Integrated Modeling of Software Product Lines with Feature Models and Classification Trees

Sebastian Oster
Real-Time Systems Group
Technische Universität Darmstadt

Florian Markert
Computer Systems Group
Technische Universität Darmstadt

Andy Schürr
Real-Time Systems Group
Technische Universität Darmstadt

Werner Müller
Global Systems Engineering Methods
Adam Opel GmbH

***Abstract*—Software Product Lines (SPLs) are an approach to improve reusability of software in a large number of products that share a common set of features. In SPLs, Feature Models (FMs) are frequently used to model commonalities and variabilities. However, according to the best of our knowledge, there are no approaches to automatically generate test cases on the basis of a stand-alone FM. We introduce a method, which fills this gap. In single system software testing, Classification Trees (CTs) are a proven approach for generating test cases derived from the original system specification. In this paper, we explore the relations and similarities between FMs and CTs and integrate both methods to a unified approach called *Feature Model for Testing* (FMT).**

I. Introduction

SPL-engineering is one of the most promising approaches of the Software Engineering community to reduce the development costs, for as well as to increase the quality of families of similar software product instances [4]. As a consequence, software product line engineering is successfully used in various domains, including the domain of automotive software development. In this area, the combination of highly parametrized software of electronic control units (ECUs), together with an abundance of configuration options of networks of ECUs will soon lead to a situation, where (1) a single ECU may be instantiated in at least 10.000 different ways and (2) the software of a network of more than 50 ECUs in a single car may exist in millions of different configurations.

As a matter of fact, we are, therefore, running into a situation where each instance of a certain brand of car possesses a unique configuration of the embedded software of all its ECUs. Testing all these millions of instances of an automotive SPL in the following traditional way is no longer feasible: create all actually used instances of an SPL and then develop for each instance a separate suite of integration test cases. Hence, the automotive industry as well as engineers from other domains are urgently looking for new methods how to systematically generate sets of software product instances that represent equivalence classes of instances with a sufficiently similar behavior from a system integration testing point of view. Furthermore, model-based and more traditional black and white box testing approaches are adapted in such a way that families of test models and derived test cases can be developed, together with semi-automatic procedures that allow one to select the appropriate test cases for a specific SPL instance.

Examining more closely the state-of-the-art of SPL development and software testing approaches in the automotive industry we see that various kinds of feature modeling concepts and tools are used for the design of SPLs and the selection of needed instances [4], [21]. On the other hand, CTs and related tools such as CTE [1] are successfully used for black-box testing of single product instances. We are not aware of any proposal how to combine feature modeling concepts used for the description and selection of features (parameters, options, ...) of SPLs with CT-concepts used for the description and selection of test case parameters of selected product instances—despite of the fact that the borderlines between feature selection at compile time and input parameter selection at runtime are blurred, and the same parameter may either be instantiated at compile time or flash time, to unlock a specific function or changed at runtime to activate a certain mode of operation.

To overcome these problems we will first present in Section II of this paper the fundamentals of SPL description by means of FMs and the specification of parameter equivalence classes for testing purposes by means of CTs. Furthermore, this section introduces our paper's running example, a subset of a case study provided by the Adam Opel GmbH which is a subsidiary of GM (General Motors) Corp. Afterwards, we explain in more detail the state-of-the-art of systematic testing approaches of SPLs and black box testing with CTs in a related work section. Section IV then systematically compares the basic modeling elements of FMs, similar to the developed FMs in FODA [14] and the classification approach as supported by the tool CTE [1]. Based on this comparison, a tight integration of both modeling languages is proposed and the abstract syntax of the resulting feature and test parameter modeling language "FMT" is presented. In Section V we describe the derivation of a product with corresponding test cases from the FMT. Additionally, we discuss an approach of testing SPLs using FMT. The conclusion summarizes the advantages of such an integrated feature and parameter equivalence class modeling approach and lists a number of ongoing and future research activities.

II. Fundamentals

The contribution of this paper is to generate variants and corresponding test cases on the basis of one representation of the SPL.

A. Running Example

Our running example is a very simple subset of hardware and software components of the recently released Opel Insignia. We restrain ourselves to four sensors, two actors (engines), and one software component. The sensors are *rain light sensor (RLS)*, *turn indicator sensor (TIS)*, *steering angle sensor (SAS)*, and the *vehicle speed sensor (VSS)*. The *RLS* detects rain and the *TIS* indicates the driver's choice to turn left or right. *SAS* senses the steering angle and the *VSS* senses the speed of the car. Two different types of engines serve as hardware features: a *1.6 liter* and a *2.0 liter turbo* engine. Additionally, we use one feature of the (Adaptive Forward Lighting) AFL+ technology. The *bending light* is a functionality which belongs to AFL+ and realizes an adaptive curve light. All parameters presented in this paper are provided by the Adam Opel GmbH. We use the running example to exemplify the differences and commonalties between FMs and CTs and to motivate our approach of integrating both to a Feature Model for Testing (FMT).

