# Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and its Evaluation

Alexander Schlie
TU Braunschweig, Germany
a.schlie@tu-braunschweig.de

David Wille
TU Braunschweig, Germany
d.wille@tu-braunschweig.de

Sandro Schulze
Otto-von-Guericke-University in Magdeburg, Germany
sandro.schulze@iti.cs.uni-magdeburg.de

Loek Cleophas
Stellenbosch University, South Africa
TU Eindhoven, The Netherlands
l.g.w.a.cleophas@tue.nl

Ina Schaefer
TU Braunschweig, Germany
i.schaefer@tu-braunschweig.de

## ABSTRACT

Model-based languages such as MATLAB/Simulink play an essential role in the model-driven development of software systems. Although they allow for problems to be lifted to a more abstract level, their development remains a complex and time-consuming process. To comply with new requirements, it is common practice to create new variants by copying and modifying existing systems. Efficient and documented reuse of systems requires dedicated variability management. If not present, the information about the relations between system variants is lost. As a result, the process of manually identifying reuse potential within existing variants might impose a greater workload than required to develop the system from scratch. Unfortunately, disregarding explicit system reusability only aggravates variant diversity. For systems to be efficiently reused, their variability information then needs to be reverse-engineered. In this paper, we propose a customizable metric and an advanced comparison procedure, the Matching Window Technique. Both allow us to overcome structural alterations commonly performed during system adaptation. We analyze related MATLAB/Simulink models and determine, classify and represent their variability information in an understandable way. With our technique, we assist model engineers in maintaining existing variants and offer guidance to derive new variants. We provide three feasibility studies with real-world models from the automotive domain and show our technique to be fast and precise. Furthermore, we perform semi-structured interviews with domain experts to assess the potential applicability of our technique in practice.

## CCS CONCEPTS

**•Software and its engineering →Software reverse engineering; Maintaining software;** *Software evolution;*

## KEYWORDS

MATLAB/Simulink · variability mining · software maintainability


*SPLC'17, Sevilla, Spain*

DOI: 10.475/123_4

**ACM Reference format:**
Alexander Schlie, David Wille, Sandro Schulze, Loek Cleophas, and Ina Schaefer. 2017. Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and its Evaluation. In *Proceedings of 21st International Systems and Software Product Line Conference, Sevilla, Spain, September 2017 (SPLC'17),* 10 pages.
DOI: 10.475/123_4

## 1 INTRODUCTION

In a variety of industrial domains, model-based languages such as *MATLAB/Simulink*[1] are of substantial importance, especially for the development of complex and safety-critical systems. Although they allow the engineers to address problems on a more abstract and intuitive level [8], the overall development of such systems remains a challenging and time-consuming task. To comply with changing requirements, i.e., new customer demands, it is common practice to reuse existing systems by copying and modifying them [19]. Commonly referred to as *clone-and-own* [21], this process can save time and workload in the beginning but it typically entails severe problems in the long-run. With this quick and easy-to-use approach, proper documentation about relations between cloned systems is usually not present. Crucial data such as version information stored within the model itself might accidentally not be updated after system adaptation. As a result, systems might be considered equal rashly although they diverge. Moreover, copying and subsequent modification can result in faulty parts to be unintentionally carried over from the initial model. If not immediately revised, these parts can accumulate undetected and account for inexplicable malfunctions at a later time. Without precise knowledge about relations between cloned systems, engineers have to manually identify every single model containing the affected parts and apply the patch accordingly. Identification and repair of such parts then require a vast manual workload and cause significant maintenance overhead.

With an emerging variant diversity and insufficient or imperfect documentation, the process of manually analyzing existing systems to identify their reuse potential might impose a greater workload than it is required to develop the entire system again from scratch. Although specific models might exist that are appropriate for reuse, they are neglected and dismissed without consideration.

[1] Mathworks® - http://www.mathworks.com/products/simulink/ - February 2017

Instead, a new model is created with almost the same functionality an existing system would have offered. Evidently, this only intensifies maintenance issues by boosting variant diversity even further.

Nevertheless, reuse might be a valid approach, especially if a system only needs minor rectification to meet the new requirements. Furthermore, structured reuse might also be desired and, by specifying common and varying parts, can enforce modularity. However, for systems to be efficiently reused, their variability information need to be reverse-engineered. This is of vital importance for deriving new variants and maintaining existing ones at the same time.

In this paper, we address these issues by analyzing and identifying variability information within related MATLAB/Simulink models. We introduce an advanced technique to properly handle structural changes commonly performed during model adaptation. Contrary to work that mainly addresses clone-detection in model-based languages (e.g., [2, 8, 19]), we not only identify common and varying parts between models but also their specific relation. With our proposed technique, we support model engineers in maintaining existing models and provide guidance to identify their reuse potential for deriving new variants. Hence, we make the following contributions:

- We outline structural changes commonly performed during model cloning and subsequent modification. Specifically, we explain *insertions* causing horizontal dispersion and *hierarchical shifts* causing vertical dispersion of related parts.
- We propose the *Matching Window Technique*, an advanced comparison procedure and a fully customizable metric to reverse-engineer variability information from related *MATLAB/Simulink* models. With this procedure, we specifically account for the outlined structural changes.
- We evaluate our approach using three feasibility studies from the automotive domain and show our technique to be fast, precise and applicable to models of industrial size. More precisely, we utilize one publicly available case study and two confidential case studies provided by our industry partner.
- We undertake an exploratory study by conducting semi-structured interviews with experts from the automotive domain to investigate the extent to which our approach meets the developers demands in an industrial environment.

The remainder of this paper is structured as follows. We provide background information on MATLAB/Simulink and briefly summarize our previous work [11, 34, 35] (Sec. 2), explain two significant structural changes commonly performed during model adaptation and introduce our proposed *Matching Window Technique* and our customizable metric (Sec. 3). We assess our technique using three feasibility studies with models from the automotive domain (Sec. 4) and discuss the results produced by our technique (Sec. 5). We conduct an exploratory study using semi-structured interviews [18] with experts from the automotive industry (Sec. 6) and state related work (Sec. 7) and future work (Sec. 8).

# 2 BACKGROUND

We provide basic terminology for representing implementation-specific variability for *MATLAB/Simulink* models. We briefly summarize our previous work and outline our preceding comparison approach, its workflow and limitations that inspired our proposed *Matching Window Technique*.

## 2.1 MATLAB/Simulink Models

*MATLAB/Simulink* is a block-based, behavioral modeling language, utilizing *functional blocks* and *signals* to specify certain software system functionality. It is used in a variety of domains, for instance in the automotive industry to develop safety-critical systems. Such models play an essential role in embedded software systems as they allow for generating development artifacts, such as source code and test cases. Each block has a set of semantical and syntactical properties that allow for their identification and comparison. Focusing on *MATLAB/Simulink*, the following block properties are of interest for the remainder of this paper:

- *name:* The textual name of a block.
- *function:* Constitutes the semantic meaning of a block, i.e., what the block is used for.
- *interfaces:* A block's interface consists of:
  - *in-ports:* For incoming data (usually signals). A block contains an arbitrary number of in-ports, each of them being connected to exactly *one* out-port.
  - *out-ports:* Constitutes the interface for outgoing data. A block contains an arbitrary number of out-ports, each associated with *one or more* in-ports.
- *connector:* A directed edge, connecting in- and out-ports.

