ITEA 3 Call 4: Smart Engineering

D3.5.2 Report on the methodology for the construction of testing models, final version

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	T3.1: Requirements-based variability modelling and abstract test case generation	
Version	V 2.0	Oct 31st, 2020
Deliverable Type	R (Report)	
Dissemination Level	Public	

Version History

Date	Description	Version	Level
Nov 30, 2019	Report on the methodology for the construction of testing models for variant and configurable products, Tools (e.g. Eclipse plugins) for modelling variability for testing and generation of abstract test cases from variability models – initial version (T3.1)	V1.0	Confidential
Oct 30, 2020	Product line viewpoint (section 4.2) enhanced and adapted to the XIVT toolchain, including BVR examples	V2.0	Public

Summary

This report presents an approach for the automatic generation of abstract test case suites from product line models, which represent collections of systems related by shared features and development history. A custom extension of SPES XT provides the conceptual framework for describing product lines; the use of UML diagrams for model artefacts allows automatic product realization, analysis and simulation, enabling the generation of abstract test suites as a set of UML Testing Profile (UTP) diagrams. Eclipse is proposed as implementation basis, due to the availability of libraries and plugins supporting variability modelling and the design and manipulation of UML artefacts.

Table of Contents

1. Introduction

Task T3.1 of XIVT Working Package 3 (WP3) is concerned with “extending existing methods for variability modelling such that feature models, base models, scenario models and threat and intrusion models are connected and allow the derivation of abstract test cases. Together with the definition of suitable meta-models, a methodology will be developed to formulate testing requirements from the use cases in these testing models” [XVT19, p. 88]. In other words, the objective is to design and implement an automated process to generate abstract test case suites from project specifications composed of:

- **Product line model**, describing a product baseline, its context and potential variants;
- **Product instances**, specifying actual product variants and their configurations.

A common conceptual and technological basis encompassing all artefacts and processing tasks is a practical requirement for effective process implementation. Moreover, automation is only possible if project documents follow a well-defined and unambiguous format that enables mechanical manipulation, and capture all the information necessary for test case generation. It must also be possible to describe all procedures in precise and objective terms that can be converted to computer algorithms.

Software Platform Embedded Systems (SPES) is a comprehensive framework for modelling project domain, requirements and architecture [SPE12]. Its extended version SPES XT includes facilities for managing product lines [SPE16, p. 197]. A set of custom extensions to SPES XT is proposed as the conceptual basis for project models.

SPES artefacts can be instantiated in the Unified Modelling Language (UML) 2.0 [UML08], making it a natural choice of common notation. UML is a mature modelling language, widely known and supported by a number of software projects, including transformation tools [UML19]. Additionally, the UML Testing Profile (UTP) extends UML to cover test concepts, enabling both project and test models to share common notation elements and structure [UTP08].

Many current UML software projects are developed in the context of the Eclipse Project [ECL19]. Noticeable among them are the UML2 library, part of the Model Development Tools (MDT) project [MDT19]; Papyrus, a MDT/UML2-based extendible graphical design editor [PAP18]; and the Query / View / Transform (QVT) implementations released by the Model to Model Transformation (MMT) project [MMT19]. These constitute the proposed foundations for implementing the test generation system.

The remainder of this report is structured as follows. The next section reviews existing approaches to automatic test generation from product line models, their features and shortcomings. The proposed product, variant and test models are then described, followed by the test case generation procedure and implementation guidelines. The report closes with a discussion on perspectives for the next project stages.

2. Related Work

The State-of-the-art report on requirements-based variability modelling and abstract test case generation — published as part of task T3.1 — provides a survey of current product line modelling and test generation methods. In the terms of that report, the method described here follows an hybrid approach: it uses a custom language (BVR) to perform feature modelling, UML for system architecture and behaviour modelling, and a variety of rule-based techniques to generate test suites from UML models.

3. Running Example

Automated Guided Vehicles (AGV's) are autonomous mobile systems often employed in industrial plants to transport goods around assembly lines, storage depots, delivery areas and so on [AGV06]. Traditionally, AGV's rely on rudimentary navigation systems requiring extensive environment retrofitting for the installation of guiding wires or location beacons. More recently, sophisticated navigation stacks based on Simultaneous Localization and Mapping (SLAM) and rich sensors such as depth cameras and LiDAR's (laser-based range sensors) have been promoted as a more robust approach, simplifying deployment while also enabling automatic handling of unplanned occurrences such as unexpected obstacles [AGV19].