B. Product Lines and Feature Models

SPLs provide a high level reuse of software in a specific problem domain [4], [21]. FMs are frequently used to describe an explicit representation of the commonalities and variabilities in an SPL. FMs consist of features each representing "a system property that is relevant to some stakeholder" as defined in [6]. One major advantage of using FMs to model SPLs is that they offer a very intuitive way to represent commonalities and variabilities. However, FMs by themselves are insufficient for a complete modeling of an SPL. Usually FMs are complemented by development artifacts such as UML diagrams or code fragments that are traced to the corresponding features. An FM provides a tree-like structure and incorporates different node notations and cross-tree-constraints. The first feature model was introduced by Kang et al. in [14] as part of FODA, in 1990. In the FM of FODA node notations like, mandatory, optional, and alternative features can be modeled. It is also possible to use require and exclude constraints between features, which are described textually. Since the introduction of FODA, further extensions of FMs were introduced to improve precision and expressiveness, including amongst others cardinalities, probabilities, and weighting. We can employ cardinalities to formulate the different notations of features and groups of features [7]. The probabilities can be based on empirical data and/or system specifications and are used to state that a certain feature is more likely than another one [8]. Weights can be used to represent cost factors of features to help the engineers to build products appropriate for a certain budget [13]. Czarnecki et al. summarize some existing extensions of the FODA FM [6]. However, our FM is very similar to the original FODA with additional cardinality extension [7]. Table I depicts the used notation. We do not want to discuss every FODA extension because our approach is not limited to a certain FM. We also aim e.g. to support the Orthogonal Variability Model (OVM) proposed by Pohl et al. [21]. A detailed description of the relation between FMs and OVMs is given in [18]. We, therefore, assume that our unified approach using FMs may also use the OVM approach.

The FM in Fig. 1 shows an excerpt of the Opel Insignia FM. All dependencies and constraints within the FM have to be taken into account when deriving a product.

Graphical Notation	Cardinality	Formal description
Single features		
edge with filled circle	1..1	feature is mandatory
edge with unfilled circle	0..1	feature is optional
Group of features		
filled circles and connected edges	1..n	choose exactly one feature
unfilled circles and connected edges	n..m	choose any combination of features, but at least one

TABLE I
Notation

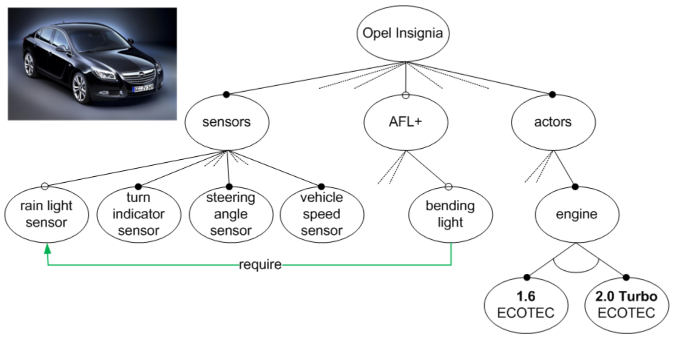

Fig. 1. Feature Model of Opel Insignia

C. Software Testing with Classification Trees

After extracting a variant from the FM, suitable test cases need to be built. For this purpose the CT-method was introduced by Grochtmann and Grimm [11]. It provides an approach to black box testing. Test cases can be extracted by defining rules for valid input combinations. A CT consists of *classifications* (boxes with bold lines), *compositions* (boxes with thin lines), and *equivalence classes* (values in brackets). Each *equivalence class* represents a disjoint subset of parameter values for a *classification*, while the *composition* splits complex input parameters into a number of subcomponents. To generate test cases from a specification using CTs the following procedure needs to be applied:

1) evaluate the specification and identify all *classifications* with the corresponding *equivalence classes*
2) build the CT
3) fill the test case table with respect to the CT using parameter value combination heuristics

4) extract all possible test cases from the test case table by identifying valid combinations of *equivalence classes* omitting illegitimate samples out of the test set

Fig. 2 depicts a variant of the Insignia SPL and a corresponding CT to test this instance. *Sensors* is a *composition*, while *turn indicator* is a *classification* of the *equivalence classes* representing test values. The model is extracted from the specification of the product instance. Test cases can be derived easily by choosing a valid combination of *equivalence classes*. This model can e.g. be used to check the output characteristics of the AFL+ dependent upon the sensor intervals. The range of the intervals is defined by the designer. It depends on suitable test scenarios chosen by the designer or extracted from the specification. A test case table is built, which incorporates all relevant test cases in an abstract way.

