Interface-Based Similarity Analysis of Software Components for the Automotive Industry

Philipp Kehrbusch
Software Engineering
Ahornstraße 55
52074 Aachen, Germany
http://www.se-rwth.de

Johannes Richenhagen
FEV GmbH
Neuenhofstraße 181
52078 Aachen, Germany
http://www.fev.com

Bernhard Rumpe
Software Engineering
Ahornstraße 55
52074 Aachen, Germany
http://www.se-rwth.de

Axel Schloßer
FEV GmbH
Neuenhofstraße 181
52078 Aachen, Germany
http://www.fev.com

Christoph Schulze
Software Engineering
Ahornstraße 55
52074 Aachen, Germany
http://www.se-rwth.de

ABSTRACT

In a software product line similar products are derived based on a common foundation. With traditional methods a number of similar products exist independently. To derive a maintainable, reusable set of software components it is necessary to understand similarities and differences. However, an adequate similarity analysis involving several products is a cost-intensive and difficult task, compromising the pursued benefit. In this paper an automated syntactical similarity analysis for software component interfaces is proposed to support the software product line extraction and maintenance. We apply this general approach specifically to the automotive domain, as the static nature of the architecture and the high number of well-defined signals makes the interfaces especially expressive. The analysis supports three use cases: the identification of similarities between two interfaces, the automated extraction of a common interface for a set of interfaces and the evaluation of interface changes during the evolution history of a software component. In addition the syntactical analysis provides the foundation for further semantical examinations. Its applicability is indicated by a case study on different variants and versions of interfaces defined in an industrial context.

CCS Concepts

•**Software and its engineering** → **Software product lines;** *Software reverse engineering;* •**Computer systems organization** → *Embedded software;*

1. INTRODUCTION

In the automotive domain, the necessary quality level of software functions demanded by standards like CMMI or ISO 26262 is increasing. In addition the market requests shorter release cycles and more complex functionality [6]. Consequently software development processes needs to become more and more efficient. Software Product Line Engineering (SPLE) [19] focuses on the establishment of a reusable platform for a particular application domain with the target to gain efficiency through reuse. As the development of a Software Product Line (SPL) is more cost intensive as the development of just a specific product [19], it is necessary to ensure that the SPL is really reused due to several projects over time. In addition, in the industry the *clone-and-own* approach is preferred as it does not need any structured planning and can be performed easily and intuitively [9]. Still this approach is executed unsystematically and known disadvantages, like increased maintenance effort, can not be solved [9].

Several approaches have been defined in the last years to improve the situation and systematize *clone-and-own* [23, 12, 2]. In addition, inspired by different results from the field of Agile Product Line Engineering (APLE) [8] we defined a reactive process for the establishment and maintenance of a SPL [24]. This application engineering focused software product line development process is an extension of PERSIST [29, 20], an AUTOSAR[1]-compliant process for product line engineering.

Main important aspect is the comparison of software components of the product line with components from other projects [24]. Similarities of structural or semantical aspects of different software components are targeted to identify potentials of reuse and avoid parallel development besides the SPLE.

This paper focus on the structural similarity analysis of different software components in the context of the automotive domain with the target to identify similar or identical components under development and to be able to extract a generic interface for the establishment of a generic component, which can be reused in the different analyzed project contexts. Therefore in a first step matches between the signals of different interfaces needs to be established. Such a match is also mandatory for further semantical analysis [25].

The outline of the paper is as follows: Sect. 2 provides

SPLC '16, September 16 - 23, 2016, Beijing, China

ISBN 978-1-4503-4050-2/16/09...$15.00

DOI: http://dx.doi.org/10.1145/2934466.2934468

[1] http://www.autosar.org/

[KRRSS16] P. Kehrbusch, J. Richenhagen, B. Rumpe, A. Schloßer, C. Schulze:
Interface-based Similarity Analysis of Software Components for the Automotive Industry.
In: International Systems and Software Product Line Conference (SPLC '16), pages 99-108, Beijing, China, 2016. ACM.
www.se-rwth.de/publications/

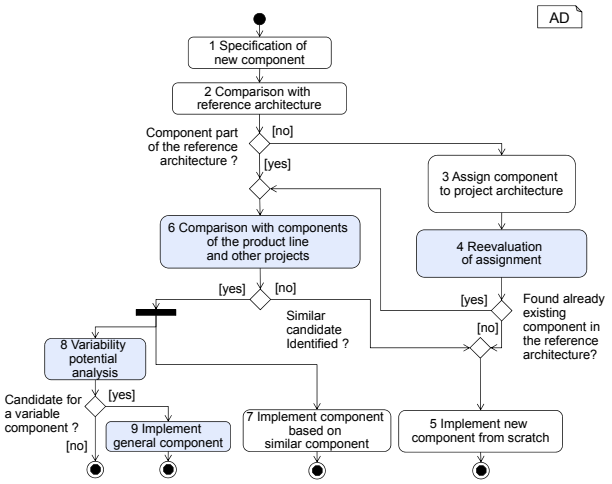

Figure 1: Reactive software product line engineering process (from [24], coloured activities indicate domain engineering related tasks).

the necessary background. In Sect. 3 the concept is explained, while in Sect. 4 its implementation is described. During Sect. 5 the the approach is evaluated and in Sect. 6 similar solutions are discussed. The paper finishes with a conclusion.

2. FOUNDATIONS

This section provides an overview about the process defined in previous work [24] which indicates the need for an automated interface-based similarity analysis. In addition, the basics regarding graph matching and aborescence are shortly described, as both are used to identify structural similarities between software component interfaces and to derive a generic interface.

2.1 Reactive Software Product Line Engineering

In previous work we have defined a reactive and agile process to synchronize a software product line with its products [24]. It is based on PERSIST [20], which is an architecture framework combined with an agile development process for the development of automotive powertrain software.

PERSIST defines a syntax for components and their interfaces consisting of several signals and parameters. Those signals are communication ports (input or output) or parameter and are defined by a signature which includes, among others, a name, a description, a width, a data type and a unit. In addition, a signal naming convention based on AUTOSAR providing an exclusive set of physical, logical and descriptive abbreviations is part of PERSIST to ensure consistent signal signatures.