Usually, *MATLAB/Simulink* models contain thousands of blocks to capture the behavior of complex systems [7, 8]. For engineers to keep an overview, logically connected blocks are commonly grouped within a *Subsystem* block which hides the contained blocks from direct view until explicitly accessed. Subsystems can be nested and constitute a *model hierarchy*. We refer to the contained blocks as *sub blocks*. They are shifted within the model hierarchy by one *hierarchical layer* with respect to the subsystem block, their *parent block*. Fig. 1 depicts two models, each one having two hierarchical layers.

## 2.2 Variability in Models

Models evolving from clone-and-own approaches diverge. We refer to locations where similar models differ as *variation points*. Fig. 1 exemplifies such a variation point between two simple models and circles in red the respective blocks. Although no difference is present on the first hierarchical layer, the two models shown in Fig. 1 diverge within their respective subsystems. Variation points can comprise numerous blocks and hierarchical layers. Depending on the extent of the performed modification, multiple variation points can be present within the analyzed models. Even when comprising just a few blocks, their correct identification is crucial. For instance with the variation point illustrated in Fig. 1 comprising just two blocks, a completely different output is produced and possibly passed on for further processing.

We use a *family model* [11] to represent variation points and to capture implementation-specific variability. Fig. 1 illustrates the family model '*FM_M1_M2*' corresponding to the depicted *MATLAB/Simulink* models *M*1 and *M*2. Within the family model, every single block of the analyzed models and their relation is represented. For instance, the two blocks labeled '*or*' and '*xOr*' shown in Fig. 3 vary but relate to each other by forming a variation point. The family model preserves this information and links the blocks together, allowing for a precise identification of the variation point.

Minor deviations, for instance to the block label such as '*Fix*' and '*Fixes*' in Fig. 1 are also captured. Family models are a valuable asset for developing and managing *software product lines (SPLs)* [20] and are used in industry standards like the *pure::variants* tool[2].

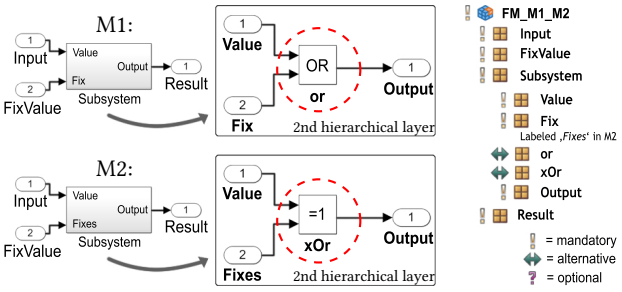


**Figure 1: MATLAB/Simulink Models and Family Model**

## 2.3 The Family Mining Framework

We introduced the *Family Mining Framework* to mine variability within related *MATLAB/Simulink* models in [34] and we depict its basic workflow in Fig. 2. We briefly recapitulate the overall process by explaining the three main phases: *Compare*, *Match* and *Merge* and we refer to [11, 34, 35] and our website[3] for more details. For every single block, the properties listed in Sec. 2.1 and information about connected signals (i.e., *label*, if present) are stored in an internal representation based on *Eclipse*[4] and its *Modeling Framework*[5]. We allow for an arbitrary number of models to be analyzed and iteratively compare and merge two models. We initially choose the model with the fewest blocks serving as the *base model* and one of the remaining *comparison models* to initiate the first iteration.

Paramount to the process, we introduce a *Compare Element (CE)* to logically associate compared blocks. Every CE features a normalized value representing the similarity of the contained blocks. A metric-based system is used to evaluate certain block properties like the *name* and *function* with each property having a certain weight.

*Compare:* Based on the implemented comparison technique, the input models are compared and an initial list of CEs is created. As shown in Fig. 2, the specific comparison technique is essential to the workflow such that all following phases rely on the created CEs.

*Match:* From the initial list of CEs created during *Compare*, a subset is retrieved that features the best matches for all evaluated blocks. During this phase, CEs can be *conflicting*, i.e., they contain the same uniquely identifiable block and for that have the exact same similarity value. We resolve such conflicts automatically if possible or display the conflict to the developer for manual resolution.

*Merge:* Based on the retrieved best matches, we merge both models. For each CE, the following user-adjustable mapping function is used to categorize the relation of the contained blocks.

$$rel(A,B) = \begin{pmatrix} similarity >= 0.95 & mandatory \\ 0 < similarity < 0.95 & alternative \\ similarity = 0 & optional \end{pmatrix} \quad (1)$$

Blocks present in both input models with a similarity of >=95% are considered *mandatory*. The 5% margin allows for minor deviations, i.e., to the name (c.f. Fig. 1). Blocks that only exist in one model are declared *optional*. Blocks, present in both models but distinctly varying are considered *alternative*, thus mutually exclusive. The resulting *150% model* is an intermediate model containing all blocks of the input models along with their identified variability information. The *150%* model is then either used as input again to be compared with another *comparison model* or exported as the final result model representing all models and their identified variability.

## 2.4 The Data Flow Approach

Integrated within our framework, the *Data Flow Approach (DFA)* explicitly exploits the data flow between blocks for their comparison.

Each hierarchical layer is divided into *stages*, i.e., horizontal traverse sections. The first stage is usually comprised of *Inport* blocks which introduce data to the system, cf. *Input* and *FixValue* in Fig. 1. Every subsequent stage consists of the successors of all blocks within the current stage. The successors are retrieved by following the outgoing signals and, thus, following the data flow exactly once. For any two *n*-th stages within different input models, our algorithm compares all blocks with each other. Each of the two models shown in Fig. 1 contain two blocks on their first stage, both times labeled *Input* and *FixValue*. The comparison results in a total of $2*2 = 4$ CEs. Subsequently following the outgoing connectors leads to the second stage, in both models comprised of a single subsystem block. Only when comparing two subsystems with each other, our approach is applied recursively for the corresponding hierarchical layer.

Every stage is processed until the end of the data flow is reached and no successors exist. After comparing the blocks labeled *Output* on the second hierarchical layer in Fig. 1, we consequently move up again to process the remaining third stage on the first hierarchical layer. Finally, for every specific hierarchical layer and stage, a separate list of CEs is retrieved, containing pairs of blocks and their similarity. In case one model has no counterpart for a block in a stage, we introduce a comparison with *void*, resulting in a similarity of 0.

For the models shown in Fig. 1, the DFA reliably identifies the variation because it remains within the same stage and hierarchical layer. However, when this is not the case, DFA might just be a too strict approach and unable to understandably identify variations.