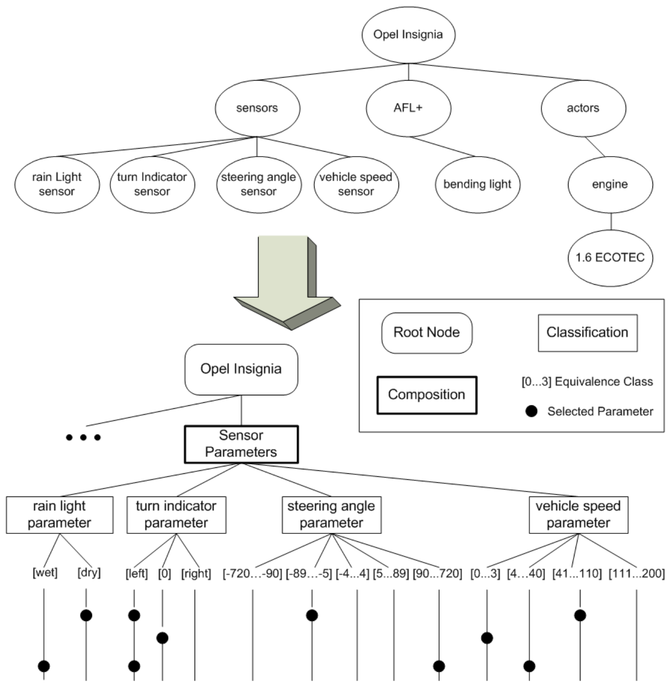

Fig. 2. Classification tree of selected product instance

In Fig. 2 three test cases for the CT of the running example are given. Each test case consists of the *equivalence classes* marked by the black dots in a horizontal line.

III. Related Work

In this section we focus on SPL-testing and the role FMs play in that context. We also present research activities related to software testing using the CT-method to generate test cases.

A. Testing SPLs

Miscellaneous approaches exist dedicated to SPL testing. Although there are no real FM-based testing approaches, many test methods partially use the FM of the SPL under test. A summary of methods is given in [28]. The authors distinguish between the following approaches to integrate testing into the development process of SPLs:

Product-by-product testing: In the majority of cases, each instance of an SPL is tested individually. This method is called product-by-product testing. Each product instance may be tested using the CT approach. However, this method is very exhaustive and normally not practicable. For instance, in the automotive field a single ECU like the engine control unit may be instantiated in at least several 10.000 different ways. Since a car may consist of up to 50 ECUs this results in millions of different configurations. An individual test of all configurations is feasible neither at the end of the assembly line nor during the development process. One method to improve product-by-product testing is to identify a minimal set of products which is representative for all other products. However, finding a minimal test set is an NP-complete problem [26]. Different heuristics are used to approximate a minimal test set. A very promising procedure is mentioned in [26]. The author uses a simplified version of the OVM approach. Members for the representative set of products are selected on the basis of requirements. That means that certain products are chosen so that all requirements are verified at least once. Optimization problems are formulated to produce a minimal test set.

In [20], an approach with a motivation similar to [26] was published. It uses dependencies derived from architecture and implementation to extend the FM of the SPL under test. Subsequently, pairwise testing that considers these dependencies is used to generate a representative set of products.

In both approaches the generation of test cases with appropriate input parameter values is out of scope.

Incremental Testing: In this approach, the first product derived from the SPL is tested individually, for instance by using the CT-methodology. With respect to the commonalities between the different products, the following products are tested using regression testing techniques [17]. The challenging part of this approach is to identify those parts of a product which stay unchanged and those which vary. The question arises if only the modified and added parts have to be tested. In addition, one has to find out if the modified parts can be tested individually or if some of them have to be tested in combination with unchanged components.

Reusable assets: The goal is to create reusable test assets. To ensure reusability, these assets are created during domain engineering. For each product these assets are customized during application engineering. Pohl et al. apply model based testing techniques. The authors use activity diagrams which are developed in domain engineering, based on requirements, and customized during application engineering to derive test cases. The so called ScenTED approach focuses on the reuse of test cases [25], [24], [22]. It uses substitution in order to derive configuration specific test cases. This is a very promising approach which we will take into account in our future work. However, test parametrization and the selection of proper values is still an open problem.

Division of responsibility: In [28], this method is described according to the levels of the development process, for instance the V-model. For example, unit testing can be executed during domain engineering and the other levels of the V-model could be executed in the application engineering phase.

All approaches confirm that testing is very challenging in SPL-engineering due to the high level of variability. However, the fact that the parametrization within an SPL leads to an additional degree of variability is often neglected as well as the fact that the borderlines between parameters that model variability at design time and parameters that are instantiated at runtime are often moving.

Furthermore, we are not aware of any SPL testing approach that combines SPL variant selection strategies with parameter value selection strategies as supported by CT. The tool pure::variants [23] offers e.g. the option to store parameter information in attributes of features, but gives no support for the definition of equivalence classes etc. The approach published in [19] is as far as we know the most similar SPL testing method to FMT. It uses one decision model to represent the variability of an SPL as well as to document input parameters of the regarded system. An integrated SPL variant and parameter value selection process is presented with rather promising evaluation results. Compared to FMT presented here the decision model is considerably less expressive and the introduced selection process is considerably less expressive than the algorithms for SPLs and the heuristics developed for CT-based black box testing purposes.

