JOT JOURNAL OF OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Automatic Generation of Configuration Files: an Experience Report from the Railway Domain

Enxhi Ferko*, Alessio Bucaioni*, Jan Carlson*, and Zulqarnain Haider†
*Mälardalen University, Sweden
†Bombardier Railway Transportation, Sweden

ABSTRACT In recent years, software product line development has been adopted by a growing number of companies. Within software product line development, one way of creating specific products is by using configuration files to control a given set of parameters of the product at run time. Often, configuration files are created manually and this may lead to a sub-optimal process with respect to development effort and error proneness. In this experience report, we describe our work in enabling the automatic generation of configuration files in the railway domain. We discuss a four-step approach whose generation mechanism uses concepts of generative programming. The approach is the outcome of a bottom-up effort leveraging the experiences and the results from our technology transfer activities with our industrial partner, Bombardier Transportation. We evaluate the applicability and the correctness of the proposed approach using the Aventra train family from Bombardier Transportation. Besides, we evaluate the ability of the proposed approach in mitigating the development effort and error proneness typical of traditional manual approaches. We performed expert interviews to assess the industrial relevance of the proposed approach and collect qualitative feedback on the perceived benefits and drawbacks. Eventually, for each of the four steps composing the proposed approach, we identify factors that might affect the adoption of the approach and use these factors for discussing the lessons we have learned.

KEYWORDS Software Product Line, Configuration File, Automatic generation.

1. Introduction

In recent years, many companies have witnessed an increasing demand for customised software-intensive systems able to address different market needs, regional standards, certifications as well as software and hardware requirements. Bombardier Transportation (BT) is one such company.[1] To meet this increasing demand for customisation, BT has been shifting its product development towards Software Product Line (SPL) (Klaus Pohl 2005). SPL is a software development paradigm where a single system is developed to meet the needs of a product family, hence several products with commonality and variability (Metzger & Pohl 2014). In a nutshell, SPL focuses on identifying and modelling similarities and differences of a product family and using this information for deriving and configuring specific products of the family. SPL differentiates between two processes, namely domain engineering and application engineering (Klaus Pohl 2005). Domain engineering focuses on defining the commonality and variability of the product family, as well as developing the common and distinguishing software assets. Application engineering uses these assets to derive specific products of the family by selecting variability options for matching product requirements and functionalities. SPL provides the opportunity of decreasing the overall development cost while providing for higher quality and shorter development time compared to the development of multiple independent systems.

In BT, the adoption of SPL has brought some challenges and difficulties, too (Metzger & Pohl 2014). One of the main open challenges that BT is facing is how to derive specific products from a family. Once a product has been designed, there are

JOT reference format:
Enxhi Ferko, Alessio Bucaioni, Jan Carlson, and Zulqarnain Haider. *Automatic Generation of Configuration Files: an Experience Report from the Railway Domain*. Journal of Object Technology. Vol. 20, No. 3, 2021.
Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2021.20.3.a4

[1] https://www.bombardier.com/en/transportation.html

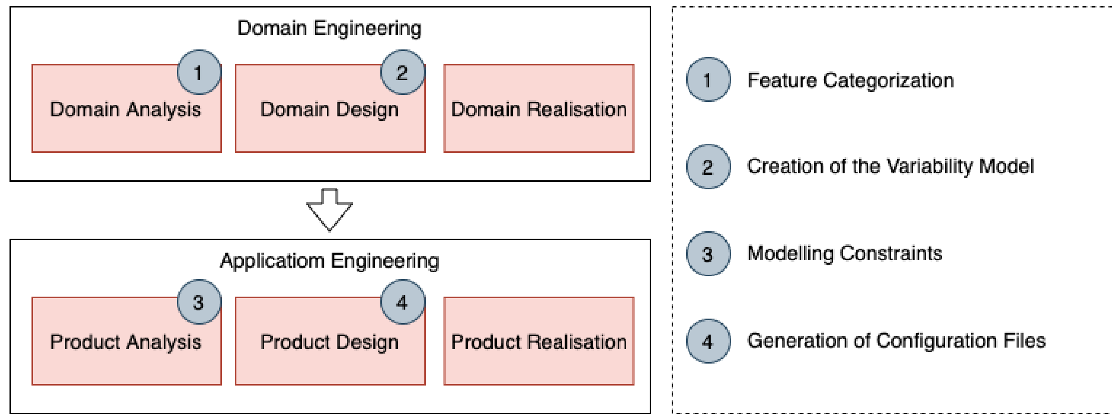

Figure 1 Mapping proposed steps to the SPL process

several ways in which the design can be turned into a concrete product. One common option, which is used by BT and is the focus of this work, is to realise individual products using configuration files. A configuration file is part of the concrete realisation of the system and determines, at run time, some of its aspects. A typical example of a configuration file would be an XML data file with parameters that enable or disable different portions of the common source code according to the features selected for that particular product. In BT, configuration files are currently created manually starting from a set of heterogeneous documents. These documents often provide for a general and informal description of the products and their features, only. Besides, they might not be stored and managed systematically. Starting from these documents, BT engineers manually create configuration files for the different products in the family. Experiences from BT show that manually creating configuration files starting from a set of heterogeneous and informal documents is time-consuming and error-prone. The most common errors reported by engineers are categorised as (i) omitting information, (ii) wrong semantics, and (iii) inconsistencies. Besides, this manual process does not allow exploiting product commonalities, meaning that common information needs to be specified for each product.

In this experience report, we describe our experience in automating the generation of SPL configuration files in the railway domain within BT. We enable the automatic generation of configuration files defining a four-steps approach whose generation mechanism leverages concepts of generative programming (Czarnecki & Eisenecker 2000). We demonstrate the applicability and correctness of the proposed approach using an industrial SPL from BT being the Aventra train SPL. Besides, we evaluate the ability of the proposed approach in mitigating the development effort and error proneness typical of traditional manual approaches. The application on the Aventra SPL suggest that the proposed approach has helped BT in lowering the development effort and mitigating the error proneness typical of the traditional, manual approaches already for SPL containing more than 3 products. Eventually, we use experts interviews to the Aventra Development team for assessing the industrial relevance of the proposed approach and for collecting qualitative feedback on the perceived benefits and drawbacks. For each step of the proposed approach, we identify a number of factors and alternatives to take into account for adapting the approach to other development process and domain. Eventually, we use these factors and alternatives for discussing the lessons we have learned.

The remainder of this paper is structured as follows. In Section 2, we present the proposed approach, its steps, and the generation mechanism. In Section 3, we show the application of the proposed approach on the Aventra SPL. In Section 4 and Section 5, we evaluate the proposed approach and discuss lessons learnt and generalisability, respectively. In Section 6, we present some related researches documented in literature. In Section 7, we conclude the paper with final remarks and possible future work.

2. Proposed Approach

In this section, we describe the proposed approach for automatically generating configuration files within the BT development process. Figure 1 provides a graphical representation of the approach in relation to a simplified version of the SPLE framework presented by Pohl et al. (Klaus Pohl 2005). the two main phases of a typical SPLE process are *Domain Engineering* and *Application Engineering*, which are represented as white rectangles in Figure 1. Each main phase consists of a number of sub-phases represented as red rectangles.

The proposed approach consists of four steps being *Feature Categorisation*, *Creation of the Variability Model*, *Variant Constraints Categorisation* and *Generation of Configuration Files*. Figure 1 represents these steps as circled numbers and places them in relation to the sub-phase they occur in. In the remainder of this section, we describe each step of the proposed approach.

2.1. Feature Categorisation

Feature Categorisation is the first step of the proposed approach and should be performed during the Domain Analysis sub-phase. Domain Analysis is responsible for defining, documenting, and validating all the shared and individual requirements of the SPL. These requirements are further mapped to features, which are used for defining the variability model of the SPL during the Domain Design sub-phase. An example of this could be the requirement that a train has two driver cabins, which would translate in the feature *direction* indicating the orientation of

the train. Industrial SPLs often consist of thousands of features, which are typically neither organised nor prioritised. Managing such a huge number of unorganised features is a complex task, which negatively affects the product derivation process (Loesch & Ploedereder 2007). The categorisation of the features identified in the Domain Analysis sub-phase is of crucial importance for BT.

