faITEA 3 Call 4: Smart Engineering

D3.5 Report on optimal test effort distribution

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	WP 3: TESTING OF CONFIGURABLE PRODUCTS	
TASK	T3.2: TEST CASE INSTANTIATION AND DISTRIBUTION OF TESTING EFFORTS AMONGST VARIANTS	
VERSION	V 1.0	OCT 31ST, 2020
DELIVERABLE TYPE	R (REPORT)	
DISSEMINATION LEVEL	PUBLIC	

Summary

A major problem to be solved for test generation is to find optimal distributions of testing efforts for given limits (e.g., time and cost), such that required test coverage can be guaranteed for all feature combinations. The following document gives an overview over different approaches developed in XIVT to optimize test efforts in various steps of the testing process.

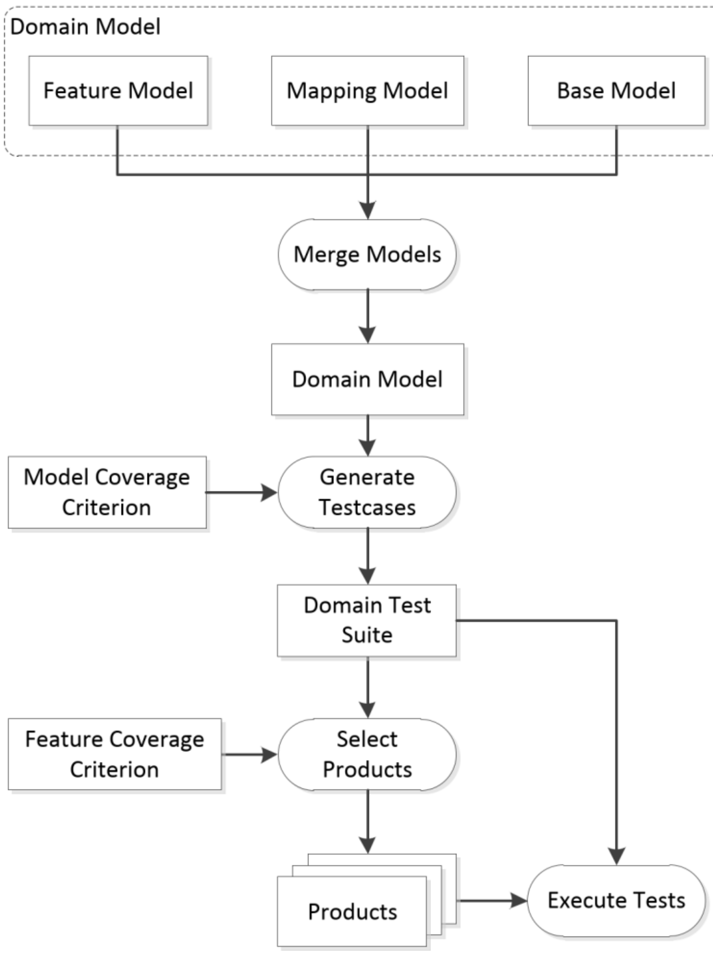
Table of Contents

Summary 2
Table of Contents 3
1. Strategies 4
  1.1 Domain-Centered Test Design (Fraunhofer).....................................4
  1.2 Delta Based Approaches (Expleo)..7
2. Optimization of test efforts 7
  2.1 Selection strategies (ifak)..7
  2.2 Reuse strategies (RISE)..15
  2.3 Optimization of test efforts in XIVT (RISE)....................................16
3. References 17

1. Strategies

1.1 Domain-Centered Test Design (Fraunhofer)

Whereas *application-centered test design* approaches as described above focus on the individual products, in *domain-centered test design* domain engineering level artefacts are used for test generation. This approach preserves the variability until a product has been selected for test execution. A major advantage of this approach is that one can focus on testing aspects of the domain model, without having to derive products first. Thus, the overlap of the results from independently generating tests for similar products can be avoided. The coverage of test targets can be maximized, which leads to high-quality test cases. Such a domain-centered test design process is depicted in the following figure.



For domain-centered test design, all domain artefacts should be taken into consideration. Using a base model as the only source of information is not adequate. The base model lacks information about the features, and about the associations of model elements to features. However, this information is important for a test generator: Features impose additional constraints on the behavior of the system. Since the base model contains implementation information for {\em all} features, it may include contradictory requirements. It might not even be a correct model according to the UML syntax. Thus, a challenge of this approach is to merge a domain model, consisting of a feature model, variability model and a base model, into a single model artefact that a standard test generator will accept as valid input.

Subsequently, we describe two solutions to this problem:

1. the *step-by-step* approach: sequentially excluding non-conforming configurations during test design-time, and
2. the *pre-configuration* approach: choosing a valid configuration before the design of individual test cases.

The step-by-step approach

The key idea of the step-by-step approach is to sustain the variability until it becomes necessary to bind it. Therefore, at the beginning of each test case design the test case is applicable to any valid product of the product line. Since not necessarily all valid paths in the base model are applicable to all products, the test designer must take account of test steps that bind variability. A test step must bind variability if not all products do conform. Subsequently, the set of valid products for this particular test case must be reduced by the set of non-conforming products. Hence, each test case is valid for any of the remaining products that do conform.

We implemented this step-by-step approach for UML state machines by introducing a Boolean variable into the system class for each feature that is not a core feature. The guards and effects on the transitions of the respective state machine can then be instrumented with these variables to include variability information in the state machine. Once the test generator executes this instrumented code, the feature is bound and it is not possible to change the binding for this test case anymore. After test generation has finished, the valid configurations for a particular test case can be read from the feature variables in each test case. Since the test cases may contain variability we obtain an *incomplete configuration* from each test case. An incomplete configuration is a configuration that supports a three-valued semantics for features instead of two values. The first two values are the same as in normal configurations (selected/unselected), the third stands for *undecided*. An undecided feature expresses variability by making no premise on the presence of the feature. Hence, each of the resulting test cases is generic for any product of the product line that conforms to the following:

For each control variable that is evaluated to true, the corresponding feature variable evaluation indicates whether this feature must be selected or unselected in the product. Features for which the respective control variable evaluates to \emph{false} are yet undecided and thus not evaluated.

The Pre-Configuration Approach

In the pre-configuration approach, test goals are selected from the domain model and also the test design is performed on this model similar to the step-by-step approach. However, during the design of an individual test case, the product configuration is fixed from the beginning of each test case and must not change before a new test case is created. Consequently, within a test case the test designer is limited to test goals that are specific to the selected product configuration. Thus, satisfying all domain model test goals is a matter of finding the appropriate configurations.

