Model-based Automation of Test Script Generation Across Product Variants: a Railway Perspective

Alessio Bucaioni*, Fabio Di Silvestro†, Inderjeet Singh†, Mehrdad Saadatmand‡, Henry Muccini§, Thorvaldur Jochumsson†
* alessio.bucaioni@mdh.se, Mälardalen University, Västerås, Sweden
† fabio.di_silvestro1@rail.bombardier.com, inderjeet.singh@rail.bombardier.com, thorvaldur.jochumsson1@rail.bombardier.com, Bombardier Transportation, Västerås, Sweden
‡ mehrdad.saadatmand@ri.se, Research Institutes of Sweden, Västerås, Sweden § henry.muccini@univaq.it, University of L'Aquila, L'Aquila, Italy

***Abstract*—In this work, we report on our experience in defining and applying a model-based approach for the automatic generation of test scripts for product variants in software product lines. The proposed approach is the result of an effort leveraging the experiences and results from the technology transfer activities with our industrial partner Bombardier Transportation. The proposed approach employs metamodelling and model transformations for representing different testing artefacts and making their generation automatic. We demonstrate the industrial applicability and efficiency of the proposed approach using the Bombardier Transportation Aventra software product line. We observe that the proposed approach mitigates the development effort, time consumption and consistency drawbacks typical of traditional strategies.**

I. Introduction

Among others, the railway sector has witnessed an increased demand for customised software-intensive systems for addressing market needs, regional standards, certifications as well as software and hardware requirements. To meet this increased customisation demand, Bombardier Transportation (BT) has been shifting towards Software Product Line (SPL) Engineering (SPLE) [1].

A SPL is a set of software-intensive systems, which share a common, managed set of features that are developed starting from a common set of core artefacts [2], e.g., requirements, test cases, etc. Traditionally, SPLE makes use of two development processes: the domain engineering and application engineering processes. The domain engineering process focuses on the creation of the SPL platform through the identification of common and variable features and the development of domain artefacts realising such features. The application engineering process focuses on the derivation of individual systems based on the SPL platform identified in the domain engineering process [2]. The main benefit of using SPLs is that all the artefacts can be systematically organised and reused [3] translating in lower cost, shorter time to market and increased quality than the development of multiple and independent systems [1]. Despite the adoption of the SPLE has brought several benefits to BT, it has also introduced new challenges to testing. Within BT, each train is customised and its software system is highly-coupled to the underlying electrical system and hardware. This means that technical artefacts, e.g., test scripts, designed for common features of the SPL need to account for the different hardware configurations of the products in the family and can not be directly reused throughout the SPL. To clarify this challenge, let us consider the case of the BT Aventra SPL.

A. A Real-world Scenario From The Railway Domain

The BT Aventra SPL consists of five electric unit trains for passengers transportation specifically designed for the British Market: the London Overground (LOT), the East Anglia (EAA), the South Western (SWR), the West Midlands (WML) and the Center-to-Coast (C2C)[1]. All the trains in the Aventra SPL share a large number of features, from basic, (e.g., driver cabin activation, doors activation, etc.) to more advanced ones (e.g., safety-related, train re-configurations, etc.). The Traction/Brake Control feature (TBC) is one of such features and is responsible for transmitting the driver inputs to the brake and propulsion systems by operating on the train communication channels, which are called signals. Despite the TBC control logic is the same for all the trains in the SPL, the signals on which it operates are different due to the different train architectures. Such heterogeneity of signals represents a major challenge for testing phases where different test scripts accounting for different signals need to be created (in fact, the behaviour of the TBC does not vary among the different trains). To tackle this challenge, BT has adopted the so-called *opportunistic reuse* of test artefacts strategy for the testing of SPLs [4]. This strategy requires test scripts to be developed for one train only and replicated for the remaining ones. Each replica is manually modified so to account for differences. However, the opportunistic reuse of test artefacts strategy carries several drawbacks, including the following.

- Development effort: manually modifying software artefacts is not only undesirable but often unfeasible for industrial SPLs.
- Error proneness: manual changes are more likely to introduce errors; besides, replicating artefacts across the SPL may propagate them.
- Consistency: manual changes make difficult to keep artefacts consistent as changes to the original artefact need to be explicitly propagated to the replicas.

B. Paper contribution

In this paper, we report on our experience in tackling the above challenge of developing test scripts stemming from common SPL features and accounting for product differences in the railway domain. To this end, together with our industrial partner BT, we define a light-weight model based approach. The approach uses metamodels, Domain-specific Languages

[1]More details on the Aventra SPL are provided in Section III.

(DSLs) and model transformations for the automatic generation of test scripts from abstract test case descriptions[2]. The main building blocks of our proposed approach are:

- a domain-independent metamodel for the functional[3] representation of SPL common features,
- a domain-independent metamodel for the representation of individual trains, train features and signals,
- a domain-independent weaving metamodel for mapping individual train signals to features inputs and outputs,
- a DSL for the specification of test cases, and
- a model to text transformation for the automatic generation of executable test scripts.

We use the BT Aventra SPL for evaluating the applicability and efficacy of the proposed approach. We evaluate the applicability of the proposed approach in industrial settings and its efficiency in generating executable test scripts which are equivalent to those created manually using the opportunistic reuse of testing artefacts strategy. We discuss the industrial relevance of our approach together with experts from the Aventra integration team using the model for assessing the industrial relevance of technology transfers introduced by Ivarsson et al. [6]. We conclude that the approach mitigates the development effort, error proneness and consistency drawbacks of the opportunistic reuse of testing artefacts strategy already for SPL containing 3 products and 2 features. The model transformation contributes to lower the effort required for the creation of test scripts and, together with the metamodels, reduces the possibility of introducing and propagating errors, and keeps the artefacts consistent.

C. Structure of the paper

The remainder of this paper is organised as follows. Section II describes the proposed approach in terms of its components and steps. Section III presents the application of the approach on the BT Aventra SPL. Section IV discusses its applicability, efficiency and industrial relevance. Besides, in this section an assessment of the development effort of the proposed approach is also provided. Section V describes potential threats to validity and related mitigation strategies. Section VI presents related work, and Section VII concludes the paper with final remarks and possible future work.

II. A MODEL-BASED APPROACH FOR THE AUTOMATIC GENERATION OF TEST SCRIPTS

In this section, we describe how the approach supports the generation of individual test scripts stemming from common SPL features and accounting for train differences. Figure 1 provides an overview of the proposed approach in terms of its main components and steps. In particular, the proposed approach includes the following components:

- *SPL metamodel (SPLmm). SPLmm* is a metamodel for defining SPL platform features.
- *Products metamodel (Pmm). Pmm* is a metamodel for designing individual trains in terms of their features and signals.
- *Weaving metamodel (Wmm). Wmm* is a metamodel for linking features and signals in *Pmm* to those in *SPLmm*.
- *Test case DSL (TcDSL). TcDSL* is a DSL for the specification of abstract test cases for features in *SPLmm*.
- *Test Script generation Transformation (TsT). TsT* is a model to text transformation for the automatic generation of executable test scripts from test cases described using *TcDSL*.

