ITEA 3 Call 4: Smart Engineering

State-of-the-art report on requirements-based variability modelling and abstract test case generation

Project References

Project Acronym	XIVT		
Project Title	eXcellence In Variant Testing		
Project Number	17039		
Project Start Date	November 1, 2018	Project Duration	36 months
Project Manager	Gunnar Widforss, Bombardier Transportation, Sweden		
Website	https://www.xivt.org/		

Document References

Work Package	WP3 — Testing of Configurable Product
Task	T3.1 — Requirements-based variability modelling and abstract test case generation Task Lead — QA Consultants
Version	V1.0 Nov. 11, 2019
Deliverable Type	R (Report)
Dissemination Level	Public.

Summary

Executive summary: This report summarizes the state of the art in requirements-based variability modelling and abstract test case generation. It describes various formalisms which have been described in the literature for feature modelling and the modelling of variability, and presents methods for the generation of abstract test cases from various informal, semi-formal and formal description techniques. Finally, the report elaborates on available industrial and academic tools both for variability modelling and model-based test generation.

Summary: The present report presents a survey of variability modelling and abstract test case generation approaches for software projects. *Feature modelling* is introduced as the common denominator among variability modelling techniques, which differ in notation (e.g. UML, XML) as well as in support for hierarchical structures, constraints on feature combinations, representation of variation points and other semantic characteristics. In contrast, abstract test generation methods are distinguished by inputs, which may be source code, requirement specifications in formal notation or natural language, UML diagrams, etc; how are tests generated, either statically via input manipulation (which may involve Machine Learning or Natural Language Processing) or dynamically through e.g. model execution; and whether test suites can run automatically or must be performed manually. It is noticed there are remarkably few available references on test generation processes for highly-variant and configurable systems. The report concludes with a review of tools for variability modelling and system testing

Table of Contents

1 Introduction 4
2 Requirement-based Variability Modelling 6
2.1 Feature Modelling 6
2.2 Variability Modelling in UML 7
2.3 XML - Variability Exchange Language 8
2.4 Software Product Line Architecture Languages 9
3 Abstract Test Case Generation 11
3.1 Machine Learning-based Test Case Generation 11
3.2 Test Case Generation from Natural Language Descriptions 13
3.3 Test Case Generation from Formal Notations and Restricted or Controlled Natural Language 15
3.4 Test Case Generation from UML Models 16
3.5 Test Case Generation from Implementation Models (e.g., Simulink) 17
3.6 Test Case Generation from Tables and Data Models 18
3.7 Mutation Testing for Variant-intensive Systems 19
4 Combined Approaches 23
5 Tools 25
5.1 Variability Modelling Tools 25
5.2 Model-based Testing Tools 26
6 References 30

1 Introduction

Reducing the amount of repeated work is a common pursuit in many areas, software development among them. Copying libraries or source code from one project to the next is a simple way to reduce implementation costs, however this is often done in an ad hoc manner. In contrast, Software Product Line Engineering (SPLE) aims to optimize development by enabling systematic asset reuse among systems sharing common characteristics. It relies on *variability modelling* methods for the cataloging of features, their dependencies and conflicts across a whole product line (i.e. a family of systems), allowing the quick identification and application of related artifacts in new projects.

Another important problem in software development is ensuring systems are properly tested before deployment. This is commonly evaluated relative to some metric, for example, how many different scenarios or what proportion of system features are covered by the test suite. Theoretically the more comprehensive the coverage the better, however in practice this has to be balanced against time and cost considerations — in particular, the effort required to create the tests themselves. In this regard automatic generation of test case suites from project artifacts — be it requirements specifications, architecture models or source code — can greatly help conserve resources while at the same time ensuring the quality of test cases. This is especially desirable for software product lines, where depending on the number of features and variety of possible combinations it may be impractical to maintain test suites manually.

Variability modelling and test case generation can be seen as largely orthogonal activities — the former focus primarily inwards into the feature structure of a whole product line, while the latter mainly seeks to audit the outward behavior of specific features or systems. On the other hand, the context of a family of systems related by common features and a shared development history opens new frontiers for test optimization: avoiding retesting the same feature across several products, or evaluating how different combinations affect the performance of individual features, become pertinent considerations. Moreover there may be advantages to adopting variability modelling and test generation methods that work on the same artifact types (e.g. UML diagrams), especially if the outputs of the former are used as inputs to the latter. Therefore it is reasonable to evaluate both processes jointly.

Within the XIVT project, the ambition of WP3 is to develop a method and tool chain for model-based testing of configurable and multi-variant systems. There are two sides to this endeavor: Model-based testing, and modelling of configurable and multi-variant systems. Consequently, this state-of-the-art report consists of two parts: In the first, we discuss approaches to model variability in software-intensive systems. In software product lines, feature models are used to describe commonalities and differences between various products in the line. For UML, the unified modelling language, specific extensions for variability modelling such as CVL, the common variability language, have been suggested. In previous projects, also XML-based languages such as VEL, the variability exchange language, have been defined.

In the second part, we review the huge amount of literature on the generation of abstract test cases from various kinds of models. Some recent approaches are based on machine learning, where supervised and unsupervised learning algorithms are used to build models and extract useful patterns based on training data sets. Test generation from natural language descriptions is still in its infancy; most of the work in this area is focussed on informal use case descriptions, from which keywords and exetion patterns are extracted to generate semi-formal artifacts. Many authors also require that for the description of tests, natural language is only used in a controlled or sytactically restricted way; different approaches, varying by the degree of formalization of the input language have been proposed. Test generation from UML models is the most advanced area, where mostly class diagrams and state machines or sequence diagrams are used to model test cases. Implementation models such as Simulink diagrams or Scade descriptions have also been used to generate test cases; however, if both the implementation and the test suite are automatically generated from the same model, the fault-detection capability of the test execution is very limited. Dedicated test description languages such as TTCN-3 or data models have been suggested, and are described in this deliverable. To conclude the second part, we review the literature on mutation testing for variant-intensive systems, where faults are artificially seeded into the application to assess the effectiveness of the testing process.

The third part of this report deals with commercial and academic tools and workbenches. Again, we present the two areas: tools for variability modelling, and tools for test case generation from various types of models.

An extensive list of references concludes the report.

2 Requirement-based Variability Modelling

On a conceptual level, there are two different modelling languages at the core of most approaches: Feature Modelling (FM) and Orthogonal Variability Modelling (OVM). While in principle they share most of their concepts and in most instances even under the same name, there is one essential difference: Feature Modelling provides a hierarchical decomposition of a system into features, denoted as a feature tree and consisting of both common and variable parts. In contrast, OVM only considers the variable parts of a system through the notions of variation points and variants. In order to increase the interoperability of approaches based on these languages, a transformation between FM and OVM models is proposed in [1].

2.1 Feature Modelling

Feature modelling is a procedure applied in product line software engineering. Product line software engineering aims to systematically manage commonalities and differences in a product family, see [2] for an overview. The goal is to exploit the commonalities for an economic production, i.e., to develop products from core assets rather than one by one from scratch. It is founded on marketing; a product plan derived from a market analysis is the primary input.

Feature modelling is the identification and classification of commonalities and differences in a domain in terms of “product features”. A survey can be found in [3] and [4]. An explicit representation of a product’s features has several advantages. First, the concept of a feature as a common language enables effective communication among stakeholders from different fields. Second, feature-oriented domain analysis is an effective way to identify variability among different products in a domain. Finally, feature models can provide a basis for developing, parametrizing, and configuring various reusable assets and enables management and configuration of multiple products in a domain.

In feature oriented domain analysis (FODA, see [1]), a feature is defined as a prominent and distinctive user visible characteristic of a system/product. It is distinguishable from functions, objects, and aspects in terms of identification viewpoints: A feature is an externally visible characteristics that can differentiate one product from another, while the former characteristics are identifiable from internal viewpoints.