We implemented the pre-configuration approach by adding a configuration signal to the very beginning of the base model. To this end, we introduce a new state to the state machine, redirect all transitions leaving the initial state to leaving this new state, and add a transition between the initial state and the new state. Due to the UML specification the redirected transitions must not have a trigger, which is why we can add a trigger for configuration purposes to each of them. The trigger listens to a configuration signal that carries a valuation for all non-core features. The guard of these transitions must protect the state machine to be configured with invalid configurations and thus contains the propositional formula corresponding to the product line's feature model. Since any configuration that is provided by the signal must satisfy the guard's condition, only valid configurations are accepted.

After validating the configuration, the parameter values of the signal will be assigned to system class variables by the transition's effect. Hence, for each non-core feature a boolean variable, indicating whether the feature is selected or not, is added to the system class. Again, transitions specific to a set of products are protected by these variables. This is similar to the step-by-step approach, where the base model behaviour is limited to a potential behaviour of an actual product. However, control variables need not to be checked during test design, since the configuration is fixed and valid from the beginning of each test case. With these transformations to the base model, a test designer can create test cases for the product line. However, each test case will be specific to one configuration. In order to create generic test cases which are applicable to more than one product, we can apply a model transformation.

Experimental results

Now, we describe the experiments we performed to evaluate our approaches. We used several example product line models: a ticket machine, an alarm system, and an app for E-commerce. For our experiments we generated tests according to both presented approaches, application-centered and domain-centered test design. For application-centered test design we sampled products according to two different feature model coverage criteria: *all-features-included-excluded* and *all-feature-pairs*. As the name indicates, the criterion *all-features-included-excluded* is satisfied if in the set of sampled products, for each feature there is one product in which it is selected, and another one in which it is deselected. The criterion *all-feature-pairs* or *pairwise* holds of a sampling, if for every pair of features (f_1, f_2) there are products where f_1 and f_2 are both selected, one where neither f_1 nor f_2 is selected, one where f_1 is selected and f_2 deselected, and one where f_1 is deselected and f_2 selected.

The commercial Conformiq test designer tool supports control-flow, branching, requirements, and path coverage criteria. For the individual state machine models as well as for the merged state machine model we required all-transition coverage from the test generator. There would be other, more sophisticated metrics for state machines to consider. However, with this criterion we maintain comparability to other studies. We were able to generate test suites for both approaches with all the aforementioned parameters for all examples. In order to compare the results, we counted

- the number of test cases,
- the test steps that were generated by the test generator, and
- the number of configurations that are necessary to execute the test cases.

These are important criteria when estimating test efforts: The number of configurations is a major factor, since every configuration must be built, set-up, and maintained for testing. The number of test steps is an indicator for the efforts for test design and maintenance. Finally, the number of tests of interest when put in relation to the test steps. This gives the average test case length, which affects the efforts for debugging. With increasing length of a test case it becomes more difficult to isolate a fault within the SUT. The results for these measures are shown in the following table.

	Example	Approach			
		AC-IX	AC-PW	DC-Pre	DC-Step
Tests	Ticket Machine	13	50	9	9
	Alarm System	21	57	12	12
	eShop	20	71	13	13
Steps	Ticket Machine	33	56	48	39
	Alarm System	55	165	62	50
	eShop	135	486	48	39
Conf.	Ticket Machine	2	6	5	1
	Alarm System	3	9	6	2
	eShop	2	7	4	2

For each example, this table shows the different approaches: application-centered with all-features-included-excluded (**AC-IX**) as well as with pair-wise (**AC-PW**) coverage and domain-centered with pre-configuration (**DC-Pre**) as well as with step-by-step (**DC-Step**). The **AC-PW** approach scores the highest values for all measures since it applies the strongest feature coverage criterion and thus covers a maximum of configurations. Consequently, more test cases and test steps are generated than for any other approach. In contrast, the **DC-Step** yields the lowest scores for any measure, while at the same time it is focused on covering every reachable transition. We take this as an indicator for domain-centered test design to scale better than application-centered approaches. We performed similar experiments on several other examples, with comparable results. From this, we conclude that domain-centered test design produces test suites with a significantly lower number of tests and test steps than application-centered test design. Thus, test execution efforts are much lower for these test suites; yet, this does not necessarily lead to a lower error detection capability.

1.2 Delta Based Approaches (Expleo)

Variant modelling tools, such as the tool BeVR developed in the XIVT context, allow to derive variants from a product line model in a structured way.

Prioritizing the derived (large) set of variants for testing or selecting variants for test due to a limited time frame can be done in analogy to test suite generation for a given product, following established coverage criteria, such as combinatorial coverage ([SPT11], [SPT13]) or requirement coverage. In opposition to test suite generation, deriving variants ad hoc from the model for testing purposes, e.g. according to delta based criteria as in [DPP17], is often not suitable: The existing variants might only a small subset of the possible configurations of the product model, as choices of variants don’t always stem from actual restrictions of the product, but from exterior circumstances such as user choices. Implementing these constraints in the model will make it overly complicated and unflexible. As too many constraints are often a crucial obstacle for the effectiveness of the corresponding algorithms, the tool VarSel will take the approach of deriving variants first, and then prioritizing them. The general approach is independent of the coverage criteria: According to the chosen coverage criteria, a set objects to cover (features, pairs, requirements) is instantiated, and in a delta-based algorithm, variants to test are chosen that contain the most yet-to-cover objects from this set. The set-based approach here ensures that at any step of testing, the highest level of coverage for the number of tested variants is reached. Other similarity based approaches take into account distance measures such as Hamming, Jacquard, Dice (see [DPP17], [SBP14]) that avoid scalability issues for higher levels of combinatorial coverage.

Apart from the interface to BeVR, the implemented tool VarSel has an interface to the Expleo in-house tool Testona. Here, test case generation for the chosen/prioritized variants to test can be conducted taking into account the same combinatorial coverage measures on the product level. A coverage approach overarching product line and product level and combining the algorithms to optimize the coverage through both levels and minimizing test efforts is in planning.

