-of-Testing-Terms-2-0*. Glossary Working Party, 2007.
- [Walo1] E. Wallmüller. *Software-Qualitätsmanagement in der Praxis*. Carl Hanser Verlag, 2001.

- [Was04] A. Wasowski. Automatic Generation of Program Families by Model Restrictions. In *Proceeding of the 3rd International Software Product Line Conference*, volume 3154, pages 73–89, 2004.
- [WDS09] J. White, B. Dougherty, and D. C. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009.
- [Weio8] J. Weiland. *Variantenkonfiguration eingebetteter Automotive Software mit Simulink*. PhD thesis, Universität Leipzig, 2008.
- [Weio9] S. Weißleder. ParTeG (Partition Test Generator), <http://parteg.sourceforge.net>, last visit August 2011, 2009.
- [Wei10] S. Weißleder. *Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines*. PhD thesis, Fraunhofer First, 2010.
- [WL99] D. M. Weiss and R. Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.
- [WSSo8] S. Weißleder, D. Sokenou, and H. Schlingloff. Reusing State Machines for Automatic Test Generation in Product Lines. In *Proceedings of the 1st Workshop on Model-based Testing in Practice (MoTiP2008)*, 2008.
- [WW09] M. Westphal and S. Wöfl. Qualitative CSP, finite CSP, and SAT: Comparing Methods for Qualitative Constraint-based Reasoning. In *Proceedings of the 21st International joint Conference on Artificial intelligence (IJCAI'09)*, pages 628–633, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.
- [ZHM97] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Survey*, 29(4):366–427, 1997.
- [Zin11] M. Zink. *Anwendung von MoSo-PoLiTe in einer Automotive SPL*. Master, Technische Universität Darmstadt, Fachgebiet Echtzeitsysteme, 2011. Referee: Prof. Dr. rer. nat. Andy Schürr, Advisor: Dipl.-Inform. Sebastian Oster.

CURRICULUM VITAE



	Professional Experience
since 2008	Research Associate at the Real-Time Systems Lab at the Technische Universität Darmstadt
06/07 - 02/08	Development Engineer at Vitronic GmbH
09/06 - 12/06	Trainee at ISRA Vision AG
08/06	Trainee at d-sire GmbH & Co. KG
06/02 - 09/02	Employee at ZENIT (Zentrum für Innovation und Technik NRW)
	Education
10/02 - 09/07	University of Duisburg-Essen, Computer Science Studies, Diploma in Computer Science
2001	University-Entrance Diploma
1998	Exchange Student in the U.S. via scholarship from the German Bundestag and the U.S. Congress

DECLARATION

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, August 2011

Sebastian Oster

APPENDIX

FLATTENING RULES

Please note that the flattening rules have been published in parallel in a technical report [Ost11].

CHILD MANDATORY

Mandatory Parent



Transformation rule pulling up a *mandatory* child beneath a *mandatory* parent

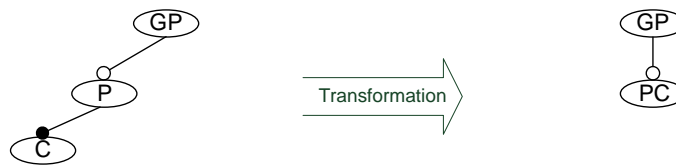
$$GP \wedge (P \rightarrow GP) \wedge (GP \rightarrow P) \wedge (C \rightarrow P) \wedge (P \rightarrow C)$$

$$= GP \wedge P \wedge C$$

$$GP \wedge (P \rightarrow GP) \wedge (C \rightarrow GP) \wedge (GP \rightarrow P) \wedge (GP \rightarrow C)$$

$$= GP \wedge P \wedge C$$

Optional Parent



Transformation rule pulling up a *mandatory* child beneath an *optional* parent

$$GP \wedge (P \rightarrow GP) \wedge (C \rightarrow P) \wedge (P \rightarrow C)$$

$$= GP \wedge (\neg C \vee P) \wedge (\neg P \vee C)$$

$$GP \wedge ((P \wedge C) \rightarrow GP) \wedge (C \rightarrow P) \wedge (P \rightarrow C)$$

$$= GP \wedge (\neg C \vee P) \wedge (\neg P \vee C)$$

Alternative Parent

Transformation rule pulling up a *mandatory* child beneath an *alternative* parent

$$\begin{aligned}
 & GP \wedge (GP \rightarrow \text{alt}(P_1, P_n) \wedge (P_1 \rightarrow GP) \wedge \\
 & (P_n \rightarrow GP) \wedge (C \rightarrow P_1) \wedge (P_1 \rightarrow C) \\
 & = GP \wedge \text{alt}(P_1, P_n) \wedge (C \rightarrow P_1) \wedge \\
 & (P_1 \rightarrow C) \\
 & = (GP \wedge P_1 \wedge C \wedge \neg P_n) \vee \\
 & (GP \wedge \neg P_1 \wedge \neg C \wedge P_n) \\
 & GP \wedge (GP \rightarrow \text{alt}((P_1 \wedge C), P_n)) \wedge \\
 & ((P_1 \wedge C) \rightarrow GP) \wedge (P_n \rightarrow GP) \\
 & = GP \wedge \text{alt}((P_1 \wedge C), P_n) \wedge (C \rightarrow P_1) \wedge \\
 & (P_1 \rightarrow C) \\
 & = (GP \wedge P_1 \wedge C \wedge \neg P_n) \vee \\
 & (GP \wedge \neg P_1 \wedge \neg C \wedge P_n)
 \end{aligned}$$

Or Parent

Transformation rule pulling up a *mandatory* child beneath an *or* parent

$$\begin{aligned}
 & GP \wedge (GP \rightarrow (P_1 \vee P_n)) \wedge (P_1 \rightarrow GP) \wedge \\
 & (P_n \rightarrow GP) \wedge (P_1 \rightarrow C) \wedge \\
 & (C \rightarrow P_1) \\
 & = GP \wedge (P_1 \vee P_n) \wedge (P_1 \rightarrow C) \wedge (C \rightarrow P_1) \\
 & = (GP \wedge \neg P_1 \wedge \neg C \wedge P_n) \vee (GP \wedge P_1 \wedge C) \\
 & GP \wedge (GP \rightarrow ((P_1 \wedge C) \vee P_n)) \wedge ((P_1 \wedge C) \rightarrow \\
 & GP) \wedge (P_n \rightarrow GP) \wedge (P_1 \rightarrow C) \wedge (C \rightarrow P_1) \\
 & (C \rightarrow P_1) \\
 & = GP \wedge ((P_1 \wedge C) \vee P_n) \wedge (P_1 \rightarrow C) \wedge (C \rightarrow P_1) \\
 & = (GP \wedge \neg P_1 \wedge \neg C \wedge P_n) \vee (GP \wedge P_1 \wedge C)
 \end{aligned}$$