Product features are identified and classified in terms of capabilities, domain technologies, implementation techniques, and operating environments, see [3]. Features that are common between the different products are modelled as

mandatory, selectable features as optional. Features are alternative if no more than one feature can be selected for a product.
To capture structural or conceptual relationships between features, they are arranged in a feature diagram. To create a feature diagram, the selection of a domain and clarification of its boundaries is the first step. Then, the relevant features are identified, organized, and refined.
There are three types of relationships that can be represented in such a diagram: The *composed-of* relationship for a feature and its constituting subfeatures, the *generalization/specialization* relationship if a feature is a generalization of a subfeature, and the *implemented-by* relationship if one feature is necessary to implement the other. The feature model can be supplemented with composition rules or constraints that indicate mutual dependency and mutual exclusion relationships, constraining the selection from optional and alternative features.
There exist extensions of the basic feature model concept, one of the most common being cardinality based feature models, see, e.g., [5], where features or feature groups are being assigned an UML-like cardinality [a..b], implementing mandatory and optionality and allowing cloning of features or groups.

2.2 Variability Modelling in UML

UML is extensively used in the modelling community, and is a widely recognized standard. During requirements analysis, use case diagrams help to identify the actors and to establish the behavior of a system by means of use cases. UML defines a small set of relationships to structure actors and use cases: extend, include, and generalization, see, e.g. [6]. The extend relationship can model conditional optionality of features, but the other variability types cannot be modelled directly with use case diagrams. Therefore new use case modelling elements are needed to explicitly express all types of variability described above. There are straightforward approaches to expand the UML model in order to allow building feature models, see, e.g., [7], [8].

However, in [9], it is emphasized that concepts and features in feature modelling are abstracted from bearing semantic concepts. Hence, a simple extension of UML based on stereotypes of classes and their relationships is ruled out. They therefore propose to go to the UML meta-model in order to derive concepts and features from more abstract elements without unwanted semantics. By deriving from meta-classes and introducing certain stereotypes, they generate features and feature relationships.

In [6], the use case meta-model is extended by the new relationships option and alternative. If the base and extension use case are in the option relationship, the behavior of the base use case may be executed in the extension use case. If they are in the alternative relationship, exactly one alternative use case must be executed.
In [10], a similar approach is pursued by allowing use cases to be marked with an option stereotype, and the extend relationship to be marked with stereotypes alternative and specialization.
Nonetheless, in [6], it is mentioned that use cases can only model system behavior, not static structures and characteristics of systems, Therefore, a combination of feature graphs and extended use case diagrams is proposed.

Resulting from a standardization effort by the OMG, the Common Variability Language (CVL) was intended to provide consistent variability modelling capabilities for diverse Domain Specific Languages (DSL) independent from the specific language of the base model. According to informed but unverifiable sources, standardization activities were discontinued because of patent issues. However, CVL has already been presented at conferences [11], [12] and was subject to evaluation with regard to its usability in [13]. A detailed description of CVL as well as three examples of its application to different DSLs, i.e. ARI, TCL and UML, is given in [14]. Further, there is an adaption of CVL for specific industrial use cases resulting from the VARIES project and the DREAMS project [15], called Better Variability Results (BVR) and presented in [16].

Another approach to modelling variants in the context of systems engineering is presented in [17]. While technically not based on UML but SysML's customized subset of UML, it uses the UML profile mechanism to extend UML with variant modelling capabilities. For that purpose, the SYSMOD profile is used, containing a set of stereotypes that reflect concepts from OVM, which serves as the theoretical foundation.

2.3 XML - Variability Exchange Language

The Variability Exchange Language [18] has been developed within the German SPES_XT project as a standardized format to exchange information about variability between different stakeholders. It is an XML based data exchange format for different software engineering tasks. Tools for variant management frequently interact with artifacts such as model based specifications, program code, or requirements documents. This is often a two-way communication: variant management tools import

variability information from an artifact, and in return export variant configurations. For example, they need to gather information about the variation points that are contained in the artifact, need to know which variants are already defined, and then modify existing or define new variants.
The VariabilityAPI serves two purposes. First, it provides a generic description of the variation points that are contained in an artifact. Variation points may come in two flavors:

- Variation points may be locations in an artifact which are removed or set inactive in a binding process. This is implemented by defining a condition for each variation point.
- Variation points may be parameters. Such variation points provide expressions which are used by the binding process to assign a value to the parameter.

Variation points may also exhibit dependencies; for example a set of variation points may be designated as a set of alternatives, which means that all but one of them will be removed during the binding process (although the actual semantics of the binding process is beyond the scope of this concept).
Second, the VariabilityAPI can define specific variant configurations. In our context, a variant configuration is an assignment of fixed values to the conditions or expressions that are associated with variation points.
The VariabilityAPI defines a number of operations for exporting and importing variability information, where tools may only implement selected parts of the specification.

2.4 Software Product Line Architecture Languages

In their “Comparison of Software Product Line Architecture Design Methods” [19] Matinlassi describes and compares five different methods for the design of SPL architectures. Two categories of their evaluation framework are particularly interesting in the context of this report: Language, i.e. whether a method defines “a language or notation to represent the models, diagrams and other artifacts it produces”, and Variability, i.e. how a method supports variability expression.
Unfortunately, they cover this aspect only very briefly, but it becomes apparent, that the methods commonly either build on UML (QADA and KobrA) or make no reference to any specific SPLAL. For FORM it is mentioned, that it uses feature models known from FODA [20] as its conceptual foundation.

In “ADLARS: An Architecture Description Language for Software Product Lines” [21] Bashroush et. al. describe an ADL for SPLs with a three-level view on the product

line: The system level, a task level and a component level. A system is modeled as a collection of concurrently running tasks, interacting via event messages. Albeit not explicitly stated in the paper, tasks resemble services in a micro-service architecture. Task and components are instances of task and component templates. Templates can refer to specified features, which can then be used in their definition of interfaces and internal structure.

As can be inferred from its name, “LightPL-ACME” [22] is based on ACME, a generic architecture description language that provides a set of basic elements for the description of (software) architectures. After its publication in 1997, many of the provided concepts were incorporated in the UML standard, in particular *Component*, *Connector*, *Port* and *Role* and mappings between ACME and UML 2.0 were suggested in [23], [24]. In order to cater to the specific needs of product line engineering, LightPL-ACME adds the elements *ProductLine*, *Feature* and *Product*, as well as a mapping of architectural elements to features through a *MappedTo* relationship. Arising from the close resemblance of ACME and later versions of UML, it suggests itself to transfer the concepts from LightPL-ACME to a UML based approach to product line engineering.

3 Abstract Test Case Generation

3.1 Machine Learning-based Test Case Generation

Machine learning techniques are mainly based on detecting useful patterns (associations) in the data or training smart agents to solve the problems (tasks). Supervised and unsupervised learning algorithms are the machine learning techniques mainly involve building models and extracting useful patterns based on training data sets, while reinforcement learning algorithms are intended to teach the agent (learner) how to solve a problem through interaction with the environment in a trial and error way.
Regarding the application of supervised and unsupervised learning algorithms, various data on software testing activities, information on execution traces and coverage can be collected at different levels of details. The main point with regard to the application of these categories of machine algorithms is that how those collected data are used to address the existing challenges in software testing, such as automated test case generation. MELBA (MachinE Learning based refinement of BlAck-box test specification) [25], [26] is a machine learning-based process which basically uses C4.5 decision tree algorithm to generate expert-level test cases based on the existing test cases or through the high-level system specification. The proposed process uses an initial set of test cases which can be either the existing test cases or generated from high-level system specification. Afterwards, the output domain of the SUT is divided into equivalence classes and the input domain of the program is modelled based on Category-Partition (CP) categories and choices. Then, the initial set of test cases are converted to abstract test cases which are presented in terms of output equivalence classes and input pairs of categories and choices. C4.5 algorithm learns the association rules connecting the inputs (pairs of category and choice) to outputs (equivalence classes). The learned rules are analyzed using some heuristics to detect potential drawbacks in the extracted rules such as misclassification of test cases, absence of certain categories or choices (absence of certain input parameters). The refinement is done iteratively to lead to generation of expert-level test cases.