2. Optimization of test efforts

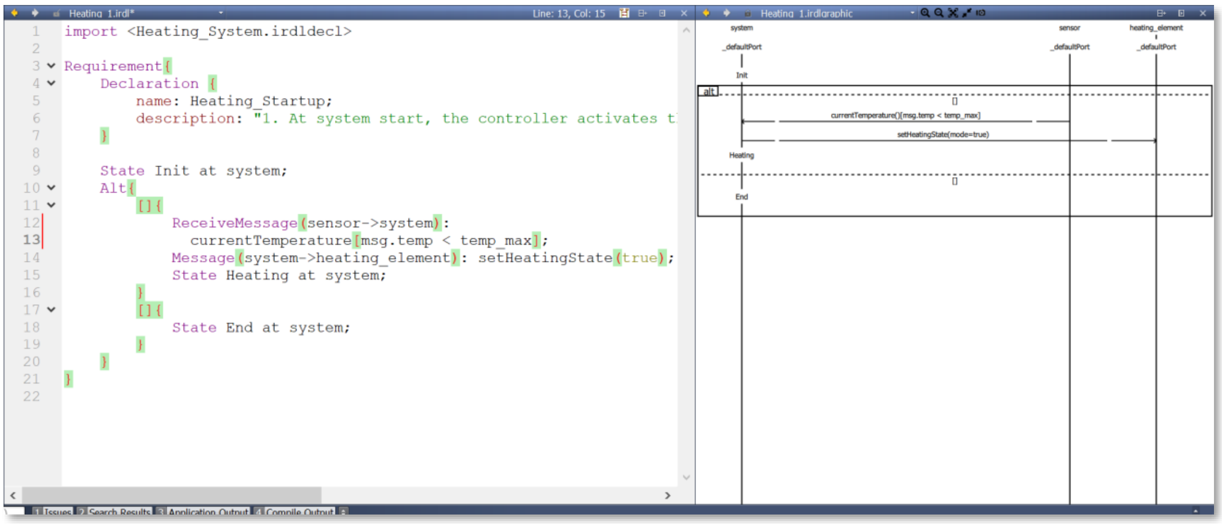
2.1 Selection strategies (ifak)

Features and requirements

Generating test cases from requirements

So far, ifak's tool chain for model-based testing is requirements based. The starting point is that one models requirements of the system one is interested using a formal language, the ifak's requirements definition language. Figure 1 shows an example of a requirement definition. From the requirements, abstract test cases are generated by ifak's tool chain (see Figure 2). An abstract test case is described by the signals sent from one component to another component including the parameters and can include timers. Furthermore, the test case describes the expected result to be returned by the target component. From the requirements defined, all possible cases are derived for testing, e.g. different values and combinations of the parameters.

As stated, this is abstract because declared names for signals, attributes and components are arbitrarily chosen and have to be mapped to real items and it is not described how the signals are transferred to the component and how they are received from the other component. A real environment is very specific and may range from a PLC, a controller until software interfaces from a certain programming language.



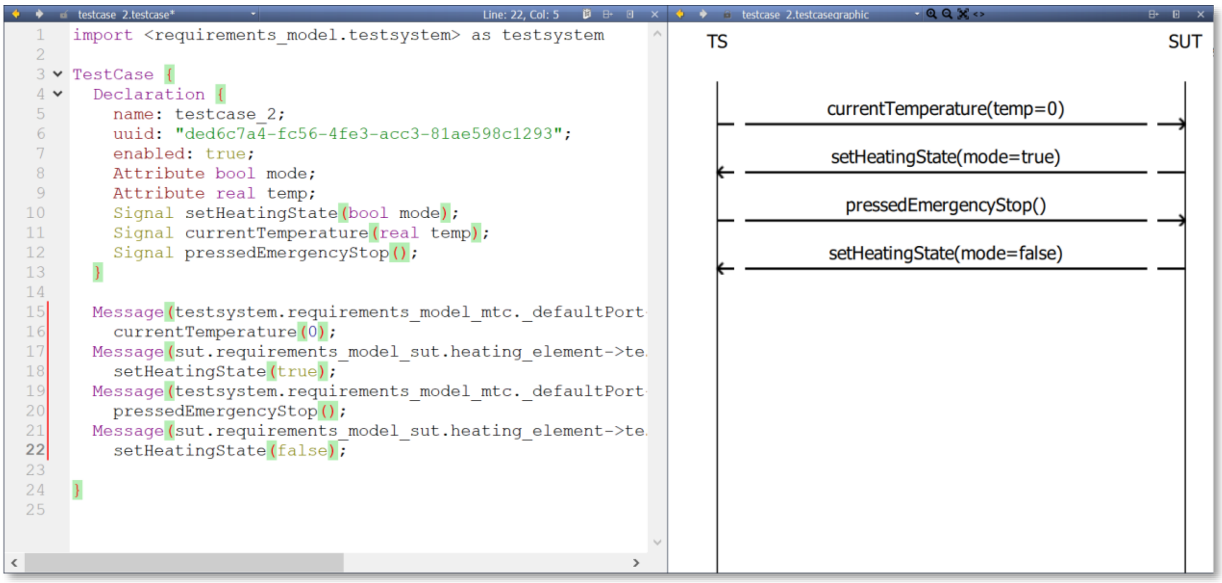
```
import <Heating_System.irdldecl>

Requirement{
    Declaration {
        name: Heating_Startup;
        description: "1. At system start, the controller activates t
    }

    State Init at system;
    Alt{
        []{
            ReceiveMessage(sensor->system):
              currentTemperature[msg.temp < temp_max];
            Message(system->heating_element): setHeatingState(true);
            State Heating at system;
        }
        []{

            State End at system;
        }
    }
}
```

Figure 1: Definition of a requirement



```
import <requirements_model.testsystem> as testsystem

TestCase {
  Declaration {
    name: testcase_2;
    uuid: "ded6c7a4-fc56-4fe3-acc3-81ae598c1293";
    enabled: true;
    Attribute bool mode;
    Attribute real temp;
    Signal setHeatingState(bool mode);
    Signal currentTemperature(real temp);
    Signal pressedEmergencyStop();
  }

  Message(testsystem.requirements_model_mtc._defaultPort
    currentTemperature(0);
  Message(sut.requirements_model_sut.heating_element->te
    setHeatingState(true);
  Message(testsystem.requirements_model_mtc._defaultPort
    pressedEmergencyStop();
  Message(sut.requirements_model_sut.heating_element->te
    setHeatingState(false);

}
```

testcase 2.testcase*
Line: 22, Col: 5
testcase 2.testcasegraphic
TS
SUT
currentTemperature(temp=0)
setHeatingState(mode=true)
pressedEmergencyStop()
setHeatingState(mode=false)

Figure 2: Test case generated from the requirement

In order to put test cases into action, we use test adapters to link the abstract behavior of test cases to concrete system components.