The main important activities of the suggested process are illustrated in Fig. 1. When a new component is developed in a project (1), its position in the reference architecture is specified in the first place (2) and only if an adequate representative could not be identified, a new one is defined (3) in the context of the project. This assignment is double checked by the domain engineering team to ensure a correct decision (4). Based on this first filter, the search space is already reduced significantly, as already evaluated in previous work [24]. Therefore, in a next step, before or during the development of a new component (semi-)automated mechanisms can be used to identify similar components (6). These mechanisms can be based on artifacts describing structural (e.g. interfaces) or semantical aspects (e.g. test cases or behavioral models).

If a similar component could be found, a variability potential analysis is also conducted (8). This analysis determines, if given similarities are worth the development of a generic component (9).

Hence, the process needs a method to structurally compare component's interfaces in order to find existing components in the software product line. Moreover, a method is needed to extract a generic core from a set of similar products, which can be fed back into the product line.

In consequence, this paper proposes an analysis technique to identify similarities between different software component interfaces based on available attributes.

2.2 Graph Matching

Each interface consists of a set of signals. To identify similarities between two interfaces it is necessary to calculate the similarity between different signals and identify the most potential signal matches afterwards. If signals of different interfaces are expressed as graph nodes and the weighted edges express the similarity between such nodes (high weight = high similarity), graph matching algorithms can be used to identify best matches.

A matching on a Graph G is defined as *"a set of independent edges of G"* [4].
Given two edges $(a, b), (c, d) \in G$, those edges are independent if and only if $(a, c), (a, d), (b, c), (b, d) \notin G$.

A weighted matching is a matching, where the edges' weights are taken into account. In a maximum weighted matching edges are chosen such that the total weight of the selected edges is maximal.

2.3 Arborescence

Once a maximum weighted matching between signals of two different interfaces is derived it is also necessary to be able to derive commonalities if more than two interfaces are involved and to extract a common interface.

An arborescence of a directed graph G is a directed spanning tree of G. "*A subgraph $T = (V, F)$ of G is an arborescence with respect to root r if and only if T has no cycles, and for each node $v \neq r$, there is exactly one edge in F that enters v*" [13].

The search for a minimum cost spanning tree is known as the minimum cost arborescence problem. Analogously, the maximum cost arborescence problem refers to the search for a maximum cost spanning tree. In order to find such an arborescence, Edmond's algorithm [10] can be applied. However, it is not always possible to find an arborescence, which contains all of the original graph's nodes. This can be the case for weakly connected graphs. It is rather possible to find a branching, which is a directed spanning forest. Fig. 2 shows a graph, whose arborescence does not cover all of the graph's nodes (original graph on the left hand side, arborescence on the right hand side).

An arborescence is defined to be an out-tree (as opposed to an in-tree). In an in-tree, all edges are directed towards the root, while they are directed away from the root node in an out-tree. Edmond's implementation generating out-

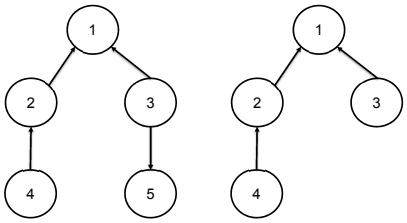

Figure 2: Graph (left) and its Arborescence (right, in-tree, inverted directions already applied)

trees can be used to generate in-trees by simply inverting the edges of the original graph, applying the algorithm, and inverting the edges of the resulting graph again.

3. CONCEPT

This section describes the conceptual aspects behind the proposed algorithm. The process can be divided into five steps: interface import, signal similarity analysis, signal mapping calculation, signal transformation path calculation and result evaluation.

In Fig. 3 simplified ingoing interfaces of three monitoring components are shown. This constructed example is used throughout the paper to motivate and demonstrate the different aspects of the proposed approach. The components intend to calculate the current system status based on the current system state (signals *Init*, *StartUp* and *SystemState*), the velocity (signals *Speed* and *Velocity* together with signal qualifier *StatusSpeed* and *VelocityQualifier*) and additional attributes like acceleration, temperature or battery voltage. The example is purely fictional and only inspired by practical examples. The connections between the different variants' signals in Fig. 3 are illustrating the intended correspondence (the matches to be identified by a structural similarity algorithm).

In *VariantA* and *VariantB* the system state is represented by just one Boolean value (init/startup), while *VariantC* defines a system state enumeration, able to represent 16 different states. Although an intended connection is given, the corresponding signal names differ. In addition, the system state is represented by different data types or different amount of signals. In case of the velocity attribute either the name, datatype (fixpoint vs. floating point arithmetic) or unit differs. In consequence also the defined range of the different signals are defined in relation to their unit representation. For the velocity signal qualifier the problem is similar to the status signal: one variant defines a simple boolean value, while the other variants represent 10 different states by an enumeration. Acceleration is considered by two different variants only, while motor temperature and battery voltage are variant specific signals.

The example shows four different important signal attributes used in the automotive industry: name, data type, unit, logical range. In comparison to the intended connections shown in Fig. 3, in Fig. 5 and Fig. 4 found matches based on name or on the datatype attribute are shown.

As for datatypes many matches are possible, only the more important ones are represented. The thickness of the edges indicates the relation strength. In addition relations are directed if an attribute can only be represented by a more general representation (e.g. Float32 can represent Int16).

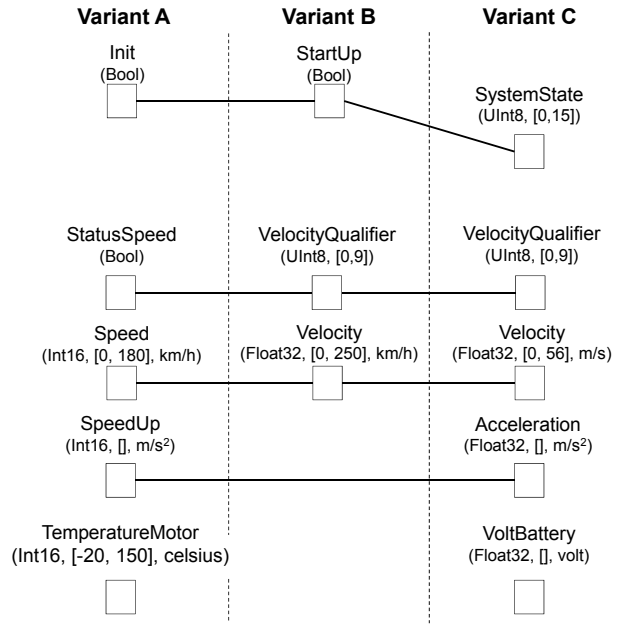

Figure 3: Suggested similarities between simplified ingoing interfaces of three monitoring components.