A product line of SLAM-based AGV's can comprise a wide diversity of variants both in software and hardware features. A number of SLAM software packages are available today, each with its particular strengths and sensory requirements. Among them RTAB-Map is characteristic of the vision-based approach for pose registration, extracting features from visual data provided by stereo or Time-of-Flight (ToF) cameras to identify different locations across an environment. In contrast, range-based Google Cartographer trades more accurate localization for stricter requirements on the quality of its input depth data, all but mandating LiDAR as the main sensor solution. Both benefit from additional sensors such as Inertial Measurement Units (IMU's) and wheel encoders that provide rough estimates of the unit's current location relative to its start point. Furthermore, environment and application requirements may demand a particular drive technology — e.g. a differential or holonomic base for added maneuverability, or a tricycle drive for extra torque. Figure 1 below illustrates the differences between drive configurations.

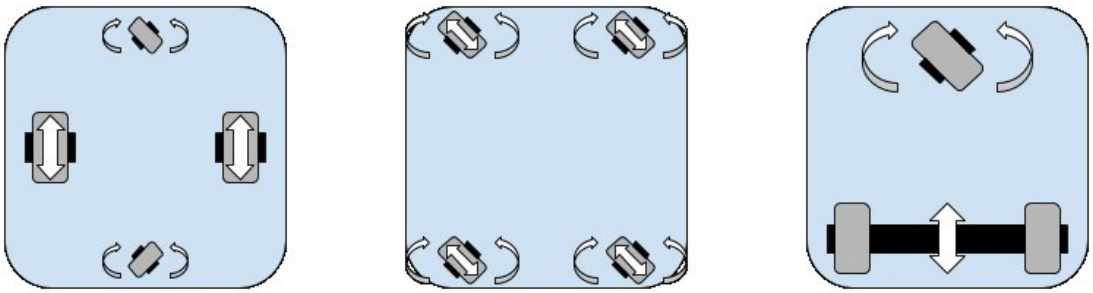


Figure 1. Mobile drive configurations. (a) In a differential drive, a pair of powered wheels can independently turn at different rates and even in opposite directions, causing the robot to move straight, turn or spin in place. A secondary pair of loose wheels is usually employed to keep the robot level. (b) In a holonomic drive, sets of independently-powered wheels allow the robot to move in arbitrary directions regardless of its heading orientation (the arrangement shown in the figure is just one of several ways to achieve that effect). (c) In a tricycle drive, a pair of fixed hind wheels powered by the same motor push the robot forward or backward, while a turning, non-powered front wheel provides direction (alternatively, the front wheel can be powered, with loose hind wheels just providing support).

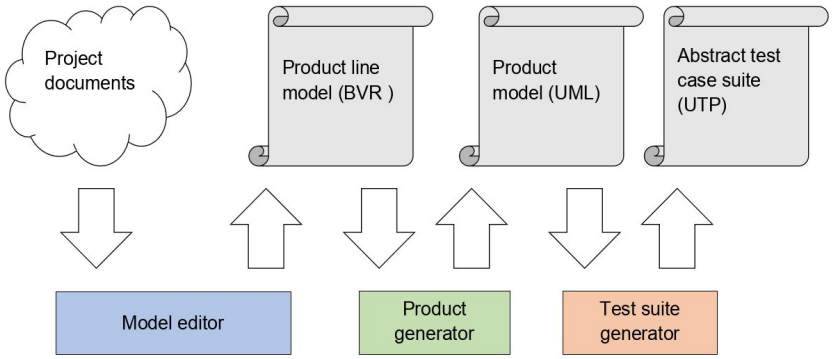


Figure 2. Abstract test case generation workflow.

4. Process Overview

The proposed test case generation process follows a multi-step workflow. Starting from a set of project documents, a *model editor* is used to generate a *product line model* that encodes relevant information about the product family. This is passed to a *product generator* that instantiates specific *product models* as guided by test metrics. Finally, a *test suite generator* instantiates a *test case suite* for each product. See Figure 2 for an illustration.

In the SPES XT modelling framework, a *viewpoint* is "a pattern or template that can be used to develop individual views on a system (and its environment)" in order "to separate the various concerns of different stakeholders during the engineering process". "Typically, the specification of a viewpoint defines that viewpoint in terms of its syntax, semantics, and pragmatics by providing, among other things, the name of the viewpoint, the corresponding stakeholder concerns, the viewpoint language (probably given by a metamodel), and techniques that can be used during the construction and analysis of the corresponding view" [SPE12, p. 36].

Accordingly, the models used to describe products, product lines and test case suites are defined as custom SPES viewpoints, inheriting the concepts of the basic framework and extending them to suit their respective purposes. The next sections will elaborate on each one of those.