BVR for variability modeling

The variability modeling language BVR (Base Variability Resolution), developed in the ARTEMIS project VARIES, has been selected to be used within the XIVT project by several partners. We treat it as a de facto standard in the XIVT project. Using BVR enables us to exchange the models among the partners easily aiming at different uses cases.

A comprehensive description of BVR can be found in *D3.2 Report on the methodology for the construction of testing models*. The following sentences repeat the key passage:

Whereas the product viewpoint describes the domain, architecture and behaviour of a single system instance, the product line viewpoint represents a family of systems in terms of feature changes (additions, removals and modifications) across a shared development history. Conceived as an application of Δ-modelling [VAR10], it is composed of four parts:

- A feature model describing the entire product line in terms of abstract functional characteristics (features), their optionality and interdependence / exclusion relations;
- A core model described in terms of the product viewpoint, representing either the product baseline (i.e. the set of user functions common to all variants) or the first member of the product line;
- A collection of resolutions describing specific configurations of the feature model;
- A collection of deltas describing changes to the core model in accordance to specific resolutions.

Feature models are described in the BVR language [BVR14] using the VSpec model.

Linking requirements and features

Since ifak's starting point in the XIVT project is to describe requirements and to generate test cases, the aim of the XIVT project is to extend this for variant-intensive systems. First, we have to introduce variability in our requirements model.

First, we introduce features. A feature simplified is a function of a system. It is up to the system designer how to choose the granularity here. Then we say that a variant of a system is a set of features. Some features may be contained in all variants some may not. Figure 3 illustrates this.

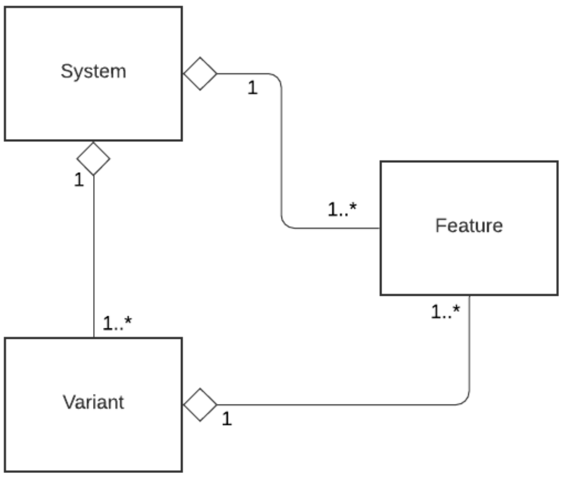


Figure 3: The role of features in a system with several variants

From our point of view, requirements are linked to features. Whereas one feature has at least one requirement and may have many requirements. On the other hand, a requirement might belong to more than one feature (Figure 4).

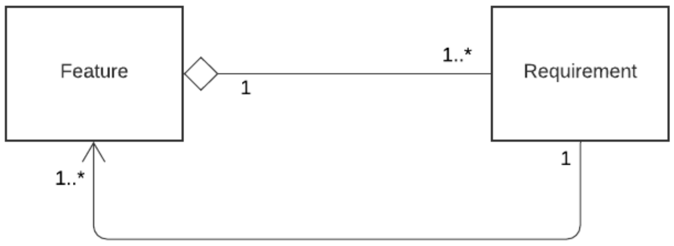


Figure 4: The relation between features and requirements

Files

The data described above are organized in three different types of files:

- XMI: UML model defining the features with core and deltas (`*.xmi, *.uml`)
- BVR: Variabilities, Resolutions and Realizations (`*.bvr`)
- IRDL: Requirements (`*.irdl`)

There are exactly one XMI and exactly one BVR file but several IRDL files.

Figure 5 illustrates the relation between these file types and the according model components.

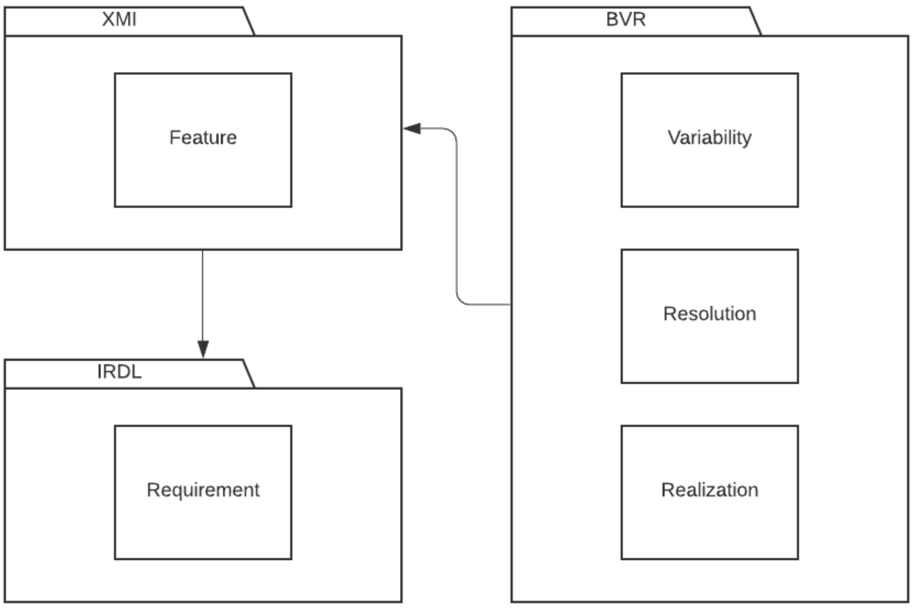


Figure 5: Relation between the model components and its files

Within XIVT, no file stands alone; all are linked to each other. The features in the XMI file contain attributes referencing the requirements. The variability and resolution elements in the BVR file point to the core and delta features and via this to the requirements too.