Similar to the other mentioned attributes, these examples shall demonstrate that the intended relation shown in Fig. 3 can not be extracted by just focusing on one of the provided attributes. In this example both attributes would provide misleading matches and only under the consideration of all provided attributes a more adequate match can be found. How this is done in detail will be explained in the following.

In the first step, the interface definitions of the interfaces to be compared are imported and represented using an internal data model. This keeps the algorithm independent of the format used to define the interface and enables different input formats to be used. Next, the interfaces are compared to each other by comparing the interfaces' signals. Interfaces are compared pairwise, and each signal from one interface is compared to each signal from the other interface. Hence, after the comparison step, there exist a similarity value between each two signals from the two interfaces. Afterwards, signal mappings are derived to map each signal from one interface to exactly one signal from the other interface, such that the total similarity is maximized. Those mappings determine, which signals shall be transformed to which other signals.

If the purpose is just to find the similarity between two interfaces, step four, the calculation of a transformation path, can be skipped. However, if the goal is to extract a generic core from a set of interfaces, an additional step is required. If the process is conducted for a set of more than two interfaces after step three, there exist groups of similar signals from different interfaces. For each group of similar signals, the goal is to find a signal that all other signals can be transformed to. If such a signal can be found, it can be included in the generic core. Hence, transformation paths between similar signals need to be derived.

Finally, the results from steps three and four need to be displayed to the user in a readable form to be used during the variability potential analysis. Therefore, a collection of

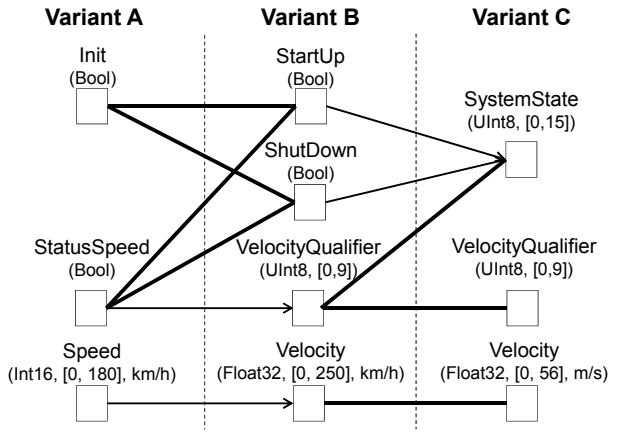

Figure 4: Data type matches for the simplified example.

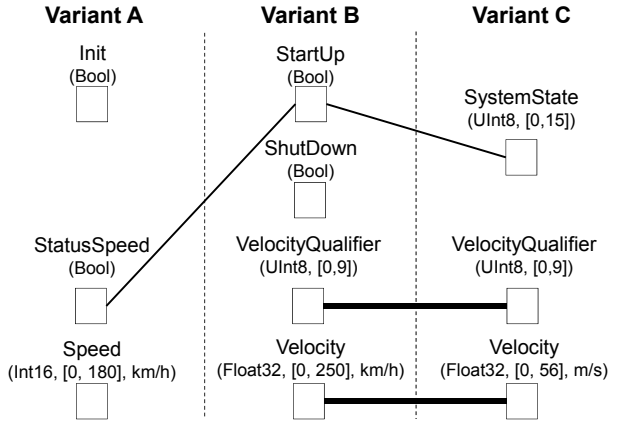

Figure 5: Name matches for the simplified example.

different views on the gathered data at different abstraction levels is provided.

3.1 Signal Similarity Analysis

As mentioned above, two interfaces (I_1 and I_2) are compared by matching their constituent signals. Let S_1 and S_2 be the set of signals and parameters of I_1 and I_2. For each pair of ingoing signals, pair of outgoing signals or pair of parameters (s_1, s_2) with $s_1 \in S_1$, a similarity value is derived. In the following it is always referred to signals (s_1, s_2), nonetheless pairs of ingoing signals, outgoing signals or parameter values are meant. The comparison is done by calculating the similarity of each of the signals' properties. The overall signal similarity is equivalent to the weighted arithmetic mean over the signal properties' similarities. The signal's properties that are considered during the comparison are its name, range (defined by a minimum and a maximum value), width, data type and unit. Let $Props$ be the set of considered properties and let $p_1 = prop(s_1, p)$ be the property value of property p of signal s_1.

On a more abstract view two signal properties can either be *equal*, *similar* or *different*, but a possible transformation from one signal to another signal is always direction dependent: it may be possible to transform a signal s_1 to a signal s_2, but not the other way around (as signal s_2 is more generic than signal s_1). To differentiate between similarities, which allow a proper transformation in both directions, and similarities, which permit only an one-sided transformation a fourth similarity level is introduced: *weakly similar*.

Let $trans_p(p_1, p_2, p)$ be a function, that determines whether or not property value p_1 is transformable to property value p_2 regarding property p. Two signal property values p_1, p_2 are defined to be *weakly similar*, if and only if $p_1 \neq p_2 \wedge trans(p_1, p_2) \neq trans(p_2, p_1)$. They are similar, if $p_1 \neq p_2 \wedge trans(p_1, p_2) \wedge trans(p_2, p_1)$ and they are different, if and only if $\neg trans(p_1, p_2) \wedge \neg trans(p_2, p_1)$.

The following defines, when two signal properties p_1, p_2 of property p are considered to be transformable, expressed by $trans(p_1, p_2, p)$:

- **name** For two names n_1, n_2, $trans(n_1, n_2, name)$ is always true.
- **width** Given two widths w_1, w_2, then $trans(w_1, w_2, width) \leftrightarrow w_1 = w_2$.
- **range** For two ranges $r_1 := [min_1, max_1]$ and $r_2 := [min_2, max_2]$, $trans(r_1, r_2, range) \leftrightarrow r_1 \subseteq r_2$.
- **data type** Given two data types d_1 and d_2, then $trans(d_1, d_2, datatype)$ if and only if values of data type d_1 are also representable using the data type d_2. For example, an eight bit signed integer value can be represented using a 16 bit signed integer, but the inverse can not be granted.
- **unit** For two units u_1, u_2, $trans(u_1, u_2, unit)$ if and only if u_1 can be converted to u_2 using SI-unit conversions [27]. For example, the unit meters per second can be converted to kilometres per hour.