RUBAR: RUle-BAsed statement Ranking [27] is a machine learning-based technique to identify the statements containing a fault leading to a failure. In other words, it is intended to generate test cases leading to failures. Like MELBA, it works with generating the rules connecting the input CP categories and choices to output

equivalence classes, but particularly it uses additional information, i.e., defining some categories in terms of relations between the parameters, to generate effective test cases resulting in failures.
In [28] an ML-based technique, which aims at learning the program behavior through data on execution traces of the program, is proposed. It uses a classifier, which produces a map between the execution statistics like branch execution profiles and the class of program behavior like “pass” or “fail”. The classifier works based on an active learning paradigm and is trained incrementally on labelled data. The training instances are Markov models of program behavior, then a hierarchical clustering technique is applied to cluster the training instances based on the behavior labels. At the last step, a final classifier is built to make a map between the generated clusters and “pass” or “fail” behavior labels. However, this approach could be considerably effective at determining if a new test case is a useful or redundant one, but less effective in generating new effective test cases. This approach requires an initial test data generator and also construction of oracle.
Another example of ML-based approaches working based on execution trace to generate test cases or extension the set of test cases is [29]. The proposed technique in [29] uses some invariants which are reversed-engineered specifications from passing test cases (correct executions), to make an operational model of the program and then examine whether the automatically generated test cases are likely to be illegal, to produce normal operation or to reveal a fault. It presents a guided test input generation which is a classification based on the operational model.

The aforementioned primary studies were mostly describing some applications of numerical-based ML approaches to test case generation in software testing. In addition to numerical-based ML approaches, automaton learning is another type of ML technique. It is a symbolic approach conforming with eXplainable AI (XAI). All the features in the final learnt model in automaton learning is traceable backwards to the data values in the training set. Once the model constructed, it is subjected to model checker to verify the behavior requirements. Automaton learning-based testing is an ML-based black box testing. It combines machine learning with model checking and provides automated test case generation, test case execution and oracle construction. This approach makes a model of the SUT and refines it iteratively. An initial model of the SUT is generated, then the automaton learning algorithm refines the model into a more detailed one and sends the partial model to a model-checker. It is subjected to model checking against a temporal logic requirement. Any counter

example resulted from the model checking could be a potential test case revealing errors [30], [31].
The role of machine learning is important since they are particularly suitable techniques to black-box testing of complex systems such as cyber-physical systems in industrial domains like automotive, railway, telecommunication. They can handle large input spaces and complex behavior patterns of software systems. They can be used with different testing techniques like model-based testing and in many different domains of testing such as model-in-loop (MIL), software-in-loop (SIL) and hardware-in-loop (HIL) testing.

Generally, to summarize, the application of the family of supervised learning algorithms, it is worth noting that the supervised learning algorithms mainly aim at finding a model on training data set (including known input and output). Then, the extracted model could be used for prediction purposes. The supervised learning algorithms work based on classification or regression. Classification techniques build models on discrete data and to predict discrete output. Whereas regression techniques are used to produce/predict continuous output. A couple of common classification algorithms are k Nearest Neighbor (KNN), Support Vector Machine (SVM), neural networks, naiive bayes and decision trees and some of the common regression techniques are Gaussian process regression models, SVM regression, regression trees and generalized linear models [32]. The classification and regression algorithms could be used for classifying the large test data, filtering the redundant ones and generating the effective test cases.
Unsupervised learning explores data to find hidden patterns/structures. It is can be useful for reducing the dimensions of data. Cluster analysis techniques are the most common algorithms in the category of unsupervised learning. A number of common algorithms for hard and soft clustering are K-means, K-Medois, hierarchical clustering, self-organizing maps, Fuzzy c-means and Gaussian mixture models [32]. They could be used for different purposes in handling the test data and generating the effective test cases.

3.2 Test Case Generation from Natural Language Descriptions

This section of the document focuses on test case generation for variant-intensive software systems using Natural-Language Requirements Description as a main artifact.

Most of the literature in this area is focused on test case generation from requirements written in form of use cases. The use case diagram is an effective way to communicate the system under development with stakeholders. To enable the use cases to be used for expressing and analysis of product lines, extensions were made to the existing use case approaches.

Product Line Use Case (PLUC) [33] is one such example of extended use cases based on the use case in literature [34]. This extension explicitly focuses on modelling the variability with the use cases using tags such as alternative, parametric and optional. Relationships between use cases can also be modeled in PLUC by referring to them in natural language text. The use case elements are supposed to be enclosed in curly brackets and tags in square brackets. Product Line Use Case Test Optimization (PLUTO) [35], [36] is a test case generation approach developed to generate test cases from the PLUC specification. PLUTO extends the category partition method for test case generation. In category partition, the use cases are parsed and analyzed for high-level functions which can be tested in isolation. The category partition method relies on the input and choices (alternative scenarios) in the use case description provided by the tester to generate a test suite which exercises all the combination of the choices and input. The PLUTO also acts the same as category partition but in PLUTO high-level functions are the use cases and the choices are the variability tags embedded into PLUC. The PLUTO can also be used to derive test cases for a specific variant of the product line. PLUTO is also extended for the dynamic software product lines [37]. This is achieved by a new controlled natural language which defines the syntax for reuse, variation point, execution steps (like loops and conditions), and control statements (such as IF). The above controlled natural language allows the description of the dynamic software product lines and enables the generation of concrete test cases. This approach also uses the category partition method to derive test cases from the use cases and the approach reduces the manual testing efforts by 40 percent.

TaRGeT [38] is an open source tool for test case generation from constrained natural language use case specification. TaRGeT guides the end user in writing complete use cases by using an integrated grammar checker. The complete use case specification written in the form of use cases can then be transformed into system models and then test cases can be generated and selected. TaRGeT allows the selection of test cases based on coverage, specific requirements/use cases, diversity in test cases and test purpose.

Delta modelling is an approach to model the system's core assets and delta separately. This modular approach helps in modelling the variabilities and

commonalities explicitly in the software product lines. Test effort reduction approaches based on delta modelling which can significantly reduce the test efforts can be found in the literature. In many cases, these modelling approaches are not used, but requirements and test description in natural language are used to capture this information. Michael et al. [39] applied the delta-oriented testing approach on the requirements level. The approach relies on human input for classification of requirements for core and variable assets. The test cases linked to the core and variable assets requirements are then classified as Invalid, New, Re-usable and Re-test. The approach was found feasible in evaluation.
Unified Modeling Language (UML) use case diagram are also extended [40] to model the variabilities and commonalities to support test case generation. The extension requires contracts of the use cases to be written in the form of a proposed first-order language. To enable the generation of tests for a specific product, the approach uses the extended use case models to generate test objectives (high-level test description). The test objectives are incomplete and are not executable test cases. The approach requires implementation details in form of sequence diagrams with pre and post conditions written in Object Constraint Language (OCL) to generate executable test cases. The approach was applied to three products for evaluation.

3.3 Test Case Generation from Formal Notations and Restricted or Controlled Natural Language

In the literature, there are different approaches to the generation of abstract test cases from formal notations and controlled natural language. The different approaches mostly differ by the degree of formalization of their input (ranging from natural language over controlled natural language to formal notations) and the degree of automation and tool support.
[41] and [42] propose methods to derive test cases from requirements specified as behavior trees (not to be confused with the behavior tree concept currently being used in the AI community) as a means of formalization for natural language requirements. It is arguable whether the input of this approach is natural language or formal notation. On the one hand, any requirements-based approach likely starts with requirements formulated in natural language, which are then refined and formalized into an appropriate input format. From this point of view, the specific input to this approach are behavior trees.

While [41] takes the behavior tree as a given, [42] emphasizes on the process of formalizing requirements and the integration and specification of different modelling

artifacts along the way. The process then results in a Testing Behavior Tree. They further suggest transforming this Testing Behavior Tree into a UML state machine, which can then serve as input for test case generation methods such as described in the following section on test case generation from UML models. Unfortunately, they do not provide concrete transformation rules, but state these as a subject for future work. An attempt at such a transformation is made in [41] by interpreting the different aspects of Behavior Tree nodes as either states or events and thus deriving a state-based transition system from a Requirements Behavior Tree. They then suggest using depth-first-search in order to find test cases, which they link to the requirements in a trace matrix in order to facilitate test case prioritization and selection.