A typical execution of the approach involves five steps. These can be grouped into two conceptual phases, as follows. Steps 1 to 3 belong to the definition phase (red box in the upper half of Figure 1), while steps 4 and 5 belong to the execution phase (grey box in the lower half of Figure 1). The steps in the definition phase are preparatory for those in the execution phase and need to be executed only once per SPL or whenever a change in the SPL occurs. It should be noted that, in the case the information captured from *SPLm*, *W* and *P* are already formalised using other artefacts or notations, tasks 1 to 3 can be skipped.

- Step 1. In this step, marked as a black circled 1 in Figure 1, engineers are required to create a model capturing the SPL platform features (*SPL platform model (SPLm)* in Figure 1). This is done by using the *SPLmm*. The goal of this step is to provide a functional representation of the common SPL features.
- Step 2. This step, marked as a black circled 2 in Figure 1, requires engineers to create a model capturing individual trains belonging to the SPL and their signals (*Products model (P)* in Figure 1) to describe possible differences among SPL products. This is done by using the *Pmm*.
- Step 3. The goal of this step, marked as a black circled 3 in Figure 1, is to relate features and signals of individual trains to the shared ones. To this end, engineers are required to create a weaving model (*Weaving model (W)* in Figure 1) linking elements of *P* to elements of *SPLm*. This is done by using the *Wmm*.
- Step 4. In this step, marked as a black circled 4 in Figure 1, test engineers are required to create test cases (*Test case (Tc)* in Figure 1) describing the checks to be performed on the common SPL functionalities as captured by the *SPLm*. This is done by using the *TcDSL*.
- Step 5. The last step, marked as a black circled 5 in Figure 1, is the execution of *TsT* for the automatic generation of executable test scripts from the test cases specified in *Tc*. For each train in the SPL, *TsT* i) translates the abstract checks specified in *Tc* into a concrete set of instructions and ii) replaces the features input and output with the train signals, using the information in *W*.

In the remainder of this section, we provide a detailed description of the enabling artefacts of the proposed approach, which can be accessed at https://github.com/fabiodisilv/Model-Based_Test_Generator_SPL . The approach execution on the BT Aventra SPL is presented in Section III.

A. SPL metamodel

SPLmm allows for the representation of the SPL features. We have developed *SPLmm* as an Ecore model within the

[2]In this paper, we use the term abstract test case to refer to a generic description of the test to be performed. We use the term test script to refer to a concrete set of instructions or short program implementing the test case. In the remainder of this paper, we refer to abstract test case simply as test case.

[3]In this paper, we borrow the concept of functional abstraction level defined in [5] for architectural frameworks.

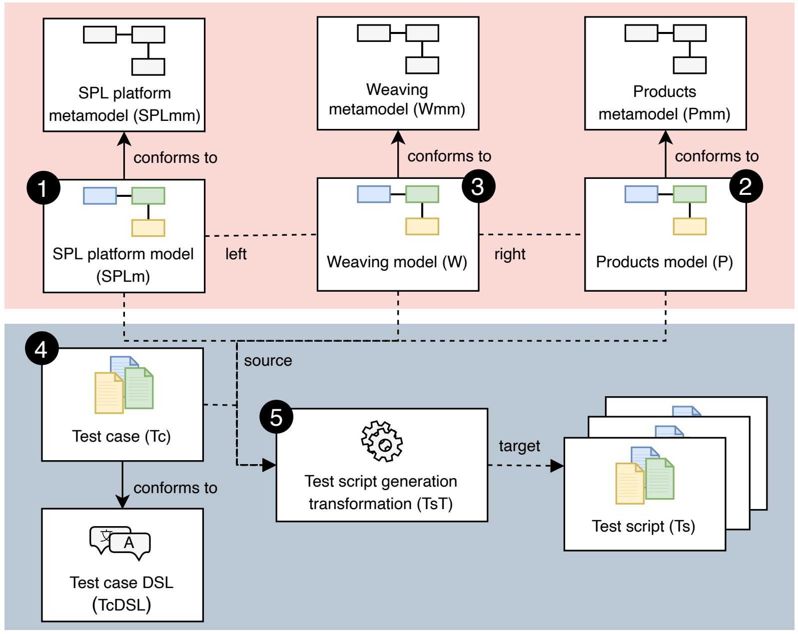


Fig. 1: Proposed approach.

Eclipse Modelling Framework (EMF)[4]. Figure 2 provides for a class diagram representation of *SPLmm*. The root metaclass is

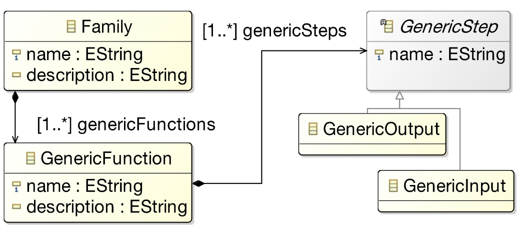


Fig. 2: Class diagram representation of *SPLmm*.

Family, which acts as a container and has two attributes: *name* and *description*. *Family* contains one or more *GenericFunction* metaclasses, which specify common features of the SPL. A *GenericFunction* class has two attributes namely *name* and *description* and contains one or more *GenericStep* metaclasses. *GenericStep* metaclasses represent the input and output of the feature and have one attribute, *name*. Accordingly, a *GenericStep* metaclass can be specialised into a *GenericInput* or a *GenericOutput* metaclass.

B. Products metamodel

Pmm allows for representing individual products of the SPL along with their differences. We have developed *Pmm* as an Ecore model within EMF and Figure 3 provides for its class diagram representation. Similarly to *SPLmm*, the *Pmm* root metaclass is *Family*. *Family* has two attributes, *name* and *description*, and contains one or more *Product* metaclasses. A *Product* metaclass specifies a train belonging to the SPL and has one attribute, *name*. A *Product* metaclass refers to one or more *ProductSpecificFunction* metaclasses, which are used to represent train specific versions of common features. A

[4]https://www.eclipse.org/modeling/emf/

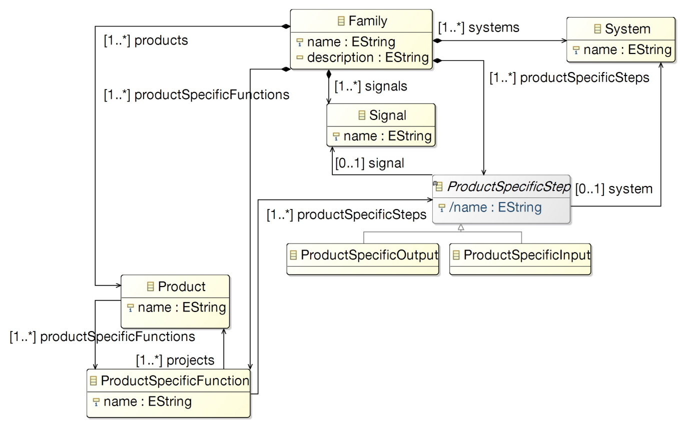


Fig. 3: Class diagram representation of *Pmm*.