Since the UML model contains the core and the delta features represented by classes, we introduced the element `requirement` in order to reference the requirements from the features [OMG15] [GDB01]. We also introduced the namespace XIVT with the prefix `xvt`. Figure 6 provides an extract of an XMI file.

```
<packagedElement xmi:type="uml:Package" name="core">
  <packagedElement xmi:type="uml:Class" name="AGV">
    <ownedAttribute         xmi:type="uml:Property"         name="platform"
aggregation="composite"/>
    <xvt:requirement xmi:id="" name="agv_req1" file="/reqs/agv_req1.irdl"/>
  </packagedElement>
  <packagedElement xmi:type="uml:Class" name="Differential">
    <xvt:requirement xmi:id="" name="df_req1" file="/reqs/df_req1.irdl"/>
    <xvt:requirement xmi:id="" name="df_req2" file="/reqs/df_req2.irdl"/>
  </packagedElement>
</packagedElement>
```

Figure 6: Adding requirements to the XMI file (bold)

The resolution models within the BVR file describe the variants of the system. The following example provides two different variants with included and excluded features (Figure 7).

```
<resolutionModels xsi:type="bvr:PosResolution" name="Differential">
  <members xsi:type="bvr:PosResolution" name="Differential"/>
  <members xsi:type="bvr:NegResolution" name="Holonomic"/>
</resolutionModels>
<resolutionModels xsi:type="bvr:PosResolution" name="Holonomic">
  <members xsi:type="bvr:NegResolution" name="Differential"/>
  <members xsi:type="bvr:PosResolution" name="Holonomic"/>
</resolutionModels>
```

Figure 7: Describing the system's variants by resolution models in the BVR file

Implementation

We developed a command line tool with the name *RIVS* (requirements modeling in multi-variant systems) which is able to manage models and objects as described. This includes creating models and objects as well as removing them or modifying attributes. We decided for a command line tool because other parts from ifak's testing tool chain also base on command line tools. We use this in the MBTCreator [MBT20] and newly in the MBTWeb, a web tool used through web browser instead of having a stand-alone tool to be installed explicitly on the computer. The advantage of this approach is that you integrate its functionality in every tool you want using the standardized BVR language no matter which operating system and GUI. In order to implement the new variability feature we take care of existing data and file formats as described above.

Basic functions

RIVS enables the user to add, remove or modify models, objects of models and attributes of models and objects. The tool operates on files only. Therefore, when launched it reads the specified XMI and

BVR file, processes the requested operation and saves the modifications (i.e. the entire model) back to the files (for now, RIVS only modifies the BVR file; see section Limitations below for details). The following tables list the options of the tool and the modifiable attributes of the models and objects.

Option	**Description**
`-f`	Specifies the XMI and the BVR file.
`-m`	Specifies the model, can be `base`, `variability`, `resolution`, `realization`.
`-c`	Creates a model or an object.
`-s`	Changes attributes of a model or an object.
`-r`	Removes a model or an object.
`-e`	Extracts variant, feature, requirement or all. Use `-e variant`, `-e feature`, `-e requirement` and `-e all` accordingly.

Attribute	**Description**
`-t`	Specifies the type of the model or object.
`-n`	Specifies the name of the model or object.
`-ibe`	Specifies the object to be an `insideBoundaryElement` within the selected realization model.
`-obe`	Specifies the object to be an `outsideBoundaryElement` within the selected realization model.
`-pn`	Specifies the attribute `propertyName` of a realization model object.
`-tp`	Specifies the attribute `toPlacement` of a realization model object.
`-tr`	Specified the attribute `toReplacement` of a realization model object.

Here are some examples of tool calls together with some explanations:

```
rivs -f xivt.uml xivt.bvr -m variability AGV -c
```

Creates a variability model with the name AGV and adds it to the model described in the xivt.uml and xivt.bvr file.

```
rivs -f xivt.uml xivt.bvr -m variability AGV -c -o Differential
```

Creates an object with the name Differential and adds it to the variability model with the name AGV. The model to be modified is read from the xivt.uml and xivt.bvr file. The modified model is written to the xivt.bvr file.

```
rivs -f xivt.uml xivt.bvr -m resolution Differential -c
```

Creates a resolution model with the name Differential adds it to the model described in the xivt.uml and xivt.bvr file.

```
rivs -f xivt.uml xivt.bvr -m resolution Differential -s -t true
```

Sets the value of the type attribute of the resolution model with the name Differential to true.

```
rivs -f xivt.uml xivt.bvr -m realization Differential -c
```

Creates a realization model with the name Differential.

```
rivs -f xivt.uml xivt.bvr -m realization Differential -s -t
bvr:PlacementFragment
```

Sets the value of the type attribute of the realization model with the name Differential to bvr:PlacementFragment.

Limitations

RIVS is currently not able to modify the XMI file. This includes classes and other elements as well as requirements. In order to manage the features, an UML modeler tool should be used. The requirements should be added by using a plain text editor.

Selection of variant, feature and requirement

When we select a variant of the system to be tested, we automatically select a set of features as well as a set of requirements. From these requirements, we can automatically generate abstract test cases and put them into action by test adapters as usual. Selecting single features and requirements is also possible.

Test coverage

Test coverage is a measure for describing the degree to which artifacts such as the source code and/or the hardware of a system is executed when a particular test suite runs. The degree of coverage is usually given as a percentage value. There are a number of code coverage criteria, the main ones being [MYE04]:

- Function coverage – has each function (or subroutine) in the program been called?
- Statement coverage – has each statement in the program been executed?
- Edge coverage – has every edge in the Control flow graph been executed?
- Branch coverage – has each branch (also called DD-path) of each control structure (such as in if and case statements) been executed? For example, given an if statement, have both the true and false branches been executed? This is a subset of edge coverage.
- Condition coverage (or predicate coverage) – has each Boolean sub-expression evaluated both to true and false?

A thorough overview of coverage criteria for variant-rich systems is given in the Deliverable D2.3 in the Section “Test optimization criteria for variant-intensive products”. There are many free and commercial tools for different programming languages which can measure the different coverage criteria. When executing the tests of the selected test suite, the measures can be determined. We intend to implement this feature soon.

Optimization

Variant-rich systems lead to an exponential increasing number of products and test cases. The variants are divided into features and each feature can be contained or not contained in different variants. The features cannot always be tested individually; most times, we have to test the entire variant instead. In order not to waste processing time and finally costs by executing the same test case again and again in different system variants, the combinatorial optimization problem is which variants we should test in order to cover all resp. as many features (including code) as possible. As an example, there is variant V1 with feature F1, F2, variant V2 with feature F2, F3 and variant V3 with F1, F2, F3. If we test V1 and V2, we do not have to test V3 again because all features are already covered. This optimization problem corresponds to the well-known set cover problem, which belongs to the list of 21 classical NP-complete problems [KAR72].

Given are a set U and n subsets S_j of U. The question is, whether there is a union of $k \leq n$ subsets S_j of U covering all elements of U. Treating this as an optimization problem, we are looking for a coverage with k is minimal. If there are costs c_j assigned to the subsets S_j, the optimization goal is to find the coverage with the least costs.