Feature Categorisation focuses on grouping the elicited features into *Primary Features* and *Secondary Features*. Primary features are all those features whose combinations can uniquely identify products in the SPL. Within BT, *voltage*, *number of cars*, and *side of consist* are primary features as the combination of their values can uniquely identify a certain train. Secondary features are all those features, which do not explicitly define a product. *Max speed* is an example of a secondary feature in the case of BT. The output of the Feature Categorisation step is a list of features labelled as primary or secondary. Based on the complexity of the SPL and the result of the categorisation, secondary features can be included or excluded from the feature model of the SPL. In the first case, the secondary features will be automatically added to the configuration files. This strategy leads to a complete, but potentially unmanageable feature model. In the second case, the secondary features need to be added manually to the configuration files. Such a strategy leads to an incomplete, but light-weight feature model. In Section 5, we discuss the advantages and drawbacks of each strategy and discuss the lessons we have learned in our work with BT.

2.2. Creation of the Variability Model

Creation of the Variability Model is the second step of the proposed approach and should be performed during both the Domain Design and Domain Realisation sub-phases. Domain Design and Domain Realisation are responsible for defining the variability model and implement the related domain artefacts. The variability model represents the commonalities and differences of the SPL products. The variability model consists of variation points, variants, and their relationships and constraints. A variation point is a representation of a location, where a feature can have different values.[2] For instance, *voltage* is a variation point with values being *Dual Voltage (DV)*, *Direct Current (DC)* and *Alternating Current (AC)*. A variant is a realisation of a variation point. Creation of the Variability Model focuses on the creation of a model capturing this information and the output of this step is a formalisation of the variability model. Currently, there is no standard way of representing variability models (Arboleda & Royer 2012) and different companies may use different notations and tools. One of the most used notations for representing the variability model is feature model (Danilo Beuche 2006). Feature modelling is a well-defined formalism, which is currently supported by several open-source and commercial tools, such as FeatureIDE (Meinicke et al. 2017). Another possibility for representing variability models is using metamodelling for building a domain-specific language (DSL) describing the SPL variation points and variants. Most of the commercial tools supporting feature modelling are built around ad-hoc DSLs. In Section 5, we discuss some aspects to consider when choosing one approach over the other that have emerged as pivotal during our work with BT.

[2] In the remainder of this paper, we use the terms feature and variation points analogously.

2.3. Variant Constraints Categorisation

Variant Constraints Categorisation is the third step of the proposed approach and should be performed during the Product Analysis and Product Design sub-phases. These sub-phases are focused on configuring all the products within the SPL. This involves the selection of a valid and complete set of variants together with possible constraints on them. We have observed that in BT, variant constraints and their management have a direct impact on the complexity of the generation of configuration files and the complexity of the product realisation. To ease the management of variant constraints, we propose to categorise them into *Technical Constraints* and *Customer-specific Constraints*. Technical constraints are all those constraints, which prevent the engineer to select erroneous or invalid variants. An example of this could be a constraint preventing the selection of *pantograph* and *diesel engine* variants for trains not having the *electrical engine* variant. Customer-specific constraints are all those constraints, which bind the choice of variants based on customer preferences. An example of this can be the orientation of a car (same or different from the leading car), which can differ from a customer to another. Technical and customer-specific constraints can refer indiscriminately to primary and secondary features. While most of the time technical constraints would be specified on primary features, there might be cases when technical constraints would be specified on a secondary feature, too, so to avoid erroneous configuration of a product. Similarly, customer-specific constraints can be specified on primary features. The output of this step is the categorisation of variant constraints into technical and customer-specific. Based on the total number of constraints, customer-specific constraints can be specified or omitted. In the case they are omitted, the customer-specific constraints need to be added directly during the configuration file generation. This solution yields towards a lighter and more malleable SPL with respect to addition, modification and deletion of products, variation points and variants. However, such a solution might reduce the degree of automation. Specifying both technical and customer-specific constraints yields to a less malleable solution with respect to the above-mentioned cases, but it enables full-fledged automation. In Section 5, we discuss the advantages and drawbacks of both solutions when applied in the Aventra SPL.

2.4. Generation of Configuration Files

Generation of Configuration File is the last step of the proposed approach and should be performed during the Product Realisation sub-phase. In this phase, products are configured based on the identified variants and variant constraints. For each derived product, technical values of the variants are expressed in the configuration files, which are part of the concrete realisation of the software and determines at run-time the set-up of the actual software. Hence this step focuses on the automatic generation of such configuration files.

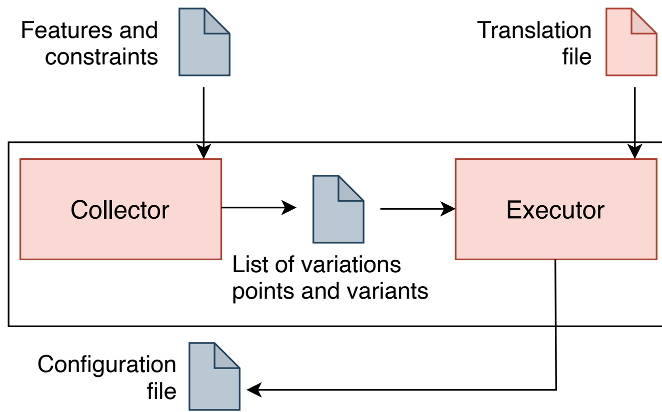

Figure 2 Mechanism for the automatic generation of configuration files

There might be several alternatives in how to achieve such an automatic generation. One option is by using transformation languages and model transformations (Sendall & Kozaczynski 2003). Model transformations are automated processes that take one or more source models as input and produce one or more target models as output, according to a set of transformation rules. An example of this could be the use of the template-based language Acceleo (Nesrine & Bennouar 2018) for the automatic generation of configuration files starting from models representing the SPL. These models should be conforming to metamodels. This solution benefits from the tooling, which usually accompanies such transformation languages. One drawback is that model-based solutions usually entail the knowledge of concepts as modelling, metamodelling, and model transformation. We have noticed that this knowledge is rarely possessed and mastered by the engineers at BT, which are also unfamiliar with model-based tooling (Liebel et al. 2014). Another drawback of this solution is the certification process. Being a safety-critical domain, the railway domain needs to comply with several safety standards, e.g., IEC 61508 (Bell 2006). In particular, the European standard EN 50128 specifies the process and technical requirements for the development of software for programmable electronic systems for use in railway control and protection applications (CENELEC 2020). EN 50128 defines three classes of tools of ascending criticality, which are T1, T2, and T3 (CENELEC 2020). According to this categorisation, modelling and transformation languages and environments fall in the T3 category as they could introduce errors in the final executable. Among other requirements, tools in the T3 category need (i) to demonstrate a suitable history of successful tool use in a similar environment, (ii) have diverse redundant code, which allows the detection and control of failures, (iii) comply with the safety integrity levels, and (iv) demonstrate a record of validation activities (CENELEC 2020). Such requirements are assessed by external certification bodies in a process, which may take up to several years and add a development cost overhead between 25 and 100% (Pop et al. 2016).

Another possibility to achieve the automatic generation of configuration files is by using traditional programming languages for realising concepts of generative programming (Czarnecki & Eisenecker 2000). In a nutshell, generative programming refers to programs that are written for creating software components in an automated fashion. Generative programming is considered the ancestor of metamodelling and model transformation. This solution may allow mitigating the certification cost in the case it can be realised using programming languages and tools already used within the company, hence already certified. In Section 5, we discuss the advantages and drawbacks of both alternatives. In this research effort, we have developed one such mechanism for the automatic generation of configuration files using C# and leveraging the concepts of generative programming.

2.4.1. Mechanism for the Automatic Generation of Configuration Files

Figure 2 shows a simplified representation of the functional architecture of the mechanism for the automatic generation of configuration files. It consists of two main modules, namely *Collector* and *Executor* (represented as red rectangles in Figure 2) and one *Translation File* (represented as a red note in Figure 2). The Executor module is implemented in C# while the Collector module represents any feature modelling environment. The mechanism takes the following as input: the list of features and constraints, and a translation file. In the first step, the Collector module is used for gathering the list of features and their values. Besides, it considers the specified technical and customer-specific constraints for establishing relationships among the selected features. These constraints might be expressed in formal or informal notation. For each product, the Collector resolves these constraints and produces an XML data file where each feature is marked as automatically selected, automatically deselected, or manually selected to specify all features selected for one final product. In particular, a feature can be automatically selected or deselected based on the resolution of a technical constraint. A feature can be manually selected as a result of the resolution of a customer-specific constraint. Listing 1 shows an excerpt of one of such files in which the *maxSpeed* feature is selected automatically and the *caro* feature is selected manually. If using a feature modelling environment, these steps correspond to the modelling of the SPL. In the second step, the Executor translates the information about selected features into a configuration file, according to the relations specified in the Translation File. In a nutshell, these relations link selected features to corresponding objects in the configuration files. An example of these relations are:

- *Parameter-to-Features* each parameter in the configuration file is associated to the list of features that might contribute to its specification. For instance, in Listing 2 *parameter1* is associated to *variant1* and *variant2*, to *variant1* or to *variant1* and *variant3*
- *Features-to-ParameterValue* the value of the parameter is selected considering the list of features actually contributing to that parameter.