The similarity $sim_p(p_1, p_2, p)$ of a property value p_1 from signal s_1 to property value p_2 of signal s_2 is defined as follows:

$$sim_p(p_1, p_2, p) = \begin{cases} 0 & \text{if } \neg trans(p_1, p_2, p) \\ 1 & \text{if } p_1 = p_2 \\ x & \text{if } trans(p_1, p_2, p) \wedge p_1 \neq p_2 \end{cases}$$

where $x \in [0, 1]$ is calculated using a property specific similarity function for each similarity criteria. The similarity functions for the different similarity criteria are defined as follows:

- **name** For two names n_1 and n_2, $sim_{name}(n_1, n_2)$ is calculated based on Levenshtein distance [15].
- **width** $sim_{width}(w_1, w_2) := 1$ for two widths w_1, w_2.
- **range** Let $r_1 = [r_{1,min}, r_{1,max}]$ and $r_2 = [r_{2,min}, r_{2,max}]$ be two ranges and $R_1 = r_{1,max} - r_{1,min}$, $R_2 = r_{2,max} - r_{2,min}$. Then $sim_{range}(r_1, r_2) := \frac{max(R_1,R_2) - |R_1 - R_2|}{max(R_1,R_2)}$.
- **data type** Let $bit(x)$ be the number of bits in a data type, then the similarity of two data types d_1, d_2 is defined as $sim_{datatype}(d_1, d_2) := \frac{min(bit(d_1), bit(d_2))}{max(bit(d_1), bit(d_2))}$
- **unit** Let u_1, u_2 be two units. [27] states that a quantity Q's unit is expressable in terms of SI base units: $[Q] := 10^n \times m^\alpha \times kg^\beta \times s^\gamma \times A^\delta \times K^\epsilon \times mol^\zeta \times cd^\eta$, where $-24 \leq n \leq 24$. Hence, $sim_{unit}(n_1, n_2) := \frac{49 - (|n_1 - n_2|)}{49}$.

Function sim_p is direction dependent, while the similarity functions for different specific properties are direction independent. In addition, a similarity threshold value can be specified for each similarity criteria. If the similarity of two properties is below the defined threshold, they are considered to be different. This way, besides the general condition of compatibility, it is possible to define a minimum degree of similarity two properties must have in order to be considered as similar.

The overall similarity $sim(s_1, s_2)$ for two signals s_1, s_2 is calculated as the weighted arithmetic mean over the signal properties' similarity:

$$sim(s_1, s_2) = \frac{\sum_{p \in Props} w_p * sim_{prop}(prop(s_1, p), prop(s_2, p))}{|Props|}$$

if

$$\forall p \in Props : trans(prop(s_1, p), prop(s_2, p)) \neq 0$$

Otherwise, $sim(s_1, s_2) = 0$.

w_p refers to the weight assigned to the property p. By assigning weights to the properties, the importance of each property for the overall similarity value can be adjusted.

Moreover, it is possible to select a priority order for signal properties, which will cause them to be compared in this order. Further comparisons can be ignored, if two properties are not considered transformable. This technique leads to an increase in performance, as less comparisons have to be carried out, and less signal pairs need to be considered during the matching step.

Results gained from the comparison step are stored in a graph structure. This enables them to be further processed and analysed in the next step. Signals form the graphs' nodes, while directed edges between nodes represent the directed similarity between the signals. The concrete similarity values are represented by the edges' weights.

Fig. 6 illustrates the derived graph structure after similarity values have been calculated for the simplified example.

Based on the similarity analysis for the property unit many different pairs are detected already and therefore the amount of possible matches reduces significantly. Only for unit less inputs multiple options exist.

3.2 Signal Mapping

The example in Fig. 6 illustrates the similarity relations between the interface of *VariantA* and *VariantB* and between *VariantB* and *VariantC*. Nevertheless, during the signal mapping step only the relation between two interfaces is considered at once. The goal of this step is to define which signals from one interface are most similar to which signals from another interface. Maximum weight graph matching is applied to the graph gained from the last step in order to retrieve a mapping, where the total similarity is maximized. The signals from Fig. 6 would be mapped as outlined in Fig. 7.

Again the matches between *VariantA / VariantB* and *VariantB / VariantC* are shown, but the mapping is calculated separately. Comparing Fig. 3 with the mapping extracted based on the described approach, all results but the mapping between the boolean values match (indicated by the red dotted lines). Based on the available attributes it is not possible to identify the correct match: for boolean values

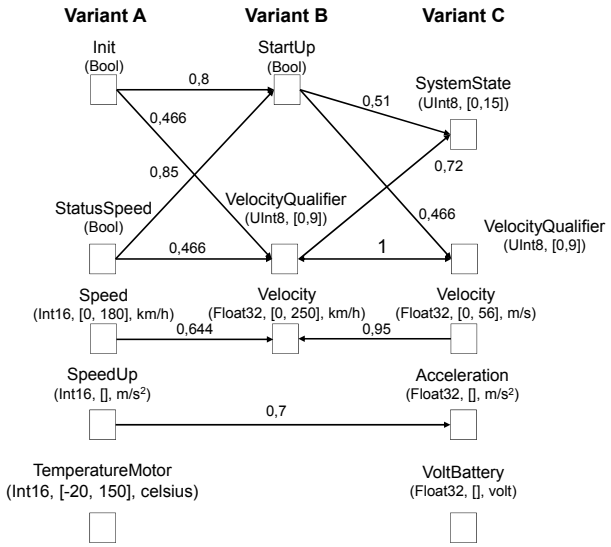

Figure 6: Calculated similarity values of the simplified example.

only name, width and datatype can be a correct indicator. Therefore, compared to other one dimensional booleans, only the name attribute can be taken into consideration for separation. This example already demonstrates that matches derived based on only three attributes and based on a very small naming similarity have a high chance to be false positives (invalid matches). To avoid false positives in this example a similarity threshold for the property name can be used, as described in Subsect. 3.1. As shown in Fig. 7, such a threshold can also result in false negatives (correct match between signal *StartUp* and signal *SystemState* is removed and intended matches between *Init/StartUp* and *StatusSpeed/ VelocityQualifer* are still not identified).