Transformation rule pulling up an *optional* child beneath a *mandatory* parent

CHILD OPTIONAL:

Mandatory Parent

$$GP \wedge (P \rightarrow GP) \wedge (C \rightarrow P) \wedge (GP \rightarrow P)$$

$$=GP \wedge P$$

$$GP \wedge (P \rightarrow GP) \wedge (C \rightarrow GP) \wedge (GP \rightarrow P)$$

$$=GP \wedge P$$

Optional Parent



Transformation rule pulling up an *optional* child beneath an *optional* parent

$$GP \wedge (P \rightarrow GP) \wedge (C \rightarrow P)$$

$$=GP \wedge (C \rightarrow P)$$

$$GP \wedge (P \rightarrow GP) \wedge (C \rightarrow GP) \wedge (C \rightarrow P)$$

$$=GP \wedge (C \rightarrow P)$$

Alternative Parent

$$GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge (C \rightarrow P_1) \wedge (GP \rightarrow \text{alt}(P_1, P_n))$$

$$=GP \wedge (C \rightarrow P_1) \wedge \text{alt}(P_1, P_n)$$

$$GP \wedge (C \rightarrow GP) (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge (GP \rightarrow \text{alt}(P_1, P_n)) \wedge (C \rightarrow P_1)$$

$$=GP \wedge (C \rightarrow P_1) \wedge \text{alt}(P_1, P_n)$$



Transformation rule pulling up an *optional* child beneath an *alternative* parent

Or Parent



Transformation rule pulling up an *optional* child beneath an *or* parent

$ \begin{aligned} & GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \\ & \wedge (GP \rightarrow P_1 \vee P_n) \wedge (C \rightarrow P_1) \\ \\ & = GP \wedge (P_1 \vee P_n) \wedge (C \rightarrow P_1) \end{aligned} $	$ \begin{aligned} & GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \\ & \wedge (GP \rightarrow P_1 \vee P_n) \wedge (C \rightarrow GP) \wedge (C \rightarrow P_1) \\ \\ & = GP \wedge (P_1 \vee P_n) \wedge (C \rightarrow P_1) \end{aligned} $
--	--

CHILD ALTERNATIVE:

Mandatory Parent



Transformation rule pulling up an *alternative* child beneath a *mandatory* parent

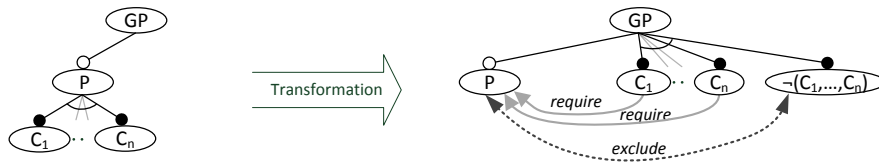
$$\begin{aligned}
 & GP \wedge (GP \rightarrow P) \wedge (P \rightarrow GP) \\
 & \wedge (C_1 \rightarrow P) \wedge (C_n \rightarrow P) \wedge \\
 & (P \rightarrow \text{alt}(C_1, C_n))
 \end{aligned}$$

$$= GP \wedge P \wedge \text{alt}(C_1, C_n)$$

$$\begin{aligned}
 & GP \wedge (GP \rightarrow P) \wedge (P \rightarrow GP) \wedge \\
 & (C_1 \rightarrow GP) \wedge (C_n \rightarrow GP) \wedge \\
 & (GP \rightarrow \text{alt}(C_1, C_n))
 \end{aligned}$$

$$= GP \wedge P \wedge \text{alt}(C_1, C_n)$$

Optional Parent



Transformation rule pulling up an *alternative* child beneath an *optional* parent

$$\begin{aligned}
 & GP \wedge (P \rightarrow GP) \wedge (C_1 \rightarrow P) \\
 & \wedge (C_n \rightarrow P) \wedge (P \rightarrow \text{alt}(C_1, C_n))
 \end{aligned}$$

$$\begin{aligned}
 & = GP \wedge (C_1 \rightarrow P) \wedge (C_n \rightarrow P) \\
 & \wedge (P \rightarrow \text{alt}(C_1, C_n))
 \end{aligned}$$

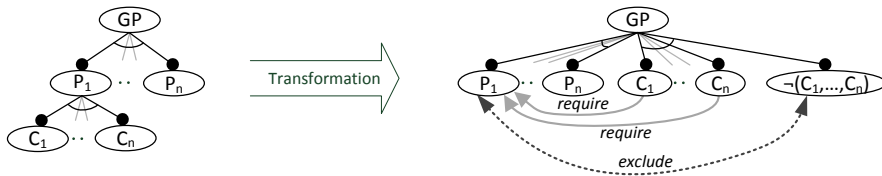
$$\begin{aligned}
 & = (GP \wedge \neg C_1 \wedge \neg C_n \wedge \neg P) \vee \\
 & (GP \wedge P \wedge C_1 \wedge \neg C_n) \vee \\
 & (GP \wedge P \wedge \neg C_1 \wedge C_n)
 \end{aligned}$$

$$\begin{aligned}
 & GP \wedge (P \rightarrow GP) \wedge (C_1 \rightarrow GP) \wedge (C_n \rightarrow GP) \wedge \\
 & (GP \rightarrow \text{alt}(C_1, C_n, \neg C_1 \wedge \neg C_n)) \wedge ((\neg C_1 \wedge \neg C_n) \\
 & \rightarrow GP) \wedge (C_1 \rightarrow P) \wedge (C_n \rightarrow P) \wedge \\
 & (\neg P \vee \neg(\neg C_1 \wedge \neg C_n))
 \end{aligned}$$

$$\begin{aligned}
 & = GP \wedge (C_1 \rightarrow P) \wedge (C_n \rightarrow P) \\
 & \text{alt}(C_1, C_n, \neg C_1 \wedge \neg C_n) \wedge (\neg P \vee \neg(\neg C_1 \wedge \neg C_n))
 \end{aligned}$$

$$\begin{aligned}
 & = (GP \wedge \neg C_1 \wedge \neg C_n \wedge \neg P) \vee \\
 & (GP \wedge P \wedge C_1 \wedge \neg C_n) \vee \\
 & (GP \wedge P \wedge \neg C_1 \wedge C_n)
 \end{aligned}$$