Listing 2 shows an excerpt of the general structure of the translation file developed for this research effort using the JavaScript Object Notation (JSON). In Section 3, we show the concrete Translation File developed for the Aventra SPL.

```
1 <configuration>
2   ...
3   <feature automatic="selected" name="maxSpeed"/>
4   <feature manual="selected" name="caro"/>
5   ...
6 </configuration>
```

Listing 1 Example of list of active feature for a product

```
1  {"parameterName1": {
2    "Type": "Type of parameter1",
3    "Comment": "Comment for parameter1",
4    "Values": [
5      {"Features": [ "variant1", "variant2" ],
6      "Value": "technicalValue1"},
7      {"Features": [ "variant1" ],
8      "Value": "technicalValue2"},
9      {"Features": [ "variant1", "variant3" ],
10     "Value": "technicalValue3"}]
11   },
12 "parameterName2": {...},
13 ....,
14  "DefaultValue": −1
15 }
```

Listing 2 Example of the translation file

The Executor uses the relations in the Translation File for automatically creating configuration files, and it is realised as a console application. It takes as inputs the path to the folder where the files from FeatureIDE are stored, the path to the folder where the generated configuration files will be stored, and the path to the translation file. First, the Executor collects the information expressed in the JSON file and stores it in a data structure called *Translation Collection*. The Translation Collection has the information expressed in the JSON file such as the parameters to be generated, their possible mapping options to features, and respective technical values. Then, starting from the list of variants and variation points, the Executor extracts the features marked as selected (either manually or automatically) and stores this information in a data structure called *Feature Collection*. Eventually, for each parameter in the Translation Collection, the Executor checks which combination of features are selected in the Feature Collection and assigns the corresponding technical value. This is done for all the products. For instance, using the relations in Listing 2, the Executor will derive *parameter1* from the features *variant1* and *variant2* with value *technicalValue1* if the features *variant1* and *variant2* are selected. Alternatively, it derives it from the feature *variant1* with value *technicalValue2* if only feature *variant1* is selected. Or, it can derive it from the features *variant1* and *variant3* with value *technicalValue3* if the features *variant1* and *variant3* are selected. A default value is added in case none of the options is triggered. Engineers can filter all configuration files with parameters that have a default value to find any possible inaccuracy when configuring the products. The result of this step is a configuration file for each product in the SPL. In Section 3, we provide an example of configuration files.

3. Applying the Proposed Approach to the Aventra SPL

In this section, we demonstrate the industrial applicability of the proposed approach using the Aventra SPL from Bombardier Transportation. The Aventra SPL is a family of passenger trains designed by Bombardier for the British market. The trains in the Aventra SPL might be of three types: the London Overground train (LOT), the East Anglia train (EAA), and the South Western Rail train (SWR). Each train might have a specific number (e.g., four, five, ten) and type (e.g., passengers carrying cabin, drivers cabin) of cars, power supply (e.g., Dual Voltage(DV), Alternative Current (AC), Direct Current (DC)) and other characteristics. In addition to differences at the train level, there might be differences at car level, too. An example of this could be the orientation (same or opposite direction as the leading car) or the position of the car. All the train functions such as, on-board communication, passenger information, entertainment, are controlled by the Train Control and Management System (TCMS), which is a complex system consisting of different software and hardware units connected via different communication links, such as Internet Protocol (IP) networks. Most of the functionalities of TCMS units are common for different types of trains. Some of the generic functions in TCMS units are:

– Doors control function — controlling the opening and closing of train doors.
– Passenger information function — displaying real-time information to the passenger
– Pantograph up/down function — indicating if the pantograph is in contact with the electric lines.

In addition, each of these units might have few modules developed differently for each train to accommodate the train differences. In total, the Aventra SPL composes of 12 different trains, 17 variation points, and 63 variants.

3.1. Aventra SPL Feature Categorisation

According to the proposed methodology, the first step to be performed is the elicitation and categorisation of the SPL features. We have elicited the Aventra SPL features from the Software Architecture and Design Specification (SAS) of Aventra TCMS Applications using the third degree of data collection technique introduced by Lethbridge et al. (Lethbridge et al. 2005). This activity has identified 63 variants grouped in 17 variation points. In sum, 12 final products are configured starting from the 4 primary features and related constraints. Table 1 shows the elicited features together with their values.

We have decided to include the secondary features with corresponding options in the variability model for a number of reasons, including the following. In the Aventra SPL, we had a small number of secondary features. Hence, including them in the variability model, did not lead to an unmanageable variability model. Omitting secondary features from the variability model would have hampered the automatic generation of configuration files and required a considerable amount of additional work for manually specifying these features in each and every configuration file. In addition, having secondary features in the

Feature name	Categorisation	Description
type	Primary	Type of train. It can be t_LOT, t_EAA, t_SWR
carNr	Primary	Number of car per consist. It can be 4, 5 or 10.
voltage	Primary	High voltage configuration in the consist. It can be AC, DC, DV.
maxSpeed	Secondary	It can be unknown, 140kph/87mph, 180kph/111mph or 250kph/155mph.
doorsPerCar	Secondary	Number of doors per car. It can be 1, 2, 3, 4 or 5.
sideOfConsist	Primary	Side of the consist. It can right or left.
cartype	Secondary	Type of car in the consist. It can be A, B, C, D, E.
carOrient	Secondary	Orientation of the car with respect the leading car. It can be unknown, same or opposite.

Table 1 Features for the Aventra SPL

variability model gave us the possibility of specifying variants constraints on such features.

3.2. Aventra SPL Variability Model

In the second step of the proposed approach, the elicited features and their values are used to create a model of the SPL variability model. We have created the model of the SPL using the FeatureIDE tool, which is an Eclipse-based framework for feature-oriented software development. FeatureIDE provides for a user-friendly visualisation of the features and their relationships in the model. It supports multi-views configuration of each product. It provides support for the evolution of the feature model simply enabling the addition or deletion of features in the feature model. In addition, FeatureIDE provides variant constraints management and constraints can be added, deleted, or modified in accordance with user needs. Figure 3 to Figure 5 show the FeatureIDE model of the Aventra SPL variability model. For the sake of readability, we have split the model into three figures: Figure 3 shows the primary features together with their possible values while Figure 4 and Figure 5 show the secondary features and their possible values. It is worth noting that the categorisation of features in primary and secondary in the feature model is only logical hence has no direct impact on the variability model or configuration files. FeatureIDE allows to mark features as *mandatory*, *optional*, *concrete* and *abstract*. Mandatory features are those features that hold for all the products in the SPL. For instance, all the trains will have the *type* feature, but only ten-cars-per-consist trains will have the *sideOfConsist* feature. Optional features are features that might be omitted for given products. Concrete features are features that will be included in the configuration file. We have used concrete features for representing variants. Abstract features represent features that will not be included in the configuration file. We have used abstract features for representing variation points. In addition, features can be assigned in the so-called *alternative group*. Features in an alternate group are mutually exclusive. In the variability model in Figure 3 to Figure 5, we have used alternative group features for representing variants of a variation point. When features are marked within an alternative group, FeatureIDE generates implicit constraints to ensure the correct semantic.