Based on the derived maximum match M a directed average similarity between interface I_1 and interface I_2 is calculated by:

$$sim_{dir}(I_1, I_2) = \frac{\sum_{(s_1, s_2) \in M} sim(s_1, s_2)}{|S_1|}$$

Again, the derived similarity is direction dependent and considers only all signals of the first interface to calculate the average similarity. In consequence, I_1 can be 100% similar to I_2, although I_2 consist of more signals, as long as all signals from I_1 are matched with 100% similarity. The undirected similarity between two interfaces is calculated by the average of the directed similarities:

$$sim(I_1, I_2) = \frac{sim_{dir}(I_1, I_2) + sim_{dir}(I_2, I_1)}{2}$$

3.3 Calculation of Transformation Paths

Based on the extracted matches between different interfaces a generic interface can be extracted. Connecting the matches found between the different interfaces, separated graphs can be extracted. In the following step for each of these graphs an arborescence is calculated to identify the most generic signal, which can be used for a generic interface suitable for all variants.

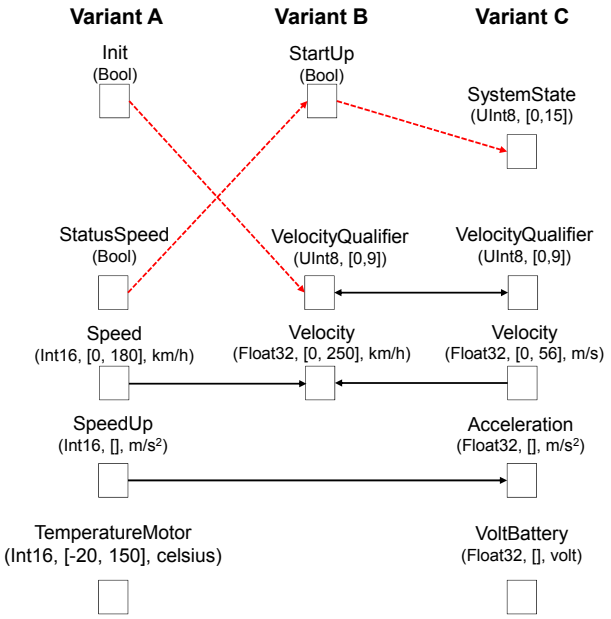

Figure 7: Calculated matches of the simplified example (filtered matches in red dotted lines).

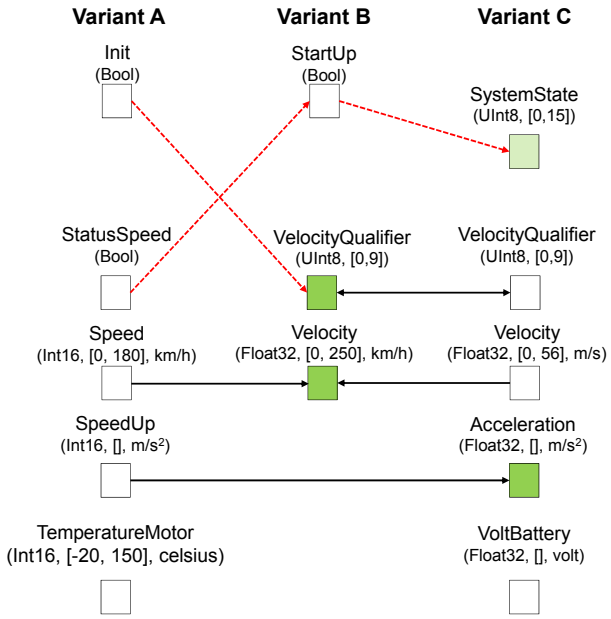

Figure 8: Transformation pathes extracted for the simplified example (filtered matches in red dotted lines, generic signals highlighted).

Considering the separated graphs from Fig. 7, Edmond's algorithm can be used to retrieve arborescences with highest similarity values for each of them. However, Edmond's algorithm generates out-trees with minimum weights. By inverting the similarity values and the edge direction before applying Edmond's algorithm, and reinverting afterwards, the required transformation paths can be extracted. As each edge between two signal nodes corresponds to a possible transformation, this would mean that there is a path of subsequent transformations from each signal node to the root node. Hence, the root node corresponds to the signal that all other signals can be transformed to. The result gained from the graph from Fig. 7 is displayed in Fig. 8. The highlighted nodes represent the extracted root nodes.

For each group of similar signals S, that are transformed into one signal s, with $s \notin S$, a transformation path similarity $sim_{path}(S, s)$, based on the derived single similarity values, is calculated:

$$sim_{path}(S, s) = \frac{\sum_{t \in S} sim(t, s)}{|S|}$$

This overall similarity value is the final indicator, if the extracted generic signal is a potential candidate for a generic interface. As Edmond's algorithm is calculating a spanning arborescence with minimum weight the derived total similarity value is the maximal similarity available by considering at many variants as possible.

In addition two preconditions needs to be fulfilled to ensure that the extracted arborescence represents a valid transformation path. First, many arborescences describe only a valid transformation path, if the relation described by the function $trans(p_1, p_2, p)$ defined in Subsect. 3.1 is transitive (in case not all nodes are connected directly). Therefore, in the following it is proven that for three signal property values p_1, p_2, p_3 of property p, the following holds:
$trans(p_1, p_2, p) \wedge trans(p_2, p_3, p) \implies trans(p_1, p_3, p)$.