4.1. Product Viewpoint

The *product viewpoint* describes a product model that captures precise specifications across three aspects:

- The product *domain*, which allows identification of *test components*;
- The product *architecture*, which allows selection of Systems Under Test (SUT's);
- The product *behaviour*, which allows derivation of *test cases*.

In the product viewpoint, the domain is represented by a Use Case diagram. Its purpose is to catalogue the entities that interact with the system (providing inputs and/or receiving its outputs) and the user functions [SPE12, p. 40] that implement such interactions.

Product architecture is represented by a Class diagram. It records the logical components that realize the functionalities provided by the user functions [SPE12, p. 41], as well as their relationships. This is also where intra-product variability is described through class inheritance relationships and object attributes.

Finally, product behaviour is represented by Sequence diagrams detailing how each user function is realized by message exchanges between system classes. These are the main input to the test generation process.

4.2. Product Line Viewpoint

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's composed of four parts, Feature model, Domain, Resolutions and Realizations.

4.2.1 Feature model

A feature model describes the entire product line in terms of abstract functional characteristics (features), their optionality and interdependence / exclusion relations.

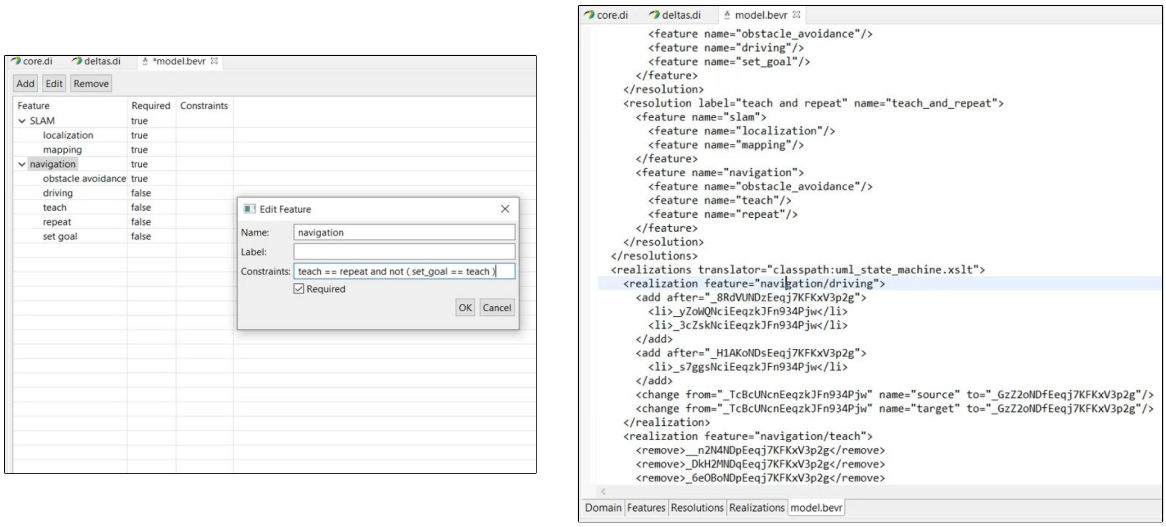
Feature models are described in the BeVR language using the State machine diagram. The BeVR language in loosely based on the BVR language [BVR14]. The BVR tool bundle [BVR15] [BVR18] is an Open Source implementation of the BVR language as a set of Eclipse IDE plugins. It integrates to Papyrus, an Eclipse UML editor [PAP18] allowing UML models to be used as realization artefacts. Together they already provide most of the features required from the model editor and product generator. However development of the BVR tool bundle has fallen behind the rest of the Eclipse ecosystem (notably Papyrus); it's also somewhat unpolished, and misses a couple relevant features — for example, adding artefacts to the core model without removing existing parts in the process.

To overcome these shortcomings, in BeVR tool, we have redesigned the variability modelling language in a free XML format which makes it easy for receiving human inputs. In this version, Eclipse Modeling Framework (EMF) has been dropped for a more human-friendly style. Also, a feature has been added which enable adding an artifact to the variant without the need to replacing existing parts. In addition, bidding process of domain artifact to the features is also simplified.

Features can be added, edited or removed and those are needed in all variance can be marked as *Required* (Figure 3). Constraints can be specified in a simple logic language for features in terms of sub-features. As the changes are made in the visual page, the BeVR model is written and updated in the plain-text file.

4.2.2 Domain

Domain describes the artifacts that are going to be manipulated. It includes the Core model which represents the minimum common set of features and Δ-models describing delta (Δ) artifacts added to the core model to generate variance. An example of core and Δ-models for SLAM-AGV product are illustrated in Figure 4 and 5. The core product is described in UML as specified in the Product Viewpoint. Figure 4 shows the SLAM-AGV core product illustrated in state diagram.



a. Visual edit feature

b. BeVR model in the plain-text file

Figure 3. Adding features in feature editor (a) and BeVR model generated in plain text file for the SLAM-AGV product line (b).