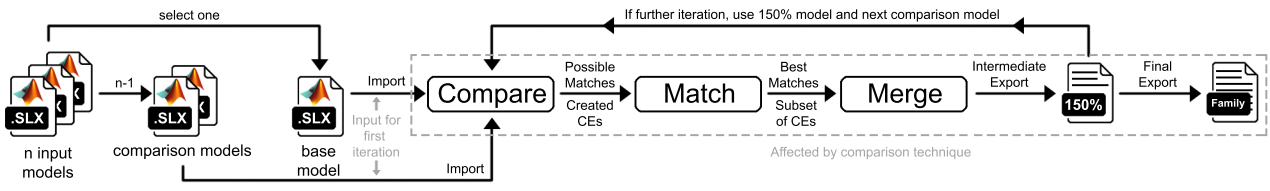


**Figure 2: Workflow of the Family Mining Approach**

[2]Pure-Systems® - http://www.pure-systems.com/ - February 2017
[3]https://www.isf.cs.tu-bs.de/cms/team/schlie/material/splc17mwt/
[4]Eclipse Foundation® - https://eclipse.org/ - February 2017
[5]Eclipse Foundation® - https://eclipse.org/modeling/emf/ - February 2017

## 3 MATCHING WINDOW TECHNIQUE

The DFA described in Sec. 2.4 performs well for simple models but is limited to structural changes that remain within the same stage and hierarchical layer. In this section, we describe alterations that cross stage and hierarchy boundaries and propose Matching Window Technique *(MWT)* to overcome the identified limitations. We further introduce an advanced similarity metric that allows for a more detailed comparison of blocks.

### 3.1 Horizontal and Vertical Dispersion

In cooperation with our industry partner, we identified two basic but vital operations likely to be performed during model adaptation.

*Insertions* disperse related parts within one hierarchical layer. Comparing the model $M1$ and the second hierarchical layer of $M2$, shown in Fig. 3, in isolation, the block *Velocity* has been added. The surrounding parts are dispersed horizontally and blocks are forced to other stages (c.f. *Gain* & *Output*). Applying DFA (c.f. Sec. 2.4) with its strict stage boundaries, their relation could not be identified.

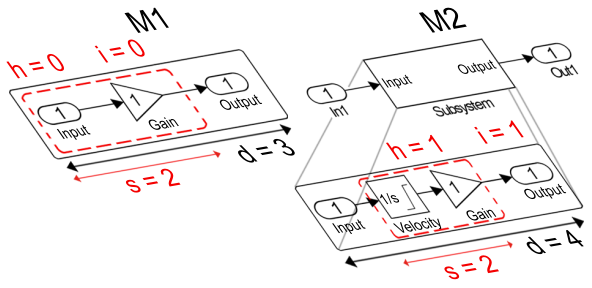


**Figure 3: Exemplified Dispersion and Matching Windows**

*Hierarchical shifts* disperse related parts across hierarchical layers. Beyond the outlined insertion in Fig. 3, $M2$ encapsulates $M1$*'s* functionality within a subsystem block. Hence, $M1$*'s* functionality has been shifted within the model hierarchy of $M2$. As illustrated, horizontal and vertical dispersion can intertwine. With *Gain* residing on the corresponding second stage in $M1$, DFA would not descend into the model hierarchy and not consider the encapsulated blocks at all.

### 3.2 Matching Window Algorithm

In our Family Mining Framework (c.f. Sec. 2.4), we introduce MWT to overcome the described limitations and to reverse-engineer variability information into understandable form even in the presence of dispersions. To this end, MWT flexibly divides each *MATLAB/Simulink* model into smaller, virtual *matching windows (MWs)*, illustrated by dashed boxes in Fig. 3. The MWs, one present in each model, can automatically decrease in size and *slide over* stages and hierarchical layers. The MWs then define new areas of the models in which DFA is applied for comparing the contained blocks. A particular window is defined by four parameters:

- ***d***, *model diameter*: maximum number of stages present in the model or subsystem respectively
- ***i***, *starting point*: index in the interval $0 \leq i < d$
- ***s***, *window size*: number of stages included in the window in the interval $s \leq (d - i)$
- ***h***, *hierarchical depth*: index in the interval $0 \leq h < h_{max}$

We introduce a *window pair (WP)* to associate the two windows, one present in each model. With the given parameters, we identify all possible windows for both, the current base and comparison model.

With any WP, the contained windows always comprise the same number of stages and, thus, always have the same size. Hence, every stage has a respective counterpart for it to be compared with. The algorithm we use to generate the WPs for a given base model $bm$ and comparison model $cm$ is shown in Procedure analyzeModels.

With the parameter $max_H$, we allow the engineers to define an upper bound up to which MWT descends into the model hierarchy. We predefine the minimal window size $s_{min} = 2$ but allow for it to be changed if desired. The maximum window size $s_{max}$ is automatically adjusted. It depends on the model or subsystem diameters respectively, as they can change across hierarchical layers (c.f. Fig. 3).

**Procedure** analyzeModels

**input** : $bm$, $cm$, $s_{max}$, $s_{min}$, $max_H$
**output**: window pair $wp$

```
1  WP ← ∅
2  foreach h_bm ∈ H(bm) with h_bm ≤ max_H do
3  |  foreach sub_bm ∈ S(h_bm) do
4  |  |  WP ← createWindowPairs(sub_bm, s_max, s_min)
5  |  end
6  end
7  foreach h_cm ∈ H(cm) with h_cm ≤ max_H do
8  |  foreach sub_cm ∈ S(h_cm) do
9  |  |  WP ← createWindowPairs(sub_cm, s_max, s_min)
10 |  end
11 end
12 return findBestMW(WP)
```

The algorithm traverses all subsystems $sub_{bm}$ in the base model and considers *all* hierarchical layers $h_{bm}$ with $h_{bm} \leq max_H$ (line 2–3). For each subsystem found within the base model, we compute a list of all possible WPs for the comparison model (line 2–4). The same process is performed for subsystems within the comparison model (line 7–9). This way, MWT is able to identify hierarchical shifts between the compared models in either direction. During this procedure, comparison of two uniquely identifiable blocks can be triggered multiple times. We avoid this by storing all created CEs in a cache. Instead of comparing the blocks a second time, the cached CE is retrieved. After traversing all hierarchical layers, we obtain a list of possible WPs. For each WP in this list, we compute its average similarity value using the following equation:

$$WP_{sim} = \frac{\sum_{x \in CE} sim(x)}{|CE| \in WP} \qquad |CE| \text{ is list of all CEs for that WP}$$

Particular blocks can be present in more than one CE within $|CE|$. For computing the overall similarity value of a WP, we only consider those CEs that provide the highest similarity value for a given block. To determine the best matches and to resolve possible conflicts, we proceed as mentioned in Sec. 2.4 and we refer to our website[3] for more details on conflict resolution. Finally, the WP with the highest similarity value is returned (line 12). Specifically using the base model as input, the corresponding algorithm is provided in the following Procedure createWindowPairs.