3.3. Avetra SPL Variant Constraints

In FeatureIDE, variant constraints (also known as cross-tree constraints (Seidl et al. 2016)) are expressed using Boolean-like formulas via the create constraint functionality. Variant constraints can be updated and new variant constraints can be added at any point in time. In the Aventra SPL, we have decided to add only technical constraints to the variability model. The main reason for such a decision was to ensure the separation of concerns between domain knowledge, expressed by technical constraints, and customer preferences. The Aventra SPL technical constraints were:

1. $t_{LOT} \rightarrow carNr_4 \vee carNr_5$
2. $t_{LOT} \wedge carNr_5 \rightarrow v_{DV}$
3. $t_{LOT} \wedge carNr_4 \rightarrow \neg v_{DC}$
4. $t_{EAA} \rightarrow v_{AC}$
5. $t_{SWR} \rightarrow carNr_5 \vee carNr_{10}$
6. $t_{SWR} \rightarrow \neg v_{AC}$
7. $carNr_{10} \rightarrow sideOfConsist$
8. $carNr_4 \vee carNr_5 \rightarrow \neg sideOfConsist$
9. $carNr_5 \vee carNr_{10} \rightarrow car5type$
10. $carNr_5 \vee carNr_{10} \rightarrow car5o$
11. $carNr_4 \rightarrow \neg car5type$
12. $carNr_4 \rightarrow \neg car5o$

Technical constraints impose restrictions on variants of different variation points. The first constraint specifies that a train of type LOT can only have four-cars or five-cars consists. The second constraint specifies that a train of type LOT having five-cars consists will have a dual voltage system. The third imposes direct current systems for trains of type LOT. The fourth constraint states that all trains of type EAA must have an alternating current system. The fifth and sixth constraints specify that trains of type SWR must have five-cars or ten-cars consists and that can not have an alternating current system. The seventh constraint specifies that the feature *sideOfConsist* can be specified only for ten-cars consists. In fact, a ten-car consist is composed of a permanent linking of two five-car sets hence the two sides of consists might have different configurations of cars and functions. The eighth constraint specifies that the feature *sideOfConsist* can not be specified for four-cars or five-cars consists. Ninth and tenth constraints specify that features *car5type* and *car5o* can be specified only for trains having five-cars or ten-cars consists, while constraints number eleven and twelve states that *car5type* and *car5o* features can not be specified for trains having four-cars consists. FeatureIDE provides a graphical editor for the

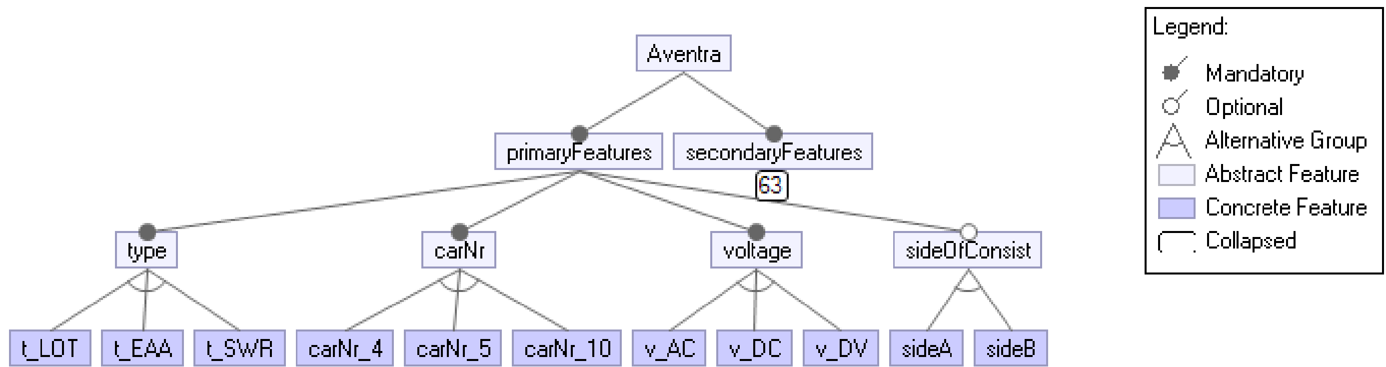

Figure 3 Primary Features of the Aventra SPL variability model.

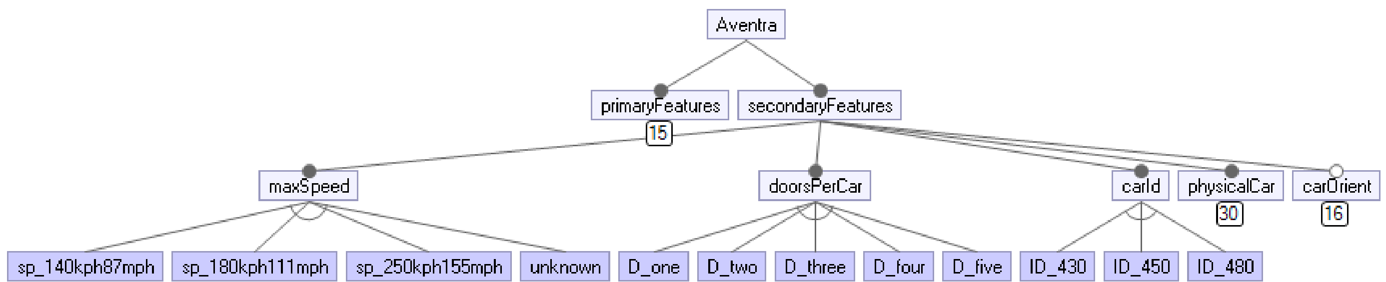

Figure 4 Secondary Features of the Aventra SPL variability model (I).

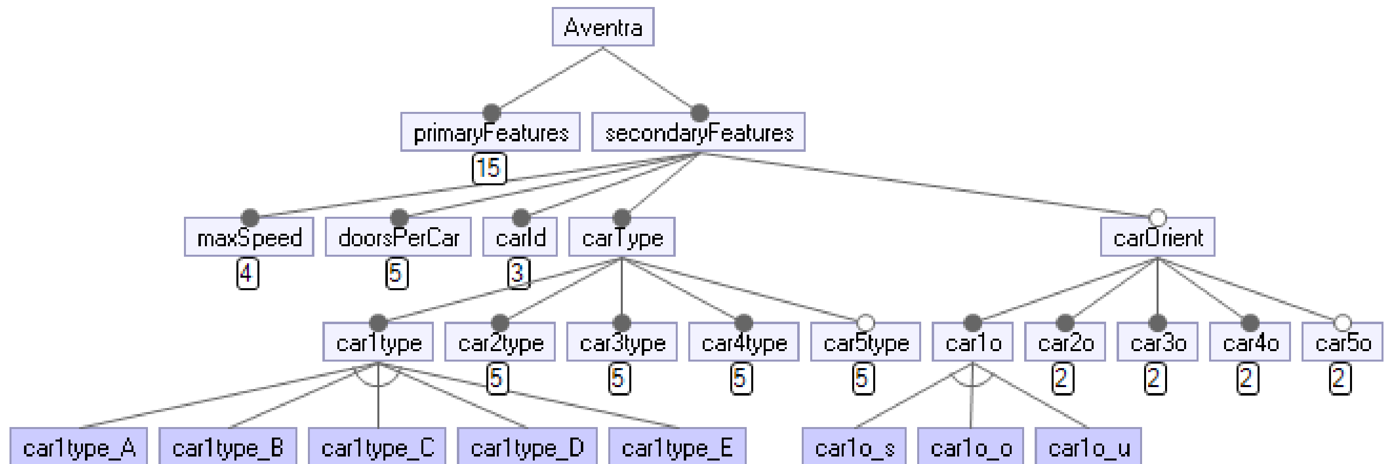

Figure 5 Secondary Features of the Aventra SPL variability model (II).

visualisation of products and their features. For each product, features are organised in a tree structure where leaves represent possible values of features as shown in Figure 6 and Figure 7. Each feature and feature value are provided with a checkbox, which can be used for selecting or deselecting that feature or value (selected features are marked with a green plus in Figure 6 and Figure 7). FeatureIDE prevents users from selecting invalid combinations of features or values meaning combinations that violate the specified constraints. All the features and the values specified as mandatory in the constraints are automatically selected. Similarly, features and values excluded by the specified constraints are automatically disabled and cannot be selected. For instance, Figure 6 shows that the value *v_DC* is deselected because of a technical constraint and that value *v_DV* is manually selected, instead. Similarly, Figure 7 shows that the feature *car5type* is automatically deselected because of the specified constraint.