ProductSpecificFunction metaclass has one attribute and refers to one or more *ProductSpecificStep* and one or more *Product*. A *ProductSpecificStep* metaclass might be specialised into *ProductSpecificInput* and *ProductSpecificOutput*. Besides, a *ProductSpecificStep* metaclass might refer to a *Signal* and a *System* metaclass, where Signals metaclasses represent concrete trains signals.

C. Weaving metamodel

Wmm allows for specifying links between elements of *P* and elements of *SPLm*. The information captured using *Wmm* is used from *TsT* to generate executable test scripts. We have developed *Wmm* as an Ecore model within EMF and Figure 4 provides for its class diagram representation. The root metaclass is *Weaving*, which has two attributes, *name* and *description* and contains one or more *FunctionLink* metaclasses. *FunctionLink* metaclasses type one *ProductSpecificFunction* to one *GenericFunction* and contain a list of *InputLink* and *OutputLink* metaclasses. The former is used to type *Product-*

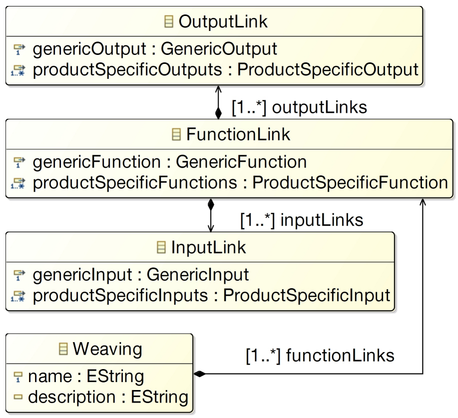


Fig. 4: Class diagram representation of *Wmm*.

SpecificInput to *GenericInput* elements while the latter for typing *ProductSpecificOutput* to *GenericOutput* elements.

D. Test case DSL

TcDSL is a language for the definition of abstract test cases. Within the proposed approach, *TcDSL* is used for writing test cases for SPL common features.

```
1  TestSuite: (testCases+=TestCase)* (productTestCases+=
       ProductTestCase)*;
2
3  TestCase: 'TestCase' name=ID 'checks' genericFunction=ID
       productException+=ProductException* '{' (steps+=Step)* '}';
4
5  ProductException: 'except' 'Product' productName=ID;
6
7  Step: Set | Check | Force | Unforce;
8
9  Set: 'Set' 'Signal' genericSignal=Signal 'to' value=Value
       productValueExceptions+=ProductValueException*;
10
11 Check: 'Check' 'Signal' genericSignal=Signal 'to' value=Value
       productValueExceptions+=ProductValueException* 'timeout'
       timeout=Timeout;
12
13 Signal: name=ID;
14
15 Value: name=ValueType;
16
17 Timeout: name=INT;
```

Listing 1: Excerpt of *TcDSL*.

We have developed *TcDSL* using the Xtext programming languages development framework[5]. Listing 1 shows an excerpt of the *TcDSL* definition describing its main concepts. *TcDSL* allows for the definition of test cases by specifying the test case name and the common feature to test (line 3 of Listing 1) as modelled in *SPLm*. The body of a test case is composed of a list of operations interacting with the feature inputs and outputs. These operations are *Set*, *Force*, *Unforce* and *Check* (line 9 of Listing 1). Lines 11 to 13 of Listing 1 show the behaviour of the *Set* and *Check* operations, respectively. The *Set* operation is responsible to specify the value of an input or output, while the *Check* operation is responsible for controlling their value within a given timeout. It might happen that a test case or some of its operations do not apply to a given train of the SPL. In this case, *TcDSL* allows for adding exceptions as shown by lines 11 and 13 of Listing 1. Listing 2 shows an example of test cases defined using *TcDSL* namely *Check_OpenDoor* and *Check_OpenDoor_Exception*. Both test cases refer to the common feature *OpenDoor* responsible for operating train doors. *Check_OpenDoor* performs two operations being a set and a check. The set operation specifies the values of *DoorLocked* to *False*, while the check operation tests that *Doorstate* is set to *OPEN* within a timeout of 5000 milliseconds. *Check_OpenDoor_Exception* defines an exception on the *Check_OpenDoor* test case for *product_A* (line 5 of Listing 2).

```
1  TestCase Check_OpenDoor checks OpenDoor{
2    Set Signal DoorLocked to False
3    Check Signal DoorState to OPEN timeout 5000}
4
5  TestCase Check_OpenDoor_Exception checks OpenDoor except
       Product product_A{
6    Set Signal DoorLocked to False (Exception Product product_A to 1)
7    ...}
```

Listing 2: Examples of test cases for the OpenDoor common functionality specified using *TcDSL*.

```
[file ('TestSuite' .concat(aProduct.name.concat( '.cs' )),false ,'UTF−8')]
[for (aTestCase : TestCase | aTestSuite.testCases /]
...
public void [aTestCase.name /](){
  [for( aStep : Step | aTestCase.steps )]
    ...
  [/for ]}
```

Listing 3: Excerpt of *TsT* transformation.

E. Test script generation transformation

TsT is the automation mechanism for the generation of executable test scripts from test cases defined using *TcDSL*. Formally, *TsT* can be described with the following function:

$$TsT < SPLm, P, W, Tc > \rightarrow n \times Ts$$

TsT takes as inputs the model of the SPL $SPLm$, the model P representing a number n of individual products of the SPL, the weaving model W and the test case Tc specified using *TcDSL*. Starting from these inputs, TsT produces one test script Ts for each of the n products modelled using P. *TsT* consists of the following main mapping rules:

- *P2Ts* creates a test script for each product in the SPL.
- *Tc2Method* creates a method in the test script for each specified test case.
- *Operation2Statement* creates a statement in the test script method for each operation in the test case.

[5] https://www.eclipse.org/Xtext/

- *Signal2Parameter*: translates each input/output of a test case operation into a signal. Besides, it marks the signal as the parameter of the test script statement.