**Procedure** createWindowPairs

**input** : subsystem $sub_{bm} \in S(H(m_{bm}))$, $cm$, $s_{max}$, $s_{min}$
**output**: all corresponding window pairs $WP$

1 $WP \leftarrow \varnothing$
2 $thres := mandatoryThreshold()$
3 **while** $s_{max} \geq s_{min}$ **do**
4  $tmpWP \leftarrow \varnothing$
5  $x := d_{cm}$
6  **repeat**
7  $s_{cm} := d_{cm} - x + 1$
8  $x := x - 1$
9  $y := d_{bm}$
10  **repeat**
11  $s_{bm} := d_{bm} - y + 1$
12  $y := y - 1$
13  $wp := createWP(s_{bm}, s_{cm}, s_{max})$
14  **if** $sim(wp) \geq thres$ **then**
15  **return** $wp$
16  **end**
17  $tmpWP \leftarrow wp$
18  **until** $y - s_{max} \geq 0$;
19  **until** $x - s_{max} \geq 0$;
20  $wp_{insertion} := getInsertionWP(tmpWP)$
21  **if** $wp_{insertion} \neq null$ **then**
22  **return** $wp_{insertion}$
23  **end**
24  $s_{max} := s_{max} - 1$
25  $WP \leftarrow tmpWP$
26 **end**
27 **return** $WP$

Procedure createWindowPairs creates all WPs for the given upper and lower bound of the window size, $s_{max}$ & $s_{min}$ (line 3). These parameters are either set manually or calculated automatically with $s_{max} = min(d_{bm}, d_{cm})$, where $d_{bm}$ is the diameter of the subsystem and $d_{cm}$ is the diameter of the comparison model. $s_{min}$ is calculated with $\lceil \frac{s_{max}}{fraction} \rceil$. The *fraction* parameter controls the fraction of stages to be analyzed. It is set to 2 by default but can be changed.

The algorithm processes all hierarchical layers, i.e., every subsystem $sub_{bm}$ of the base model in relation to the comparison model. For the opposite direction, the algorithm proceeds identically.

Starting with $s_{max}$, all window combinations of the compared models are created and the size of the windows is reduced until $s_{min}$ is reached (line 3–26). For every created WP its overall similarity value is evaluated. If it exceeds the threshold defined for mandatory blocks (cf. Eq. 1), the contained windows contain identical blocks.

Hence, the algorithm found a direct match between the analyzed windows. The algorithm then returns the corresponding CEs as the best match (line 15). To detect horizontal dispersion, the algorithm takes all best matched CEs from the considered WPs and executes the matching for this set again. That way, the algorithm checks whether related parts exists that are horizontally separated within the currently considered window pairs and returns the corresponding WP (line 20–22). If the threshold for mandatory blocks is not reached and no insertion exists, a list of all possible WPs is returned.

## 3.3 Enhanced Metric

Extending previous work [35], we propose an enhanced metric that utilizes the parameters listed in Tab. 1 to compare blocks in greater detail than before. Specifically, we now further evaluate both the number and *function* of all directly connected blocks.

For instance, the block labeled *Gain* in $M1$ from Fig. 3 connects to two blocks, labeled *Input* and *Output*. Thus, the *function* of *Gain's* in-port is *Inport*, the internal function of the associated block *Input*. Consequently, the function of *Gain's* out-port is *Outport*, the internal function of the block *Output*. The distinction between these characteristics of ports allows for a more fine grained comparison of the blocks' interfaces and is a major improvement. By taking both the number and function of comprised ports into consideration, the surrounding structure of the compared blocks is evaluated to an extent that produces a more sound similarity for the compared blocks.

**Table 1: Properties used for the Metric-Based Comparison.**

| Property | Weight | Computation |
|---|---|---|
| name | 5% | $LD^*$ [14] of the blocks' names |
| function | 75% | $sim(f_A, f_B) = \begin{pmatrix} 1 & type(A)=type(B) \\ 0 & else \end{pmatrix}$ |
| #inports | 5% | $\sum_{i \in IN} (i) \,/ \lvert IN \rvert$ |
| #inport-functions | 5% | $\sum_{t \in T_{IN}} \left( \frac{\#t}{max(t)} \right) / \lvert T_{IN} \rvert$ |
| #outports | 5% | $\sum_{o \in OUT} (o) \,/ \lvert OUT \rvert$ |
| #outport-functions | 5% | $\sum_{t \in T_{OUT}} \left( \frac{\#t}{max(t)} \right) / \lvert T_{OUT} \rvert$ |

*Levenstein Distance, IN/OUT - set of in-/outports of a model block
$T_{IN}/T_{OUT}$ – set of functions of predecessor (IN)/successor(OUT)

## 3.4 Horizontal Dispersion Example

Horizontal and vertical dispersion can intertwine. Illustrated in Fig. 3, $M1$ has been shifted within the model hierarchy of $M2$ and insertion of *Velocity* caused *Gain* and *Output* to relocate across stages.

Focusing on horizontal dispersion, we list the corresponding excerpt from all generated WPs in Tab. 2. A complete list of all WPs is provided on our website[3], along with a dedicated screencast on vertical dispersion detection. Indicated by the parameter $h$ in Tab. 2, MWT has descended into the hierarchy of $M2$ while remaining on the first hierarchical layer in $M1$. Based on $M1$'s diameter, the window size is automatically set to $s = 3$. For both WPs $P_0$ and $P_1$, the contained CEs and their similarity value as well as the resulting WPs' overall similarity value are provided in Tab. 2.

**Table 2: Excerpt from Window Pairs for Models from Fig. 3**

| Pair | Windows | i | s | h | Compare Elements | Similarity |
|---|---|---|---|---|---|---|
| $P_0$ | $w_{bm}$ | 0 | 3 | 0 | Input - Input: 0.95 | 0.4% |
| | $w_{cm}$ | 0 | 3 | 1 | Gain - Velocity: 0.2<br>Output - Gain: 0.05 | |
| $P_1$ | $w_{bm}$ | 0 | 3 | 0 | Input - Velocity: 0.1 | 0.67% |
| | $w_{cm}$ | 1 | 3 | 1 | Gain - Gain: 0.95<br>Output - Output: 1 | |

$w_{bm}$ – window in base model M1, $w_{cm}$ – window in comparison model M2

For both WPs listed in Tab. 2, the default 95% threshold for mandatory blocks (c.f. Eq. 1) is not reached. If enabled by the engineer, MWT then initiates the horizontal dispersion detection procedure.

For that, MWT performs a second matching on all CEs contained within the WPs. Highlighted gray in Tab. 2 are the best matches retrieved by MWT for the respective CEs. Hence, we identify *Input*, *Gain* and *Output* to be mandatory. Since no counterpart for *Velocity* exists in *M1*, it is matched with *void* and, thus, is considered optional. The *Velocity* block then structurally separates two mandatory elements and as a result is correctly identified as an insertion. The procedure returns a WP containing the identified insertion and the expected matching as highlighted in Tab. 2. The procedure then terminates and does not create any further windows. If insertion detection is not enabled or no insertion was found within the WPs, MWT would further decrease the window size as illustrated in Fig. 3.

# 4 EVALUATION

To assess the feasibility of our proposed technique, we conducted three case studies with real-world models from the automotive domain. In this section, we provide our objectives, information about the analyzed models and the methodology used for the evaluation.

## 4.1 Objectives

With our proposed MWT, we reliably and understandably identify relations between *MATLAB/Simulink* model variants. By that, we aim to support engineers in maintaining and evolving existing models and to further provide guidance for their efficient and documented reuse. We focus on the following research questions:

**RQ1:** *Is the performance reasonable when scaling up and how do both approaches differ in terms of performance?* For applicability in practice, our proposed technique should exhibit a reasonable performance. That is, an overall runtime preferably within seconds or minutes.