3.4. Generating Aventra SPL Configuration Files

The generation of the Aventra SPL configuration files is entrusted to the automation mechanism described in Section 2. FeatureIDE provides a mechanism for creating XML data files containing the list of features along with their values. These files are created starting from the three structures described above

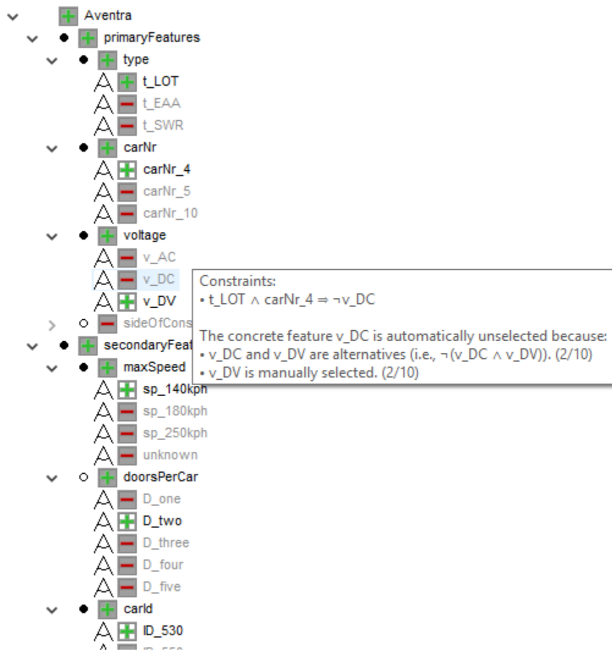

Figure 6 Tree constraint in configuration editor for *LOT4carDV*

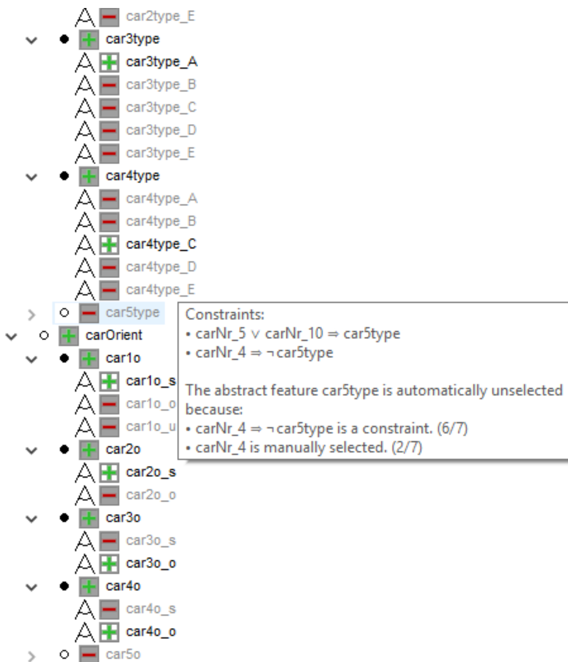

Figure 7 Cross-tree constraint in configuration editor for *LOT4carDV*

and for each product of the SPL. In the context of the mechanism presented in Section 2 these files are equivalent to the *list of variation points and variants* created from the *Collector* module. We have used such a feature for creating the files containing the list of features that are selected for one product and their values of the Aventra SPL. Listing 3 describes a fragment of the file containing the list of variation points and variants for the *LOT4carDV* product.

```
<configuration>                                                              1
  ...                                                                        2
  <feature automatic="selected" name="type"/>                                3
  <feature manual="selected" name="t_LOT"/>                                  4
  <feature automatic="unselected" name="t_EAA"/>                             5
  <feature automatic="unselected" name="t_SWR"/>                             6
  <feature automatic="selected" name="carNr"/>                               7
  <feature manual="selected" name="carNr_4"/>                                8
  <feature automatic="unselected" name="carNr_5"/>                           9
  <feature automatic="unselected" name="carNr_10"/>                          10
  <feature automatic="selected" name="voltage"/>                             11
  <feature automatic="unselected" name="v_AC"/>                              12
  <feature automatic="unselected" name="v_DC"/>                              13
  <feature manual="selected" name="v_DV"/>                                   14
  <feature automatic="selected" name="maxSpeed"/>                            15
  <feature manual="selected" name="sp_140kph87mph"/>                         16
  <feature automatic="unselected" name="sp_180kph111mph"/>                   17
  <feature automatic="unselected" name="sp_250kph155mph"/>                   18
  <feature automatic="selected" name="DoorsPerCar"/>                         19
  <feature automatic="unselected" name="D_one"/>                             20
  <feature manual="selected" name="D_two"/>                                  21
  <feature automatic="unselected" name="D_three"/>                           22
  <feature automatic="unselected" name="D_four"/>                            23
  <feature automatic="unselected" name="D_five"/>                            24
  ...                                                                        25
</configuration>                                                             26
```

Listing 3 List of variation points and variants for the *LOT4carDV* product.

This product is a LOT train characterised by four cars and a dual voltage system. Its max speed is 140 kilometres per hour and it has two doors per car. Accordingly, the variation points and variants corresponding to these values are marked as selected. For the sake of space, we only show a portion of the file containing the list of variation points and variants for the *LOT4carDV* product. The interested reader can find the other files here.[3] It should be noted that the higher the number of constraints in the SPL model, the higher is the number of features that can be automatically selected or deselected. However, the complexity of specifying new constraints increases with the total number of constraints as each of these is specified through Boolean expression (which needs to take into account already specified ones). All these files are provided as input to the generation mechanism together with the translation file described in Section 2. In Listing 4, we report an excerpt of the translation file developed for the Aventra SPL. In particular, Listing 4 shows the relations responsible for the translation of *S_B_Project* and *ST_B_MaxSpdHmi* parameters. For the sake of space, we only report portions of these files. Listing 5 shows the generated configuration file for the *LOT4carDV* product. In particular, we can see that the features have been translated into parameters according to the relations specified in the translation file in Listing 4. For instance, the combination of features *type* and *carNr* has been translated into the *S_B_Project* parameter and their values *t_LOT* and *carNr_4* into the parameter value *1* (lines 1 to 9 of Listing 4).

[3] https://zenodo.org/record/4737049#.YJFBwy0Rrxh

```
1   {"S_B_Project": {
2      "Type": "Integer",
3      "Comment": "Customer project: 1: Lotrain 4 cars, 2: Lotrain 5 cars, 3: East
        Anglia 5 cars, 4: East Anglia 10 cars (side A or B of 5 car consist), 5:
        intentionally empty, 6: South West 5 cars, 7: South West 10 cars (side A
        or B of 5 car consist)...",
4      "Values": [
5         { "Features": [ "t_LOT", "carNr_4" ],
6          "Value": 1},
7         {"Features": [ "t_LOT", "carNr_5" ],
8          "Value": 2},
9         ...]},
10  "ST_B_MaxSpdHmi": {
11     "Type": "Integer",
12     "Comment": "Max speed dial value;Value interpretation: 0–unknown, 1–
        140kph/87mph, 2–180kph/111mph, 3–250kph/155mph,,",
13     "Values": [
14        {"Features": [ "sp_140kph87mph" ],
15        "Value": 1},
16        {"Features": [ "sp_180kph111mph" ],
17        "Value": 2},
18        {"Features": [ "sp_250kph155mph" ],
19        "Value": 3},
20        ]},...,
21  "DefaultValue": –1
22  }
```

Listing 4 Example of the translation file for the Aventra SPL

```
1  <ParameterSet>
2  ...
3    <Parameters>
4      <Parameter Name="S_B_Project" Type="Integer" Value="1" />
5      <Parameter Name="ST_B_MaxSpdHmi" Type="Integer" Value="1" />
6      <Parameter Name="HV_config" Type="Integer" Value="1" />
7      <Parameter Name="Nr_of_cars_in_consist" Type="Integer" Value="4" />
8      <Parameter Name="Nr_of_doors_per_car" Type="Integer" Value="2"/>
9      <Parameter Name="Type_car1" Type="String" Value="D" />
10     <Parameter Name="Orientation_car1" Type="Integer" Value="2" />
11     ...
12    </Parameters>
13 </ParameterSet>
```

Listing 5 Generated configuration file for the *LOT4carDV* product.

4. Evaluation

In this experience report, we describe our experience in automating the generation of SPL configuration files in the railway domain with BT.

In this section, we evaluate the correctness of the proposed approach together with its ability in lowering the error-proneness and the developing effort typical of the manual approach. We use expert interviews for assessing the industrial relevance of the proposed approach and collecting qualitative feedback on its benefits and drawbacks. Eventually, we discuss the main threats to validity for this research and related mitigation strategies.

When defining the automation mechanism, we have created both the Executor and the Translation file so that they would generate configuration files identical to the handcrafted ones that were previously used within BT. It is important to note that having identical configuration files was of crucial importance for BT. In fact, differences in the configuration files would affect the software running in the trains, which would need to be updated. Assuming that the correct set of variation points and variants are selected, there is only one factor that could undermine the generation of identical configuration files, which is the presence of errors in the Translation file. In order to show that the Translation file is free from errors, we have compared the 12 generated configuration files for the Aventra SPL with the handcrafted ones using a visual difference tool focusing on files, directories, and version controlled projects comparison. The results of the comparison have shown that all the generated configuration files were 100% identical to the manually created ones. The interested reader can access the comparison here.[4] Hence, the proposed mechanism is correct. One may argue that, in general, there is still the possibility to introduce errors when writing the mappings in the Translation file. While this is a valid concern, such a risk can not be completely removed although we have identified several mitigation strategies that could be added to our automation mechanism (we discuss this in the following section). Moreover, it should be noted that this risk is more likely to happen in a traditional manual approach as the engineer would need to rely on the mappings that we formalise in the translation file each time she writes the configuration file for a product in the SPL (while in the proposed process the Translation file is written only once per SPL).