[CMR14] tells of three main approaches for solving this problem in the context of testing multi-variant systems. The combinatorial testing determines variants based on the interaction coverage of features, e.g. pairwise coverage [OST12]. The approach exclusively uses information from the variability model but achieves an effective reduction of the product subset for testing. The second strategy prioritizes variants according to criteria like most critical, most sold etc. [MAN12]. In this sense, [LLL15, SCH19] use incremental delta-testing in order to reduce the test effort between product variants by identifying effective test cases for reuse.

The third method presented in [CMR14] selects variants based on requirements and architecture coverage. In order to find the minimal number of subsets covering all features, [CMR14] implemented a Greedy algorithm as well as a Simulated Annealing algorithm and compared the results. A Greedy algorithm for polynomial time approximation to the set cover problem has been presented in [CHV79]. It chooses the subsets according to one rule: at each stage, choose the subset that contains the largest number of uncovered elements. The results for two real world cases studied in [CMR14] showed that the Greedy algorithm is superior to Simulated Annealing in terms of quality of results and run time. Even though Simulated Annealing is capable of leaving local optima in the solution space, which the Greedy algorithm is not, Simulated Annealing never found a better solution than the Greedy Algorithm.

Within the XIVT project, we plan to implement the Greedy algorithm into our RIVS tool.

Implementation in RIVS

The RIVS tool can also be used to select single features, variants or requirements as well as an approximately minimal subset of the requirements covering all variants as described above. Some examples are listed in the following:

```
rivs -f xivt.uml xivt.bvr -e variant Differential
```

Selects all requirements of the variant with the name Differential.

```
rivs -f xivt.uml xivt.bvr -e feature Holonomic
```

Selects all requirements of the feature with the name Holonomic.

```
rivs -f xivt.uml xivt.bvr -e requirement df_req1
```

Selects the requirement with the name df_req1.

```
rivs -f xivt.uml xivt.bvr -e all
```

Selects the approximately minimal subset of requirements covering all variants as described in the previous section.

The output is a set of requirements for which the existing methods of test case generation and creation, such that test instances and configuration parameter values can be created for concrete testing. Figure 8 provides a selection output sample.

```
requirement agv_req1 /reqs/agv_req1.irdl
requirement df_req1 /reqs/df_req1.irdl
requirement df_req2 /reqs/df_req2.irdl
```

Figure 8: Selection output sample

2.2 Reuse strategies (RISE)

Generally, reuse of any artifact could be done in three main ways [RI1], as follows. Black-box reuse approaches reuse artifacts as-is with no modification what-so-ever. In contrast, the white-box approaches modify the artifacts' internal structure (such as modification of test sequence). Finally, the grey-box reuse approaches are mainly focused on reusing artifacts with modified configuration parameters.

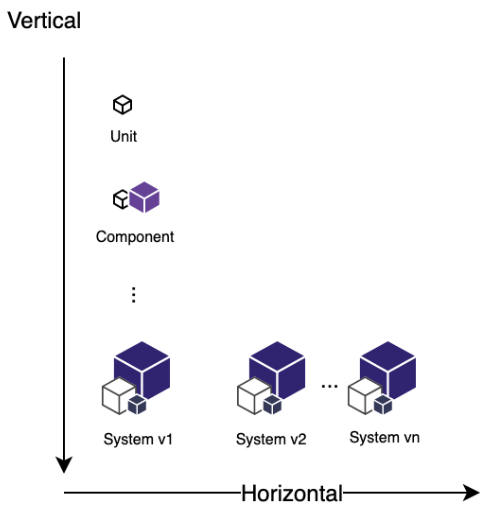


Figure TestReuse: Dimensions of test reuse

Reuse of test cases is an active area of research and can optimize test efforts. The test reuse can be done in two main dimensions as shown in Figure TestReuse. Vertical test reuse approaches reuse test cases across different integration levels with in the same product variant [RI4]. While horizontal test reuse approaches reuse test artifacts across product versions and variants [RI6]. Reuse of test specifications and test scripts across product variants is a common industrial practice [RI0]. As reported by Abbas et al. [RI0], engineers find it easier to tailor tests than writing new ones. In the railway domain, Variability-Aware Reuse Analysis (VARA) [RI2] method is used to identify product line features that could be reuse to realize new customer requirements. As a by-product of this method, the reuse of test specification and test script is also aided in a black-box manner. In practice, these test cases are then manually adapted to the new product variants.

As noted, most of the (74%) test cases becomes obsolete after very small modifications to the system [RI4]. In this regard, white-box and grey-box approaches borrow concepts from test evolution and test repair to detect test cases that can be tailored automatically [RI3]. These, approaches then either recommend repair actions or automatically apply repair actions to the test cases.

2.3 Optimization of test efforts in XIVT (RISE)

One way to optimize test efforts can be by effectively reusing test cases across product variants. In addition, test reuse can help in avoiding redundancies. This way the efforts could be targeted towards the delta parts of the variants and would avoid redundant testing. In the XIVT project, efforts have been made to optimize the test efforts. We discuss the test optimization approaches in the remainder of this section.

Test reuse Analysis and recommendAtion (TARA) is a planned decision support system for test reuse analysis across variants. It takes product line assets (models), its tests and modified versions of the assets to classify the test cases for reuse. This tool's output is a list of test cases that could be reused for the modified version of the asset. Figure TARA shows an overview of the TARA.

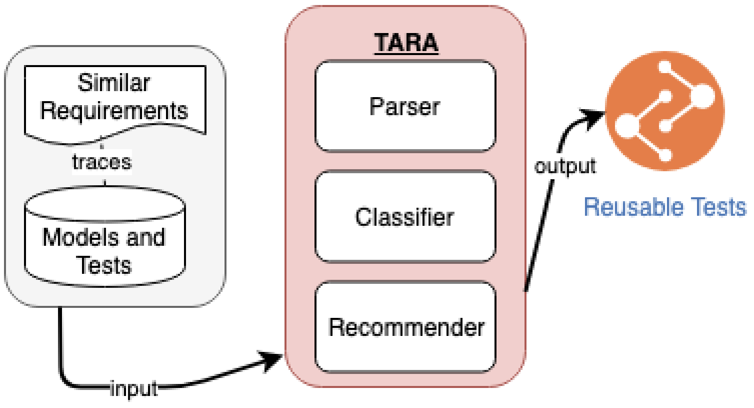


Figure TARA: Overview of TARA on PL models

Delta-Oriented Test Analysis and selection (DoTA) is another planned decision support systems for test selection of modified variants. DoTA basically takes in a model with tests and a modified version of the model and selects test cases such that it will execute the modified parts of the model, thus targeting the testing efforts towards the modified parts of the variants. Figure DoTA shows a high-level view of the DoTA approach for test selection.