- **Name** A name can always be transformed into any other name.
- **Range** For three ranges r_1, r_2, r_3, the following holds: $trans(r_1, r_2, range) \wedge trans(r_2, r_3, range) \implies r_1 \subseteq r_2 \wedge r_2 \subseteq r_3$. Hence, $r_1 \subseteq r_3$ and thus $trans(r_1, r_3, range)$.
- **Width** Let w_1, w_2, w_3 be three widths. Widths are only transformable if they are equal, hence $trans(w_1, w_2, width) \wedge trans(w_2, w_3, width) \implies w_1 = w_2 = w_3$ and therefore $trans(w_1, w_3, width)$.
- **Unit** For three u_1, u_2, u_3, $trans(u_1, u_2, unit)$ and $trans(u_2, u_3, unit)$ implies that u_1 is transformable to u_2 and u_2 is transformable to u_3. Hence, u_1 is transformable to u_3 and thus $trans(u_1, u_3, unit)$.
- **Data Type** Let d_1, d_2 and d_3 be three data types. $trans(d_1, d_2, datatype) \wedge trans(d_2, d_3, datatype)$ implies that values of data type are representable by data type d_2. Those values again are representable by data type d_3, hence $trans(d_1, d_3, datatype)$.

Hence, all relations $trans(p_1, p_2, p)$ are transitive.

Second, the extracted graph can not contain two edges extracted from the same variant. If this is the case, the resulting transformation path would indicate to transform two signals of one interface to one signal of another interface. E.g. in Fig. 9 the identified matching between *Init* and *SystemState* would result in a transformation path, which transforms both signals *Init* and *StatusSpeed* to signal *SystemState*. In addition, in a separate arborescence the signal *Init* is transformed to signal *VelocityQualifier*.

Instead of performing a correction upfront, the resulting path is highlighted as invalid to indicate a possible conflict: the transformation path maybe invalid, but already provides

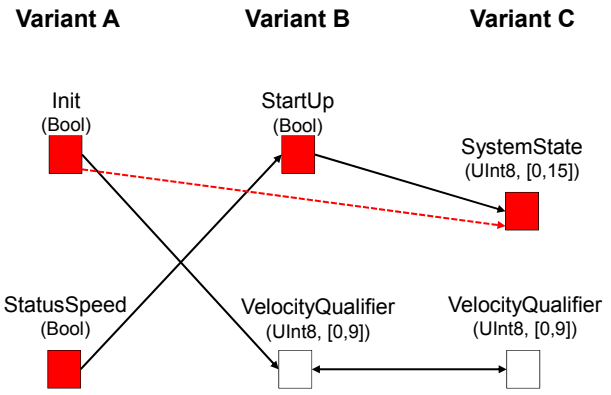

Figure 9: Invalid transformation path indicating false positives.

useful information to identify the source issue. In this example, if the invalid matches have not already been detected by an established threshold of the name property, the additional information gained by connecting several matches can be used to indicate false positives and therefore avoid invalid generic interface extractions.

4. IMPLEMENTATION

A prototype implementation of the proposed algorithm has been developed in Python in order to evaluate the approach.

The implementation focus on precision, instead of performance. The target is to be able to easily compare, exchange and modify parts of the proposed logic to increase the correctness of the identified signal matches and the derived generic interfaces.

The application has been divided into three major parts: the model, the controllers and the view module.

The model contains the data model for the internal representation of interfaces and their signals. Moreover, it contains an abstraction layer for the library independent representation of graphs. In the current implementation the library networkx[2] is used.

The controllers operate on the model and provide basic services like import, the comparison of signals or the generation of signal mappings. The logic importing the data, the metrics calculating the similarity values and the graph based algorithms (matching and Edmonds algorithm) are separated from each other and can easily be exchanged.

The comparison controller is responsible for the comparison of two or more signals and stores the results in a graph structure, which can be used for further analysis. In addition, metrics, which extract similarity relations based on the provided model, can easily be exchanged to support a fast comparison of different approaches. Another controller is responsible for the analysis of the comparison results. It generates signal mappings and calculates signal transformation paths.

The view module is responsible for the generation of reports, which display the gathered data like similarity values for signals or signal properties, signal mappings or transformation paths.

[2]https://networkx.github.io/

5. EVALUATION

The described approach has been evaluated on preselected software components currently under development at FEV GmbH. The preselection of software components has been performed based on available extrinsical matches. The available components with multiple extrinsical matches have already been identified in preliminary work [24]. The evaluation focus on measuring the precision of the described approach. Therefore, each derived mapping is evaluated by experts to identify all false positives (wrong signal matches) and false negatives (missed signal matches). In consequence, to be able to perform a qualified evaluation in a suitable time frame the amount of derived signal matches has been reduced to an adequate size. From the preselected components, 6 components with together 15 variants have been chosen, resulting in 721 signal matches (including unmatched signals).As already indicated by the simplified example the amount of available attributes in relation to the overall defined attribute seems to influence the correctness of the derived signal mapping. Therefore, during the evaluation for each signal mapping the minimum data accuracy of the involved signals has been measured to identify a possible correlation between the approach's precision and the degree of available information. Considering the name property as an information which is always provided and the min and max value of the range attribute as two separately defined attributes, the data accuracy of a signal represents the percentage of defined attributes *width*, *min*, *max*, *data type* and *unit*. In consequence, boolean signal's highest data accuracy value is $0,4$ (no unit and no range), while integers reach only $0,8$ (suggesting no unit is defined). In addition, a threshold on the name property could decrease the amount of false positives, but maybe increase the amount of false negatives at the same time. Therefore different thresholds for the name property have been evaluated to check if such a threshold is a proper mechanism to handle low data quality. All properties are considered with the same weight of 1 during the evaluation.

The results of the evaluation considering the data accuracy of the provided data set and the precision (percentage of correct matches) of the proposed approach are shown in Fig. 10. A calculated Spearmen correlation coefficient of $0,23$ and a p-value below $0,001$ indicate a relation between data accuracy and precision. Analyzing the provided data, attribute definition occurrences are provided in the following order: name, width, data type, unit and range. In most cases the definition of a physical unit is enough to achieve a correct signal matching, while the consideration of name, width and data type only results in a failure rate around 50 %. In Fig. 11 the precision of the derived matches is shown in relation to the applied name similarity threshold.