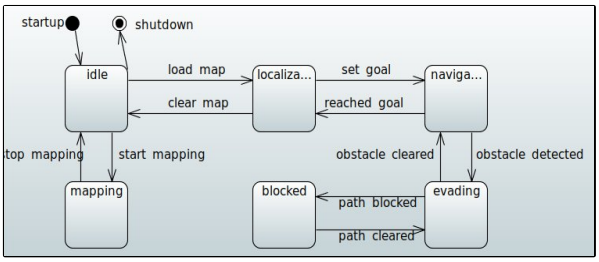


Figure 4. The core model represents the minimum set of common features across the product line for the SLAM-AGV product line, represented in a Papyrus Model.

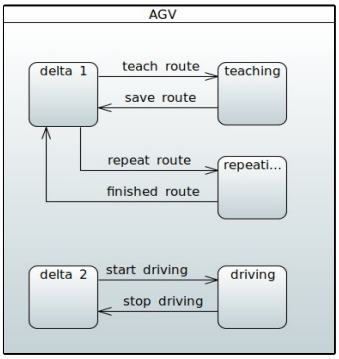


Figure 5. The model contains dummy states ‘delta 1’ and ‘delta 2’ which are used to define transition to the core model.

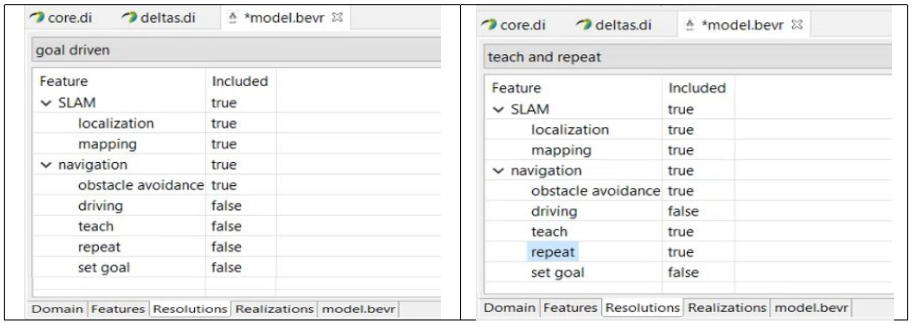
Starting state represented by an idle state from the idle state the AGV can either map the environment or use the map for sub localization and then proceed to driven navigation and so on to different states.

The Δ-models (Figure 5) contains some optional features (states) that can be added to introduce a variance. The core and Δ-models are combined in the BeVR Model. Deltas are represented by BeVR *fragment substitutions*, which bind a *placement* (a collection of core model parts) to a *replacement* (a collection of delta parts) and the feature it realizes.

4.2.2 Resolutions

Resolutions are instances of the feature models that represent a specific product. When creating a resolution, features that are required are automatically added to the model. While creating new resolutions, additional features can be marked as *Required* (see Figure 6).

In this example, “goal driver” resolution requires features that are automatically included (Figure 6a). While for creating a new resolution for “teach and repeat”, new features need to be added as shown in (Figure 6b).



a. b.

Figure 6. Creating resolutions for the proposed product line.

4.2.2 Realizations

Realizations bind specific resolutions to the domain artifact. In the realization editor resolutions are specified for selected features (Figure 7). Delta states from the Δ-models are linked to states in the core model connecting source and target transitions.

The new BeVR model showing the variance with respect to the core model is depicted in Figure 8. The new states and transitions can be observed under region group.

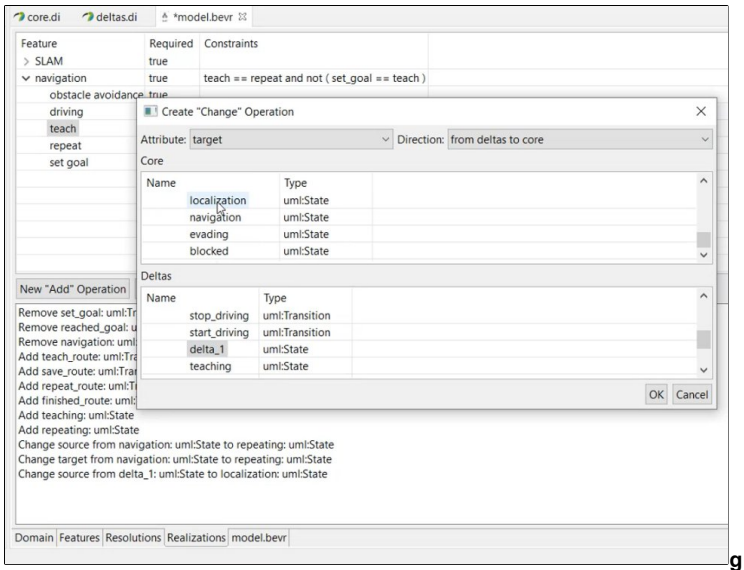


Figure 7. Specifying states and transition mapping from Δ-models to core model product realization involves creating the variance.