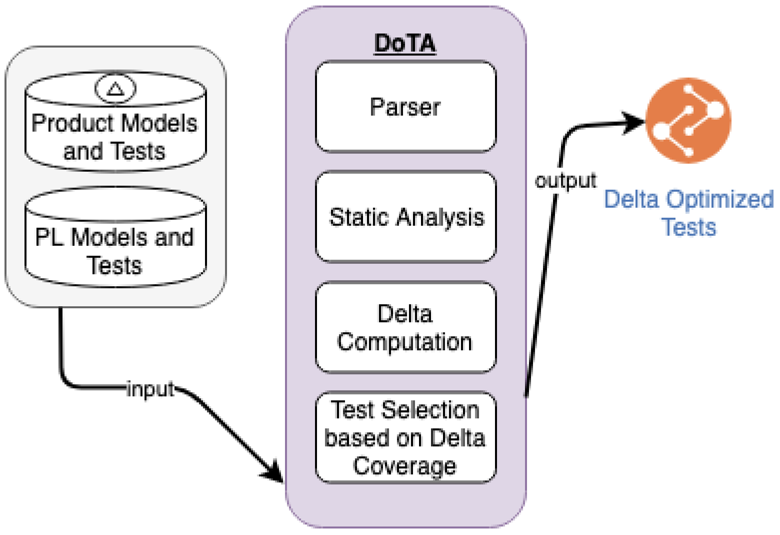


Figure DoTA: Overview of the DoTA approach

TOpic Modeling Towards Effective test Case Selection (TOM-TEC) is an approach developed in XIVT to select test cases that are highly likely to be testing Extra-Functional Properties (EFPs, for example, safety-related tests) [RI7]. It might not be possible to execute all the tests in some cases, and thus selection might be required. As test selection can be made based on a variety of criteria, TOM-

TEC uses the topical coverage of the test cases towards EFPs. Figure TOM-TEC shows the high-level view of the approach for test case selection using Natural Language Processing (NLP).

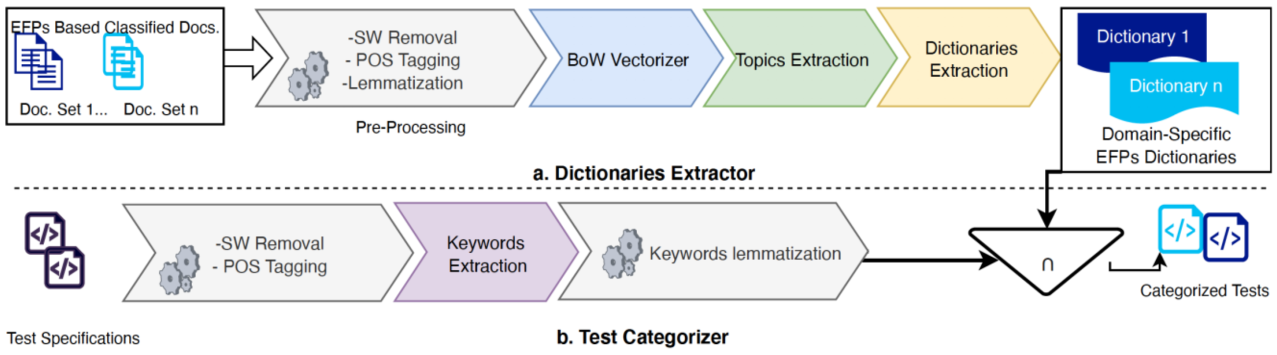


Figure TOM-TEC: Test case selection approach for EFPs

3. References

[AGV06] Vis, Iris FA. "Survey of research in the design and control of automated guided vehicle systems." *European Journal of Operational Research* 170.3 (2006): 677-709.

[AGV19] Weng, Jian-Fu, and Kuo-Lan Su. "Development of a SLAM based automated guided vehicle." *Journal of Intelligent & Fuzzy Systems* 36.2 (2019): 1245-1257.

[AMT09] Lamancha, Beatriz Pérez, et al. "Automated model-based testing using the UML testing profile and QVT." *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*. 2009.

[BVR14] Haugen, Øystein, and Ommund Øgård. "BVR – better variability results." *International Conference on System Analysis and Modeling*. Springer, Cham, 2014.

[BVR15] Vasilevskiy, Anatoly, et al. "The BVR tool bundle to support product line engineering." *Proceedings of the 19th International Conference on Software Product Line*. 2015.

[BVR18] “SINTEF-9012 / bvr.” *SINTEF-9012*, 22 January 2020, https://github.com/SINTEF-9012/bvr.

[COM04] Kuhn, D. Richard, Dolores R. Wallace, and Albert M. Gallo Jr. “Software Fault Interaction and Implications for Software Testing”. *IEEE Transactions on Software Engineering* 30(6), pp. 418-421. 2004.

[COM11] Nie, Changhai, and Hareton Leung. “A Survey of Combinatorial Testing,” *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29. 2011.

[COM16] Rogstad, Erik, and Lionel Briand. “Cost-effective strategies for the regression testing of database applications: Case study and lessons learned”. *J. Systems and Software* 113, pp. 257-274. 2016.

[DPP17] Al-Hajjaji, Mustafa, et al. "Delta-oriented product prioritization for similarity-based product-line testing." *2017 IEEE/ACM 2nd International Workshop on Variability and Complexity in Software Design (VACE)*. IEEE, 2017.

[ECL19] “Eclipse (software).” *Wikipedia*, 21 August 2019, https://en.wikipedia.org/wiki/Eclipse_(software).

[GTC12] Johansen, Martin Fagereng, Øystein Haugen, and Franck Fleurey. "An algorithm for generating t-wise covering arrays from large feature models." *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 2012.

[MDT19] “Model Development Tools (MDT).” *Eclipse Foundation*, 21 August 2019, https://www.eclipse.org/modeling/mdt/.

[MMT19] “Model to Model Transformation (MMT).” *Eclipse Foundation*, 21 August 2019, https://www.eclipse.org/mmt/.

[PAP18] “Eclipse Papyrus.” *Eclipse Foundation*, 21 August 2019, https://www.eclipse.org/papyrus/index.php.