TsT accounts for the exceptions defined using the *TcDSL* by skipping the products specified for the exceptions. We have implemented *TsT* using a template-based technology called Acceleo[6]. Listing 3 shows an excerpt of *TsT* implementing part of the *P2Ts* (line 1) and *Tc2Method* (line 4) rules. Acceleo allows the definition of a model transformation as a mix of static and dynamic elements. Static elements are expressed in the syntax of the target programming language and will not be changed at transformation time. As test scripts in BT are written in C#, *TsT* uses static elements from the C# programming language. Dynamic elements represent placeholders, which will be replaced with elements from the source/target models at transformation time. Listing 3 shows an example of static elements from C# in line 4.

III. The Aventra Family: a use case from the railway domain

In this section, we demonstrate the industrial applicability of the proposed approach using the BT Aventra SPL. As described in Section I, the Aventra SPL is a family of multiple electric unit trains for passengers transportation specifically designed for the British market. The Aventra SPL consists of five kinds of electric trains called LOT, EAA, SWR, WML and C2C. The trains belonging to the Aventra SPL share a considerable number of features. In Section I, we have introduced one such features called TBC, which is responsible for transmitting the driver inputs to the brake and propulsion system. When we have started the work on the case study, the smoke tests for the Aventra SPL have already identified 18 features shared among all the trains. For the sake of brevity, in the remainder of this section we focus only on two common features (TBC and Activate Cabin) and three trains (LOT, EAA and SWR). However, the interested reader can access the full implementation of the BT Aventra SPL at https://github.com/fabiodisilv/Model-Based_Test_Generator_SPL.

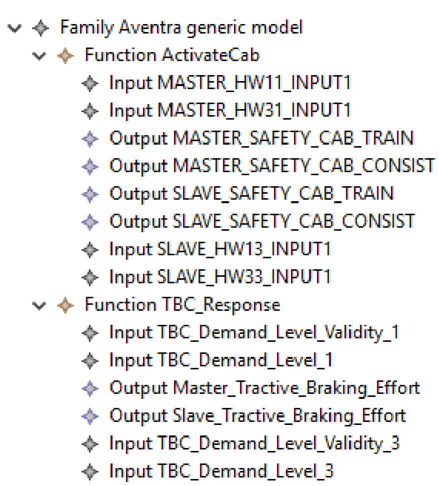


Fig. 5: Model of the Aventra SPL common features and their steps.

[6]https://www.eclipse.org/acceleo/overview.html

According to the proposed approach, the first step is to capture the SPL features and their generic steps, using *SPLmm*. Figure 5 shows a tree-based representation of the excerpt of the model describing the Activate Cabin and TBC features (named as *Function ActivateCab* and *Function TBC_Response* in the figure) along with their generic steps (named as. e.g., *Input MASTER_HW11_INPUT1*, *Input TBC_Demand_Level_Validity_1*, etc., in the figure).

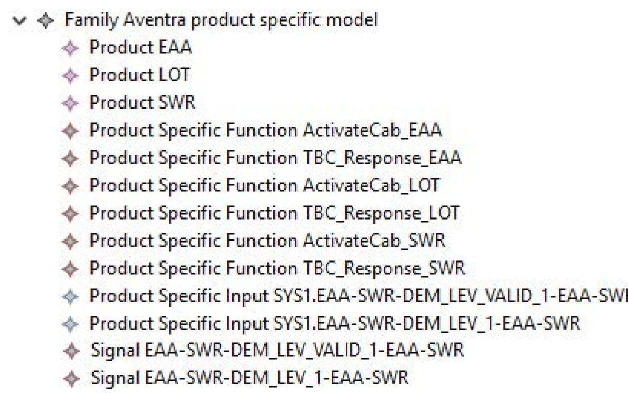
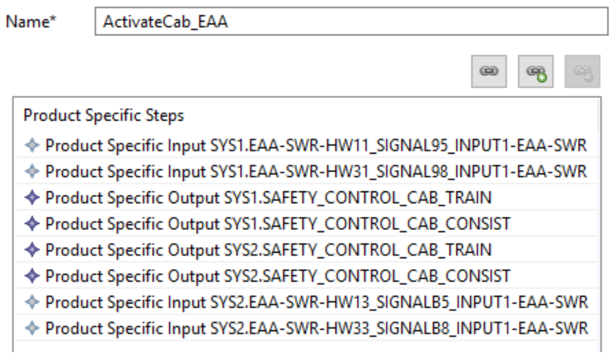


Fig. 6: Model of the trains in the Aventra SPL, their features, steps and signals.

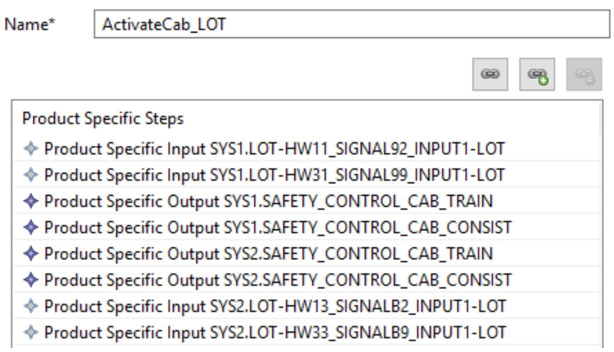
The second step of the proposed approach is modelling of the trains in the SPL in terms of their train specific implementation of the features, steps and signals, using *Pmm*. Figure 6 shows a tree-based representation of the excerpt of the model describing the EAA, LOT and SWR trains (named as *Product EAA*, *Product LOT* and *Product SWR* in the figure) along with their train specific implementation of the features (named as, e.g., *Product Specific Function ActivateCab_EAA*, *Product Specific Function TBC_Response_EAA*, etc., in the figure). Besides, Figure 6 shows some of the train specific steps for such features (named as e.g., *Product Specific Input SYS1.EAA-SWR-DEM_LEV_VALID_1-EAA-SWR*, *Product Specific Input SYS1.EAA-SWR-DEM_LEV_1-EAA-SWR*, etc., in the figure) along with some of their signals (named as e.g., *Signal EAA-SWR-DEM_LEV_VALID_1-EAA-SWR*, *Signal EAA-SWR-DEM_LEV_1-EAA-SWR*, etc., in the figure)[7]. Here we can see how *Pmm* allows capturing train differences. For instance, Figure 7 shows that *ActivateCab_EAA* and *ActivateCab_LOT* use four inputs and four outputs, while *ActivateCab_SWR* uses two inputs and four outputs. Besides, the inputs and outputs used by *ActivateCab_EAA* differ from those used by *ActivateCab_LOT*, as shown in Figure 7a and Figure 7b.