Alternative Parent

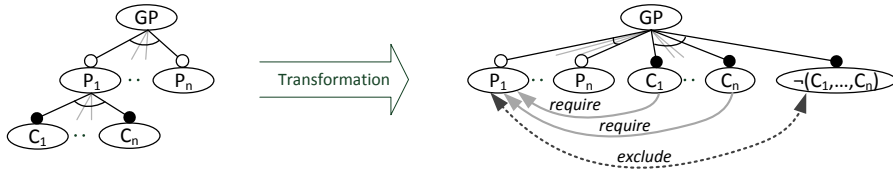


Transformation rule pulling up an *alternative* child beneath an *alternative* parent

$$\begin{aligned}
& GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge \\
& (GP \rightarrow \text{alt}(P_1, P_n)) \wedge (C_1 \rightarrow P_1) \wedge \\
& (C_n \rightarrow P_1) \wedge (P_1 \rightarrow \text{alt}(C_1, C_n)) \\
\\
& = GP \wedge \text{alt}(P_1, P_n) \wedge (C_1 \rightarrow P_1) \wedge \\
& (C_n \rightarrow P_1) \wedge (P_1 \rightarrow \text{alt}(C_1, C_n)) \\
\\
& = (GP \wedge P_n \wedge \neg P_1 \wedge \neg C_1 \wedge C_n) \vee \\
& (GP \wedge P_1 \wedge \neg P_n \wedge C_1 \wedge \neg C_n) \vee \\
& (GP \wedge P_1 \wedge \neg P_n \wedge \neg C_1 \wedge C_n)
\end{aligned}$$

$$\begin{aligned}
& GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge \\
& (GP \rightarrow \text{alt}(P_1, P_n)) \wedge (C_1 \rightarrow GP) \wedge \\
& (C_n \rightarrow GP) \wedge ((\neg C_1 \wedge \neg C_n) \rightarrow GP) \wedge \\
& (GP \rightarrow \text{alt}(C_1, C_n, (\neg C_1 \wedge \neg C_n))) \wedge \\
& (C_1 \rightarrow P_1) \wedge (C_n \rightarrow P_1) \wedge \\
& ((\neg C_1 \wedge \neg C_n) \vee \neg P_1) \\
\\
& = GP \wedge \text{alt}(P_1, P_n) \wedge \text{alt}(C_1, C_n, (\neg C_1 \wedge \neg C_n)) \wedge \\
& (C_1 \rightarrow P_1) \wedge (C_n \rightarrow P_1) \wedge ((C_1 \vee C_n) \vee \neg P_1) \\
\\
& = (GP \wedge P_n \wedge \neg P_1 \wedge \neg C_1 \wedge C_n) \vee \\
& (GP \wedge P_1 \wedge \neg P_n \wedge C_1 \wedge \neg C_n) \vee \\
& (GP \wedge P_1 \wedge \neg P_n \wedge \neg C_1 \wedge C_n)
\end{aligned}$$

Or Parent



Transformation rule pulling up an *alternative* child beneath an *or* parent

$$\begin{aligned}
& GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge \\
& (GP \rightarrow (P_1 \vee P_n)) \wedge (C_1 \rightarrow P_1) \wedge \\
& (C_n \rightarrow P_1) \wedge (P_1 \rightarrow \text{alt}(C_1, C_n)) \\
\\
& = GP \wedge P_1 \vee P_n \wedge (C_1 \rightarrow P_1) \wedge \\
& (C_n \rightarrow P_1) \wedge (P_1 \rightarrow \text{alt}(C_1, C_n)) \\
\\
& = (GP \wedge P_n \wedge \neg P_1 \wedge \neg C_1 \wedge \neg C_n) \vee \\
& (GP \wedge P_1 \wedge C_1 \wedge \neg C_n) \vee \\
& (GP \wedge P_1 \wedge \neg C_1 \wedge \neg C_n)
\end{aligned}$$

$$\begin{aligned}
& GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \\
& (C_1 \rightarrow GP) \wedge (C_n \rightarrow GP) \wedge \\
& ((\neg C_1 \wedge \neg C_n) \rightarrow GP) \wedge (GP \rightarrow (P_1 \vee P_n)) \wedge \\
& (r \rightarrow \text{alt}(C_1, C_n, (\neg C_1 \wedge \neg C_n))) \wedge (C_1 \rightarrow P_1) \wedge \\
& (C_n \rightarrow P_1) \wedge ((\neg C_1 \wedge \neg C_n) \vee \neg P_1) \\
\\
& = GP \wedge (P_1 \vee P_n) \wedge (C_1 \rightarrow P_1) \wedge (C_n \rightarrow P_1) \wedge \\
& \text{alt}(C_1, C_n, (\neg C_1 \wedge \neg C_n)) \wedge ((C_1 \vee C_n) \vee \neg P_1) \\
\\
& = (GP \wedge P_n \wedge \neg P_1 \wedge \neg C_1 \wedge C_n) \vee \\
& (GP \wedge P_1 \wedge C_1 \wedge \neg C_n) \vee \\
& (GP \wedge P_1 \wedge \neg C_1 \wedge \neg C_n)
\end{aligned}$$

CHILD OR:

Mandatory Parent



Transformation rule pulling up an *or* child beneath a *mandatory* parent

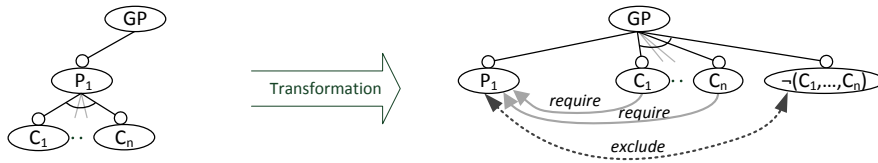
$$GP \wedge (GP \rightarrow P) \wedge (P \rightarrow GP) \wedge \\ (C_1 \rightarrow P) \wedge (C_n \rightarrow P) \wedge (P \rightarrow (C_1 \vee C_n))$$

$$= GP \wedge P \wedge (C_1 \vee C_n)$$

$$GP \wedge (GP \rightarrow P) \wedge (P \rightarrow GP) \wedge (C_1 \rightarrow GP) \wedge \\ (C_n \rightarrow GP) \wedge (GP \wedge P \wedge (C_1 \vee C_n))$$

$$= GP \wedge P \wedge (C_1 \vee C_n)$$

Optional Parent



Transformation rule pulling up an *or* child beneath an *optional* parent

$$GP \wedge (P \rightarrow GP) \wedge (C_1 \rightarrow P) \wedge \\ (C_n \rightarrow P) \wedge (P \rightarrow (C_1 \vee C_n))$$

$$= GP \wedge (C_1 \rightarrow P) \wedge (C_n \rightarrow P) \wedge \\ (P \rightarrow (C_1 \vee C_n))$$