**RQ2:** *What level of precision and recall can we achieve with our proposed technique?* Precision is paramount regarding both reliability and usability of our technique. We refer to precision as the extent to which the assigned relations between blocks are understandable to the user and, thus, considered valid. We refer to recall as the extent to which blocks have been processed and assigned a relation by MWT.

**RQ3:** *What specific requirements do engineers have regarding variability mining in an industrial environment?* We want to identify current and future challenges which variability mining for *MATLAB/Simulink* models faces in an industrial environment. We further want to assess the extent to which our proposed technique meets those requirements.

## 4.2 Analyzed Models

To assess the feasibility of our proposed technique in an industrial environment, we conducted a total of three case studies with real-world models from the automotive domain. Specifically, we used one generated case study and two case studies provided to us by our industry partner and their passenger car and truck devision.

*4.2.1 Generated Case Study.* Using an exemplary *driver assistance system (DAS)* from the publicly available SPES_XT [6] project, we artificially generated a set of mid-scale *submodels (SMs)* by identifying delimitable parts within the *DAS* model and extracting them. The extracted SMs we used for composing large-scale models are listed in Tab. 3 along with information on their size and complexity.

**Table 3: Basic Properties of the Extracted Submodels**

| Model name & *Abbreviation* | #blocks | $\#B_{Sub}$ | $D_{Hierarchy}$ |
|---|---|---|---|
| EmergencyBreak *'EB'* | 409 | 43 | 7 |
| FollowToStop ($req.$ CC) *'FTS'* | 699 | 77 | 11 |
| SpeedLimiter *'SL'* | 497 | 57 | 10 |
| CruiseControl *'CC'* | 671 | 74 | 11 |
| Distronic ($.req$ CC) *'DT'* | 728 | 78 | 11 |

$B_{Sub}$ – subsystem blocks, $D_{Hierarchy}$ – max. hierarchical depth, $req.$ - requires

Using the project documentation, we identified dependencies for *FTS & DT* that prohibit using them in isolation. Respecting the identified dependencies listed in Tab. 3, we combined these SMs and created a total of 19 different variants that explicitly address a model adaptation scenario. For instance, the largest model created contains all SMs listed in Tab. 3. Other models contain only one SM, e.g., *FTS* or two SMs respectively, e.g., *FTS* and *EB*. From an evolutionary point of view, the latter could be an extension of the former by adding *EB* to the model. For the 19 large-scale models created, manual evaluation of all possible 171 combinations[7] is infeasible. Hence, we randomly selected a subset of XX models, equally comprised of all large-scale model sizes in terms of contained SMs.

*4.2.2 Industrial Case Study.* For the industrial studies, we had access to a total of four models provided to us by our industry partner. From their passenger car devision, we were provided with two models constituting an *Exterior Light Front (ELF)*. We were further provided two models constituting a *Drive Train (DTM)* from their trucks devision. The ELF models contain ≈ 30.000 blocks each, the DTM models ≈ 40.000 blocks each. With the help of the domain experts of our industry partner, the models were partitioned into smaller SMs for processing. Logically associated SMs were clustered in groups, totaling 7 groups for the DTM and 3 groups for the ELF models. All models are confidential and only abstracted information on them can be provided. An excerpt of these groups is listed in Tab. 4 along with information on the contained submodels' sizes and complexity. The full list is available on our website[3].

## 4.3 Methodology

To evaluate both approaches regarding their applicability in an industrial environment, we specifically focus on analyzing the overall performance and precision of the generated results.

For performance, we examine the overall runtime and its distribution over the involved phases *Import*, *Compare and Match*, *Merge*,

[6] *Software Plattform Embedded Systems "XT"*, TU München - http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html - March 2017

[7] $(19 \cdot 18/2)$ because the input order does not matter for pairwise comparisons.

**Table 4: Excerpt from Groups for Industrial Studies**

| Group name | #submodels | #blocks min – max | $\#B_{Sub}$ min – max | $D_{Hierarchy}$ |
|---|---|---|---|---|
| **DriveTrain *(DTM)*:** | | | | |
| AGP | 13 | 6 – 821 | 1 – 257 | 3 |
| ISC | 25 | 63 – 18393 | 9 – 2429 | 6 |
| MLS | 11 | 119 – 5158 | 8 – 539 | 8 |
| ⋮ | | | | |
| **ExteriorLightFront *(ELF)*:** | | | | |
| AFB | 11 | 18 – 1575 | 2 – 201 | 5 |
| MSS | 8 | 37 – 362 | 8 – 48 | 3 |
| RFL | 10 | 29 – 1922 | 3 – 277 | 4 |

$B_{Sub}$ – subsystem blocks, $D_{Hierarchy}$ – max. hierarchical depth

and *Export*. Each comparison was performed 10 times and the average was calculated to account for runtime deviations inherently present in a non-closed system.

For precision, we face the problem of a missing ground truth. Because of their sheer size and complexity, it is infeasible to consider every possible combination of input models. We thus focused on certain combinations and manually identified the variability *before* applying our algorithm to specifically compare the results with our previous findings. For the generated study, we selected ≈ 25% of the comparisons and had two experts – well familiar with these *MATLAB/Simulink* models – examine the results of our family mining approach. For the industrial study, we had one of those two experts examine 22 SM combinations from the DTM models and 10 SM combinations from the ELF models. Particularly, we investigated whether the conflation of blocks and their assigned relation were understandable to each analyst and met their perception of variability.

## 5 RESULTS AND INTERPRETATION

The generated case study was evaluated on a Dual-Core i7 processor with 12 GB of RAM while for the industrial study, a Dual-Core i5 with 4GB of RAM was used. Both systems ran Windows 7 on 64bit. We can only show aggregated data in this section, but detailed results are provided online[3].

**Performance (RQ1):** To assess the proposed MWT and to evaluate its impact on runtime compared to our initial approach, we applied both methods to all pairwise combinations of the generated models (c.f. Sec. 4.2). The results are displayed in Fig. 4 and are ordered from left to right by the total size of both input models in terms of SMs.