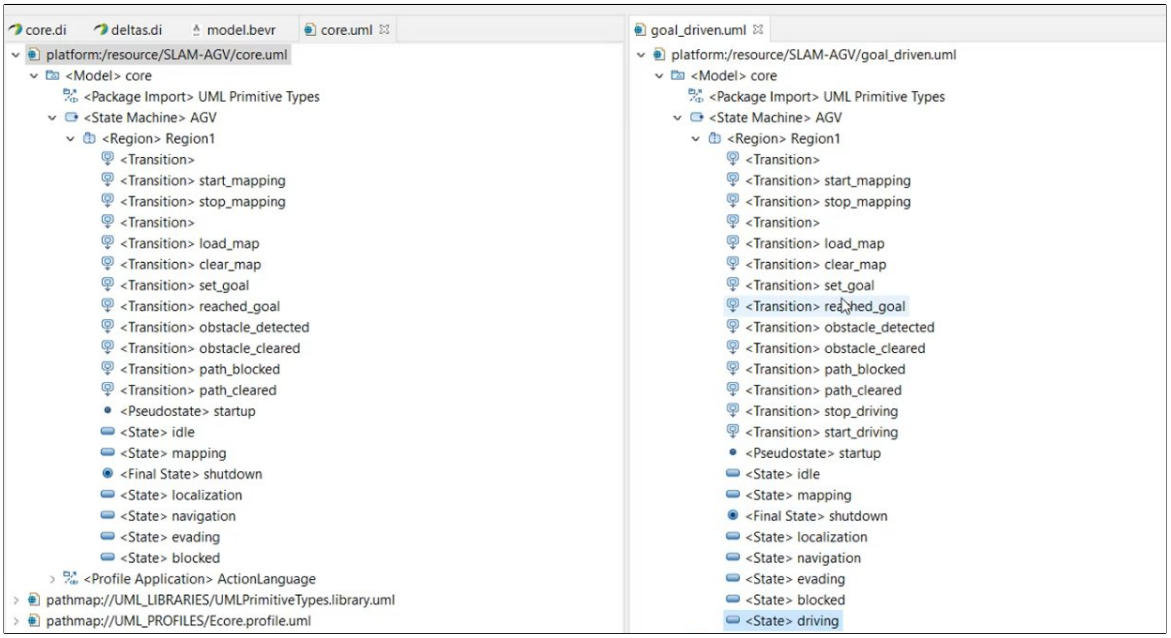


Figure 8. Comparing the new BeVR model shows the variance with respect to the core model. The BeVR model showing states and transitions that were added.

4.3. Test Viewpoint

The *test viewpoint* specifies an abstract test suite for an associated product, and is composed of:

- An *architecture* presenting the set of *test contexts* used to collect individual *test cases*, the *test components* that simulate external actors, the *data pools* that enable retrieval of required data to reproduce specific scenarios, and the product's *Systems Under Test* (SUT's) — i.e. the logical components that are to be exercised by the test cases;
- The *test cases* proper, describing the sequences of interactions among SUT's, test components and data pools that constitute tests, as well as expected results.

The architecture and test cases are represented as UML Testing Profile (UTP) diagrams. Specifically, the architecture is recorded as a Class diagram, whereas individual test cases are written as Sequence diagrams.

5. Test Suite Generation

A test suite can be generated by mechanically transforming a product model [AMT09]. Algorithm 1 below describes a procedure for performing such a transformation.

```
Create a Test Suite

For each Actor in the Product Model:
  Create a Test Component in the Test Suite

For each Use Case in the Product Model:
  Create a Test Context in the Test Suite

For each Sequence diagram Si in the Product Model:
  Add a Test Case Ti in the Test Context TXu of the corresponding Use Case
  Create a Sequence diagram TSi in the Test Suite
  Copy all non-Actor lifelines and exchanged messages from Si to TSi
  Mark all lifelines copied into TSi with the <<SUT>> stereotype

  For each Actor lifeline Aij in Si:
    Add a lifeline for the corresponding Test Component TAij in TSi

  Create a Data Pool DPi in the Test Suite
  For each message Mij from an Actor Aij to a lifeline SUTik in Si:
    For each parameter Pijk in Mij:
      Create a data selection operation for Pijk in DPi
      Add to TSi a call from TAij to DPi retrieving a value for Pijk
    Add to TSi a Mij exchange from TAij to SUTik set up with the given Pijk

  For each message or return value sent to TAij in TSi:
    Create a validation action in TXu
    Add to TSi a validation action call for the message or return value
```

Algorithm 1. Converting a product model into a test suite.