B. Black-Box Testing Products with CT

CTs are a widely used technique for test case generation. There are many publications related to this topic. A CT can be used to test a single product instance derived from the FM. The resulting product instance has specific actors and sensors, which interact complying to the specification. A CT splits the input domain of the sensors into relevant *equivalence classes*. These classes can be used to test an actor that is part of the product instance. Several approaches to improve and extend CTs like the Classification Hierarchy Table [3], the Classification Tree Transformer [27], Class Graphs [15], adding attributes [16], and introducing a time line [5] have been discussed. There are tools like the Classification Tree Editor for Embedded Systems (CTE/ES) that support the designer when trying to build a CT and its test table. The CTE/ES provides a graphical user interface that enables the designer to build a CT and the corresponding combination table.

All CT-based testing approaches we are aware of share the drawback that they deal with single product instances only and thus are only compatible with a product-by-product SPL testing approach.

IV. Unified Approach - FMT

As introduced in the preceding sections FMs and CTs have been used in software engineering for rather different purposes until now. An in-depth comparison of the modeling language constructs of FMs and CTs in the sequel reveals that both modeling approaches share a majority of their concepts. Therefore, this section is structured as follows: the first subsection starts with the in-depth comparison of FM and CT modeling constructs. Based on the results of this comparison, the following subsection then selects a minimal number of language constructs that correspond to a superset of the concepts of both FM and CT. Finally, the last subsection introduces a metamodel that captures the essential design of the new integrated FMT modeling language from an abstract syntax point of view.

A. FMT Language Constructs

In this section we develop language constructs which present a minimal list of abstract language concepts that is a superset of the concepts of FM and CT. Table II lists the constructs of FMs and CTs and the corresponding constructs for the unified approach: FMT. First, we have to define which kinds of nodes the FMT needs to support. CTs distinguish between *compositions*, *classification*, and *equivalence classes*. *Composition* and *classification* (line 1 in Table II) in CTs are nodes with child elements and *equivalence classes* (line 3 in Table II) are leafs in a CT. In FMs only *compound features* contain child elements and leafs are always *features*. To integrate CTs and FMs with regard to the different kinds of nodes, we need to examine the differences. *Compositions* are always mandatory and *classifications* are always optional nodes. As a consequence, we can use the *compound feature* known from FMs, and use cardinalities to state that the *compound feature* is either optional (0..1) or mandatory (1..1). Therefore, we adopt the *compound feature* for the FMT approach. The leafs of CTs and FMs differ obviously. On the one hand, an *equivalence class* represents a value or range of values of a parameter necessary for testing. On the other hand a *feature* in FMs can be any feature or property of an SPL. To realize an integration we need a leaf, which is capable to represent both information: a *feature* in general and a representation of values of parameters. Another difference, which we have to take into account for the integration, are the

	Feature Model	Classification Tree	FMT
1.	compound feature	composition, classification	compound feature
2.	Feature	none	Feature
3.	none	equivalence class	atomic feature
4.	mandatory feature (1..1)	composition	mandatory feature (1..1)
5.	optional feature (0..1)	classification	optional feature (0..1)
6.	(1..n) features	equivalence class	(1..n) features
7.	(n..m) features	none	(n..m) features
8.	cross-tree-constraints	only between equivalence classes	cross-tree-constraints
9.	feature attributes	none	feature attributes
10.	feature types	none	feature types

TABLE II
Language Constructs

cardinalities. In FMs *features* and *compound features* may have four different types of cardinalities to describe commonalities and variabilities. In CTs three of them are present: *composition* (1..1), *classification* (0..1), and *equivalence classes* (1..n). We adopt the missing cardinality of FM to ensure that the FMT is as expressive as the original FM. Furthermore, all four types of cardinalities may be used on all levels of an FMT tree in contrast to the much more restrictive approach of CT.

In addition, we allow cross-tree-constraints between all nodes of the FMT (line 8 in Table II). Finally, we adopt the node attributes and node types from feature modeling (line 9 and 10 in Table II).

B. Concept

We now describe the integration of FM and CT by means of a metamodel depicted in Fig. 3. We developed this class diagram on the basis of Table II. Please note that the depicted class diagram is a small extract of the complete FMT metamodel that illustrates the concept of the unified approach. It does e.g. not contain any information concerning the test case generation. The characteristics of the nodes of the FMT approach are described using the following classes: `Compound Feature`, `Feature`, and `Atomic Feature`. The last one can either be a feature representing its property in form of a literal or an interval. This is realized using the two classes `Literal` and `Interval`. These classes may represent:

- a value or a range of values of test parameters as known from equivalence classes in CTs
- a value or a range of values of parameters needed for product instantiation purposes
- the labels of features known from FMs.