[PIT06] Bryce, Renée, and Charles J. Colbourn. ”Prioritized interaction testing for pair-wise coverage with seeding and constraints” *Inf. Softw. Technol.* 48 (10), pp. 960–970. 2006.

[SBP14] Henard, Christopher, et al. "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines." *IEEE Transactions on Software Engineering* 40.7 (2014): 650-670.

[SDM18] Halim, Shahliza Abd, Dayang Norhayati Abang Jawawi, and Muhammad Sahak. “Similarity distance measure and prioritization algorithm for test case prioritization in software product line testing.” *Journal of Information and Communication Technology*, 18(1), pp. 57-75. 2018.

[SPE12] Pohl, Klaus, et al., eds. *Model-based engineering of embedded systems: The SPES 2020 methodology*. Springer Science & Business Media, 2012.

[SPE16] Pohl, Klaus, et al. "Advanced model-based engineering of embedded systems." *Advanced Model-Based Engineering of Embedded Systems*. Springer, Cham, 2016. 3-9.

[SPT11] Hervieu, Aymeric, Benoit Baudry, and Arnaud Gotlieb, “PACOGEN: Automatic Generation of Pairwise Test Configurations from Feature Models”. *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, Hiroshima, Japan, pp. 120–129. 2011.

[SPT13] Marijan, Dusica, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu, “Practical pairwise testing for software product lines”. *Proceedings of the 17th International Software Product Line Conference*, Tokyo, Japan, pp. 227-235. 2013.

[UML08] Miles, Russ, and Kim Hamilton. *Learning UML 2.0*. " O'Reilly Media, Inc.", 2006.

[UML19] “List of Unified Modeling Language Tools.” *Wikipedia*, 21 August 2019, https://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools.

[UTP08] Baker, Paul, et al. *Model-driven testing: Using the UML testing profile*. Springer Science & Business Media, 2007.

[VAB14] Abal, Iago, Claus Brabrand, and Andrzej Wasowski. "42 variability bugs in the linux kernel: a qualitative analysis." *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014.

[VAR10] Schaefer, Ina. "Variability Modelling for Model-Driven Development of Software Product Lines." *VaMoS* 10 (2010): 85-92.

[XVT19] XIVT Project Consortium, “XIVT Full Project Proposal Annex.” 03 August 2019.

[RI0] Muhammad Abbas, Robbert Jongeling, Claes Lindskog, Eduard Paul Enoiu, Mehrdad Saadatmand, & Daniel Sundmark (2020). Product Line Adoption in Industry: An Experience Report from the Railway Domain. In *24th ACM International Systems and Software Product Line Conference*.

[RI1] de Almeida, E. S. (2019). Software Reuse and Product Line Engineering. In *Handbook of Software Engineering* (pp. 321-348). Springer, Cham.

[RI2] Muhammad Abbas, Mehrdad Saadatmand, Eduard Paul Enoiu, Daniel Sundmark, & Claes Lindskog (2020). Automated Reuse Recommendation of Product Line Assets based on Natural Language Requirements. In the 19th *International Conference on Software and Systems Reuse*.

[RI3] Imtiaz, Javaria, et al. "A systematic literature review of test breakage prevention and repair techniques." *Information and Software Technology* 113 (2019): 1-19.

[RI4] B. Daniel, V. Jagannath, D. Dig, ReAssert: suggesting repairs for broken unit tests, in: International Conference on Automated Software Engineering, 2009, pp. 433– 444.

[RI5] D. Flemström, D. Sundmark and W. Afzal, "Vertical Test Reuse for Embedded Systems: A Systematic Mapping Study," *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, Funchal, 2015, pp. 317-324, doi: 10.1109/SEAA.2015.46.

[RI6] R. Ramler and W. Putschögl, "Reusing Automated Regression Tests for Multiple Variants of a Software Product Line," *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, Luxembourg, 2013, pp. 122-123, doi: 10.1109/ICSTW.2013.21.

[RI7] M. Abbas, A. Rauf, M. Saadatmand, E. P. Enoiu and D. Sundmark, "Keywords-based test categorization for Extra-Functional Properties," *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Porto, Portugal, 2020, pp. 153-156

[BVR14] 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.

[VAR10] Schaefer, Ina. "Variability Modelling for Model-Driven Development of Software Product Lines." VaMoS 10 (2010): 85-92.

[XVT19a] XIVT D3.2 Report on the methodology for the construction of testing models

[MBT20] MBTCreator, https://github.com/ifak/mbtcreator

[OMG15] Object Management Group: XML Metadata Interchange (XMI) Specification Version 2.5.1. https://www.omg.org/spec/XMI/2.5.1/PDF

[GDB01] Timothy J. Grose, Gary C. Doney, and Stephen A. Brodsky. 2001. Mastering XMI: Java Programming with XMI, XML and UML. John Wiley & Sons, Inc., USA.

[MYE04] Glenford J. Myers (2004). The Art of Software Testing, 2nd edition. Wiley. ISBN 0-471-46912-2.

[CMR14] Cmyrev, Anastasia; Reissing, Ralf. "Efficient and effective testing of automotive software product lines." Applied Science and Engineering Progress 7.2 (2014): 53-57.

[OST12] Oster, S."Feature Model-based Software Product Line Testing," Doctoral Thesis, Technical University Darmstadt, 2012.

[MAN12] Manicke, O. “Variability management for mastering complexity in the development process of mechatronical vehicle functions,” Doctoral Thesis, Technical University, Dresden, 2012. (in German)

[VAZ13] Vazirani, Vijay V. *Approximation algorithms*. Springer Science & Business Media, 2013.

[KAR72] Karp, Richard M. "Reducibility among combinatorial problems." Complexity of computer computations. Springer, Boston, MA, 1972. 85-103.

[CHV79] Chvatal, V. A Greedy Heuristic for the Set-Covering Problem. Mathematics of Operations Research Vol. 4, No. 3 (Aug., 1979), pp. 233-235

[FID20] FeatureIDE, https://featureide.github.io/

[SCH19] Schaefer, Ina. How to test the Universe? Presentation at the 25th Nederlandse Testdag, 20 November 2019, Capgemini Nederland, Utrecht, Netherlands, https://testdag2019.github.io/pdf/ina_schaefer.pdf

[LLL15] Lachmann, Remo; Lity, Sascha; Lischke, Sabrina; Beddig, Simon; Schulze, Sandro; Schaefer, Ina: Delta-oriented test case prioritization for integration testing of software product lines. SPLC 2015: 81-90