Another approach based on a more formalized description of the requirements, i.e. using controlled natural language, is presented in [43]. There, the requirements are formulated in SysReq-CNL, “a Controlled Natural Language (CNL) specially tailored for editing unambiguous requirements of data-flow reactive systems”. They are then transformed into an intermediary Software Cost Reduction (SCR) requirements format, serving as an input to test case generation using T-VEC.

Advancing from (controlled) natural language towards formal notations, [44] presents an approach based on UML state machines, the UML Testing Profile, and a formal notation, which serves as an input format for Microsoft Spec Explorer. While the UML aspect is discussed in the following section, the rest of this approach may be considered independent. The notation is based on C# and consists of a set of annotations that help to denote methods as part of the model, e.g. as rule methods. This is being used in order to write so called model programs, representing the behavior of the system under test. Whereas the model program expresses all legal sequences of actions and behaviors of the system under test, an additional “Cord” script defines constraints that are necessary for the generation of practical test cases. The Spec Explorer tool then builds an internal state-based representation of the model and explores it according to the provided configuration. This results in a set of test sequences.

3.4 Test Case Generation from UML Models

The Unified Modeling Language (UML) is a general-purpose, developmental, modelling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system. UML has gained a lot of attention

from researchers in the field of testing and the UML Testing Profile (UTP) was developed and standardized as a testing oriented extension to UML. UML offers different types of integrated diagrams such as use case, activity and sequence diagrams providing different perspectives of the model. Even though each type of UML diagram offers a view of the system, there are some limitations to each type of diagram in generating test cases. Each type offers unique features and is useful in certain scenarios but holds some limitation to generate test cases when they are used in a different scenario.
Model-Based Testing (MBT) uses the design models for software testing. It provides an efficient way of testing as it provides a methodology with a combination of source code and system requirements for software testers to test the software. In MBT, test cases are generated using the models. One of the major advantage of MBT is its ability to detect errors from the early stage of development and generating test case without being dependent on any implementation of the design.
From the literature review, we could observe that there are different methods available to generate test cases using different UML diagrams: that are activity diagram [45], sequence diagram [46], [47] and state chart diagram [48]–[50]. Every research has used its own method of test case generation using different UML diagrams. Some research studies use a single UML diagram where as other use a combination of UML diagrams. Most of the researches have used a technique of converting the UML diagram into an intermediate graph which is used to generate test cases. The graph is traversed based on coverage criteria and fault model to generate test cases. In some studies, the researchers use a combination of UML diagrams as sequence diagram and state chart diagram are not alone sufficient to generate test cases. Because of this limitation an integration of two UML diagrams is required in some cases.

3.5 Test Case Generation from Implementation Models (e.g., Simulink)

Implementation-based testing is usually performed at unit level to manually or automatically create tests that exercise different aspects of the program structure. To support developers in testing code, implementation-based test generation has been explored in a considerable amount of work [51] in the last couple of years from code coverage criteria to mutation testing.

Numerous techniques for automatic test generation based on code coverage criteria [52]–[56] have been proposed in the last decade. An example of such an approach is EvoSuite [52], a tool based on genetic algorithms for automatic test generation of

Java programs. Another automatic test generation tool is KLEE [53] which is based on dynamic symbolic execution and uses constraint solving optimization as well as search heuristics to obtain high code coverage. CompleteTest [57], a tool developed for IEC 61131-3 implementation models, is an example of a whitebox tool focusing on embedded software for generating timed sequences of inputs.

Another very popular implementation model is Simulink that has gained a lot of attention. Matinnejad et al. proposed an approach for generating test cases for Simulink models [58]. Ben Abdesalem et al., focused on the automated generation of test cases based on multi-objective search algorithms combined with artificial intelligence techniques for autonomous vehicles [59]. Other approaches use mutation testing, where the idea lies on generating faulty versions of the system and measuring the number of faults detected by the test suite [60]–[63]. Nevertheless, these approaches focused solely on generating test inputs for testing CPSs at a single test level (MiL or SiL), and require significant manual effort to concretise test cases for their execution in further test levels (e.g., HiL). Furthermore, the lack of test oracles makes the evaluation after the test cases are executed mostly manual, which is a non-systematic and error-prone process.

Simulink has, Model in the Loop (MIL), Software in the Loop (SIL), Processor in the Loop (PIL) and Hardware in the Loop (HIL) support with one test case (with back to back support) [64], [65]. Using code coverage settings, automatic test cases can be generated [66]. Using simulink test assessment blocks, test oracles are created and can verify the model performs as expected [67].

3.6 Test Case Generation from Tables and Data Models

Combination test generation techniques are test generation methods where tests are created by combining the input data values of the software based on a certain combinatorial strategy. Combinatorial testing can be helpful in the creation of test cases by generating certain combinations among parameter values. Several techniques have been proposed for combinatorial testing [68]–[72] in order to generate a test suite which covers the combinations of t parameter values at least once or in a certain interaction strategy (e.g., each-used, pairwise, t-wise, base choice). One of the most used combinatorial criteria is pairwise or 2-wise.

Recently, researchers have shown an increased interest in combinatorial software testing. There are a number of studies in which combinatorial testing tools and

techniques are being evaluated (e.g., [73]–[75]) in their use of combinatorial modelling and testing of industrial systems. Borazjany et al. [73] performed a case study in which they applied combinatorial testing on an industrial system by using ACTS to generate tests. The purpose of their study was to apply combinatorial testing to a real-world system and evaluate its effectiveness, as well as gaining experience and insight in the practical application of combinatorial testing, including the input modelling process. The tests are generated for testing both the functionality and the system robustness. Another study conducted at Lockheed Martin [75] reports about an introduction of combinatorial testing in a large organization. The applicability of combinatorial testing was evaluated by comparing different features contained in a set of combinatorial test tools and then applying these tools to real world systems. A number of pilot projects were conducted where ACTS was used as the primary tool. According to the results of this study, ACTS continued to be used by a number of teams once the pilot projects ended. Lei et al. [73] conducted a study to generalize the pairwise IPO strategy to t-way testing, and implemented this new strategy in FireEye. The tool was evaluated in terms of efficiency by using different system configurations. The experiments showed that the number of tests increased rapidly with the t-strength. FireEye and four other test generation tools were applied to a Traffic Collision Avoidance System (TCAS) module. The results show that FireEye performed considerably better in both size of test suites and generation time for higher strength t-way testing. Another approach used for embedded systems is timed-base choice criterion by Bergstrom et al. [76], in which timing information is incorporated in the input space used for test generation.

3.7 Mutation Testing for Variant-intensive Systems

Testing a software product line may be very tricky since the product line can be used to derive a combination of products. The products can grow significantly and testing all possible combination might not be feasible. Mutation testing is used to improve the test suite or evaluate existing test suite. The mutation testing approaches are based on mutation operators, meaning that they rely on injecting potential bugs (mutation operators) into the software (modified versions are called mutants) and see if the test suite detects them. If a test is failed to execute on a modified (potentially buggy) version of the software, the mutant is considered as “killed”. If the mutant is not killed, then either the test suite needs to be improved or the mutant is equivalent to the original software. Mutation testing is generally applied at the code-level but evidence of applying mutation testing at model-level can also be found in the literature. This

section of the document focuses on the use of mutation testing for software product lines.

Christopher et al. [77] applied the mutation testing on the feature model-level to assess the quality of test suites. Particularly, Christopher et al. proposed two mutation operators for feature models to evaluate if diverse test cases kill more mutants. This work uses the formula representation of the feature model and proposes two mutation operators, one operator replaces a literal in a clause and the other operator replace a clause with two newly generated clauses. Products are then derived from the original feature model, using SAT solvers. The derived products are then checked with the formulas of the mutants to calculate the mutation score. Diverse test cases were found to be more effective in killing the generated mutants.

Variability can be realized in different ways. One way to handle variability in C++ software systems is using the preprocessor constructs (such as #if def and #if defined etc.).