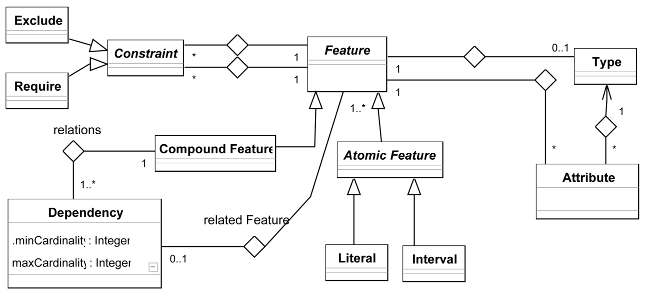

Fig. 3. Simplified metamodel of the unified approach

A node in the FMT can only be a `Compound Feature` or a leaf node (`Literal` or `Interval`), because the classes `Feature` and `Atomic Feature` are abstract. `Compound Feature`, `Literal`, and `Interval` inherit properties from `Feature`. All nodes in the FMT may have a `Type` and an arbitrary number of `Attributes`. In a `Type` the information of the data type of a feature is stored if existent. This is important, for instance, to distinguish between parameters of an integer or real data type. `Attributes` can store any information of the node. To obtain sufficient information to properly plan the test effort it is for example important for vehicle OEMs to embed information about realizing a feature in hardware or software. Additionally, we can apply constraints between `Features` and, therefore, also between `Compound Features` and `Atomic Features`. According to FMs we allow `Require` and `Exclude` constraints between all nodes. Since we want to adopt the cardinalities, describing the relation of features or groups of features to their parent node, we model a relation between `Features` and `Compound Features` by means of the class `Dependency`, which contains the cardinality constraints as attributes (minimum and maximum cardinality).
Fig. 4 depicts an object diagram that shows how dependencies and cardinalities are used to distinguish between the four different categories of subfeatures of Table II. A `Feature` is

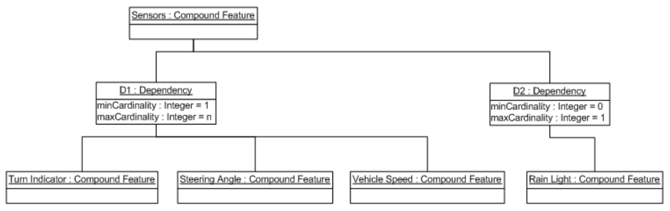
Sensors : Compound Feature
D1 : Dependency
minCardinality : Integer = 1
maxCardinality : Integer = n
D2 : Dependency
minCardinality : Integer = 0
maxCardinality : Integer = 1
Turn Indicator : Compound Feature
Steering Angle : Compound Feature
Vehicle Speed : Compound Feature
Rain Light : Compound Feature

Fig. 4. Object Diagram describing cardinality in FMT

always connected to its `Compound Feature` by a dependency. Therefore, `Dependencies` with different cardinalities can be placed beneath a `Compound Feature`.

V. Application

In this section we briefly describe the application of our FMT approach by generating test cases for our running example. Additionally, we discuss a tool, which is under development in our research groups.

A. Generating a test case

We demonstrate the ability to derive products and test cases on the basis of the FMT using our running example depicted in Fig. 5. We now derive two different products and present the handling of the parameters. The two products differ because they use different types of engines which results in different configurations. The 2.0 Turbo ECOTEC engine allows a vehicle maximum speed of 240km/h and, therefore, needs to be tested above 200km/h. The 1.6 ECOTEC engine is limited to 192km/h and does not require the test instance for vehicle speeds above 200km/h. Therefore, when testing the product containing the 1.6 ECOTEC engine, the *equivalence class* representing a speed range between 201 and 250 km/h has to be disregarded.
For both instances we derived some example test cases, which was done according to the well-known procedure described in section II-C. In the following we will describe the FMT in Fig. 5 which integrates the information of the FM of Fig. 1 and the CT of Fig. 2. Leaf nodes of the FMT, therefore, either represent basic features of the Opel Insignia SPL or *equivalence classes* of (runtime) parameter values. All non-leaf nodes of the FMT are inherited from the FM of Fig. 1, whereas leaf nodes are inherited from Fig. 1 and 2. The nodes of the new FMT have to be interpreted as follows:

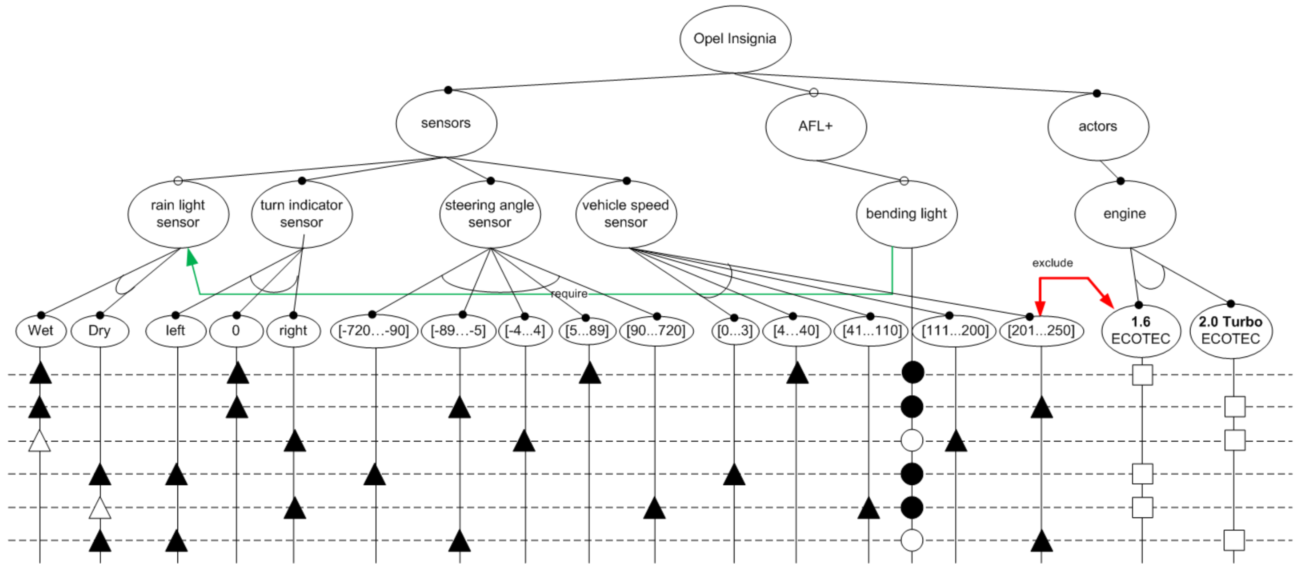

Fig. 5. FMT of the running example

- All nodes below the *sensors* node represent optional or mandatory Opel Insignia sensors together with their output parameter value definitions which are used as input parameters for control function test cases.
- All nodes below the *actors* node represent available actors (actuators) for the Opel Insignia such as different types of engines. Input parameters that control the behavior of these actors have been omitted due to lack of space. These missing parameters are output parameters of to be tested control functions. Specifications of their values can be used as oracles during the execution of test cases.
- A node like *bending light* represents a group of control functions that shall be tested. Again due to lack of space we do assume that *bending light* consists of a single function only which consumes output values of a subset of all sensors and produces input values for a subset of all actors defined in Fig. 6.

Node attributes (which are not visualized in Fig. 5) are used to distinguish these different categories of nodes of FMTs as well as for other purposes like the definition of additional node selection constraints (cf. metamodel of Fig. 3). Furthermore, Fig. 5 shows that the optional *bending light* functionality still requires the optional rain light sensor (as well as all other mandatory sensors of our SPL). The fact that the *bending light* control function also requires input values from the three other mandatory sensors is not visualized in Fig. 5. A more realistic FMT splits the *bending light* functionality into a number of subfunctionalities such as curve light, rain light, city light, or highway light which have to be tested separately. As a consequence we have to distinguish between features (functions, parameters, sensors, actors) that are part of a regarded product instance, but irrelevant for a specific test case, and features that are directly involved in a specific test case. The test case specifications in the lower part of Fig. 5 use black and white shapes for these two different purposes.

The selection of a specific variant and associated test cases is specified in a style adopted from CTs (due to lack of space the FMT metamodel of Fig. 3 does not cover these elements). Different shapes on vertical lines are used to distinguish between the following three cases, when a certain feature or parameter is selected:

1) square: selection at design time
2) circle: selection at installation time (flash time)
3) triangle: selection at runtime

The first two options correspond to the selection of a certain variant in an FM, whereas the third option corresponds to the selection of test cases with parameter values in a CT.

Regarding the bottom part of Fig. 5 we can see that the type of engine is of course selected at design time. The *bending light* functionality can be added or removed by firmware updates as long as the optional rain light sensor is present. The fact that all presented variants with their associated test cases do contain the optional rain light sensor is implicitly represented by the fact that all test cases possess a parameter value definition for this type of sensor (Wet or Dry). Black triangles represent Wet and Dry values that are actually used in a specific test case, whereas white triangles represent the fact that the *rain light sensor* is present and computes output values that are not needed as input for the just regarded test cases. Finally, Fig. 5 shows that four of the six depicted test cases are related to the functionality of the *bending light* (black bending light circles). In general, the distinction between black and white shapes related to other nodes of the FMT reflect the information which features and parameter values are relevant for which test case. It consists of nodes describing the product line and nodes for the test cases. Test case values are depicted in brackets.