In short, the algorithm above applies the following set of simple rules:

- Each Use Case is transformed into a Test Context;
- Each Actor is transformed into a Test Component;
- For each Sequence diagram, a corresponding Test Case is created, containing all lifelines and message exchanges in the original diagram, with the following changes:
  - Every Actor is replaced with the corresponding Test Component;
  - For messages sent from a Test Component to the SUT, any parameter values are retrieved from a data pool created for this particular test case;
  - Return values and messages sent from the SUT to a Test Component are passed to a corresponding validation action.

5.1. Variant Selection

Given a test suite, the order in which it is executed has influence on certain quality measures like early error detection or requirement coverage. So, in the same way, given a list of variants to test, it is of interest in which order to test them in order to optimize the above properties. Also, time for testing is limited and it often might only be feasible to test a subset of all possible variants. So achieving high coverage with this subset is necessary.

In the following, we will explicate solutions for the situation where a set of variants is fixed and has to be prioritized for testing. Note that this differs from the approach to generate variants from a feature model for testing purpose.

If the latter variant generation approach is chosen and a test suite generation algorithm is implemented, a selection of product models must be instantiated from the product line model. For the generation of variants for testing purposes, methods for, e.g., achieving combinatorial coverage can be adopted from their original application area of input configurations, as done in [SPT11], [SPT13]. These methods are thoroughly studied and widely implemented, see e.g. [DPP17]. Nonetheless, in most applications, the existing variants only form a small subset of the possible configurations of the product model. To implement this fact in the model, a variety of constraints would necessarily have to be set for the model. These constraints don't stem from actual restrictions of the product, but rather from exterior circumstances that are difficult to foresee. The logic of the model is therefore affected. Furthermore, too many constraints are often a crucial obstacle for the effectiveness of the corresponding algorithms.

We will take the approach of deriving variants first, and then prioritizing them as explicated in the following. First, we will elaborate on the testing goals in terms of coverage criteria.

5.2. Coverage Criteria

In order to give an algorithm for variant selection, the coverage goal of the test suite has to be chosen in order to appropriately adapt the algorithm. For error detecting, combinatorial, in particular pairwise testing has proven to be effective, see, e.g., [COM11], and the case studies [COM04] and [COM16], as the majority of errors arises from features or the interaction of two features. Furthermore, it was shown that in certain circumstances, the presence as well as the absence of features can affect the behavior of other features [VAB14]. For the different modelling approaches described above (feature vs. delta modelling), different potential coverage criteria arise from these considerations.

In terms of feature models, the following approaches to coverage can be taken:

- Combinatorial coverage for features (simple, pairwise, etc.);
- Combinatorial occurrence coverage for features (also simple, pairwise etc.). This means that the absence of a feature is assessed like an alternative feature, as it can affect the behavior of other features [VAB14];
- A mixed approach is also feasible, where the absence of a feature is weighted in a different way (possibly less impacting) than the presence of a feature.

In terms of Δ-models, these criteria are equivalent to the following approaches to coverage:

- Delta coverage without taking removal deltas into account.
- Delta coverage with taking removal deltas into account.
- As in feature coverage, a mixed approach is feasible, where removal deltas are weighted in a different way.

A natural goal for industrial application is requirement coverage. This presumes that requirements are stated globally, i.e., are not phrased for a specific variant but for the feature model itself. That means particularly that requirements have to be explicitely linked to features, describing the functionality of a feature or the interaction of a system of features.

If these prerequisites are met, a more elaborate algorithm is necessary for accomplishing requirement coverage, as requirements can be linked to varying numbers of features, possibly with specific parameter values as an input.

5.3. Variant Selection Algorithm

Note again that by the line of arguments in 5.1, we are not considering generation of variants from a model for testing purposes, but rather selection/prioritization of variants from a given set of already generated variants. Given a set of variants chosen from the feature model, possibly represented by a core model and Δ-models, there are several ways to prioritize them according to the above coverage goals. A natural approach is to consider the set of objects to cover (features, deltas, pairs, requirements...) and choose the variant that covers the most objects, then the one that covers the most additional objects etc., in order to ensure the hightest possible coverage after each additionally tested variant. This is the approach that we will choose for now. The method can be refined by weighting the objects according to their “importance” to test them, as done in [PIT06] for pairwise coverage. In the article, furthermore, constraints for pairs are modelled by negative weights.