The next step is the linking of the train specific implementations of the SPL features, steps and signals, defined in the previous step, to the generic ones, defined in the first step. Figure 8 shows some excerpts of the model describing this linking. In particular, Figure 8a shows two *Function Link* elements typing *Product Specific ActivateCab_EAA*, *Product Specific*

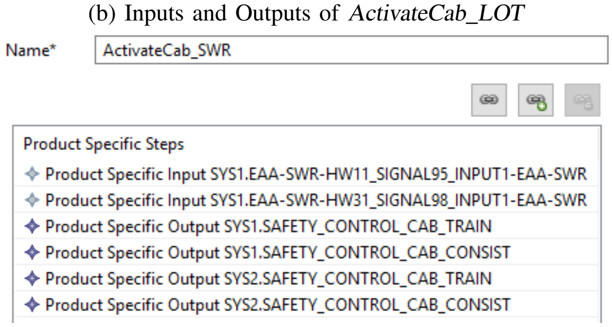
[7]For the sake of brevity, we omit all the steps, signals, linking information and test scripts for all the trains and features. The interested reader can access the complete implementation at https://github.com/fabiodisilv/Model-Based_Test_Generator_SPL



(a) Inputs and Outputs of *ActivateCab_EAA*

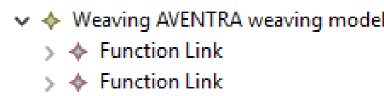


(b) Inputs and Outputs of *ActivateCab_LOT*

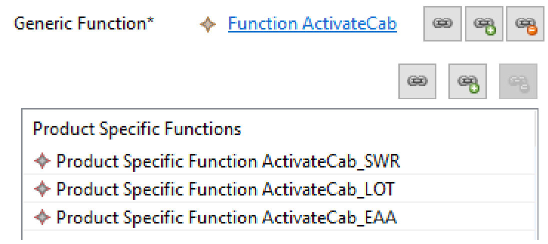


(c) Inputs and Outputs of *ActivateCab_SWR*

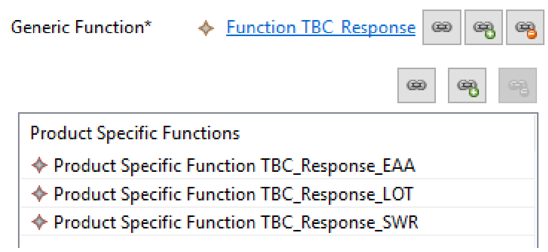
Fig. 7: Model of the train specific implementations of the *ActivateCab* feature.



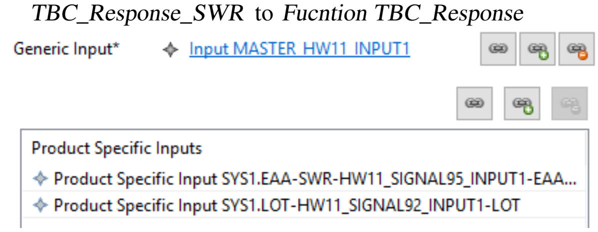
(a) Weaving model of the Activate Cabin and TBC features.



(b) Typing of *Product Specific ActivateCab_EAA*, *Product Specific ActivateCab_LOT* and *Product Specific ActivateCab_SWT* to *Function ActivateCab*



(c) Typing of textslProduct Specific TBC_Response_EAA, *Product Specific TBC_Response_LOT* and *Product Specific TBC_Response_SWR* to *Fucntion TBC_Response*



(d) Example of inputs linking

Fig. 8: Excerpts of the weaving model for the Aventra SPL

ActivateCab_LOT and *Product Specific ActivateCab_SWR* to *Function ActivateCab* (Figure 8b) and *Product Specific TBC_Response_EAA*, *Product Specific TBC_Response_LOT* and *Product Specific TBC_Response_SWR* to *Function TBC_Response* (Figure 8c), respectively[7]. Figure 8d shows an example of steps linking where the train specific inputs *Product Specific Input SYS.EAA-SWR-HW11_SIGNALS95_INPUT1-EAA-SWR* and *Product Specific Input SYS.LOT-HW11_SIGNALS92_INPUT1-LOT* of *ActivateCab_EAA*, *ActivateCab_LOT* and *ActivateCab_SWR* (Figure 7) are typed to the generic input *Input MASTER_HW11_INPUT1* of *Function ActivateCab* (Figure 6). The fourth step is the specification of abstract test cases for the generic features TBC_Response and ActiveCab. Listing 4 shows an abstract test case for ActivateCab namely, *Check_ActivateCab*, and two abstract test cases for TBC_Response, namely *Check_TBCResponse1* and *Check_TBCResponse3*. *Check_ActivateCab* consists of two force and four check operations. *Check_TBCResponse1* and *Check_TBCResponse3* consists of two force and two check operations each. The last step is the generation of executable test scripts using the TsT transformations described in Section II. The execution of TsT produces three C# files, one for each train in the Aventra SPL. Each of these files contains the set of instructions implementing the abstract test cases defined in the previous step. Listings 5 describes a portion of the generated C# file for the EAA train containing the *Check_ActivateCab*, *Check_TBCResponse1* and *Check_TBCResponse3* methods derived from the corresponding abstract test cases reported in Listing 4[7]. These methods show how the linking information captured by the model in Figure 8 is used for substituting the general steps used in the abstract test cases with train specific signals. For instance, the generic *MASTER_HW11_INPUT1* used in the definition of the *Check_ActivateCab* abstract test case in Listing 4, is substituted with the EAA specific signal *EAA-SWR-HW11_SIGNAL95_INPUT1-EAA-SWR* in Listing 5 using the information captured by the model in Figure 8d.

IV. Discussion

The automatic generation of test scripts stemming from common SPL features is one of the most prominent challenges hampering the full-fledged adoption of SPLE withing BT. In this paper, we propose a light-weight approach, which tackles such a challenge using model-based techniques such as metamodelling and automation by model transformation. Metamodelling allows increasing abstraction and separation of concerns. Besides, it enables automation by model transformation. The proposed approach mitigates the drawbacks of the opportunistic reuse of test artefact strategy, which are development effort, error proneness and consistency. One may argue that the number of development artefacts required for the proposed approach is higher than the number of development artefacts required for the opportunistic reuse of test artefact strategy.

```
TestCase Check_ActivateCab checks ActivateCab {
        Force Signal MASTER_HW11_INPUT1 to True
        Force Signal MASTER_HW31_INPUT1 to True
        Check Signal MASTER_SAFETY_CAB_TRAIN to 1 timeout 10000
        Check Signal MASTER_SAFETY_CAB_CONSIST to 1 timeout
            10000
        Check Signal SLAVE_SAFETY_CAB_TRAIN to 4 timeout 10000
        Check Signal SLAVE_SAFETY_CAB_CONSIST to 3 timeout 10000
            }

TestCase Check_TBCResponse1 checks TBC_Response {
        Force Signal TBC_Demand_Level_Validity_1 to true
        Force Signal TBC_Demand_Level_1 to 100
        Check Signal Master_Tractive_Braking_Effort to −10 timeout 1000
        Check Signal Slave_Tractive_Braking_Effort to −10 timeout 1000 }

TestCase Check_TBCResponse3 checks TBC_Response except Project
      SWR {
        Force Signal TBC_Demand_Level_Validity_3 to true
        Force Signal TBC_Demand_Level_3 to 100
        Check Signal Master_Tractive_Braking_Effort to −10 timeout 1000
        Check Signal Slave_Tractive_Braking_Effort to −10 timeout 1000 }
```