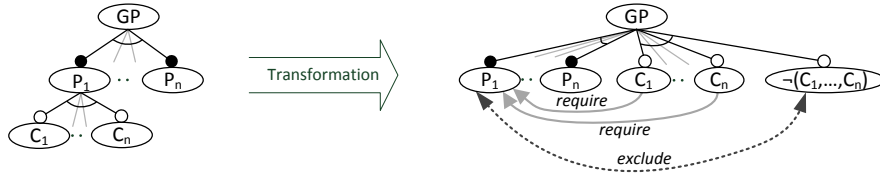
$$= GP \wedge (C_1 \rightarrow P) \wedge (C_n \rightarrow P) \wedge \\ ((C_1 \vee C_n) \vee \neg P)$$

$$GP \wedge (P \rightarrow GP) \wedge (C_1 \rightarrow GP) \wedge \\ (C_n \rightarrow GP) \wedge ((\neg C_1 \wedge \neg C_n) \rightarrow GP) \wedge \\ (GP \rightarrow (C_1 \vee C_n \vee (\neg C_1 \wedge \neg C_n))) \wedge (C_1 \rightarrow \\ P) \wedge (C_n \rightarrow P) \wedge (\neg(\neg C_1 \wedge \neg C_n) \vee \neg P)$$

$$= GP \wedge (C_1 \rightarrow P) \wedge (C_n \rightarrow P) \wedge (C_1 \vee C_n \vee \\ (\neg C_1 \wedge \neg C_n)) \wedge ((C_1 \vee C_n) \vee \neg P)$$

$$= GP \wedge (C_1 \rightarrow P) \wedge (C_n \rightarrow P) \wedge \\ ((C_1 \vee C_n) \vee \neg P)$$

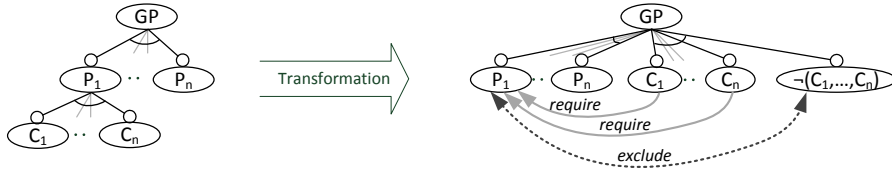
Alternative Parent



Transformation rule pulling up an *or* child beneath an *alternative* parent

$$\begin{aligned}
 & GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge \\
 & (GP \rightarrow \text{alt}(P_1, P_n)) \wedge (C_1 \rightarrow P_1) \wedge \\
 & (C_n \rightarrow P_1) \wedge (P_1 \rightarrow (C_1 \vee C_n)) \\
 & = GP \wedge \text{alt}(P_1, P_n) \wedge (C_1 \rightarrow P_1) \wedge \\
 & (C_n \rightarrow P_1) \wedge (P_1 \rightarrow (C_1 \vee C_n)) \\
 & = GP \wedge \text{alt}(P_1, P_n) \wedge (C_1 \rightarrow P_1) \wedge \\
 & (C_n \rightarrow P_1) \wedge (\neg P_1 \vee (C_1 \vee C_n)) \\
 & GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge (GP \rightarrow \text{alt}(P_1, P_n)) \\
 & \wedge (C_1 \rightarrow GP) \wedge (C_n \rightarrow GP) \wedge ((\neg C_1 \wedge \neg C_n) \rightarrow GP) \\
 & \wedge (GP \rightarrow (C_1 \vee C_n \vee (\neg C_1 \wedge \neg C_n))) \wedge (C_1 \rightarrow P_1) \wedge \\
 & (C_n \rightarrow P_1) \wedge (\neg(\neg C_1 \wedge \neg C_n) \vee \neg P_1) \\
 & = GP \wedge \text{alt}(P_1, P_n) \wedge (C_1 \rightarrow P_1) \wedge \\
 & (C_n \rightarrow P_1) \wedge (\neg(\neg C_1 \wedge \neg C_n) \vee \neg P_1) \\
 & = GP \wedge \text{alt}(P_1, P_n) \wedge (C_1 \rightarrow P_1) \wedge \\
 & (C_n \rightarrow P_1) \wedge (\neg P_1 \vee (C_1 \vee C_n))
 \end{aligned}$$

Or Parent



Transformation rule pulling up an *or* child beneath an *or* parent

$$\begin{aligned}
 & GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge \\
 & (GP \rightarrow (P_1 \vee P_n)) \wedge (C_1 \rightarrow P_1) \wedge \\
 & (C_n \rightarrow P_1) \wedge (P_1 \rightarrow (C_1 \vee C_n)) \\
 & = GP \wedge (P_1 \vee P_n) \wedge (C_1 \rightarrow P_1) \wedge \\
 & (C_n \rightarrow P_1) \wedge (\neg P_1 \vee (C_1 \vee C_n)) \\
 & GP \wedge (P_1 \rightarrow GP) \wedge (P_n \rightarrow GP) \wedge (C_1 \rightarrow GP) \wedge \\
 & (C_n \rightarrow GP) \wedge ((\neg C_1 \wedge \neg C_n) \rightarrow GP) \wedge \\
 & (GP \rightarrow (P_1 \vee P_n)) \wedge (GP(C_1 \vee C_n \vee (\neg C_1 \wedge \neg C_n))) \wedge \\
 & (C_1 \rightarrow P_1) \wedge (C_n \rightarrow P_1) \wedge (\neg P_1 \vee (\neg(\neg C_1 \wedge \neg C_n))) \\
 & = GP \wedge (P_1 \vee P_n) \wedge (C_1 \rightarrow P_1) \wedge (C_n \rightarrow P_1) \wedge \\
 & (\neg P_1 \vee (C_1 \vee C_n))
 \end{aligned}$$

SDM TRANSFORMATION RULES

In the following we present the SDM diagrams implementing the flattening algorithm. Figure 74 shows the metamodel used for the SDM implementation. It consists of a feature model, features, dependencies (*mandatory*, *optional*, *or*, and *alternative*), and constraints (*require* and *exclude*).