The left entry for instance represents all model comparisons involving a total of 2 SMs whereas the right represents those involving a total of 9 SMs. For each entry, the left bar represents the data flow oriented approach, while the right represents the same comparisons performed using the MWT. For both approaches, our data shows a quadratic increase in runtime for a growing overall size of the compared models with the MWT requiring an average of 148% more time to process the same models and to identify the structural dislocations. Both techniques take only seconds to completely process the input models. Even for comparing the largest

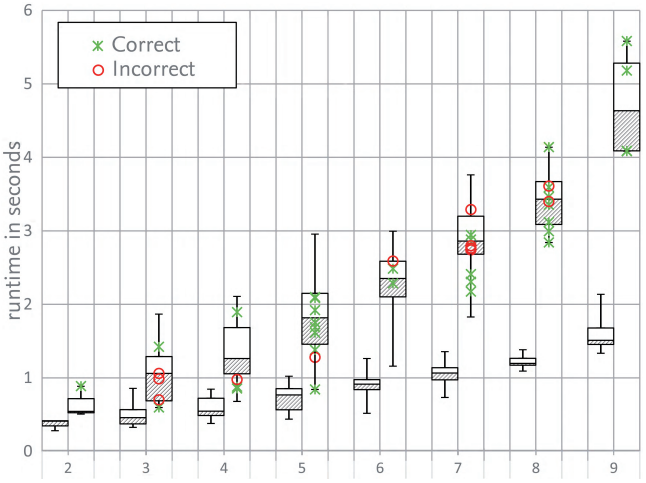


**Figure 4: Performance Results for the Generated Study**

models based on Tab. 3, MWT only takes ≈ 5 seconds. The results of the pairwise comparisons for all grouped ELF models listed in Tab. 4 are displayed in Fig. 5. Considering the correlation between

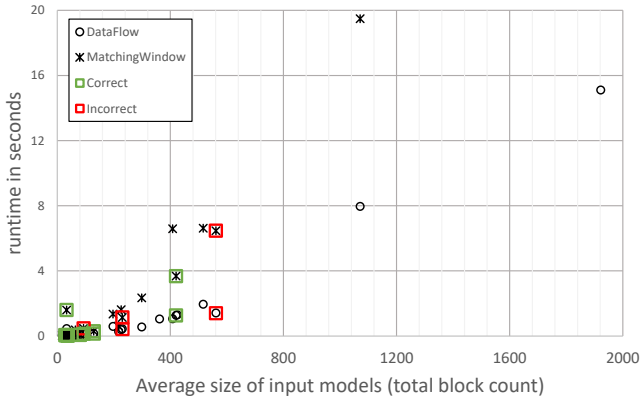


**Figure 5: Performance Results for *ELF* Industrial Models**

the total size of the input models and a runtime thats well within an acceptable time frame, we argue that our proposed method is applicable and efficient for real-world models.

**Precision (RQ2):** To evaluate the precision of our the MWT, we manually inspected the results of 47 comparisons for the generated study as described in Sec. 4.3. The results are displayed in Fig. 4 with green stars indicating results classified as correct and with red circles indicating false results. The data shows that our method produces mostly correct results; in numbers, 33 out of the inspected models have been analyzed correctly (≈ 70%). Beyond that, correct and incorrect results are scattered across comparisons of different model sizes and different runtimes. Hence, the quantitative analysis does not provide any explanation for incorrect results. When investigating the false results, we observed the *Merge Phase* to cause some blocks not being stored in the result model as expected. Displayed in Fig. 5, the green boxes indicate results for the ELF models

assessed to be correct while red boxes depict results that are correct but have minor flaws, i.e., a single missing signal name. For both studies, the problem of incorrect results originates in the *Merge Phase* and can easily be fixed. The crucial comparison and matching process during which the relation of blocks is analyzed works as expected. In summary, we argue that for pairwise comparisons, our method exhibits a relatively high precision for the inspected models.

**Scalability (RQ3):** Based on Fig. 4 and 5, our MWT scales and performs well in terms of runtime. Fig. 6 displays the comparisons for the ELF models and the corresponding total number of created CEs for the initial data flow oriented and our MWT. The initial approach is approximated by the blue line, the MWT is approximated by the red one. Following the trend indicated in Fig. 4 and 5, there is a quadratic increase in the total number of created CEs for a growing size of the input models. Especially for the industrial models, we found the memory space required for storing all CEs to be the major factor limiting the scalability, rather than the runtime itself. **Threats to validity:** Although we designed, implemented, and evaluated our methods with care, certain threats to validity are inherently present. First, we analyzed models from the automotive domain only. Other domains may entail different peculiarities. However, as automotive systems exhibit a relatively high complexity, we argue that our results are representative to some extent for pairwise comparisons. Although initiated by feedback from the domain experts from the automotive industry, we developed our methods independently and prevented ourselves from being biased to that specific domain. Hence, our method should be applicable to other models as well and, thus, be generalizable. Our similarity metric uses weights and thresholds that might be hard to specify upfront and that may influence the results. Given the domain knowledge of model engineers, the metric can either be adopted or replaced by a more suitable one. While our approach does not provide us with a universally valid result regarding precision, it strongly strengthens the confidence in our approach and its ability to produce understandable and valid results for related models.

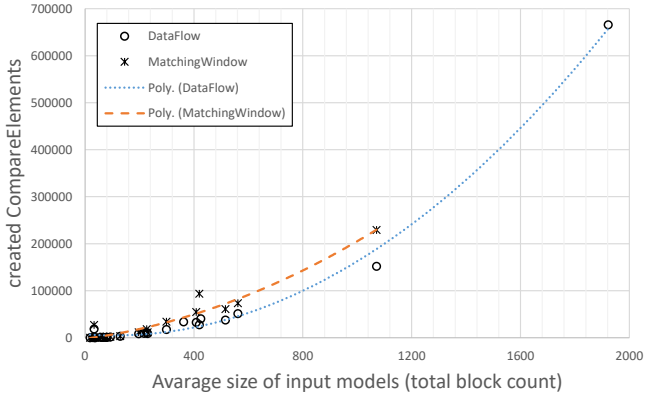


**Figure 6: Scalability of the ELF Models from Industrial Study**

## 6 EXPERT INTERVIEWS

To gain insight on how models emerge and evolve in an industrial environment in the automotive domain, but also to evaluate and discuss our findings, we conducted an exploratory study, performing semi-structured interviews [18] with domain-experts. Our objective was to get an impression of the current use cases for variability mining techniques, of the requirements engineers have regarding those techniques and the extent to which our proposed technique fulfills those demands. Specifically targeting the results produced by our approach, we asked for the interviewees' initial perception of usability, general applicability and the potential for further improvements regarding our proposed technique.

### 6.1 Participants

As summarized in Tab. 5, we interviewed eight domain-experts from our industry partner with the participants being active in the areas of model-based development (MBD), static analysis (SA) and research (Res) for 10 to 30 years.

**Table 5: Interview participants**

| Participant | Fields | Role |
|---|---|---|
| p1, p6 | MBD, SA | Senior Developer, Architect |
| p2 – p5 | MBD, SA | Senior Developer |
| p7 | MBD, SA | Technical Leader |
| p8 | MBD, SA, Res. | Senior Developer |

### 6.2 Data Collection

All interviews were performed separately with the participants not knowing the questions in advance and with answers not being accessible to others over the course of the interviews. Although all participants were familiar with our work, a short presentation on the overall approach was given prior to questioning to refresh the interviewees' memories. The full questionnaire we used to conduct the interviews is provided on our website[3].

*6.2.1 Questionnaire.* A total of nine questions were asked, covering three fields of interest, *General Expectations*, *Fields of Application*, and *Usability of the Results*.

*6.2.2 Interview.* The performed interviews included open-ended questions only and took between 25 and 35 minutes each. Before starting the interview, two models from the SPES-XT case-study were provided and the interviewee was asked to identify the variability. This was done to get an unbiased impression of the participants' perception of variability. For the interview, these models were compared using our *Matching Window Technique*, the generated result model presented to and the interviewee asked to what extent our approach captured the previously identified variability.