One of the main goals of the proposed approach is to mitigate the error proneness typical of manual approaches. When working on the Aventra SPL, engineers in BT have identified three main categories of errors that can be made when manually creating the configuration files, being (i) omitting information: parameters or technical values are omitted from configuration files, (ii) wrong semantics: the parameters and their value are inserted in the document, but they are interpreted wrongly, (iii) inconsistencies: parameters or technical values are inconsistent among the configuration files. Errors in the first and third categories are implicitly avoided using the proposed approach. In fact, both parameters and values are automatically generated starting from the same SPL model and Translation file. This means that it is not possible to have two or more configuration files having, for instance, a different set of parameters. Besides, all the parameters and their value are generated taking into account the set of identified variation points and variants. Hence, no information is omitted or lost in the process of generating configuration files. The above mentioned comparison among generated and manually created configuration files shows that the proposed mechanism is free from inconsistencies and information loss. Within the proposed approach, errors in the second category may arise if an erroneous Translation file is created. However, compared to the traditional manual approach, the proposed mechanism drastically reduces the possibility of such errors. In fact, the Translation file is only written once per

[4] https://zenodo.org/record/4737049#.YJFAXC0Rrxg

SPL as opposite to the manual approach where such errors can happen each time an engineer writes a configuration file for a product in the SPL.

Another important goal of the proposed approach is to lower the development effort of the configuration files. In order to evaluate this aspect, we have compared the number of artefacts that need to be manually created with the proposed approach and with the traditional one. If n is the number of products in a SPL, then the number N of manually created or modified artefacts for the proposed approach is $N_{proposedapproach} = 3$. Within the proposed approach, an engineer needs to manually create the Collector, Executor, and Translation file. If n is the number of products in a SPL, then the number N of manually created or modified artefacts for the traditional manual approach is $N_{manualapproach} = n$. With manual approaches, the engineers would need to manually create a configuration file for each of the n products in the SPL. By estimating the number of manually created artefacts with both approaches, it is evident that the proposed mechanism is able to reduce the development effort for the configuration files. This suggests that the proposed approach discloses the opportunity of lowering development effort already for SPLs containing three products. In this comparison, we assume that the effort of eliciting and managing features is the same with both approaches and that the SPL is newly created. If that is not the case, then the number N of manually created or modified artefacts for the proposed approach is $N_{proposedapproach} = 4$. In fact, an engineer would need to manually create the SPL model besides the already mentioned artefacts. It is worth mentioning that the complexity of the SPL model is directly related to the number n of products in the SPL. However, we can reasonably state that for reasonable n-values, the development effort related to manual models in traditional approaches is greater than the effort related to the creation of the SPL model, Collector, Executor, and Translation file in the proposed approach.

In order to discuss the development effort needed in the case of evolving SPLs, we have identified three different scenarios. The first scenario involves adding new functionality for any existing product in a SPL as the result of a customer need. In the context of the Aventra SPL, this could be the installation of a new air conditioning system for the EAA trains. This scenario requires updating all the configuration files of all the products in the SPL so to reflect the new parameter along with its technical value. If n is the number of products in the SPL, manual approaches would require n modifications while the proposed approach would require 1 modification. In fact, manual approaches would require the modification of all the configuration files. The proposed approach would require modification of the translation file, only, as the configuration files will be updated automatically by a subsequent execution of the automation mechanism. The second scenario involves changing a feature for a subset of existing products in the SPL. In the context of the Aventra SPL, this could be requiring that all the LOT trains would switch from having two doors per car to four doors per car. This scenario requires the modification of the configuration files of the involved products, only. If k is the number of the products in the subset being modified, manual approaches would require the modification of k configuration files, while the proposed approach would require 1 modification for modifying the constraint related to the feature. The third scenario involves the addition of a new product to the SPL. This scenario has two sub-scenario. In the first sub-scenario, the product is added with the same set of features as the existing ones. In the Aventra context, this sub-scenario could be to have a LOT train with 10 cars. This scenario would require the creation of the configuration file for the new product. Manual approaches would require the manual creation of one configuration file, while the proposed approach would not require any manual modification as the new product would be captured by the variability model prior to the generation phase. The second sub-scenario is when the new product is added with additional features. In the Aventra context, an example of this scenario could be a customer requiring a LOT train with 10 cars and a new maximum speed of 300kph/186mph. This scenario would require the modification of all the configuration files. If n is the number of products in the SPL, manual approaches would require n modifications, while the proposed approach would require the modification of the translation file, only.

We have performed expert interviews for assessing the industrial relevance of the proposed approach and collecting feedback on perceived benefits and drawbacks. The questionnaire encompasses thirteen questions being a mix of open-ended and closed-ended questions. The questions assessing the industrial relevance draw on the model for assessing the industrial relevance of technology transfers introduced by Ivarsson et al. (Ivarsson & Gorschek 2011). This model focuses on four aspects being subjects, context, scale, and research method. The survey was shared with the engineers from the BT Aventra development team and we got six respondents. For the sake of space, we omit the list of the questions, which can be found here.[5] To sum up, 66,7% of the respondents have found the problem of manually creating configuration files to be relevant and 16,7% extremely relevant (on a scale from not relevant at all to extremely relevant). One respondent has said that: "*this is a problem that will only be more extensive in the future, when trying to provide generic software*". When asked about the suitability of the proposed approach, 83,4% of the respondents found it to be suitable (on a scale from not suitable at all to extremely suitable). Some of the perceived benefits are: "easy to maintain and extend", "this approach can improve quality of Aventra software deliveries: reducing time for generating the configuration files, mitigating risk for a failure, etc.". Among the main drawbacks, the respondents have identified that: "in the (unlikely) event of major changes in the parameters, the feature model and the automation pipeline could require huge and time consuming refactoring" and " challenging to have it reused in other domains/projects". When evaluating the industrial relevance of the proposed approach, 83,7% of the respondents have found the subjects, the context, the method, and the scale aspects to be industrially relevant.

[5] https://zenodo.org/record/4737049#.YJFAXC0Rrxg

4.1. Threats to Validity

Hereafter, we discuss and classify potential threats to validity and describe our mitigation strategies according to the scheme proposed by Runeson et al. (Runeson & Höst 2009). The work presented in this experience report is an example of applied research. According to Wohlin et al., the threats to validity for applied researches can be prioritised as follow (from the most to the least important): internal, external, construct, and conclusion validity (Wohlin et al. 2012).

4.1.1. Internal Validity Threats to internal validity affect the ability to draw correct relationships between treatment and outcome. In order to mitigate possible threats to internal validity, we have decided to work on a real-life example coming from our industrial partner: the Aventra SPL. This has ensured the validity of the artefacts and of the use case set up. When developing the proposed approach, we have made an effort not introducing new tools and limiting the impact of tool performance. For instance, we have decided to measure the number of created/modified artefacts rather than the developing or modifying time as this could have been affected by the tools and underlying hardware. Although we have used only a single group, we believe this did not affect the response of the subjects involved as all of them had prior and established experience in BT and SPL development.

4.1.2. External Validity Threats to external validity affect the ability to generalise the results beyond the experiment settings. In order to mitigate such threats, we have made an effort for selecting the most representative subjects. Within BT, we have only had subjects that had prior experience with SPL development. When it comes to the selection of the use case, we have referred to the real-world use case from BT being the Aventra SPL. The Aventra SPL was among the latest SPLs being developed at BT. We have designed, developed, and applied the proposed mechanism within the BT premises within the same context where the production takes place. Eventually, we have proposed a discussion for each step composing the proposed approach for adapting it in different scenarios as a way of generalising the finding of this research.

4.1.3. Construct Validity Threats to construct validity relate to the extent to which the setting of an experiment reflects the theory. In order to mitigate such threats, we have tried to evaluate different aspects of the proposed approach as opposite in focusing on one single factor. Besides, we have complemented the evaluation with a qualitative survey. The design, development, and application of the proposed approach have been carried out within BT and using a real-world example. When preparing the questionnaire, we have made an effort in avoiding any possible hypothesis guess and treatment testing.

4.1.4. Conclusion Validity Threats to conclusion validity affect the ability to derive a correct conclusion from the relations between treatment and outcome. To minimise threats to conclusion validity, we have asked independent practitioners to analyse the outcome of our experience report. Besides, we have made an effort in evaluating the proposed mechanism using objective measures as in the case of the number of artefacts. We have found the number of artefacts to be more reliable since it does not involve human skills. When constructing the qualitative questionnaire, we have tried to use simple wording and have remained available for possible explanations. Eventually, we have submitted the questionnaire only to the BT engineers having a proven experience in SPL development.