Listing 4: *Check_ActivateCab*, *Check_TBCResponse1* and *Check_TBCResponse3* abstract test cases

While this concern might be valid, the results from our evaluation suggest that this holds for small SPLs only. If F is the number of features and P the number of products in an SPL, then the number N of developed artefacts for the proposed $N_{proposedapproach} = 3 + F$. Regardless of the size of the SPL, the proposed approach requires to create *SPLm*, *W* and *P* models. Besides these models, the proposed approach requires the creation of a *Ts* model for each of the F features contained in the SPL. If F is the number of features and P the number of products in an SPL, then the number N of developed artefacts for the opportunistic reuse of test artefacts strategy is $N_{opportunisticresue} = P \times F$ as the opportunistic reuse approach requires the creation of an artefact for each feature of each product in the SPL. Figure 9 compares the graphs of these two functions where the blue solid line represents $N_{proposedapproach}$ while the red solid line represents $N_{opportunisticresue}$. It is evident that initially, the proposed approach requires a higher number of development artefacts, which makes it less suitable for relatively small SPLs. However, the initial higher effort becomes negligible when the SPLs increase in size. In particular, Figure 9b and Figure 9c show that the proposed approach requires a fewer development artefacts for SPLs containing 3 products and 2 features or 5 products and 1 feature, already.

```
//Generic function ActivateCab
//Function to activate the cab
public void Check_ActivateCab(){
    //Force MASTER_HW11_INPUT1 True
    SYS1["EAA−SWR−HW11_SIGNAL95_INPUT1−EAA−SWR"].
        Force(true);
    //Force MASTER_HW31_INPUT1 True
    SYS1["EAA−SWR−HW31_SIGNAL98_INPUT1−EAA−SWR"].
        Force(true);
    //Check MASTER_SAFETY_CAB_TRAIN 1
    SYS1["SAFETY_CONTROL_CAB_TRAIN"].WaitForSignal(1,
        10000);
    //Check MASTER_SAFETY_CAB_CONSIST 1
    SYS1["SAFETY_CONTROL_CAB_CONSIST"].WaitForSignal(1,
        10000);
    //Check SLAVE_SAFETY_CAB_TRAIN 4
    SYS2["SAFETY_CONTROL_CAB_TRAIN"].WaitForSignal(4,
        10000);
    //Check SLAVE_SAFETY_CAB_CONSIST 3
    SYS2["SAFETY_CONTROL_CAB_CONSIST"].WaitForSignal(3,
        10000); }
//Generic function TBC_Response
//Forward input reference from TBC to brake and propulsion during
     normal conditions
public void Check_TBCResponse1(){
    //Force TBC_Demand_Level_Validity_1 true
    SYS1["EAA−SWR−DEM_LEV_VALID_1−EAA−SWR"].Force(
        true);
    //Force TBC_Demand_Level_1 100
    SYS1["EAA−SWR−DEM_LEV_1−EAA−SWR"].Force(100);
    //Check Master_Tractive_Braking_Effort −10
    SYS1["TB_EFFORT"].WaitForSignal(−10, 1000);
    //Check Slave_Tractive_Braking_Effort −10
    SYS2["TB_EFFORT"].WaitForSignal(−10, 1000); }
//Generic function TBC_Response
//Forward input reference from TBC to brake and propulsion during
     normal conditions
public void Check_TBCResponse3(){
    //Force TBC_Demand_Level_Validity_3 true
    SYS2["EAA−DEM_LEV_VALID_3−EAA"].Force(true);
    //Force TBC_Demand_Level_3 100
    SYS2["EAA−DEM_LEV_3−EAA"].Force(100);
    //Check Master_Tractive_Braking_Effort −10
    SYS1["TB_EFFORT"].WaitForSignal(−10, 1000);
    //Check Slave_Tractive_Braking_Effort −10
    SYS2["TB_EFFORT"].WaitForSignal(−10, 1000); }
```

Listing 5: Generated C# file containing the test scripts for EAA

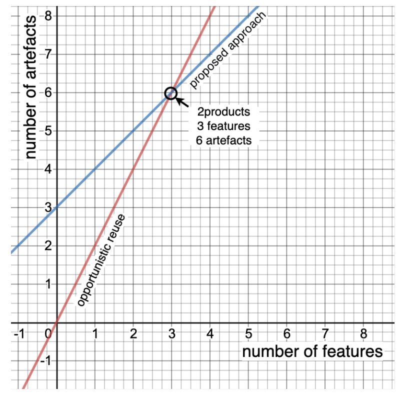
Previous studies report that industrial SPLs typically consist of tens of products and hundreds of features [7] [8]. While this is a valid concern, it is not straightforward to practically measure the required development time as this might depend on several factors, which are hard to control, such as, e.g., proficiency of developers with different technologies. However, we are planning to investigate this aspect in future work. With the proposed approach, error proneness is mitigated by construction as test scripts are always generated automatically starting from the abstract test cases. Similarly, consistency is achieved by construction as changes in the SPL features, products or abstract test cases would lead to a new generation of executable test scripts. Another concern might be that the mere comparison of the number of development artefacts might be inaccurate as it does not take into account the complexity and size of the artefacts as well as their development time.

In the previous section, we have demonstrated the applicability of the proposed approach on the Aventra SPL. In particular, we have shown how the proposed approach can automatically generate test scripts stemming from common SPL features and accounting for product differences. To show the efficiency of the proposed approach, we have compared the generated test scripts to those written manually using the *difflib* Python module [9]. Prior to this research, the test scripts where created manually as described in Section I. The results from the similarity checks are as follows:

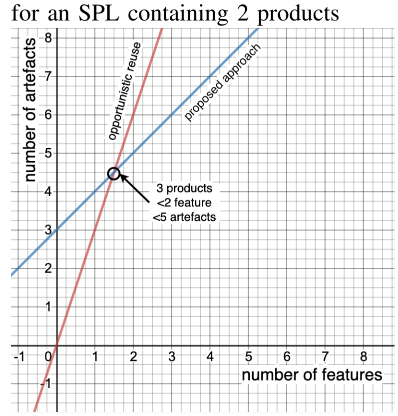
- EAA artefacts: 67% sequences' similarity
- LOT artefacts: 67% sequences' similarity
- SWR artefacts: 68% sequences' similarity

It is important to note that the differences between the scripts are due to the comments injected in the generated test scripts (e.g., lines 1 and 2 in Listings 5). We use comments in the generated test scripts as a way for increasing understandability and maintainability of the code. For integrity and security reasons, we cannot release all the handcrafted and generated test scripts. However, the interested reader can find examples of these at https://github.com/fabiodisilv/Model-Based_Test_Generator_SPL, which can be used for running the similarity check.