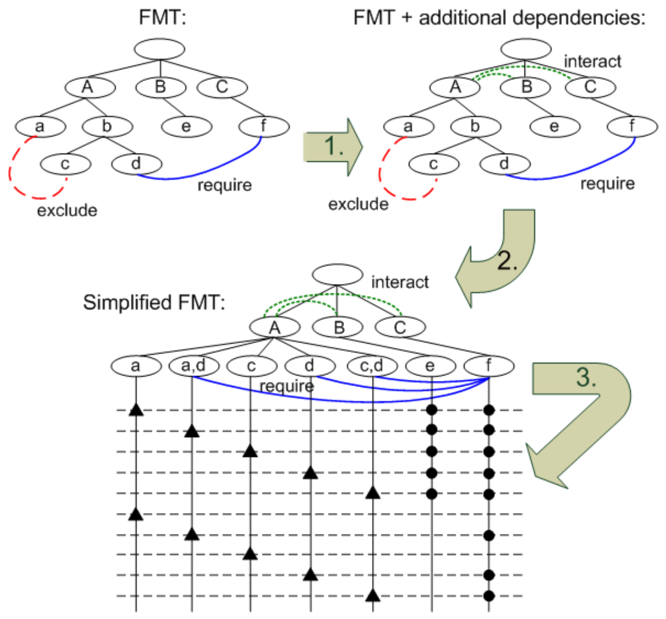

Fig. 6. Using FMT for SPL testing

B. Testing SPLs using FMT

In this subsection we describe in more detail how the FMT approach is used for SPL testing purposes, i.e. we sketch a methodology how to generate the bottom part of Fig. 5 automatically. For this purpose we describe the current state of development of our MoSo-PoLiTe project, which realizes the test methodology described in [20], but use FMT instead of FM. The generation of a representative set of products is subdivided into three steps [20].

1) Adding dependencies derived from system architecture and code as additional edges in the FMT.
2) Simplification of the FMT such that the resulting FMT uses a minimal set of modeling concepts.
3) Integrated selection of variants and runtime parameter values using the pairwise combination approach of [20]

Fig. 6 depicts the individual steps. We refer to [20] for further details. To use the FMT approach we are currently developing an FMT tool suite using MOFLON and GEF [10]. MOFLON is a meta-CASE tool for rapid development of CASE tools and tool adapters [2] developed at the Technische Universität Darmstadt. MOFLON supports model analysis, model transformation, and model integration for standard modeling languages like UML or domain-specific modeling languages. The abstract syntax of the new FMT modeling language as well as its static semantics rules are described using the OMG metamodeling approach supported by MOFLON, i.e. a combination of MOF 2.0 and OCL 2.0. Using this description as input we can generate an FMT model repository implementation together with all specified static semantics rules in Java. Furthermore, a generic text-oriented user interface for the definition of FMT instances is generated, too. The implementation of the user interface of a visual FMT editor on top of GEF is ongoing work. Model transformation rules (graph transformations) can be used to implement automatically executable FMT transformations. The last processing step of Fig. 6 has been implemented by modifying an existing Java implementation of a pairwise testing tool. The modified implementation in addition takes all kinds of dependencies between FMT nodes into account. The incorporation of CT-parameter value selection heuristics dealing e.g. with illegal or stress parameter *equivalence classes* (cf. Section II-C) is subject of ongoing research activities. The same is true for the first processing step depicted in Fig. 6. Right now dependencies that capture the fact that certain features or parameters interact from an implementation point of view have to be added manually. The adaption of ideas how to automatically derive this information from SPL architectures or code is also subject to future research activities.
According to the approach described in [20] we use the pairwise combination method to generate a representative set of products. At this point the major advantages of the FMT approach comes into play. We can generate the test cases for the selected products semi-automatically as described in the previous section. To summarize, we benefit from several advantages using FMT instead of FM. The FMT is more precise than FM when it comes to parametrization and for each product of the representative set we can derive the corresponding test cases semi-automatically. Likewise, the FMT describes parameters explicitly, therefore, we can consider dependencies which only occur between certain values of parameters.

VI. Conclusion and Future Work

Within this paper we have presented a new approach how to integrate SPL engineering with feature models (FMs) and black-box testing of system functions with CTs. Our motivation for this line of research is based on the fact that both FM- and CT-based methods are, e.g., well established in the automotive industry for embedded software system development purposes. On the one hand FMs are a suitable modeling method to describe commonalities and variabilities of an SPL. CTs, on the other hand, support the generation of test cases, using equivalence classes of parameter values of a regarded system function. We are not aware of any integrated approach that combines both methods for the generation of variants as test candidates and the associated test cases. Today, FM-based methods are first used to select one variant after the other; then for each of these variants a separate CT has to be defined which is then used to guide the related test case selection process. The integration of FM and CT in the form of the presented new FMT (Feature Model for Testing) method seems to be the perfect symbiosis of two very promising and widely used techniques. The key advantages are: (1) we use a single model for SPLs and test case generation, (2) approaches known from FMs and CTs can still be applied, and (3) generation algorithms of variants and test cases can be combined. We are currently working on different projects using FMT for SPL testing purposes including the BMBF project feasiPLe. Ongoing and future research activities will address the following problems:

- Checking the consistency of the FMT with respect to an additionally available specification of the behavior of the studied system. The FMT needs to incorporate the complete functionality described in the specification. This must be done before generating test cases.
- In embedded systems, our main application area, real-time constraints play an important role. Furthermore, test cases often have to be executed in a specified order and continuous parameter values reflecting physical properties of a controlled system and its environment have to be synthesized from sequences of selected discrete parameter values using well-known interpolation methods.
- Defining a measure of completeness for the generated test scenarios with respect to an additionally available system behavior specification is challenging, too [26]. Completeness checkers are useful to evaluate the generated test cases and to find gaps and corner cases.
- The nodes of the FMT have to be extended to describe the cost of creating configurations and requirements priority which is essential for complex SPLs in the automotive sector.
- We are currently applying our approach to several SPL scenarios. These are real world examples and FMTs generated randomly.
- We apply the pairwise testing approach [20] to generate a representative set of test cases and measure the coverage using appropriate and well known coverage metrics.