Mustafa et al. [78] proposed mutation operators for preprocessors-based variable systems. A proposed taxonomy is used to derive the mutation operators, which includes variability model faults (faults in features and their dependencies), variability mapping faults (faults in the mapping of code to the configuration) and domain artifact faults (faults in the code of feature interaction). The operators remove a random feature from the feature model, modify a feature dependency, add an additional condition to the feature, conditionally use traditional mutation operators, remove `#ifdef` blocks, and move the code around an `#ifdef` block. The authors then discussed how their mutation operators represent the real faults in the domain of variant-intensive software systems.

Dennis et al. [79] also proposed a variety of mutation operators on feature model level. Their mutation operators include creating a feature, moving features, setting the feature as optional, creating feature groups, moving feature groups and many others. These simple mutation operators reflect the small syntactic errors. The authors also addressed the problem of mutant selection using both random and similarity-based strategies. For similarity, the authors considered equality as a measure and considered the definition of it in the future. Tool support for the mutation sampling was provided and the approach was evaluated for applicability, effectiveness, and efficiency.

To deal with the huge number of combinations and to enable the use of model checkers for product lines, the Feature Transition Systems (FTS) was proposed [80]. Xavier et al. [81] based their work on the FTS and feature models to present a vision for efficient mutation testing. The mutation is supposed to be described in feature

diagrams and it can be replicated in the behavioral FTSs. After that, configurators can help in deriving the mutated version of concrete products and test cases can be executed on all mutants at the same time. Thus, the approach may speed the process of mutation analysis in the software product lines.
Mutation Testing is also used for the reduction of testing efforts in terms of test configurations. An approach [82] in the literature uses mutated feature models to evaluate test configurations being generated by an evolutionary algorithm. The evolutionary algorithm (1+1 evolutionary) is guided by a maximization fitness function for the mutation score. The proposed search-based approach for the generation of test configuration was evaluated for effectiveness (in terms of mutation score) and the number of test configuration generated were also compared to random strategy. On average, the search-based test configuration generation approach improved the mutation score by 68%.
Feature mapping models are used to model the features on system design (UML models). Mutation operators targeting the mapping models and UML models are also proposed in the literature [83]. The operators include deleting a mapping, deleting target elements of the mapping, inserting an extra element to the target of the mapping and flipping the features of the mapping. The operators for UML models focused on the transition of UML state machines. The mutation operators were applied to three software product lines case studies. Their test suites were analyzed against each mutation operator. The authors concluded that transition coverage should not be used as a criterion for test case selection in product lines because it might result in a low mutation kill rate.
It might be really hard to find a benchmark for evaluation of a testing strategy for a software product line. Work on generating a benchmark with potential bugs is reported in the literature [84]. The work uses the code of products and generates benchmark test cases using EvoSuite [85]. The products are then mutated using random and direct mutation, and test cases are executed to find the mutation score. This approach can be very useful when evaluating new testing strategies.

Mutation testing has been used also for embedded software by Enoiu et al. [86]. This technique is used for producing test cases using an automated test generation approach that operates using mutation testing for software written in IEC 61131-3 language, a programming standard for safety-critical embedded software, commonly used for Programmable Logic Controllers (PLCs). This approach uses the UPPAAL model checker and is based on a combined model that contains all the mutants and the original program.

4 Combined Approaches

In [87], [88] the model-based software product line testing (MoSo-PoLiTe) concept is introduced, providing a method for combinatorial SPL testing and its implementation by a tool chain.
State charts are used as a test model for the SPL, whose states and transitions are then mapped to features of a feature model. This model is flattened and transformed into a constraint satisfaction problem, and a greedy ad-hoc algorithm, the AETG (Automatic Efficient Test Case Generator System, see [89]) is applied in order to generate a set of test cases fulfilling the desired combinatorial coverage criteria.

The following problems of testing SPL are identified in [90], and approached with an algorithm defined in [91]: The time provided for software tests is inflexible, test cases have to be provided and executed in a restricted time frame. The testing of invalid software configurations of an SPL naturally leads to errors. Hence, a form of constraint management is necessary. Also, test sets often lack in measurable test coverage criteria if they are provided manually and chosen from experience.
The authors provide methods to identify the variation points of a SPL and construct a variability feature model. Subsequently, they present an algorithm to automatically generate a – with regards to a given amount of time - minimal set of configurations covering all pairwise interactions between the features and satisfying all constraints. For this purpose, the feature model is converted into a constraint model: A matrix with columns representing the features and rows representing possible configurations, where the number of rows is dynamic. Three types of constraints can be modelled: Inheritance links that represent hierarchical relations between features (opt, and, or, xor), cross-tree links (mutex, requires) and the constraints that enforce a certain level of coverage (e.g. pairwise coverage). A constrained anytime minimization algorithm is applied. The result is a so-called mixed level covering array that represents test cases sufficient for the desired level of coverage, of minimal size with respect to the assigned generating time.
The approach was tested on a case study of a video conferencing SPL. Timewise, the combination of constructing the feature model and performing the algorithm generated test sets 7 times faster than the manual approach. Those test sets were 17% smaller and did not contain any invalid test cases. They provided full pairwise coverage, compared with around 19% pairwise coverage of the manual approach.

Both approaches presented above have the disadvantage that the notion of variability is very narrow: It is basically synonymous with configurability. Furthermore,

they can only be applied if the variability is fixed from the beginning. The case that variants are added one after another is not covered, although it is a relevant scenario in industry.

Delta modelling [92] takes the model of a core product as a base and models variants of it by expressing the addition, removal and modification of features by deltas. Here, in principle, multiple deltas can implement a feature, and, vice versa, one delta can be used to implement multiple features. So, instead of having to deal with a full model for each variant, the latter can be represented by a (much smaller) set of deltas.
[93] investigate the prioritization of variants for delta models. As a measure of distance of two variants they choose the Hamming distance, i.e., up to scaling, the number of deltas resp. features in which the two variants differ. As a first product to test, the core product (preferably the largest/most error prone product) is chosen. Further variants to test are selected by computing the minimum distance to the set of already tested products for all possible not-yet-tested candidates. The candidate with the largest minimum distance is then chosen as the next test object.
The error-finding capabilities are measured against a random approach and the MoSo- PoLiTe algorithm, both of which they outperform.
This approach also has limitations. Using the Hamming distance does not lead to optimal feature coverage properties, as removed features are weighted as much as additional features. Nonetheless, in [94], it is mentioned that the absence of features can also trigger errors.
Also, measuring the distance between a variant and a set of variants by taking the minimal distance to any element of the set does not lead to optimal feature coverage. A variant might have a relatively large distance to all elements of the set, yet, it might not contain any features that are not in the set yet.

5 Tools

Following the line of discussion from the previous two chapters, we describe two types of tools: First, we review tools with the main focus on variant modelling and management. Then, we review available tools for the generation of abstract test cases from different types of models.

5.1 Variability Modelling Tools

5.1.1 pure::variants

Pure::variants by pure.systems is a major product for variant management. It allows to generate feature models, define instantiations, and link the features to many different types of artefacts in the development process, including requirements, software modules, and test cases. It contains connectors to DOORS, Rational Rhapsody, Enterprise Architect, Magic Draw, Simulink, and other software engineering tools. A free community edition, which is limited in the size of the models, is available.

5.1.2 BigLever Gears

Gears by BigLever Inc. is one of the main competitors of pure::variants and has a large overlap in functionality. The main features of Gears are:

- Feature models and product feature profiles: The modelling language used by Gears differs in large parts from the one used by pure::variants and models are not interchangeable. For any product in the portfolio a profile is created, which unambiguously identifies the product by its features according to the feature model.
- Configuration and variance points: In Gears, variance points are „intelligent casings for product variability". Gears provides a language to describe variance points such that they can be configured according to a product profile. A product configurator allows to assemble the required components such as software modules, requirements and tests, for the individual products of the SPL.
- Self-contained IDE: In contrast to pure::variants, Gears offers a "console" for the portfolio-specific development aspects. This ensures a wide degree of independence from external influences. Similar to pure::systems there are connectors to established software development environments. In particular, there is a tight integration between Gears and Rhapsody for model-based development.

Additionally, there is a “bridge” to DOORS, Rational Quality Manager and Clear Case.