Figure 75 depicts the SDM diagram providing the implementation to iterate through the feature model and to search for a three level subtree. The SDM diagram implements a bottom up approach searching for leaves within the feature model. Those leaves are incrementally pulled up using the flattening rules and stops until all leaves are directly situated beneath the root feature of the feature model.

After finding a three-level subtree, the corresponding flattening rules are called depending on the relationship between the parent feature and its child feature(s). Figure 76 shows the implementation of the three different flattening rules. The first block processes *mandatory* child features and was explained in detail in the Implementation and Evaluation part. The second block processes *optional* child features and the third block flattens subtrees with child features within an *or* or *alternative* group.

MANDATORY CHILD

Figure 77 depicts the SDM-based flattening for *mandatory* child features. The SDM diagram implements the following 4 steps as denoted in Figure 77:

1. Check whether the precondition is fulfilled: child *mandatory*.
2. Concatenation of the parent name and the child name to a new name of the former parent feature.
3. Update of constraints. The constraints in which the former child feature was involved are transferred to the parent feature.
4. Destroy *mandatory* child and the corresponding relationship.

OPTIONAL CHILD

The graph transformation depicted in Figure 78 processes *optional* child features. The SDM diagram implements the following 4 steps as denoted in Figure 78:

1. Check precondition: child feature is *optional*.

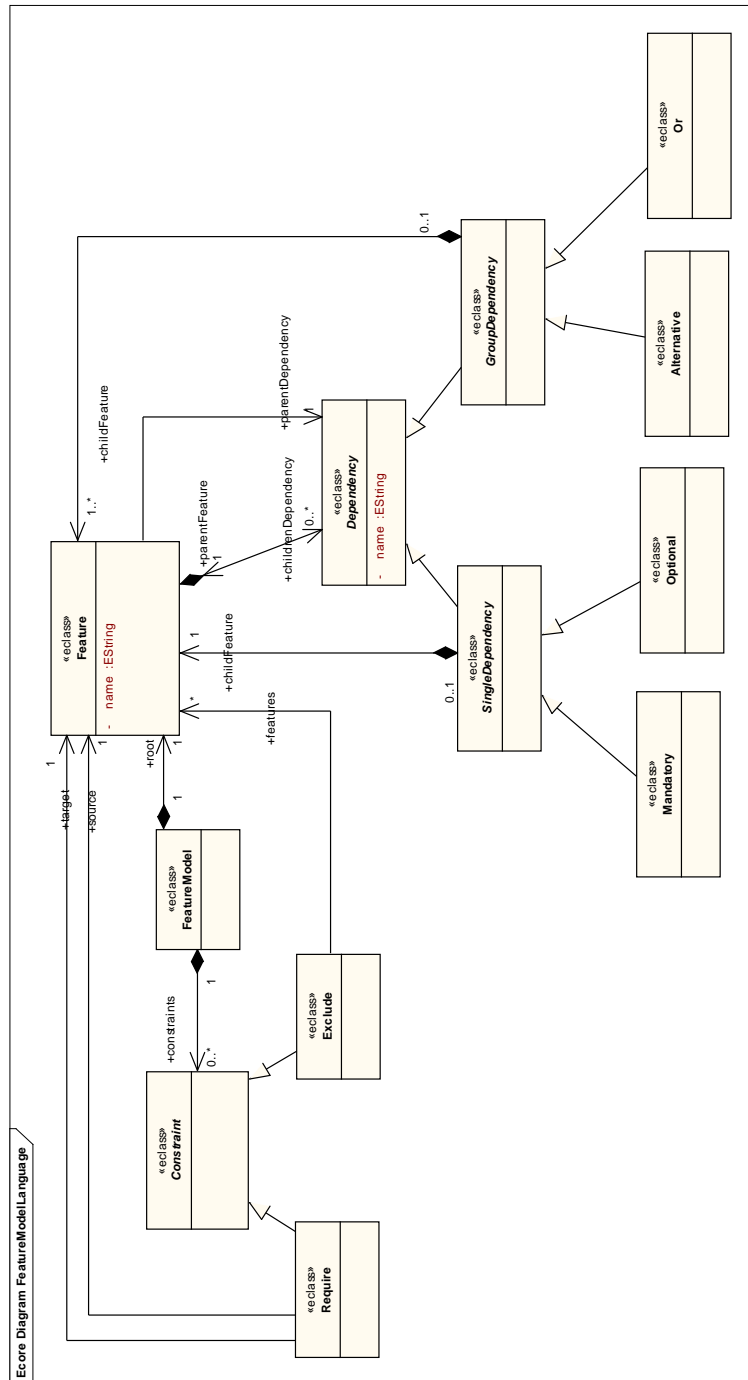


Figure 74: Appendix: Metamodel for the SDM implementation

2. If precondition holds, the child feature is pulled up beneath the grandparent feature.
3. If the parent feature is *mandatory* then the transformation is complete.