5. Discussion and Lessons Learnt

In Section 2, we have introduced the proposed approach and its steps. For each of these steps, we have identified a set of alternatives or decisions that could potentially impact the generation process. In this section, we discuss the benefits and drawbacks of these alternatives with the aim of highlighting important aspects contributing to the generalisability of the proposed approach. In addition, we discuss our choices and describe lessons learnt.

The first step of the proposed approach is the categorisation of features in primary and secondary features. Here the decision is whether or not to include the secondary features in the variability model to build in the second step. Including the secondary features in the variability model leads to a more detailed model of the SPL and enables the automatic generation of configuration files. However, including the secondary features can contribute to a very complex and potentially unmanageable variability model especially when secondary features are the majority in the architecture. Omitting secondary features from the variability model would reduce the complexity and the size of the architecture leading to a more manageable model. However, it would dramatically increase the number of manual modifications as the secondary features and their values would need to be added later as variation points and variations of each product. In addition, omitting secondary features would also hamper the specification of constraints on these features hence the automatic generation of valid configuration files. Here the trade-off is between completeness and manageability. The relevance of such a trade-off has been stressed by an expert in the survey: "*in the (unlikely) event of a major change in the parameters, the feature model could require a huge and time consuming refactoring*" In the case of the Aventra SPL, we have decided to include the secondary features and mark them as mandatory features in the feature model developed using FeatureIDE. In this way, they were automatically selected for every product and we could easily configure them and add put constraints. Our decision was mainly affected by two factors being the level of automation we wanted to achieve and the limited number of secondary features for the Aventra SPL.

The second step of the proposed approach is creating the variability model. The main decision to take is on the notation to be used for representing the variability model hence variation points and variants. We have identified two alternatives being to use feature modelling or to use metamodelling. Feature modelling is a well-established formalism, which has been widely used for SPLs. The second alternative is using metamodelling for creating an ad-hoc language. It is worth mentioning that feature modelling is often achieved using metamodelling techniques. In our experience, the crucial factors affecting this step

are the set of competencies and skills in the company and the available resources in terms of time and costs. In the survey, one respondent has highlighted that: "*It's not easy to find practitioners with modelling skills.*" Here the trade-off is between development effort, flexibility, and prior knowledge. In the case of the Aventra SPL, we have decided to use feature modelling and the FeatureIDE tool as it was already known within Bombardier Transportation and did not require further training. Other reasons behind our choice were ease of use, graphical interface, and support for SPL evolution.

The third step of the proposed approach is the categorisation of variant constraints in technical and customer-specific. The main decision in this step is whether or not to specify customer-specific constraints on the variability model. Specifying customer-specific and technical constraints leads to a precise definition of products, whose configuration files can be derived using automation mechanisms. However, this would require more development effort for engineers to write all the constraints and deep knowledge of the products. In addition, this could have an impact even when a product is added or modified as a higher number of constraints might potentially need to be updated. Defining only technical constraints has resulted to be a more flexible solution and supports the evolution of SPL with less effort than in the first option. Moreover, with this approach, tasks can be easily divided through different teams. Engineers with domain knowledge are involved in creating technical constraints and other engineers would deal with the customer requirements in later stages and at the product level. In this step, the trade-off is between development effort and completeness. In the case of Aventra SPL, we have decided to specify only technical constraints to ease possible future extensions and to be able to separate tasks into teams with different levels of domain knowledge and customer-specific requirements knowledge.

The last step of the proposed approach is the generation of configuration files. The main discussion here is how to realise the automation mechanism. The first alternative is to use ad-hoc transformation languages. The second alternative is to use traditional programming languages leveraging concepts of generative programming. The first alternative can rely on a wide set of transformation languages and supporting tools, may enable a round-trip process (Eramo & Bucaioni 2013), and would seem the natural choice in the case metamodelling is used in the second step. However, this solution comes with several practical drawbacks, including the following. Metamodeling and transformation languages require specific skills to be mastered. Usually, these skills are rarely found in the industry. In addition, tools supporting transformation languages are usually not in the technological stack of companies hence they would require prior training to be effectively used. Finally, transformation languages and supporting tools would need to undergo the certification processes and this is a major drawback for a company operating in safety-critical domains such as the railway domain as discusses in Section 2. Metamodelling and domain-specific languages (DSLs) could have been used for creating the translation file, with a concise syntax and semantics. A simple tooling infrastructure could have build on this language so to achieve e.g., code completion, type checking, etc. As mentioned above, we have decided not to use metamodelling techniques for the lack of competencies in the company, which would have made it difficult to maintain and evolve the automation mechanism. Besides, the use of a DSL would have not improved the mappings between parameter and features value that can not be changed in favour of more meaningful ones as the feature technical values come from the actual software (which in many cases is inherited from previous related projects). Eventually, we have provided the translation file with a default value that is added in case none of the correct mappings is triggered. With such a mechanism, engineers can filter all configuration files with parameters that have a default value to find any possible inaccuracy or mistakes when configuring the products. It should be noted that despite we do not use external (modelling) tools, the proposed mechanism still needs to undergo verification and validation activities in accordance with the safety standards.

6. Related Work

In this section, we survey the principal strands of research that relate to the work we present in this paper.

In the last decades, researchers and practitioners have proposed several notations and approaches for representing the variability model of SPLs along with its variability. In this respect, one of the most used techniques is feature modelling (Danilo Beuche 2006). Other commonly used techniques are modelling and metamodelling. In their work, Fang et al. have presented an approach for automating software derivation where feature modelling is integrated with domain-specific modelling languages for variability expression (Fang et al. 2016). Bergen et al. in their study present a comparison of Kconfig and Component Description Language (CDL), two variability modelling languages that adopt the concepts of feature modelling. Both languages are used in practice to describe the variability of the Linux kernel and eCos operating system for embedded devices, respectively (Berger et al. 2010). Compare to our approach, the work by Fang et al. and Bergen et al. have a narrower scope as they mostly focus on variability expression and not on product derivation.

The work by Czarnecki et al. proposes to split the configuration process into different stages. Each stage yields a feature model where a subset of the system described on the initial feature model can be specified. An example of this could be to address the requirements from individual manufacturers in the first stage while configuring product components in a later stage. The process by Czarnecki et al. has some similarities with Variant Constraint Categorisation in our proposed process. Both approaches propose to split the configuration process into stages. While correspondences between staged-configurations and primary and secondary features can be drawn, in the general case they are not equivalent. To the best of our understanding, our categorisation of features is more restrictive, while in the process by Czarnecki et al. an engineer can decide to split the requirements in any way she wants and assign each category to a given stage (Czarnecki et al. 2004).

In the domain of train signaling, Svendsen et al. (Svendsen et al. 2010) have proposed an approach for representing SPLs and generating their products, which uses the interplay of two domain-specific modelling languages being the Train Control Language and the Common Variability Language (CVL). Vasilevskiy et al. have investigated the use of domain-specific modellling languages for representing variability and deriving new products (Vasilevskiy et al. 2015). In particular, they have introduced the Base Variability Resolution language as an extension of CVL and have provided the BVR tool bundle as a set of Eclipse plugins. The BVR tool bundle supports the resolution, realisation, and derivation of products. The same authors have introduced an approach to build robust realisations, too (Vasilevskiy et al. 2016). In the scope of their work, realisations define mappings between abstract features in a feature tree and their implementation artefacts. The goal of their approach was to ensure that small changes in the SPL model would result in small changes in the realisations. Although targeting product realisation, the approaches of Svendsen et al. and Vasilevskiy et al. use different techniques as compared to our approach. The approach in this paper achieves product realisation using configuration files. Besides being a possible technique, this was a strong requirement inherited from our business partner. In the proposed approach we use the concepts of generative programming, while in their approaches Svendsen et al. and Vasilevskiy et al. leveraged the interplay of modelling languages and code generation.

Several works have been investigating the use of Aspect-Oriented Programming (AOP) for product derivation. Lee et al. (Lee et al. 2009) leverage the interplay of AOP and feature modelling for introducing automation in the product realisation. Voelter and Groherin propose to use model-driven and aspect-oriented development to support the product derivation process (Voelter & Groher 2007). Feature models and domain-specific models are used to represent the problem and the solution domain, while model transformations and AOP techniques are used for the automatic manipulation of these models leading to product realisation. Another research focusing on model transformations has been presented by Tawhid and Petrin (Tawhid & Petriu 2011). Their approach tackles the problem of automatically generating a model for a specific product while focusing on performance aspects. To this end, it relies on two model-to-model transformations. The first one uses the Atlas Transformation Language (ATL) and takes as input a feature model of the SPL and produces a UML model with performance annotations realised as MARTE annotations. Starting from this model, the second transformation generates a Layered Queueing Network (LQN) model that will be used to analyse the performance of the SPL.