5.1.3 Eclipse FeatureIDE, Eclipse EMF Feature Model/Feature Diagram Editor

The FeatureIDE project [https://featureide.github.io] aims to provide a complete Software Product Line Engineering solution, implemented as a collection of Eclipse plugins. The toolchain covers the whole engineering lifecycle, from domain and requirements analysis all the way to feature implementation, product generation and testing.
The code base is licensed under L-GPL 3, and as of 2019 still sees active development — the latest version 3.6 was released August 30. Software packages can be installed on an existing Eclipse environment or downloaded alongside the base IDE as a self-contained application. Online documentation is sparse, but the *Mastering Software Variability with FeatureIDE* book (written for version 3.3) seems to be a satisfactory guide.
FeatureIDE supports variability modelling through Feature Models, and deltas at programming level, but automated test generation does not seem to be included yet — though automated *configuration* generation and integration to JUnit are touted as features. The toolchain is also extensible, with a number of external projects having been built on top of it. All factors considered, FeatureIDE seems at first glance a comprehensive SPLE solution, and merits closer examination as a possible basis for implementing a test generation tool for variant systems.

5.1.4 Other approaches

Wikipedia [https://en.wikipedia.org/wiki/Feature_model] lists 27 tools supporting the editing and/or analysis of feature models (including the ones mentioned above). Many of these tools are outdated, no longer supported, or not well-integrated into modern development processes.

5.2 Model-based Testing Tools

5.2.1 Expleo Modica

MODICA is a test generation tool that employs a usage model as a source. A usage model describes a system from the perspective of its usage, with states as its nodes and state transitions as its edges. To this model, the test case generation algorithm can be applied that aims to comply with specified coverage criteria, using the smallest possible number of test steps in the test cases. Coverage criteria can be given by requirements, the request that (certain) states, state transitions or paths are

covered, or the choice of special test sequences that are otherwise hard to reach. In MODICA, there is also a variant handling available that allows to specify test generation strategies for different variants of the usage model. See [95] for details.

5.2.2 Expleo Testona

TESTONA is a test case generation tool based on the classification tree method [96]. The classification tree of a test object, e.g. a product, specifies its functionality by dividing it into its aspects/parameters (classifications) and their specifications/parameter values (classes). Furthermore, constraints on the combination of these classes can be expressed in dependency rules.
Possible test cases are then admissible combinations of classes from different classifications. For the generation of a suite of test cases, there are different modes available that represent different levels of test coverage: In *minimal combinatio*n, every class appears in at least one test case, in *pairwise* and *threewise combination*, the same holds for every pair and triple of classes, respectively. In *maximal combination*, all possible test cases are generated.
In addition, it is possible to weight the classes depending on their frequency or error risk and consequently obtain a prioritization of test cases. A variability management is built in, allowing the user to specify variants from the generic model and apply TESTONA-applications specifically to them. See [97] for details.

5.2.3 Expleo Meran

MERAN is an integration tool for requirement management that also supports variant management. It allows the creation of generic entities of requirements or test specifiations, in a way that their properties are fragmented in small units. Once a specific variant is chosen, the requirements or test specifications can be adapted by choice of parameters or text segments. See [98] for details.

5.2.4 Ifak MBT Creator

The MBTCreator is a tool suite that combines various functionalities for model-based testing and test prioritization. MBTCreator offers editors and a graphical user interface for a toolchain that covers all steps from requirements to test case generation, prioritization and execution.
In a first step, the tool features methods for formalization of requirements using a notation language, the IRDL (Ifak Requirement Description Language). A state machine can then be generated from formalized requirements, which models all behavior of the SUT as described in the requirements. Abstract test cases are then generated for the state machine using one of several coverage based test goals.

Additionally, MBTCreator features a test prioritization method which prioritizes test cases via a combination of model-based cluster analysis and a requirements-based evaluation procedure to enable optimization of the test execution process.
To execute the abstract test cases, a test manager has recently been included into MBTCreator. It evaluates the abstract test cases and derives a suitable test program. To run the test cases for the SUT, a test adapter has been developed, which enables communication between the MBT Creator and the SUT. Currently, communication via OPC UA and Shared Memory is supported.

5.2.5 SaFRel: ML-based performance (stress) test case generation (RISE)

SaFReL is a self-adaptive fuzzy reinforcement learning-based performance (stress) testing framework which makes the tester agent able to learn the optimal policy for generating stress test conditions without having a performance model of the system. Finding the performance breaking point of the software under test (SUT), at which the system functionality breaks, or the performance requirements are not satisfied anymore, is the main objective of the stress testing in this framework. In stress testing, providing extreme (stress) test conditions involves changing (manipulating) the platform- and workload-wise factors affecting the performance. The current prototype mainly focuses on stress testing regarding manipulating the resource availability.
It assumes two learning phases, i.e., initial and transfer learning. First, it learns the optimal policy through the initial learning and then reuses the learnt policy for observed software systems with performance sensitivity analogous to already observed ones while still keeping the learning running in the long-term. The current prototype uses a performance prediction module to estimate the effects of the applied actions. It gets the initial resource utilization and nominal response time of the system, which have been measured in an isolated, contention free execution environment, and the performance sensitivity indicators as inputs. This framework could be executed on a virtual machine containing the SUT, and it would be augmented by an actuator doing the resource scaling within the VM. In this case, it will be able to use the (resource) monitoring tools (services) like Percepio Tracealyzer to receive the status data.

5.2.6 IntegrationDistiller: Automating generation of Integration Test Cases (RISE)

IntegrationDistiller is a solution and tool for automatic generation of integration level test cases for object-oriented .NET applications. It has a static analysis engine, implemented using Roslyn .NET compiler platforms APIs, which can automatically

parse the source code and based on the concept of coupling-based testing, identifies different coupling relationships that can exist between classes, methods and their parameters, and generate test paths to cover different interaction scenarios. More details can be found in http://www.es.mdh.se/pdf_publications/5282.pdf.

5.2.7 Other

There is a number of tools which have been developed, but are either out of date, no longer maintained, or otherwise unavailable. We mention them mostly for completeness, since some of the underlying ideas are (still) relevant for XIVT:

- Imbus Variant Test, https://www.imbus.ca/testbench/variant-test/
- IT Power Contino Prova, https://itpower.de/de/produkt/continoprova/
- T-VEC, http://www.t-vec.com/
- vEXgine, http://caosd.lcc.uma.es/vexgine/
- Fokus!MBT
- Behavior Engineering Support Environment

6 References

[1] F. Roos-Frantz, D. Benavides, and A. Ruiz-Cortes, “Feature Model to Orthogonal Variability Model Transformation towards Interoperability between Tools,” p. 9.
[2] L. M. Northrop and P. C. Clements, “A Framework for Software Product Line Practice, Version 5.0,” p. 258.
[3] K. Lee, K. Chul Kang, and J. Lee, “Concepts and Guidelines of Feature Modeling for Product Line Software Engineering,” presented at the 7th International Conference on Software Reuse: Methods, Techniques and Tools, 2002, pp. 62–77.
[4] K. C. Kang and H. Lee, “Variability Modeling,” in *Systems and Software Variability Management*, R. Capilla, J. Bosch, and K.-C. Kang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 25–42.
[5] K. Czarnecki and C. H. P. Kim, “Cardinality-Based Feature Modeling and Constraints: A Progress Report,” p. 9.
[6] T. von der Maßen and H. Lichter, “Modeling Variability by UML Use Case Diagrams,” in *Proceedings of International Workshop on Requirements Engineering for Product Lines*, Essen, Germany, 2002.
[7] M. Clauß, “Modeling variability with UML,” p. 5.
[8] A. Bragança and R. J. Machado, “Deriving Software Product Line’s Architectural Requirements from Use Cases: an Experimental Approach,” p. 15.
[9] V. Vranic, “Integrating Feature Modeling into UML,” p. 13.
[10] S. Azevedo, R. J. Machado, A. Bragança, and H. Ribeiro, “On the refinement of use case models with variability support,” *Innovations Syst Softw Eng*, vol. 8, no. 1, pp. 51–64, Mar. 2012.
[11] F. Fleurey, “The Common Variablility Language (CVL),” p. 28.
[12] Ø. Haugen, A. Wąsowski, and K. Czarnecki, “CVL,” in *Proceedings of the 17th International Software Product Line Conference on - SPLC ’13*, New York, New York, USA, 2013, p. 277.
[13] J. Echeverria, J. Font, C. Cetina, and O. Pastor, “Usability evaluation of variability modeling by means of common variability language,” in *CEUR Workshop Proceedings*, 2015.
[14] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen, “Adding Standardized Variability to Domain Specific Languages,” in *2008 12th International Software Product Line Conference*, 2008, pp. 139–148.
[15] O. Haugen, “Variability Modelling in DREAMS,” p. 21.
[16] Ø. Haugen and O. Øgård, “BVR – Better Variability Results,” in *System Analysis and Modeling: Models and Reusability*, vol. 8769, D. Amyot, P. Fonseca i Casas, and G. Mussbacher, Eds. Cham: Springer International Publishing, 2014, pp. 1–15.
[17] T. Weilkiens, *Variant Modeling with SysML*. Leanpub, 2014.
[18] “Variability Exchange Language | Variability-Exchange-Language.” [Online]. Available: https://www.variability-exchange-language.org/. [Accessed: 30-Jul-2019].