A first significant increase of precision is measured by a threshold value of 0.5, while the peak of 95% for the evaluated data set is achieved with thresholds 0.78, 0.79 and 0.8. Starting with threshold 0.81 the validity is decreasing again. Nevertheless, a threshold of 1.0 still provides a precision of 87%. Analysing the concrete amount of false positives (incorrect matches) and false negatives (missing matches) an increase of false negatives is introduces at threshold 0.65 (1 false negative), as can be seen in Fig. 12. A more or less even relation between false positives and false negatives is provided in the same area where the highest overall precision is given: between thresholds of 0.77 and $0,78$ the amount

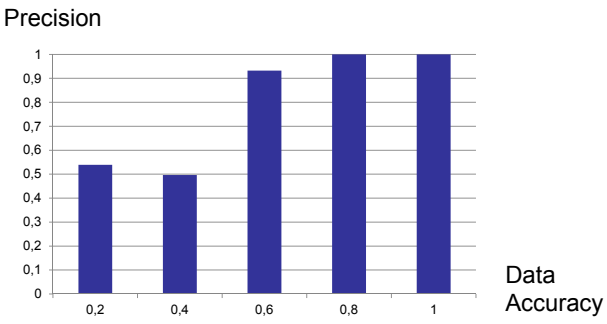

Figure 10: Precision in relation to data accuracy.

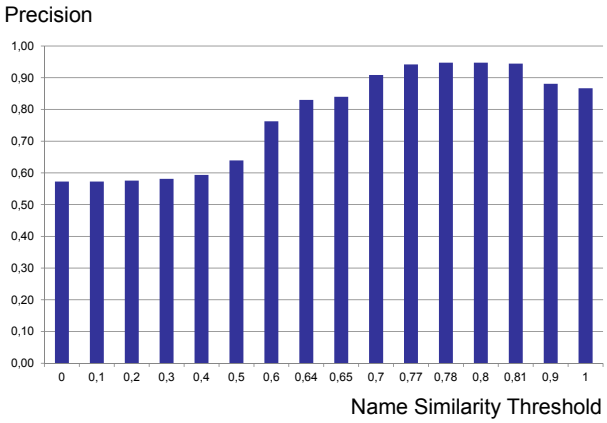

Figure 11: Precision for different name similarity thresholds.

of false negatives becomes higher than the number of false positives. If only a general similarity value needs to be extracted to identify potential candidates for reuse the overall precision is important, while the origin of the inaccuracy is secondary. If in a next step, a generic interface shall be extracted and further evaluation consider semantical similarity based on the extracted matches, false positives are less problematic than false negatives. False positives can be identified either manually or with additional automated semantical analysis, while false negatives are much harder to detect afterwards. In addition, all false positives extracted by a threshold higher than 0.65 have not been matched with the wrong signal, but would not match with any other signal at all. In consequence, for the given data set all valid matches are detected if no false negative is provided. Therefore, a different weighting of the properties during the similarity analysis should not provide better results for the given data set.

The main reason for the strong relation between precision and the name similarity threshold in the context of low data accuracy is the consequent application of a signal naming convention within PERSIST[29, 20]. This convention provides additional standardized semantical information based on abbreviated keywords of the automotive domain. Consequently, correct matches with no name similarity, as illustrated in Fig. 3, can occur hardly. Although the results are quite promising they can not be generalized. The examined components are all developed in the context of the FEV GmbH, specified by specific teams and relate to specific sub

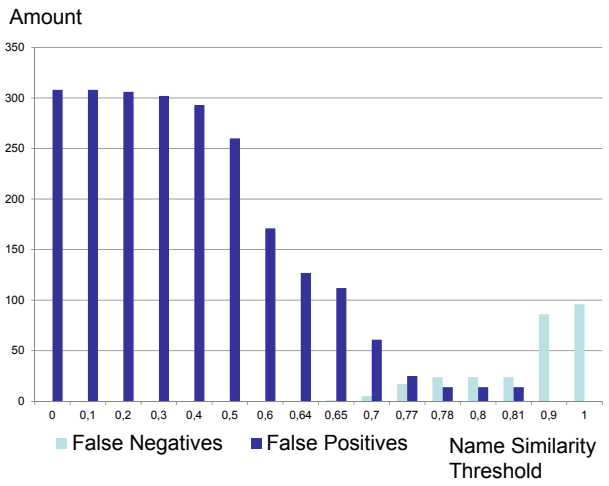

Figure 12: Amount of false positives and false negatives in relation to applied name similarity threshold.

domains. Therefore, further investigations are necessary to consolidate the observed.

Based on the extracted matches arborescences are calculated to extract common interfaces. In only two cases several signals of the same interface are included in one transformation as illustrated in Fig. 9. In these cases the corresponding paths targeted false positive signal matches. Reports are generated which illustrate either an overall interface suitable for all variants or which define the generic core: an interface consisting of a signals shared by all considered variants. As long as the derived signal matches are correct, the extracted transformation pathes represent a valid generic interface. The measured average structural similarity between different variants of a software component is 29%, while the specific similarity for all variants of one software component reach between 11% and 67%. Therefore, several indicator for the implementation of general components, as requested in [24], are provided automatically. In addition, corresponding interfaces are derived, too. Besides the identification of similarities between different variants the described signal matching is also used to extract similarities between two version of the same interface. Thereby, a change analysis can be performed as shown in Tab. 1 to be able to monitor the evolution of a software product line and its components. The example is directly taken from an evaluated component, but revision numbers, history length and comments are abstracted. Column two and three indicate the average similarity from the older to the newer revision (↑) or the other way around (↓). The average similarity is calculated by the average similarity of all found matches from one interface to another (see $sim_{dir}(I_1, I_2)$ in Sect. 3). If all signals of the older version are matched by equal signals of the newer interface the directed average similarity is 1.0, even some new signals of the newer interface are not matched at all. Comparing these directed similarity values in the version history shown in Tab. 1 it can directly be seen that only between version one and version two small modifications of already provided signals are performed. In all other cases only additional signals or parameters are defined. The changes between revision 2/3, 3/4 and 6/7 are highlighted as major changes. During the evaluation the history of 7 components

Rev.	↑	↓	Comment	IN	OUT	CAL
8	1.0	1.0	description	29	25	28
7	1.0	0.84	branch merge	29	25	28
6	1.0	0.98	new signal	21	23	25
5	1.0	1.0	description	20	23	25
4	1.0	0.911	new signals	20	23	25
3	1.0	0.85	new signals	20	17	25
2	0.99	0.99	typos	19	17	17
1	-	-	creation	19	17	17

Table 1: Revision history including directed similarity relations between different versions.

have been analyzed in detail. In all cases the extracted similarity relations expressed the applied changes, although without any name similarity threshold applied. This observation can be explained by the stronger cohesion between elements of the same history, but also by the kind of changes applied: in no case a wild mix of rename, add and delete signal operations are performed at the same time.