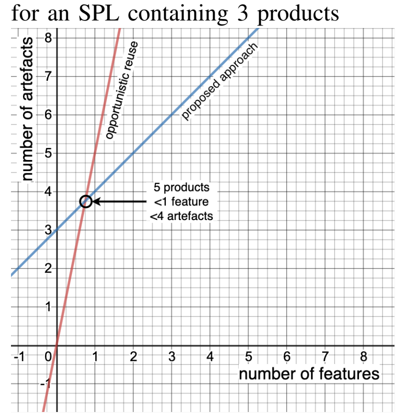
We have used expert surveys for assessing the industrial relevance of the proposed approach. The surveys contains five questions and draws on the model for assessing the industrial relevance of technology transfers introduced by Ivarsson et al. [6]. The model focuses on four aspects being subjects, context, scale, and research method. The pool of respondents included more than twenty practitioners from the BT Aventra integration team. For the sake of space, we omit the list of the questions, which can be found at https://github.com/fabiodisilv/Model-Based_Test_Generator_SPL. 100% of respondents have found the subject and context of this research to be industrially relevant. One respondent has remarked that "domain knowledge contributes significantly in the evaluation of a method, which generally lacks in students compared to practitioners". 90% of the respondents have evaluated the scale and research method aspects of this research as industrially relevant. A remark on the scale aspect states that "the functions/features have inherently similar nature in the aspects which are addressed in this research. So no negative bearing can be foreseen on the method when scaling". Overall, 60% of the respondents have found the proposed approach highly relevant, while 40% of the respondents found it relevant. A final remark from a respondent has agreed "that the current industrial testing approach is complex, error-prone and require



(a) Graph showing number of artefacts for an SPL containing 2 products



(b) Graph showing number of artefacts for an SPL containing 3 products



(c) Graph showing number of artefacts for an SPL containing 5 products

Fig. 9: Graphs comparing the number of artefacts required from the two approaches for SPLs containing 2,3 and 5 products

much development and maintenance effort. The work this research provides an interesting alternative that is promising in terms of test reusability and should, therefore, decrease cost."

V. Threats to Validity

We have defined, developed, and validated the proposed approach following an adaptation of the research methodology introduced in [10]. Such a methodology focuses on maximising the technology transfer between academia and industry using an iterative process emphasising the evaluation of the technology to be transferred. We have validated the applicability and efficacy of the proposed approach using the Aventra SPL use case from BT as discussed in Section III.

Besides, we have discussed the industrial relevance of the approach using experts interviews as described in Section IV. In the following, we discuss and classify potential threats to validity as well as our mitigation strategies according to the scheme proposed in [11].

*Internal.*To mitigate possible threats to internal validity in the expert interviews, we have selected practitioners with proven and extensive experiences in testing, MDE and railways domains. We have made an effort to ask questions in a neutral way so not to bias respondents with more positive answers in favour of the proposed approach.

External. To produce a general solution, the proposed approach and its constituents have been defined by performing an in-depth study of the state-of-the-art and -practice, as discussed in Section VI. The expert interviews involved researchers and practitioners from academia and industry with different nationalities and levels of experience. Hence, we believe that the results of interviews are agnostic of the country of origin and level of experience of the participants. It is important to note that validation on the Aventra SPL use case has been carried out entirely at the Bombardier premises in Västerås.

Construct. In this work, we focus on the Aventra SPL use case. Such a use case reflects real-world challenges as experienced from BT. All the authors of this work have prior and established experience in the fields of MDE, testing and railway transportation, which has helped in ensuring construct validity. The expert interviews have been opened by an informal discussion on the proposed approach and a questions and answers session for mitigating the risks of misunderstandings. Besides, it is worth to mention that all the authors have a longstanding collaboration with the practitioners involved in the study and this has resulted in insightful feedback characterised by mutual trust. *Conclusion.* To mitigate potential threats to conclusion validity, we have validated the proposed approach on the Aventra SPL use case provided from our business partner, BT. It is worth to mention that, the execution of the proposed approach on the Aventra SPL has been driven from the Aventra integration group, involving senior software engineering with more than 10 years of experience in the railway domain, so to avoid researchers bias. The validation has shown that the proposed approach was able to generate test scripts equivalent to those created manually using the opportunistic reuse of test artefacts strategy.