Several frameworks and commercial tools support product configuration in the context of SPL. One such tool is ArchStudio an Eclipse-based development platform based on ArchFeature (Gharibi & Zheng 2016). ArchFeature provides for an architectural model integrating feature specification, product line architecture, and their relationships. This is achieved by extending an existing XML-based architecture description language namely xADL. xADL is mostly used for modelling system architectures consisting of components and connections. It includes a graphical modelling environment that visualise relationships of feature in a SPL and supports the automatic derivation of the products. An assessment for other frameworks and commercial tools supporting product configuration in a SPL such as GEARS, pure:variants , Captor, CIDE, MSVCM, XVCL, GenArch is done in (Torres et al. 2010).

7. Conclusion and Future Work

In this experience report, we have presented our work for automating the generation of configuration files in the context of software product lines. The proposed approach revolves around a mechanism leveraging concepts of generative programming and is composed of four main steps spanning through the whole software product line development process. We have validated the applicability of the proposed approach using an industrial use case from the railway domain: the Bombardier Transportation Aventra train family. Besides, we have evaluated the ability of the proposed approach of lowering the development effort and mitigating the error proneness typical of manual approaches. The application of the proposed approach on the Aventra train family has shown that the development effort and the error proneness are mitigated already for software product lines containing a limited number of products and features. We have complemented the evaluation, with a questionnaire on the Bombardier engineers. We have discussed lessons learnt and, for each of the steps of the proposed approach, we have identified a list of factors and alternatives that could affect its adoption in other contexts.

Future work might encompass several research directions. One direction is to extend the translation file with a proper JSON schema for improving drawbacks related to type-checking and consistency. Another possible extension improving the specification of the mappings could be to develop a domain-specific language. Eventually, a further extension is to investigate the possibility of realising the proposed approach within FeatureIDE. Another research direction encompasses the development of the generation mechanism using model-transformation languages. In particular, we are already investigating the use of the Acceleo transformation language. Finally, we are investigating how to automatically collect or elicit features from requirements for enabling the automatic derivation and categorisation of features and constraints. This could be achieved with the interplay between formal notations for the specification of requirements and a feature mining approach.

Acknowledgments

The work in this paper is supported by the Swedish Knowledge Foundation (KKS), through the projects A-CPS and MINEStrA, and by the Swedish Governmental Agency for Innovation Systems (VINNOVA), through the project PANORAMA.

References

Arboleda, H., & Royer, J.-C. (2012). *Model-driven and software product line engineering*. John Wiley & Sons, Incorporated.

Bell, R. (2006). Introduction to iec 61508. In *Acm international conference proceeding series* (Vol. 162, pp. 3–12). Australian Computer Society, Inc.

Berger, T., She, S., Lotufo, R., Wasowski, A., & Czarnecki, K. (2010). Variability modeling in the real: A perspective from the operating systems domain. In *Ase '10* (p. 73–82). Association for Computing Machinery.

CENELEC. (2020). *Railway applications - communication, signalling and processing systems - software for railway control and protection systems.* (https://standards.globalspec.com/std/14317747/EN%2050128)

Czarnecki, K., & Eisenecker, U. W. (2000). Generative programming: Methods, tools, and applications. *Addison-Wesley*.

Czarnecki, K., Helsen, S., & Eisenecker, U. W. (2004). Staged configuration using feature models. In *Software product lines. splc 2004. lecture notes in computer science, vol 3154.* Springer, Berlin, Heidelberg.

Danilo Beuche, M. D. (2006). Software product line engineering with feature models. *Software Development Magazine - Project Management, Programming, Software Testing*.

Eramo, R., & Bucaioni, A. (2013). Understanding bidirectional transformations with tggs and jtl. *Electronic Communications of the EASST*, *57*.

Fang, M., Leyh, G., Doerr, J., & Elsner, C. (2016). Multi-variability modeling and realization for software derivation in industrial automation management. In *Proceedings of the acm/ieee 19th international conference on model driven engineering languages and systems* (p. 2–12). New York, NY, USA: Association for Computing Machinery.

Gharibi, G., & Zheng, Y. (2016). Archfeature: Integrating features into product line architecture. In *Proceedings of the 31st annual acm symposium on applied computing* (p. 1302–1308). New York, NY, USA: Association for Computing Machinery.

Ivarsson, M., & Gorschek, T. (2011, 06). A method for evaluating rigor and industrial relevance of technology evaluations. *Empirical Software Engineering*, *16*, 365-395.

Klaus Pohl, F. v. d. L., Günter Böckle. (2005). *Software product line engineering*. Springer, Berlin, Heidelberg.

Lee, K., Botterweck, G., & Thiel, S. (2009, May). Feature-modeling and aspect-oriented programming: Integration and automation. In *2009 10th acis international conference on software engineering, artificial intelligences, networking and parallel/distributed computing* (p. 186-191).

Lethbridge, T. C., Sim, S. E., & Singer, J. (2005). Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, *10*, 311–341.

Liebel, G., Marko, N., Tichy, M., Leitner, A., & Hansson, J. (2014). Assessing the state-of-practice of model-based engineering in the embedded systems domain. In *International conference on model driven engineering languages and systems* (pp. 166–182).

Loesch, F., & Ploedereder, E. (2007). Optimization of variability in software product lines. In *11th international software product line conference (splc 2007)* (p. 151-162).

Meinicke, J., Thüm, T., Schrter, R., Benduhn, F., Leich, T., & Saake, G. (2017). *Mastering software variability with featureide* (1st ed.). Springer Publishing Company, Incorporated.

Metzger, A., & Pohl, K. (2014). Software product line engineering and variability management: achievements and challenges. In *Future of software engineering proceedings* (pp. 70–84). Association for Computing Machinery.

Nesrine, L., & Bennouar, D. (2018). On the use of model transformation for the automation of product derivation process in SPL. *Acta Universitatis Sapientiae, Informatica*, *10*, 43-57.

Pop, P., Scholle, D., Hansson, H., Widforss, G., & Rosqvist, M. (2016). The safecop ecsel project: Safe cooperating cyber-physical systems using wireless communication. In *2016 euromicro conference on digital system design (dsd)* (pp. 532–538).

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, *14*(2), 131.

Seidl, C., Winkelmann, T., & Schaefer, I. (2016). A software product line of feature modeling notations and cross-tree constraint languages. In A. Oberweis & R. Reussner (Eds.), *Modellierung 2016* (p. 157-172). Bonn: Gesellschaft für Informatik e.V.

Sendall, S., & Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *IEEE software*, *20*(5), 42–45.

Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, Ø., Møller-Pedersen, B., & Olsen, G. K. (2010). Developing a software product line for train control: A case study of cvl. In J. Bosch & J. Lee (Eds.), *Software product lines: Going beyond* (pp. 106–120). Berlin, Heidelberg: Springer Berlin Heidelberg.

Tawhid, R., & Petriu, D. C. (2011). Automatic derivation of a product performance model from a software product line model. In *2011 15th international software product line conference* (p. 80-89).

Torres, M., Kulesza, U., Sousa, M., Batista, T., Teixeira, L., Borba, P., . . . Masiero, P. (2010). Assessment of product derivation tools in the evolution of software product lines: An empirical study. In *Proceedings of the 2nd international workshop on feature-oriented software development* (p. 10–17). New York, NY, USA: Association for Computing Machinery.

Vasilevskiy, A., Chauvel, F., & Haugen, u. (2016). Toward robust product realisation in software product lines. In *Proceedings of the 20th international systems and software product line conference* (p. 184–193). New York, NY, USA: Association for Computing Machinery.

Vasilevskiy, A., Haugen, u., Chauvel, F., Johansen, M. F., & Shimbara, D. (2015). The bvr tool bundle to support product line engineering. In *Proceedings of the 19th international conference on software product line* (p. 380–384). New York, NY, USA: Association for Computing Machinery.

Voelter, M., & Groher, I. (2007). Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th international software product line conference* (p. 233–242). USA: IEEE Computer Society.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

About the authors

Enxhi Ferko is a PhD student at Mälardalen University (Sweden). You can contact him at enxhi.ferko@mdh.se.her

Alessio Bucaioni is an assistant professor at Mälardalen University (Sweden). You can contact him at alessio.bucaioni@mdh.se.him

Jan Carlson is a professor at Mälardalen University (Sweden). You can contact him at jan.carlson@mdh.se.him

Zulqarnain Haider is a software engineer at Bombardier Railway Transportation.