### 6.3 Results

The following sections provide a brief overview of the interviewees' feedback. Key aspects mentioned by the participants are shortly discussed at the end of this section.

*6.3.1 General Expectations.* The overall expectation expressed by all participants is our approach supporting them during model development and maintenance. Participants *p1*, *p2*, *p7* and *p8* explicitly mention lacking documentation during model evolution as

a problem for maintainability and the additional work necessary to recover or validate performed changes. The interviewees do not demand a flawless but a precise analysis and expect the generated result to always require some manual inspection until enough trust is established in the approach. Participants *p1*, *p2* and *p4* specifically refer to identifying both common and varying parts as crucial and highlight our approach's ability to combine that information as an advantage over common clone-detection and diff tools.

*6.3.2 Fields of Application.* Overall, the opinions on specific use cases diverge between the participants. Participant *p1* sees our approach as a preprocessing step prior to creating a *pure::variants* model, *p2*, *p4* and *p7* categorize our approach as valuable in a re-engineering scenario with *p4* and *p7* specifically mentioning the ability of our approach to identify reusable artifacts in existing models when developing a new system from scratch. On the other hand, *p3* does not see refactoring scenarios as imminent but refers to variability as already being considered during the design phase. Participant *p5* supports *p3*'s statements, saying that variability is currently approached in a constructive and not in a refactoring-oriented manner. The use of certain constructs provided by *MATLAB/Simulink* itself to explicitly model variability is confirmed by *p8* [32]. It is stated by *p1* to *p5* that the current primary use case revolves around analyzing two models. *p4* specifically mentions the traceability of changes performed to models in a repository as crucial. However, *p3*, *p5* and *p8* along with the other interviewees foresee a significant potential for our approach in the upcoming years when new systems are modeled and reusing existing artifacts gains more importance.

*6.3.3 Usability of the Results.* When asked about whether or not the result model reflects the participants initial perception of variability, interviewees, without exception agreed with the contained information but were skeptical about the scalability of the result model's form of representation (cf. Fig. 1). Participants *p1*, *p3* and *p5* highlight our categorization and annotation of variability as valuable for model maintenance, stating that all relevant information is captured to support developers in getting a first impression about the relation of models. Although the ELF and DTM models had to be partitioned for processing, *p6*, *p7* and *p8* do not see this as a drawback but specifically mention the users' domain knowledge allowing for a precise targeting of certain model parts for comparison and aggregation of the results. The lack of horizontal and vertical dispersion in those models were confirmed by all participants. *p1* and *p2* pointed out that a parameterizable 150% model is initially developed rather than different variants, thus explaining the few differences in the evaluated models. These structural peculiarities are usually not present when following the development guidelines of our industry partner, but stated by *p7* and *p8*, to occur when changes are performed in critical situations and this information is then required later. In conclusion, our approach is generally appreciated by the interviewed domain experts and the result model is considered understandable. The customizable metric and the possibility to introduce additional algorithms to the framework are also welcomed. Nevertheless, the form of representation can be improved and the participants would like the approach to be integrated within *MATLAB/Simulink*.

# 7 RELATED WORK

A variety of approaches exist in the literature to identify variability in models and clone detection is a prominent one. For instance, Deissenboeck et al. [8] operate on graphs and search for model clones using graph-based algorithms integrated into *ConQAT* [30]. Using *ConQAT*, Al-Batran et al. detect syntactic and semantic clones in *MATLAB/Simulink* models with semantic-preserving graph transformations [1]. With *ModelCD*, Pham et al. apply two algorithms *eScan* and *aScan* to labeled graphs to detect exact clones similar to Deissenboeck et al. and also *near-miss clones* [19]. Liang et al. use graph representations to identify the longest *common-subsequences*, but unlike our approach, without classifying the commonalities and differences [15]. Alalfi et al. use *SIMONE*, an extended version of their source code clone detection tool *NiCAD* [2, 3]. Using a TXL grammer for *MATLAB/Simulink*, they find clones on different levels of granularity with a recent extension allowing to consolidate differences on the subsystem level [4]. We do that too, but also detect, classify and relate similarities. Much like our approach, Ryssel et al. use a similarity criterion to describe both a block and its surrounding blocks, but then apply a clustering algorithm to detect *variation points* and store components in libraries for easy reuse [27]. Ryssel et al. use feature models to store the dependencies between such variation points, unlike our approach that uses a family model [25]. Another prominent technique aims at detecting differences between models. Unlike SiDiff [12, 31], these are usually commercial tools like SimDiff [9] and do not focus on the relation between model elements. Besides, several approaches exist to identify problem space variability (low level abstractions [5, 6]) by analyzing development artifacts and storing the gathered information in feature models. Weston et al. identify possible feature models from natural-language requirements [33]. Zhang et al. and Font et al. identify variability by utilizing EMF Compare [10, 36]. The feature-based variability is stored by means of the *common variability language (CVL)* but only one hierarchy level is considered. Building upon *MoVaC* [17], Martinez et al. propose *MoVa2PL* to semi-automatically identify features from a set of model variants, also using the CVL [16]. She et al. and Ryssel et al. reverse engineer feature models from product maps which are rather abstract and do not provide insights into concrete artifacts [26, 29]. With feature models, they provide problem space variability, while we infer solution space variability (high level abstractions [5, 6]) by means of family models. Rubin and Chechik define an operator that allows to merge different products into a SPL [22], extending it in [24] by adding measurable metrics to allow for a creation of different SPLs. However, this operator does not allow to derive a family model. We also acknowledge related work that is not directly dedicated to block-based models. Klatt et al. propose variability mining in source code using *abstract syntax trees (ASTs)* [13]. By using the AST, they consider more fine-grained variability but are limited in terms of application fields. In contrast to that, our approach can easily be adapted to other block-based modeling languages. Rubin and Chechik propose n-way merging to compare multiple models at once [23]. Although working for UML class diagrams and similar structured diagrams, applicability for more complex and large-scale models still has to be evaluated.

## 8 CONCLUSION AND FUTURE WORK

In this paper, we improve upon our existing variability mining approach by introducing an advanced, user-adjustable similarity-metric and a new comparison procedure called *Matching Window Technique*. For related *MATLAB/Simulink* models, both enhancements allow for a more detailed and accountable analysis. With MWT, we reliably reverse-engineer variability information, even in the presence of dispersions, and display that information in a way that is understandable and valuable to model engineers.

The provided feasibility studies show that we can now handle large-scale systems in both the presence and absence of dispersions and by that, increase applicability of our technique. The feedback we got from the interviewed domain experts clearly supports our technique and affirms MWTs' capability to actively ease maintenance of large-scale systems and to support model engineers in developing new systems – either from scratch or by reusing existing artifacts.

In the future, we plan to improve scalability to handle industrial models in their entirety and to display the generated results directly within the MATLAB/Simulink environment. Furthermore, we aim to find suitable ways of pre-processing models to filter out unnecessary comparisons beforehand. Ultimately, we also want to identify industrial use cases for the comparison of entire sets of related models and to put our approach's ability to do so to the test.