VI. Related Work

In this work, we discuss a model-based approach for test-scripts generation in the context of software development of complex industrial systems. In the past decades, several approaches for test artefacts generation have been proposed. Asaithambi and Jarzabek propose an approach known as the *generic adaptable test cases for SPLs* [12]. Such an approach aims at reducing the number of test cases needed for testing all the products within a given family. To this end, test cases are generated after analysing different assets of the SPL, including already existing test cases. The analysis aims at spotting and removing duplicated test cases. What is more, similar test cases are generalised and grouped based on different parameters. The main drawback of the work of Asaithambi and Jarzabek is the lack of any practical application of such an approach, which remains only theorised. Compared to our solution, the approach of Asaithambi and Jarzabek mainly differs in the test artefacts generation. In our approach, such a task is completely automatic as it is entrusted to a model transformation. Reuys et al. propose a model-based approach to test case derivation in the system test of software product lines [13]. The approach is known as *Scenario based Test case Derivation* (ScenTED) and requires test artefacts for the whole family to be designed for extensions so as to cover the variability of each product. What is more, within ScenTED test cases are automatically generated from system models such as the Unified Modeling Language (UML) activity diagrams or sequence diagrams. Similar to ScenTED, our approach generates test artefacts starting from a structured representation of information families and products within families. However, our approach does not rely on behavioural system models, but on lightweight models. This makes our approach more flexible and suited for those contexts where behavioural information is not available. Lochau et al. illustrate an application of the so-called *delta-oriented testing* technique, which is an incremental testing technique relying on state machines describing the products behaviours [14]. The approach first generates test artefacts based on the state machines representing a product. Later, it evolves the generated test artefacts based on modifications calculated on the state machines. Dukaczewski et al. present another delta-oriented testing technique, which replaces state machines with textual requirements [14]. Similar to the above delta-oriented testing techniques, the goal of our approach is to enhance (automatic) test artefacts generation. However, our proposed approach differs from the above-mentioned ones as executable test scripts are generated for each product starting from a single test case for the whole family. Several approaches to test artefacts generation are based on the use of the UML Testing Profile (UTP), which allows the specification of tests for both static (structural) and dynamic (behavioural) aspects of a software system [15], [16]. Bagnato et al. describes an industrial application of UTP within the field of future internet application [17]. The test definition through UTP is carried out using a graphical representation, in contrast with the textual representation provided by our approach. Moreover, the use of UTP requires a MDE background since the test specification is directly linked to UML model(s). Even though we make use of MDE techniques, once set up, our approach can be used without MDE knowledge as the DSL ease the test-case definition due to its similarity to the natural language. Iber et al. present another approach based on UTP [15]. In this work, the authors build a textual domain-specific language from which UTP models are automatically generated using a model transformation. In turn, the generated UML model(s) could be further transformed into test-scripts using external transformations. Our approach is similar to the one from Iber et al. as they both rely on DSLs and model transformation. However, the main difference is that in our approach executable test scripts are automatically generated from a test case written using a DSL. UTP could be used for the automatic generation of executable test scripts, too. In this context, the work in [18] provides a two-step transformation process, which generated Java executable test scripts from UML 2.0 Testing Profile (U2TP). In particular, the U2TP is first translated into TTCN-3 using a model transformation. Then the TTCN-3 is translated into a Java skeleton which has to be completed manually. The authors argue that achieving a fully automated generation is difficult, if not impossible, due to the different levels of abstraction of system models and test specifications. Another approach using UTP is AGEDIS by Cavarra et al. [19]. The approach make use of UML system models, e.g., class diagram, object diagram, etc., and a profile defined by the authors. The key novelty of this strategy is the possibility to create an additional model containing the test directives. These directives are used to tune the test generator to allow test engineers to perform an appropriate test selection for budget, time and test campaign constraints. Our strategy does not require such a tune as generic test-cases are directly specified by test engineers. Test artefacts generation is also presented in the approach by Tahat et al. [20]. To avoid the complexity of UML systems models, such an approach use requirements specified using the Specification Description Language (SDL) as the starting point of the generation process. In particular, the requirements written using SDL are first gathered and then transformed into an Extended Finite State Machines (EFSMs). In turn, EFSMs are transformed into test cases. According to the authors, such an approach can account for requirements changes without the need for regenerating all the test cases. Similar to our approach, the work of Tahat et al. uses a structured representation of products as the base for the generation process. However, in our approach, the test engineer is required to write a single test case from which the test scripts are generated automatically based on the information represented in the models.

VII. Conclusion and Future Work

In this paper, we have tackled the challenge of automatically generating test scripts from shared SPL features by introducing a model-based approach using metamodelling and automation by model transformation. We have leveraged the BT Aventra SPL for demonstrating that the proposed approach is applicable in industrial settings and it can generate executable test scripts that are equivalent to those created manually. We have discussed how the proposed approach mitigates the development effort, error proneness and consistency drawbacks of the opportunistic reuse of test artefacts strategy for SPLs containing three products and two features. We have reported the practitioners evaluation on the industrial relevance of the proposed approach.

One line of future work encompasses the extension and refinement of the involved metamodels and DSL to capture interfaces and signals from different application domains. Besides improving their expressiveness, this would positively impact the usability of the proposed approach and different industries could use it without trade-offs or heavy modifications, regardless of the application domain. Besides, we are working of further extensions so as to enable the automatic generation of different development artefacts rather than test scripts. Another line of future work encompasses refinements to the model-to-text transformation to support the generation of test scripts in several target programming languages. Finally, we are planning to evaluate the development complexity and the time consumption of the proposed approach with respect to the opportunistic reuse of test artefacts strategy.

Acknowledgment

References

[1] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[2] A. Metzger and K. Pohl, "Software product line engineering and variability management: achievements and challenges," in *Future of Software Engineering Proceedings*, 2014.

[3] W. B. Frakes and K. Kang, "Software reuse research: status and future," *IEEE Transactions on Software Engineering*, 2005.

[4] P. A. da Mota Silveira Neto, I. do Carmo Machado, J. D. McGregor, E. S. de Almeida, and S. R. de Lemos Meira, "A systematic mapping study of software product lines testing," *Information and Software Technology*, vol. 53, no. 5, pp. 407 – 423, 2011.

[5] M. Broy, M. Gleirscher, P. Kluge, W. Krenzer, S. Merenda, and D. Wild, "Automotive Architecture Framework: Towards a Holistic and Standardised System Architecture Description," Tech. Rep., 2009.

[6] M. Ivarsson and T. Gorschek, "A method for evaluating rigor and industrial relevance of technology evaluations," *Empirical Softw. Engg.*, vol. 16, no. 3, p. 365–395, Jun. 2011.

[7] J. Bosch, "Product-line architectures in industry: a case study," in *Proceedings of the 21st international conference on Software engineering*, 1999.

[8] D. Nestor, L. O'Malley, A. Quigley, E. Sikora, and S. Thiel, "Visualisation of variability in software product line engineering," 2007.

[9] Python library, "difflib," https://docs.python.org/3/library/difflib.html#difflib.SequenceMatcher, accessed: 2020-10-05.

[10] T. Gorschek, P. Garre, S. Larsson, and C. Wohlin, "A model for technology transfer in practice," *IEEE Software*, 2006.

[11] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, 2009.

[12] S. P. R. Asaithambi and S. Jarzabek, "Generic adaptable test cases for software product line testing: Software product line," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, 2012.

[13] A. Reuys, E. Kamsties, K. Pohl, and S. Reis, "Model-based system testing of software product families," in *Advanced Information Systems Engineering*, 2005.

[14] 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)*, May 2013, pp. 49–52.

[15] J. Iber, N. Kajtazović, G. Macher, A. Höller, T. Rauter, and C. Kreiner, "A textual domain-specific language based on the uml testing profile," in *Model-Driven Engineering and Software Development*, P. Desfray, J. Filipe, S. Hammoudi, and L. F. Pires, Eds. Springer International Publishing, 2015.

[16] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch, "The uml 2.0 testing profile and its relation to ttcn-3," in *Testing of Communicating Systems*, D. Hogrefe and A. Wiles, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 79–94.

[17] A. Bagnato, A. Sadovykh, E. Brosse, and T. E. J. Vos, "The omg uml testing profile in use–an industrial case study for the future internet testing," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013.

[18] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din, "From u2tp models to executable tests with ttcn-3 - an approach to model driven testing -," in *Testing of Communicating Systems*, F. Khendek and R. Dssouli, Eds. Springer Berlin Heidelberg, 2005.

[19] A. Cavarra, C. Crichton, J. Davies, A. Hartman, T. Jeron, and L. Mounier, "Using uml for automatic test generation," 01 2002.

[20] L. H. Tahat, B. Vaysburg, B. Korel, and A. J. Bader, "Requirement-based automated black-box test generation," in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, 2001.