We focus on model checking techniques to address some of the problems stated above. Hence, another field of our research is the use of model checkers for test case generation and FMT validation. For this purpose, we need to develop a tool that is able to translate the FMT and the resulting test cases into boolean formulas, which can be evaluated by a model checker. Methodologies to translate an FM into a boolean formula have already been introduced in [12], [9].
Finally, we have to address the problem that realistic complete FMT models cannot be displayed directly as depicted in this paper. Various methods have to be developed how to collapse/hide irrelevant substructures efficiently.

Acknowledgment

This work was partially funded by the *feasiPLe* project, "Feature-driven, aspect-oriented and model-driven Software Product Line Development", Federal Ministry of Education and Research (BMBF), Germany.

References

[1] S. Alekseev, R. Tiede, and P. Tollkühn, "Systematic approach for using the classification tree method for testing complex software-systems," in *SE'07: Proceedings of the 25th conference on IASTED.* Anaheim, CA, USA: ACTA Press, 2007, pp. 261–266.

[2] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr, "Adapting FUJABA for Building a Meta Modelling Framework," in *Proc. 1st International Fujaba Days*, H. Giese and A. Zündorf, Eds., vol. tr-ri-04-247, 10 2003, pp. 29–34.

[3] T. Y. Chen, P. L. Poon, and T. H. Tse, "A new restructuring algorithm for the classification-tree method," in *STEP '99.* Washington, DC, USA: IEEE Computer Society, 1999, pp. 105–114.

[4] P. Clements and L. Northrop, *Software product lines: practices and patterns.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[5] M. Conrad and A. Krupp, "An extension of the classification-tree method for embedded systems for the description of events." *Electr. Notes Theor. Comput. Sci.*, vol. 164, no. 4, pp. 3–11, 2006.

[6] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multilevel configuration of feature models," *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005.

[7] ——, "Formalizing cardinality-based feature models and their specialization," in *Software Process: Improvement and Practice*, 2005, pp. 7–29.

[8] K. Czarnecki, S. She, and A. Wasowski, "Sample spaces and feature models: There and back again," in *SPLC '08.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 22–31.

[9] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *SPLC '07.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 23–34.

[10] G. E. F. (GEF). [Online]. Available: http://www.eclipse.org/gef/

[11] M. Grochtmann and K. Grimm, "Classification trees for partition testing," *Software Testing, Verification and Reliability*, vol. 1993, pp. 63–82, 1993.

[12] M. Janota and J. Kiniry, "Reasoning about feature models in higher-order logic," in *SPLC '07.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–22.

[13] B. D. Jules White and D. C. Schmidt, "Selecting highly optimal architectural feature sets with filtered cartesian flattening," *Journal of Systems and Software to appear*.

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

[15] K. R. P. H. Leung and W. Wong, "Towards a more efficient way of generating test cases: Class graphs," in *APAQS '00: Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00).* Washington, DC, USA: IEEE Computer Society, 2000, pp. 285–293.

[16] S. Lützkendorf and K. Bothe, "Attributierte Klassifikationsbäume zur Testdatenbestimmung," *Softwaretechnik-Trends*, vol. 23, no. 1, 2003.

[17] J. D. McGregor, "Testing a software product line," Tech. Rep. CMU/SEI-2001-TR-022, 2001.

[18] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval, "Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis," in *RE '07. 15th IEEE International*, 2007, pp. 243–253.

[19] E. M. Olimpiew and H. Gomaa, "Model-based testing for applications derived from software product lines," in *A-MOST '05.* New York, NY, USA: ACM, 2005, pp. 1–7.

[20] S. Oster and A. Schürr, "Architekturgetriebenes Pairwise-Testing für Software-Produktlinien," in *Workshop SE '09: Produkt-Variabilität im gesamten Lebenszyklus*, March 2009.

[21] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[22] K. Pohl and A. Metzger, "Software product line testing," *Commun. ACM*, vol. 49, no. 12, pp. 78–81, 2006.

[23] pure::systems. [Online]. Available: http://www.pure-systems.com

[24] S. Reis, A. Metzger, and K. Pohl, "Integration testing in software product line engineering: A model-based technique," in *FASE*, 2007, pp. 321–335.

[25] A. Reuys, E. Kamsties, K. Pohl, and S. Reis, "Model-based system testing of software product families," in *CAiSE*, 2005, pp. 519–534.

[26] K. Scheidemann, "Verifying families of system configurations," *Doctoral Thesis*, vol. TU Munich, 2007.

[27] I. Stürmer, H. Dörr, H. Giese, U. Kelter, A. Schürr, and A. Zündorf, "Das MATE Projekt visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen," Dagstuhl Seminar Modellbasierte Entwicklung eingebetteter Systeme, Januar 2007.

[28] A. Tevanlinna, J. Taina, and R. Kauppinen, "Product family testing: a survey," *ACM SIGSOFT Software Engineering Notes.*, vol. 29, pp. 12–12, 2004.