## ACKNOWLEDGMENT

We thank Richard Jockusch, Sönke Holthusen, Remo Lachmann and the domain experts for their help and strong support.

## REFERENCES

[1] Bakr Al-Batran, Bernhard Schätz, and Benjamin Hummel. 2011. Semantic Clone Detection for Model-Based Development of Embedded Systems. In *Proc. of the Intl. Conference on Model Driven Engineering Languages and Systems (MODELS) (Lecture Notes in Computer Science)*, Vol. 6981. Springer, 258–272.

[2] M.H. Alalfi, J.R. Cordy, T.R. Dean, M. Stephan, and A. Stevenson. 2012. Models are code too: Near-miss clone detection for Simulink models. In *Proc. of the Intl. Conference on Software Maintenance (ICSM)*. IEEE, 295–304.

[3] M.H. Alalfi, James R. Cordy, and Thomas R. Dean. 2014. Analysis and Clustering of Model Clones: An Automotive Industrial Experience. IEEE, 375–378.

[4] M.H. Alalfi, E.J. Rapos, A. Stevenson, M. Stephan, T.R. Dean, and J.R. Cordy. 2014. Semi-automatic Identification and Representation of Subsystem Variability in Simulink Models. In *Proc. of the Intl. Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 486–490.

[5] Kathrin Berg, Judith Bishop, and Dirk Muthig. 2005. Tracing Software Product Line Variability: From Problem to Solution Space. In *Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries (SAICSIT '05)*. South African Institute for Computer Scientists and Information Technologists, Republic of South Africa, 182–191.

[6] K. Czarnecki and U. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley.

[7] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. 2010. Model Clone Detection in Practice. In *Proceedings of the 4th International Workshop on Software Clones (IWSC '10)*. ACM, New York, NY, USA, 57–64. DOI: http://dx.doi.org/10.1145/1808901.1808909

[8] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. 2008. Clone Detection in Automotive Model-based Development. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 603–612.

[9] EnSoft. 2016. SimDiff. (Oct. 2016). http://www.ensoftcorp.com/simdiff/

[10] Jaime Font, Manuel Ballarín, Øystein Haugen, and Carlos Cetina. 2015. Automating the Variability Formalization of a Model Family by Means of Common Variability Language. In *Proc. of the Intl. Software Product Line Conference (SPLC) (SPLC '15)*. ACM, 411–418.

[11] Sönke Holthusen, David Wille, Christoph Legat, Simon Beddig, Ina Schaefer, and Birgit Vogel-Heuser. 2014. Family Model Mining for Function Block Diagrams in Automation Software. In *Proc. of the Intl. Workshop on Reverse Variability Engineering (REVE)*. ACM, 36–43.

[12] Udo Kelter, Jürgen Wehren, and Jörg Niere. 2005. A Generic Difference Algorithm for UML Models. *Software Engineering* 64, 105-116 (2005), 4–9.

[13] Benjamin Klatt, Martin Küster, and Klaus Krogmann. 2013. A Graph-Based Analysis Concept to Derive a Variation Point Design from Product Copies. In *Proc. of the Intl. Workshop on Reverse Variability Engineering (REVE)*. ACM, 1–8.

[14] Vladimir I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady* 10, 8 (1966), 707–710.

[15] Zhengping Liang, Yiqun Cheng, and Jianyong Chen. 2014. A Novel Optimized Path-Based Algorithm for Model Clone Detection. *Journal of Software* 9, 7 (2014), 1810–1817.

[16] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. l. Traon. 2015. Automating the Extraction of Model-Based Software Product Lines from Model Variants (T). In *Proc. of the Intl. Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE, 396–406.

[17] Jabier Martinez, Tewfik Ziadi, Jacques Klein, and Yves le Traon. 2014. Identifying and Visualising Commonality and Variability in Model Variants. In *Proc. of the European Conference on Modeling Foundations and Applications (ECMFA) (Lecture Notes in Computer Science)*, Vol. 8569. Springer, 117–131.

[18] M.Q. Patton. 1987. *How to Use Qualitative Methods in Evaluation*. Sage Publications.

[19] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Complete and Accurate Clone Detection in Graph-based Models. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 276–286.

[20] Klaus Pohl, Günter Böckle, and Linden, F. J. van der. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[21] Christian Del Rosso and Claudio Riva. 2003. Experiences with Software Product Family Evolution. *Principles of Software Evolution, International Workshop on* 00 (2003), 161. DOI: http://dx.doi.org/doi.ieeecomputersociety.org/10.1109/IWPSE.2003.1231223

[22] Julia Rubin and Marsha Chechik. 2012. Combining Related Products into Product Lines. In *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE) (Lecture Notes in Computer Science)*, Vol. 7212. Springer, 285–300.

[23] Julia Rubin and Marsha Chechik. 2013. N-way Model Merging. In *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 301–311.

[24] Julia Rubin and Marsha Chechik. 2013. Quality of Merge-Refactorings for Product Lines. In *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE) (Lecture Notes in Computer Science)*, Vol. 7793. Springer, 83–98.

[25] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. 2010. Automatic Variation-point Identification in Function-block-based Models. In *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 23–32.

[26] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. 2011. Extraction of Feature Models from Formal Contexts. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 4:1–4:8.

[27] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. 2012. Automatic library migration for the generation of hardware-in-the-loop models. *Science of Computer Programming* 77, 2 (2012), 83–95.

[28] Alexander Schlie, David Wille, Sandro Schulze, Loek Cleophas, Peter Manhart, and Ina Schaefer. 2016. FamilyMining additional material website. (Oct. 2016). https://goo.gl/25ao5Y

[29] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. 2011. Reverse Engineering Feature Models. In *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 461–470.

[30] CQSE – Continuous Quality in Software Engineering. 2016. ConQAT. (Oct. 2016). https://www.conqat.org

[31] Software Engineering Group, University of Siegen. 2016. The SiDiff project. (Oct. 2016). http://pi.informatik.uni-siegen.de/sidiff/

[32] Jens Weiland and Peter Manhart. 2013. A Classification of Modeling Variability in Simulink. In *Proc. of the Intl. Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*. ACM, 7:1–7:8.

[33] Nathan Weston, Ruzanna Chitchyan, and Awais Rashid. 2009. A Framework for Constructing Semantically Composable Feature Models from Natural Language Requirements. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. ACM, 211–220.

[34] D. Wille. 2014. Managing Lots of Models: The FaMine Approach. In *Proc. of the Intl. Symposium on the Foundations of Software Engineering (FSE)*. ACM, 817–819.

[35] David Wille, Sönke Holthusen, Sandro Schulze, and Ina Schaefer. 2013. Interface Variability in Family Model Mining. In *Proc. of the Intl. Workshop on Model-Driven Approaches in Software Product Line Engineering (MAPLE)*. ACM, 44–51.

[36] Xiaorui Zhang, Øystein Haugen, and Birger Møller-Pedersen. 2011. Model Comparison to Synthesize a Model-Driven Software Product Line. In *Proc. of the Intl. Software Product Line Conference (SPLC)*. IEEE, 90–99.