[19] M. Matinlassi, “Comparison of software product line architecture design methods: COPA, FAST, FORM, KobrA and QADA,” in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 127–136.
[20] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study:,” Defense Technical Information Center, Fort Belvoir, VA, Nov. 1990.
[21] R. Bashroush, T. J. Brown, I. Spence, and P. Kilpatrick, “ADLARS: An Architecture Description Language for Software Product Lines,” in *29th Annual IEEE/NASA Software Engineering Workshop*, Greenbelt, MD, USA, 2005, pp. 163–173.
[22] E. Silva, A. L. Medeiros, E. Cavalcante, and T. Batista, “A Lightweight Language for Software Product Lines Architecture Description,” in *Software Architecture*, vol. 7957, K. Drira, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 114–121.
[23] M. Goulão, “Bridging the gap between Acme and UML 2.0 for CBD,” p. 5.
[24] C. Mokarat and W. Vatanawood, “UML Component Diagram to Acme Compiler,” in *2013 International Conference on Information Science and Applications (ICISA)*, 2013, pp. 1–4.
[25] L. C. Briand, “Novel Applications of Machine Learning in Software Testing,” in *2008 The Eighth International Conference on Quality Software*, 2008, pp. 3–10.
[26] L. C. Briand, Y. Labiche, and Z. Bawar, “Using Machine Learning to Refine Black-Box Test Specifications and Test Suites,” in *2008 The Eighth International Conference on Quality Software*, 2008, pp. 135–144.
[27] L. C. Briand, Y. Labiche, and X. Liu, “Using Machine Learning to Support Debugging with Tarantula,” in *The 18th IEEE International Symposium on Software Reliability (ISSRE ’07)*, 2007, pp. 137–146.
[28] J. F. Bowring, J. M. Rehg, and M. J. Harrold, “Active Learning for Automatic Classification of Software Behavior,” p. 11.
[29] C. Pacheco and M. D. Ernst, “Eclat: Automatic Generation and Classification of Test Inputs,” in *ECOOP 2005 - Object-Oriented Programming*, vol. 3586, A. P. Black, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 504–527.
[30] H. Khosrowjerdi and K. Meinke, “Learning-based Testing for Autonomous Systems Using Spatial and Temporal Requirements,” in *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, New York, NY, USA, 2018, pp. 6–15.
[31] K. Meinke, “Learning-Based Testing: Recent Progress and Future Prospects,” in *Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, A. Bennaceur, R. Hähnle, and K. Meinke, Eds. Cham: Springer International Publishing, 2018, pp. 53–73.
[32] “Machine Learning with MATLAB.” [Online]. Available: https://www.mathworks.com/campaigns/offers/machine-learning-with-matlab.html. [Accessed: 12-Aug-2019].
[33] A. Bertolino, A. Fantechi, S. Gnesi, and G. Lami, “Product Line Use Cases: Scenario-Based Specification and Testing of Requirements,” in *Software Product*

Lines, T. Käköla and J. C. Duenas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 425–445.

[34] A. Cockburn, *Writing Effective Use Cases*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[35] A. Bertolino and S. Gnesi, "PLUTO: A Test Methodology for Product Families," in *Software Product-Family Engineering*, vol. 3014, F. J. van der Linden, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 181–197.

[36] A. Bertolino and S. Gnesi, "Use Case-based Testing of Product Lines," in *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2003, pp. 355–358.

[37] I. L. Araújo, I. S. Santos, J. B. F. Filho, R. M. C. Andrade, and P. S. Neto, "Generating test cases and procedures from use cases in dynamic software product lines," in *Proceedings of the Symposium on Applied Computing - SAC '17*, Marrakech, Morocco, 2017, pp. 1296–1301.

[38] F. Ferreira, L. Neves, M. Silva, and P. Borba, "TaRGeT: a Model Based Product Line Testing Tool," p. 7.

[39] M. Dukaczewski, I. Schaefer, R. Lachmann, and M. Lochau, "Requirements-based delta-oriented SPL testing," in *2013 4th International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*, 2013, pp. 49–52.

[40] C. Nebut, Y. L. Traon, and J.-M. Jezequel, "System Testing of Product Lines: From Requirements to Test Cases," in *Software Product Lines*, T. Käköla and J. C. Duenas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 447–477.

[41] Y. I. Salem and R. Hassan, "Requirement-based test case generation and prioritization," in *2010 International Computer Engineering Conference (ICENCO)*, 2010, pp. 152–157.

[42] M.-F. Wendland, I. Schieferdecker, and A. Vouffo-Feudjio, "Requirements-Driven Testing with Behavior Trees," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 2011, pp. 501–510.

[43] G. Carvalho *et al.*, "Test case generation from natural language requirements based on SCR specifications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing - SAC '13*, 2013, p. 1217.

[44] M.-F. Wendland, A. Hoffmann, and I. Schieferdecker, "Fokus!MBT: a multi-paradigmatic test modeling environment," in *Proceedings of the workshop on ACadeMics Tooling with Eclipse - ACME '13*, New York, New York, USA, 2013, pp. 1–10.

[45] C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic test case generation for UML activity diagrams," in *Proceedings of the 2006 international workshop on Automation of software test - AST '06*, New York, New York, USA, 2006, p. 2.

[46] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado, "Test case generation by means of UML sequence diagrams and labeled transition systems," in *Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics*, 2007.

[47] M. Sarma, D. Kundu, and R. Mall, “Automatic Test Case Generation from UML Sequence Diagram,” in *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, 2007, pp. 60–67.
[48] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, “Generating test data from state-based specifications,” *Softw. Test. Verif. Reliab.*, vol. 13, no. 1, pp. 25–53, Jan. 2003.
[49] S. Gnesi, D. Latella, and M. Massink, “Formal test-case generation for UML statecharts,” in *Proceedings. Ninth IEEE International Conference on Engineering of Complex Computer Systems*, 2004, pp. 75–84.
[50] P. Samuel, R. Mall, and A. K. Bothra, “Automatic test case generation using unified modeling language (UML) state diagrams,” *IET Software*, vol. 2, pp. 79–93, 2008.
[51] A. Orso and G. Rothermel, “Software Testing: A Research Travelogue (2000–2014),” in *Proceedings of the on Future of Software Engineering*, New York, NY, USA, 2014, pp. 117–132.
[52] G. Fraser and A. Arcuri, “EvoSuite: Automatic Test Suite Generation for Object-oriented Software,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, New York, NY, USA, 2011, pp. 416–419.
[53] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” p. 14.
[54] N. Tillmann and J. de Halleux, “Pex-White Box Test Generation for .NET,” in *TAP*, 2008.
[55] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, “Combined static and dynamic automated test generation,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA ’11*, Toronto, Ontario, Canada, 2011, p. 353.
[56] Y. Kim, Y. Kim, T. Kim, L. Gunwoo, Y. Jang, and M. Kim, “Automated unit testing of large industrial embedded software using concolic testing,” presented at the 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings, 2013, pp. 519–528.
[57] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson, “Automated test generation using model checking: an industrial evaluation,” *Int J Softw Tools Technol Transfer*, vol. 18, no. 3, pp. 335–353, Jun. 2016.
[58] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann, “Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior,” *IEEE Transactions on Software Engineering*, vol. PP, pp. 1–1, Mar. 2018.
[59] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, “Testing advanced driver assistance systems using multi-objective search and neural networks,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, Singapore, Singapore, 2016, pp. 63–74.
[60] R. Rana, M. Staron, C. Berger, J. Hansson, M. Nilsson, and F. Törner, “Increasing Efficiency of ISO 26262 Verification and Validation by Combining Fault Injection and Mutation Testing with Model Based Development,” 2013.