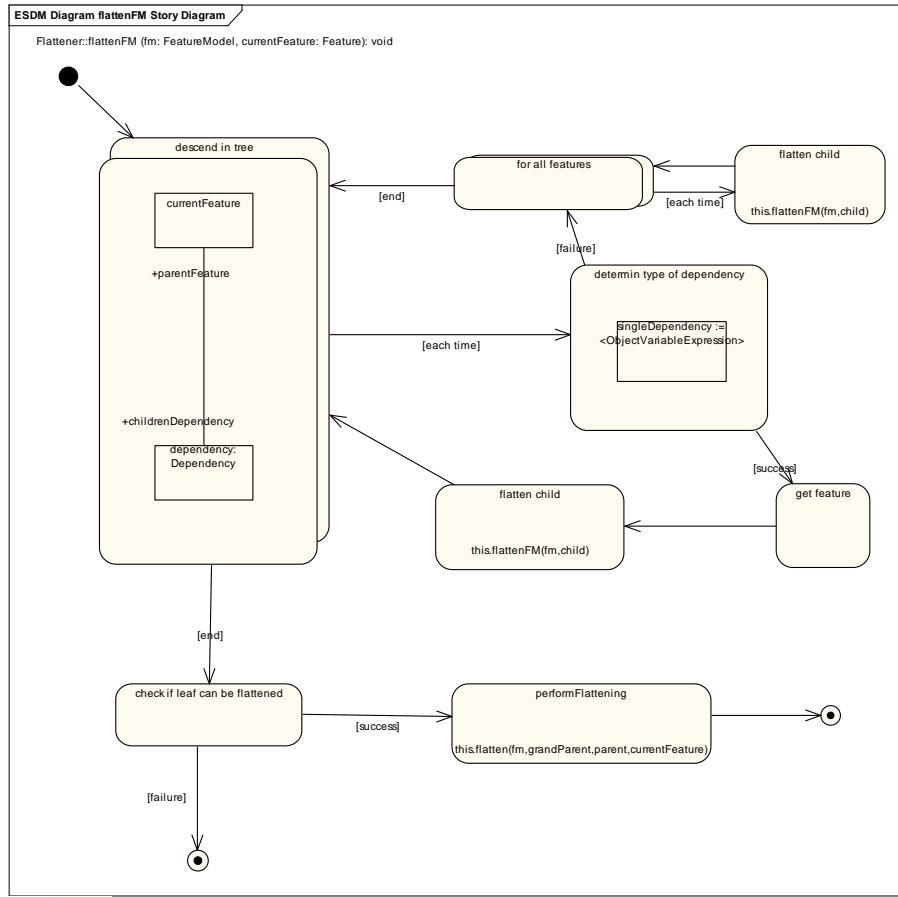


Figure 75: Appendix: Iteration through the feature model including pattern matching

4. If the parent is *optional* or within an *or* or an *alternative* group an additional *require* constraint is added pointing from the former child feature towards the former parent feature.

ALTERNATIVE AND OR CHILD

The graph transformation depicted in Figure 79 processes child features within *or* or *alternative* groups.

The SDM diagram implements the following 5 steps as denoted in Figure 79:

1. Check precondition: child feature is within a group.
2. Pull up the child feature.
3. Check if parent feature is *mandatory*: if the parent feature is *mandatory* the transformation is complete