It should be mentioned here that to avoid scalability issues for higher levels of combinatorial coverage, a feasible concept to variant selection and prioritization is a similarity based approach as described in [DPP17] or [SBP14]: Variants that differ in many features are heuristically prone to give a better level of the desired coverage than similar ones. To measure the distance between variants, there are different measures at hand (Hamming, Jacquard, Dice, etc.). Examples, evaluations and references can be found in [SDM18]. In the delta based prioritization algorithm in [DPP14], the Hamming distance is applied.

As elaborated earlier, here, a greedy algorithm for variant selection will be implemented that covers the maximal additional number of objects (features, deltas, pairs) in each step, i.e., for each newly selected variant. That is, given the case the test suite is aborted at any step, for time or other reasons, the corresponding coverage reached is as high as possible. Algorithm 2 below provides a pseudocode description of the procedure.

```
# Select a subset of a list of variants for testing purposes/prioritize the list.
#
# Notes:
# * V1,...,Vn denotes the variants derived from the product model.
# * S1,...,Sn denote the sets of target objects covered by these variants,
#   e.g. S1 is the set of features or pairs covered by V1.
# * S=S1∪...∪Sn is the set of all objects to be covered.
#
# Arguments:
# * V=[V1,...,Vn] list of variants
#
SELECT_VARIANTS(V):
  # Initialize the set of already tested objects and an empty list.
  Stested=∅
  Vnew=[]

  Until Stested=S do:

    # Select the variant Vk that gives the most new/not yet tested objects.
    k=argmaxi=1,...,n|Si∩(S\Stested)|

    # Add the variant Vk to the (end of the) new list of prioritized variants
    # and add the newly covered objects to the set Stested.
    Vnew=Vnew+Vk
    Stested=Stested∪Sk

  # If S=Stested, and not all variants are in Vnew yet, either output the
  # shorter list, or add the remaining variants to Vnew (possibly in an
  # order determined by similarity.

  RETURN Vnew
```

Algorithm 2. Selecting products for test.

Note that this algorithm is a variant of an algorithm that appears in the literature several times, e.g. in [DPP17] and [SBP14]. It varies in the way that the distance of the not-yet tested variants to the set of tested variants is measured: Instead of taking the maximum of the distances of a new variant to all already tested variants, or summing these distances up, we take a set-based approach. The features/deltas etc. of the already tested variants are gathered in a set and the distance to this set is measured. This is a natural approach for reaching the coverage goal. However, for, e.g., pairwise or higher combinatorial coverage, this becomes more expensive, as sets of pairs of features have to be managed and checked.

Another issue here is that the algorithm stops if the desired coverage is reached. A resolution here would be to go on with similarity-based algorithms as in [SBP14] as soon as the initial coverage goal is reached. The distance of a variant to the already tested set is computed by taking the minimum over all distances, or summing up the distances. Therefore, the distance of a variant to the already tested set only becomes 0 when the variant is already contained in the set. More and more variants can be added that are heuristically most unsimilar to the set and therefore extend the error-detecting properties of the test suite.

Note that the above algorithm is in the most general form and applicable for any of the coverage criteria in 5.2. Objects to be covered can be absent/present features, pairs, deltas, or requirements. We will give to concrete examples below.

For the two coverage goals feature/delta coverage, the distance measure $|S_j \cap (S \setminus S_{tested})$ specifies as follows:

1. Feature coverage: Let $S = F(PL)$ be the set of all possible features in the product line. Let $S_{tested} = F(P_{tested})$ be the set of all features that are contained in already tested products, and $S_j = F(P_j)$ be the features of the product P_j. Then, the corresponding metrics are:
   a. $|(F(PL) \setminus F(P_{tested})) \cap F(P_j)|$
   b. $|(F(PL) \setminus F(P_{tested})) \cap F(P_j)| + |(F(PL) \setminus F(P_j)) \cap F(P_{tested})|$
   c. $(1-\alpha)|(F(PL) \setminus F(P_{tested})) \cap F(P_j)| + \alpha|(F(PL) \setminus F(P_j)) \cap F(P_{tested})|$

   Note that b. is just the Hamming distance.