[61] L. Thi My Hanh, T. Khuat, and B. Nguyen, “Mutation-based Test Data Generation for Simulink Models using Genetic Algorithm and Simulated Annealing,” *International Journal of Computer and Information Technology (IJCIT)*, vol. 03, pp. 763–771, Jul. 2014.
[62] Y. Zhan and J. A. Clark, “Search-based mutation testing for *Simulink* models,” in *Proceedings of the 2005 conference on Genetic and evolutionary computation - GECCO ’05*, Washington DC, USA, 2005, p. 1061.
[63] Y. Zhan and J. A. Clark, “A search-based framework for automatic testing of MATLAB/Simulink models,” *Journal of Systems and Software*, vol. 81, no. 2, pp. 262–285, Feb. 2008.
[64] “Speedgoat Hardware Support for Real-Time Simulation and Testing from Simulink Real-Time.” [Online]. Available: https://se.mathworks.com/hardware-support/real-time-simulation-and-testing-with-speedgoat-hardware.html. [Accessed: 24-Oct-2019].
[65] “Create and Run a Back-to-Back Test - MATLAB & Simulink - MathWorks Nordic.” [Online]. Available: https://se.mathworks.com/help/sltest/ug/create-and-run-a-back-to-back-test.html. [Accessed: 24-Oct-2019].
[66] “Automatically Create a Set of Test Cases - MATLAB & Simulink - MathWorks Nordic.” [Online]. Available: https://se.mathworks.com/help/sltest/ug/generate-test-cases-from-model-components.html#mw_590f3b16-953e-423e-b049-b11bfe19330e. [Accessed: 24-Oct-2019].
[67] “Assess simulation testing scenarios, function calls, and assessments - Simulink - MathWorks Nordic.” [Online]. Available: https://se.mathworks.com/help/sltest/ref/testassessment.html. [Accessed: 24-Oct-2019].
[68] C. Nie and H. Leung, “A Survey of Combinatorial Testing,” *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011.
[69] D. R. Kuhn, R. N. Kacker, and Y. Lei, “SP 800-142. Practical Combinatorial Testing,” National Institute of Standards & Technology, Gaithersburg, MD, United States, 2010.
[70] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, “ACTS: A Combinatorial Test Generation Tool,” in *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*, 2013, pp. 370–375.
[71] D. Kuhn, R. Kacker, yu Lei, and J. Hunter, “Combinatorial Software Testing,” *Computer*, vol. 42, pp. 94–96, Sep. 2009.
[72] D. R. Kuhn, R. N. Kacker, and Y. Lei, “Combinatorial Testing,” *Encyclopedia of Software Engineering*, p. 30.
[73] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, “IPOG: A General Strategy for T-Way Software Testing,” in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS’07)*, 2007, pp. 549–556.
[74] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn, “Combinatorial Testing of ACTS: A Case Study,” in *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*, 2012, pp. 591–600.

[75] J. D. Hagar, T. L. Wissink, D. R. Kuhn, and R. N. Kacker, “Introducing Combinatorial Testing in a Large Organization,” *Computer*, vol. 48, no. 4, pp. 64–72, Apr. 2015.
[76] H. Bergström and E. P. Enoiu, “Using Timed Base-Choice Coverage Criterion for Testing Industrial Control Software,” in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017, pp. 216–219.
[77] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, “Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 2013, pp. 188–197.
[78] M. Al-Hajjaji, F. Benduhn, T. Thüm, T. Leich, and G. Saake, “Mutation Operators for Preprocessor-Based Variability,” in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS ’16*, Salvador, Brazil, 2016, pp. 81–88.
[79] D. Reuling, J. Bürdek, S. Rotärmel, M. Lochau, and U. Kelter, “Fault-based Product-line Testing: Effective Sample Generation Based on Feature-diagram Mutation,” in *Proceedings of the 19th International Conference on Software Product Line*, New York, NY, USA, 2015, pp. 131–140.
[80] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin, “Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking,” *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1069–1089, Aug. 2013.
[81] X. Devroey, G. Perrouin, M. Cordy, M. Papadakis, A. Legay, and P. Y. Schobbens, “A Variability Perspective of Mutation Analysis,” 2014.
[82] C. Henard, M. Papadakis, and Y. Le Traon, “Mutation-Based Generation of Software Product Line Test Configurations,” 2014, pp. 92–106.
[83] H. Lackner and M. Schmidt, “Towards the Assessment of Software Product Line Tests: A Mutation System for Variable Systems,” in *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, New York, NY, USA, 2014, pp. 62–69.
[84] S. Fischer, R. E. Lopez-Herrejon, and A. Egyed, “Towards a Fault-detection Benchmark for Evaluating Software Product Line Testing Approaches,” in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, New York, NY, USA, 2018, pp. 2034–2041.
[85] G. Fraser and A. Arcuri, “EvoSuite at the SBST 2016 Tool Competition,” in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, New York, NY, USA, 2016, pp. 33–36.
[86] E. P. Enoiu, D. Sundmark, A. Čaušević, R. Feldt, and P. Pettersson, “Mutation-Based Test Generation for PLC Embedded Software Using Model Checking,” in *Testing Software and Systems*, vol. 9976, F. Wotawa, M. Nica, and N. Kushik, Eds. Cham: Springer International Publishing, 2016, pp. 155–171.
[87] S. Oster, T. Darmstadt, and F. Elektrotechnik, “Feature Model-based Software Product Line Testing,” p. 236.

[88] S. Oster, I. Zorcic, F. Markert, and M. Lochau, “MoSo-PoLiTe: tool support for pairwise and model-based software product line testing,” in *VaMoS*, 2011.
[89] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, “The AETG System: An Approach to Testing Based on Combinatorial Design,” *IEEE Trans. Softw. Eng.*, vol. 23, no. 7, pp. 437–444, Jul. 1997.
[90] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, “Practical pairwise testing for software product lines,” in *Proceedings of the 17th International Software Product Line Conference on - SPLC ’13*, Tokyo, Japan, 2013, p. 227.
[91] A. Hervieu, B. Baudry, and A. Gotlieb, “PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models,” in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, Hiroshima, Japan, 2011, pp. 120–129.
[92] I. Schaefer, “Variability Modelling for Model-Driven Development of Software Product Lines,” p. 8, 2010.
[93] M. Al-Hajjaji, S. Lity, R. Lachmann, T. Thum, I. Schaefer, and G. Saake, “Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing,” in *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*, Buenos Aires, Argentina, 2017, pp. 34–40.
[94] I. Abal, C. Brabrand, and A. Wasowski, “42 variability bugs in the linux kernel: a qualitative analysis,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE ’14*, Vasteras, Sweden, 2014, pp. 421–432.
[95] P. Kruse and J. Reiner, “Usage Models for the Systematic Generation of Test Cases with MODICA,” 2016, pp. 113–121.
[96] M. Grochtmann and K. Grimm, “Classification trees for partition testing,” *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63–82, 1993.
[97] P. Kruse and M. Luniak, “Automated Test Case Generation Using Classification Trees,” *Software Quality Professional*, vol. 13, pp. 4–12, Jan. 2010.
[98] C. Robinson-Mallett, M. Grochtmann, J. Wegener, J. Köhnlein, and S. Kühn, “Modelling Requirements to Support Testing of Product Lines,” presented at the ICSTW 2010 - 3rd International Conference on Software Testing, Verification, and Validation Workshops, 2010, pp. 11–18.