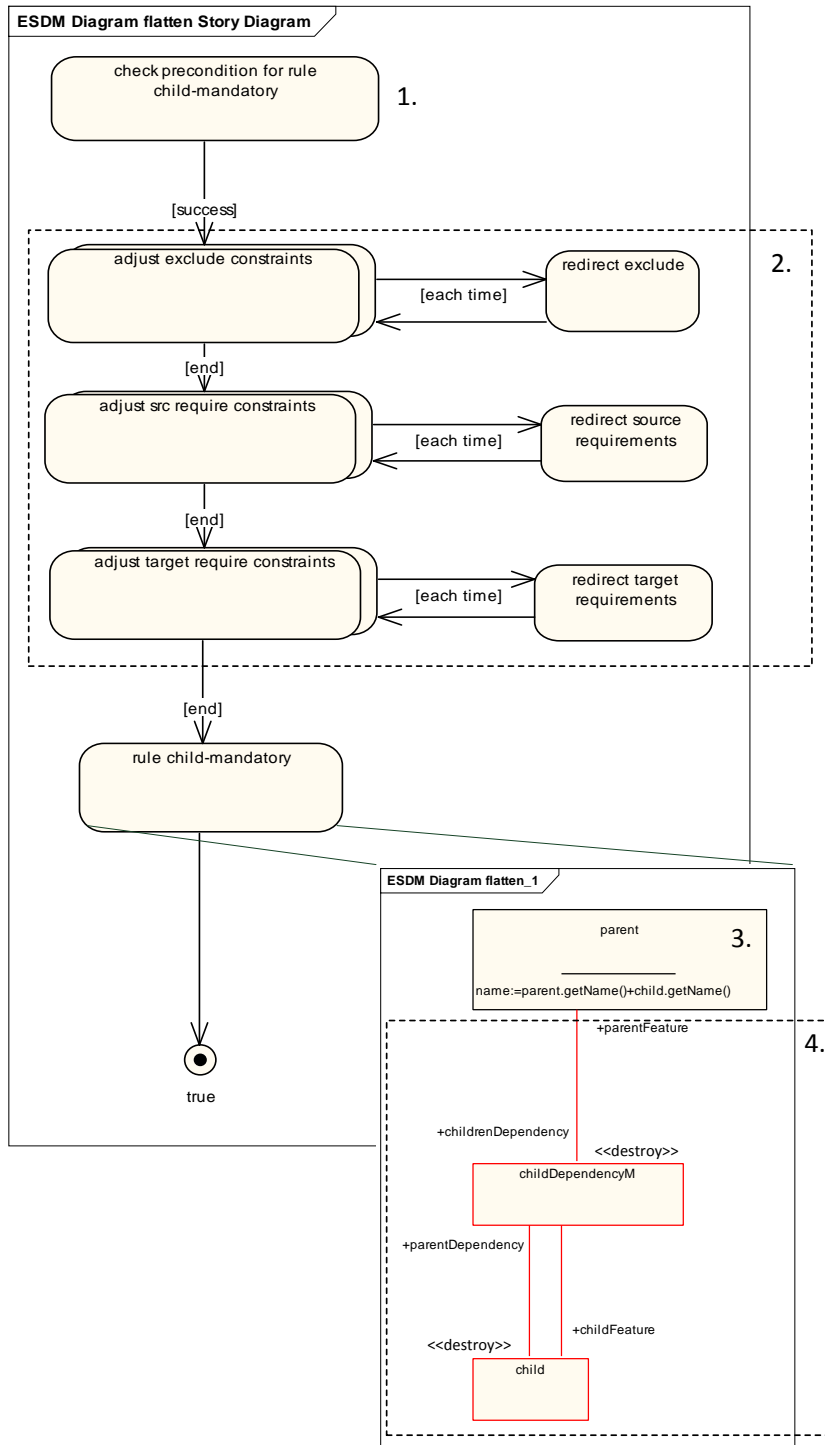


Figure 77: Appendix: Transformation rule for *mandatory* children

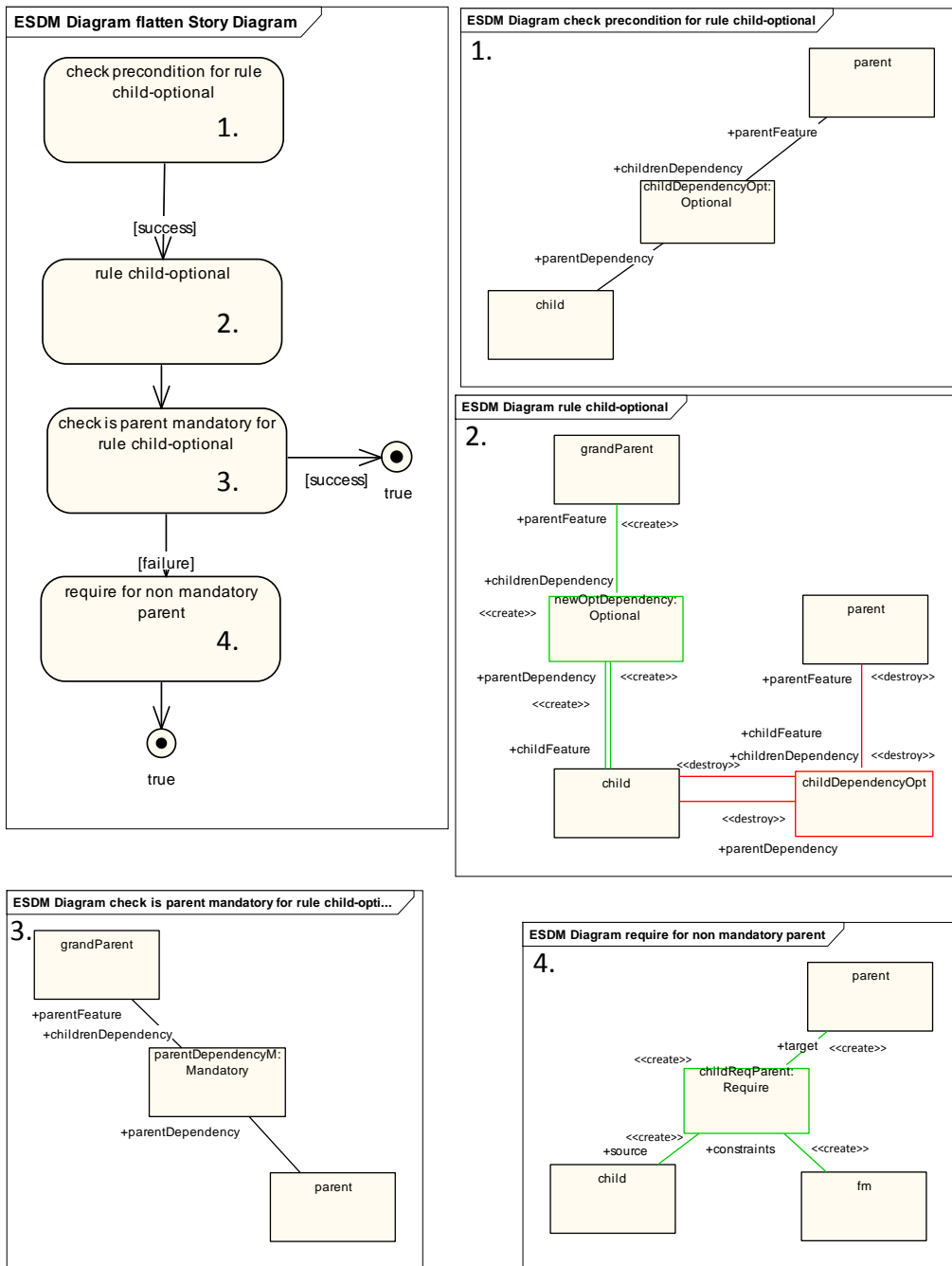


Figure 78: Appendix: Transformation rule for *optional* child features

5. The negation feature is added and an *exclude* constraint between the former child feature and the negation features is added.