2. Delta coverage: Let $D_{add}(PL)$, $D_{mod}(PL)$, $D_{rem}(PL)$ be the sets of addition, modification and removal deltas in the product line, and correspondingly for the already tested products P_{tested} or a product P_j. Then, the corresponding metrics are:
   a. $|D_{add}(PL) \setminus D_{add}(P_{tested}) \cap D_{add}(P_j)| +$
      $|D_{mod}(PL) \setminus D_{mod}(P_{tested}) \cap D_{mod}(P_j)|$
   b. $|D_{add}(PL) \setminus D_{add}(P_{tested}) \cap D_{add}(P_j)| +$
      $|D_{mod}(PL) \setminus D_{mod}(P_{tested}) \cap D_{mod}(P_j)| +$
      $|D_{rem}(PL) \setminus D_{rem}(P_{tested}) \cap D_{rem}(P_j)|$
   c. $(1-\alpha)|D_{add}(PL) \setminus D_{add}(P_{tested}) \cap D_{add}(P_j)| +$
      $(1-\alpha)|D_{mod}(PL) \setminus D_{mod}(P_{tested}) \cap D_{mod}(P_j)| +$
      $\alpha|D_{rem}(PL) \setminus D_{rem}(P_{tested}) \cap D_{rem}(P_j)|$
3. Requirement coverage: For requirement coverage, one could consider a set S that consists of subsets of features that are linked to requirements. For each requirement, one identifies the feature/s that have to be present in order to test that requirement. As the sets can be of various size, and other factors can be relevant for requirement coverage, we do not further elaborate this for now.

Evaluations of the efficiency of the algorithm with respect to reducing testing efforts can be found in D3.5.1.

6. Implementation

As mentioned in the overview section, the proposed test case generation process can be implemented as a set of three tools:

- A *model editor* for creating the Product Line Model (PLM);
- A *product generator* that instantiates Product Models from a PLM;
- A *test suite generator* that produces a Test Suite from an individual Product Model.

As the names suggest, the model editor is a user-facing application that requires input from a human operator, while the product and test suite generators are automatic tools that only require an input model (and possibly configuration parameters) to work. Accordingly, the first tool will require a relatively elaborate GUI interface, while the latter two can be implemented as command-line tools or plugins with a comparatively simple interface.

Test suite generation will require extensive manipulation of product models. The UML2 library (part of the Model Development Tools project [MDT19]) enables programmatic access to UML project files created in Papyrus, while the Query / View / Transform (QVT) implementations released by the Model to Model Transformation project [MMT19] can be used to simplify the implementation of the test case generation process outlined in Algorithm 1.

It goes without saying that as the language of implementation of most of the above components, Java will be used to write the model generators and any code customizations that might be required by the model editor.

7. Conclusions

This report proposed a methodology for automatically generating a suite of abstract test cases for a product line — a family of artefacts related by shared features and development history. The presentation was roughly divided in three parts: definition of models for product lines and test suites, techniques for automatic generation of abstract test cases, and considerations on implementation.

Theoretical concepts are built upon the SPES XT methodology. Custom Product Line, Product and Testing viewpoints were defined, providing the notation for modelling product lines, individual products and abstract test suites respectively. The Product Line viewpoint is largely an application of the BeVR language, while the Product and Testing viewpoints are built upon UML and the UML Testing Profile (UTP).

An algorithm for automatic generation of test cases through transformation of UML models was then presented, followed by a discussion on criteria for test coverage determination. Different options for variant selection were presented and contrasted, from heuristic to optimal. An argument was made for an optimal approach that is flexible enough to allow different coverage criteria to be used according to the needs of specific projects.

Finally, implementation was considered. A summary outline of the system architecture was given, and existing tools and libraries that can be used as basis for realizing it are presented. A case was made for implementing the tool suite in Java as a set of Eclipse IDE plugins, following the example of other projects in the area of variability modelling.

As this report was mostly concerned with the theoretical side of the proposed automatic test generation method, following activities must concentrate on implementation. The initial discussion developed here must be complemented by a more detailed analysis of system requirements and architecture, identification of shortcomings in existing tools, implementation of missing features and experimental evaluation. In particular, since increased productivity is a central objective in the XIVT project, the matter of user friendliness is expected to become increasingly important as we move from design into implementation and then deployment.

8. 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.

Appendix A - Tool Releases

Tool releases are hosted at https://gitlab.com/xivt/itea. Repositories can be accessed using the following credentials:

- **Username:** `ITEA3XIVT`
- **Password:** `20222018XIVT`

See the next sections for further details.

BeVR

https://gitlab.com/xivt/itea/bevr

BeVR (pronounced "beaver") is a fork of the original Base Variability Resolution (BVR) tool set. It is a set of plug-ins for Eclipse that implements and supports the BVR language. It enables feature modelling, resolution and realization of UML products.