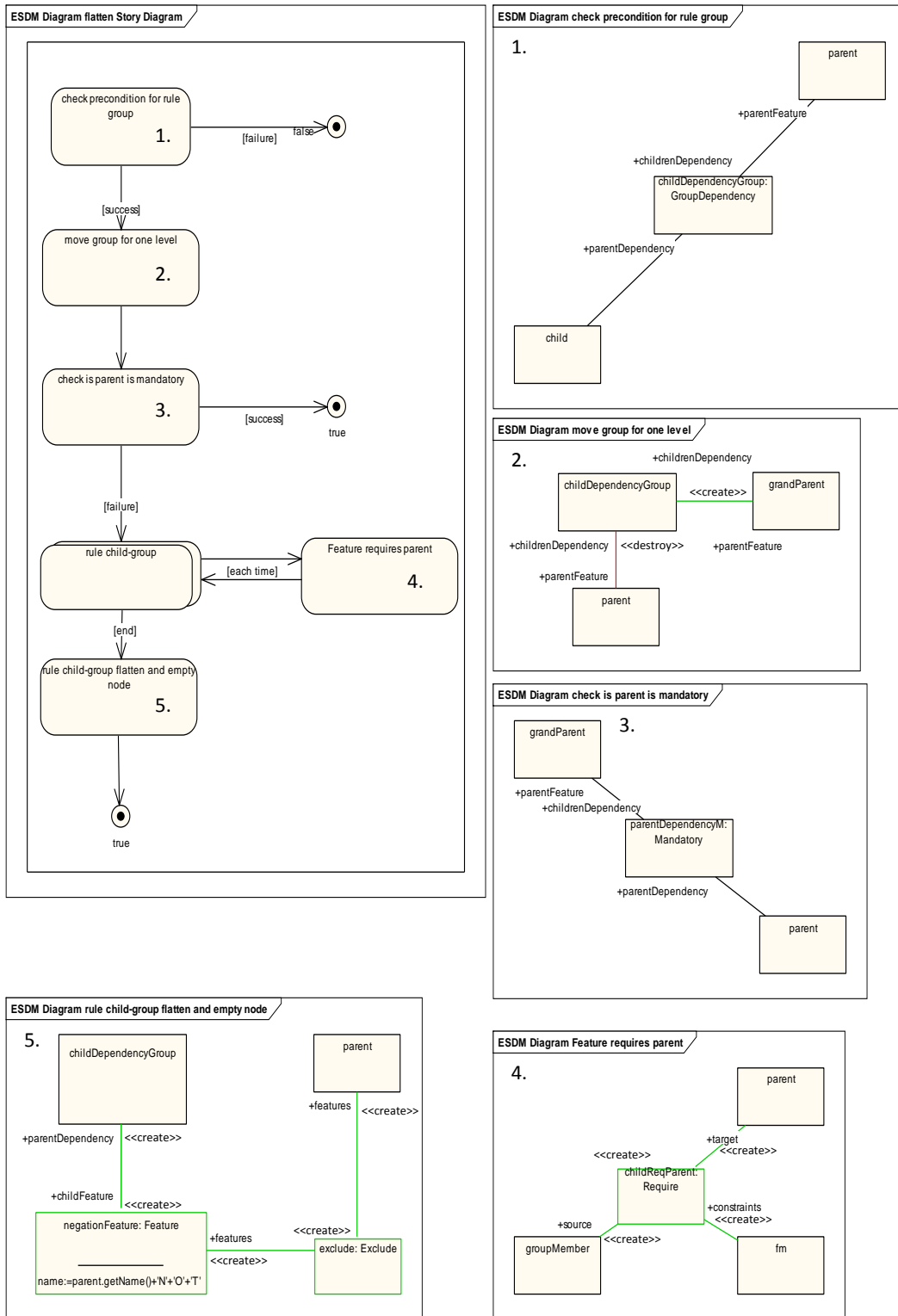


Figure 79: Appendix: Transformation rule for child features within *or* or *alternative* groups

GENERATED SOURCE CODE FOR CHILD OPTIONAL

Listing 7: Appendix: Generated Source Code for child *optional*

```
// story pattern check precondition for rule child-optional
try {
    fujaba__Success = false;
4
    // check object child is really bound
6    JavaSDM.ensure(child != null);
    // check object parent is really bound
8    JavaSDM.ensure(parent != null);
    // check isomorphic binding between objects parent and child
10   JavaSDM.ensure(!parent.equals(child));

12   // bind object
    _TmpObject = child.getParentDependency();
14

    // ensure correct type and really bound of object childDependencyOpt
16   JavaSDM.ensure(_TmpObject instanceof Optional);
    childDependencyOpt = (Optional) _TmpObject;
18

    // check link childrenDependency from childDependencyOpt to parent
20   JavaSDM.ensure(parent.equals(childDependencyOpt.getParentFeature()));

22   fujaba__Success = true;
} catch (JavaSDMException fujaba__InternalException) {
24   fujaba__Success = false;
}
26
if (fujaba__Success) {
28   // story pattern rule child-optional
    try {
30       fujaba__Success = false;

32       // check object child is really bound
        JavaSDM.ensure(child != null);
34       // check object childDependencyOpt is really bound
        JavaSDM.ensure(childDependencyOpt != null);
36       // check object grandParent is really bound
        JavaSDM.ensure(grandParent != null);
38       // check object parent is really bound
        JavaSDM.ensure(parent != null);
```

```
40      // check isomorphic binding between objects grandParent and
      // child
      JavaSDM.ensure(!grandParent.equals(child));
42
44      // check isomorphic binding between objects parent and child
      JavaSDM.ensure(!parent.equals(child));
46
48      // check isomorphic binding between objects parent and
      // grandParent
      JavaSDM.ensure(!parent.equals(grandParent));
50
52      // check link childFeature from childDependencyOpt to child
      JavaSDM.ensure(child.equals(childDependencyOpt.getChildFeature
      ()));
54
56      // check link childrenDependency from childDependencyOpt to
      // parent
      JavaSDM.ensure(parent.equals(childDependencyOpt.
      getParentFeature()));
58
60      // check link parentDependency from child to
      // childDependencyOpt
      JavaSDM.ensure(childDependencyOpt.equals(child.
      getParentDependency()));
62
64      // destroy link
      org.moflon.util.eMoflonEMFUtil.removeOppositeReference(child,
      childDependencyOpt, "parentDependency");
66      // destroy link
      childDependencyOpt.setChildFeature(null);
68      // destroy link
      parent.getChildrenDependency().remove(childDependencyOpt);
70      // delete object childDependencyOpt
      org.moflon.util.eMoflonEMFUtil.remove(childDependencyOpt);
72
74      // create object newOptDependency
      newOptDependency = FeatureModelLanguageFactory.eINSTANCE.
      createOptional();
76      // create link
      org.moflon.util.eMoflonEMFUtil.addOppositeReference(child,
      newOptDependency, "parentDependency");
      // create link
      newOptDependency.setChildFeature(child);
      // create link
      newOptDependency.setParentFeature(grandParent);
      fujaba__Success = true;
```

```

78     } catch (JavaSDMException fujaba__InternalException) {
79         fujaba__Success = false;
80     }

82     // story pattern check is parent mandatory for rule child-optional
83     try {
84         fujaba__Success = false;

86         // check object grandParent is really bound
87         JavaSDM.ensure(grandParent != null);
88         // check object parent is really bound
89         JavaSDM.ensure(parent != null);
90         // check isomorphic binding between objects parent and
91         // grandParent
92         JavaSDM.ensure(!parent.equals(grandParent));

94         // bind object
95         _TmpObject = parent.getParentDependency();

96         // ensure correct type and really bound of object
97         // parentDependencyM
98         JavaSDM.ensure(_TmpObject instanceof Mandatory);
99         parentDependencyM = (Mandatory) _TmpObject;

100        // check link childrenDependency from parentDependencyM to
101        // grandParent
102        JavaSDM.ensure(grandParent.equals(parentDependencyM.
103            getParentFeature()));

104        fujaba__Success = true;
105    } catch (JavaSDMException fujaba__InternalException) {
106        fujaba__Success = false;
107    }

108    if (fujaba__Success) {
109        return true;
110    } else {
111        // story pattern require for non mandatory parent
112        try {
113            fujaba__Success = false;

114            // check object child is really bound
115            JavaSDM.ensure(child != null);
116            // check object fm is really bound
117            JavaSDM.ensure(fm != null);
118            // check object parent is really bound
119            JavaSDM.ensure(parent != null);

```

```
122         // check isomorphic binding between objects parent and
           child
           JavaSDM.ensure(!parent.equals(child));
124
           // create object childReqParent
126         childReqParent = FeatureModelLanguageFactory.eINSTANCE
           .createRequire();
           // create link
128         childReqParent.setSource(child);
130
           // create link
           org.moflon.util.eMoflonEMFUtil.addOppositeReference(
           childReqParent, parent, "target");
132         // create link
           fm.getConstraints().add(childReqParent);
134
           fujaba__Success = true;
136     } catch (JavaSDMException fujaba__InternalException) {
           fujaba__Success = false;
138     }
140     return true;
142 }
```