6. RELATED WORK

This section gives a brief overview of related work in the context of software component retrieval, software product line extraction or clone detection.

Chen et al. [7] propose a method to retrieve software components from a component library based on signature matching. A database is used to store a pool of reusable software components. Software components defined in an object-oriented fashion are searched for by constructing database queries that check for signature matches. Mili et al. use specification based matching to retrieve software components from a database [17]. The method distinguishes between exact retrieval, where the retrieved function is expected to fully comply to the given specification, and approximate retrieval, where a function's behaviour is similar to a given specification and can be modified to satisfy it. The retrieval approach for exact retrieval is refinement based, meaning a component is a suitable candidate, if its specification refines the given specification. For approximate retrieval, the goal is to find a specification, that maximizes the information that it has in common with the search argument, and hence minimizes the effort to adjust a component. Similar approaches to preselect software components based on signature matching are defined by Zaremski et al. [31, 32] or Fischer et al. [11]. In addition, different methods and metrics are proposed to retrieve web services based on interface similarities [30, 28].

In the context of software product line extraction Koschke et al. [14] suggest an apporach based on the reflexion method [18]. Thereby the Levenshtein distance is used to derive similarities between functions and global variables. Even the related case study is connected to the automotive industry, no further attributes except data type are considered to identify proper matches. Berger et al. [3] propose a method for the comparison of software components' interfaces. Structural and semantic comparison are used to determine whether two interfaces are equal or similar. Two interfaces are considered to be equal, if their names and signatures perform an exact match. Further semantical comparisons are performed manually. Based on the theory of software product line refinement [5] Rubin and Chechik suggest [22] a general product line refactoring approach for any type of models represented in XMI. Similar to the proposed approach the overall process is divided into a compare, a match and a merge operation step. Nevertheless in a proposed example for UML statecharts only the attributes name, type, depth, actions and transitions are considered and weighted separately. A detailed evaluation is not provided. Ryssel et al. analyze the similarity of Simulink models to extract generic models [26]. Regarding the interface the proposed similarity metric does consider only the average of identical parameters. In addition, due to the calculation of a neighbourhood criteria the port connections between different components are evaluated.

According to Roy et al.[21] metric-based clone detection approaches (e.g. [1, 16] use a technique called fingerprinting functions: different metrics are derived for specific syntactical units to identify clones. Nevertheless these metrics do not compare detailed attributes, but measure general differences regarding the amount of parameter declarations or changed data types.

All mentioned approaches have in common that they are not considering additional signal properties common in the automotive industry, like unit or logical range, to derive adequate signal matches. In addition, even lexical similarity of parameter names or descriptions are evaluated, no naming convention standard is available to significantly increase the precision of the proposed approach.

7. CONCLUSION

This paper describes an approach for the automotive industry to derive a similarity statement between different variants or versions of software component interfaces. By focusing on the additional signal properties provided in this specific domain and applying a name similarity threshold to make use of applied signal naming conventions, a very precise signal matching could be established. The performed evaluation indicate a relation between precision and data accuracy and a relation between precision and applied name similarity threshold. The approach can be used to indicate reuse potentials and to derive a generic interface automatically. In addition, the similarity analysis can be applied to monitor the evolution of software component interfaces. Even the results are quite promising, further investigations are necessary to consolidate the observed and further improve the described method. Considering both observed relations, a dynamic threshold in relation to available data accuracy could further improve the correctness of the results. In addition, further information could be extracted in a grey-box approach to increase available data accuracy: e.g. the cohesion between different signals of one component could be considered due to a control flow analysis to improve the overall precision of the derived signal matches.

8. REFERENCES

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the IEEE Symposium on Software Metrics*, pages 292–303, 1999.

[2] C. Berger, H. Rendel, and B. Rumpe. Measuring the ability to form a product line from existing products. In *Variability Modelling of Software-intensive Systems (VaMos)*, 2010.

[3] C. Berger, H. Rendel, B. Rumpe, C. Busse, T. Jablonski, and F. Wolf. Product line metrics for

legacy software in practice. In *Proceedings of the 14th International Software Product Line Conference (SPLC 2010)*, volume 2, 2010.

[4] B. Bollobás. Handbook of combinatorics (vol. 2). chapter Extremal Graph Theory, pages 1231–1292. MIT Press, Cambridge, MA, USA, 1995.

[5] P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. *Theoretical Computer Science*, 455:2–30, Oct. 2012.

[6] M. Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

[7] P. Chen, C. R. Hennicker, and M. Jarke. On the Retrieval of Reusable Software Components. In *Proceedings of the Second International Workshop on Software Reusability*, pages 99–108. IEEE, 1993.

[8] J. Díaz, J. Pérez, P. P. Alarcón, and J. Garbajosa. Agile product line engineering - a systematic literature review. *Software: Practice and Experience*, 41(8):921–941, July 2011.

[9] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An exploratory study of cloning in industrial software product lines. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, CSMR '13, pages 25–34, Washington, DC, USA, 2013. IEEE Computer Society.

[10] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

[11] B. Fischer, M. Kievernagel, W. Struckmann, et al. *VCR: A VDM-based software component retrieval tool*. Citeseer, 1994.

[12] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 391–400, Washington, DC, USA, 2014. IEEE Computer Society.

[13] J. Kleinberg and É. Tardos. *Algorithm Design*. Pearson Education India, 2006.

[14] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17(4):331–366, 2009.

[15] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics Doklady*, volume 10, page 707, 1966.

[16] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 244–, Washington, DC, USA, 1996. IEEE Computer Society.

[17] R. Mili, A. Mili, and R. T. Mittermeir. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering*, 23(7):445–460, 1997.

[18] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, Apr 2001.

[19] K. Pohl, G. Böckle, and F. J. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, 2005.

[20] J. Richenhagen, H. Venkitachalam, S. Pischinger, and I. A. Schloßer. Persist–a scalable software architecture for the control of diverse automotive hybrid topologies. In *15. Internationales Stuttgarter Symposium*, pages 37–56. Springer, 2015.

[21] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, May 2009.

[22] J. Rubin and M. Chechik. Combining related products into product lines. In *Fundamental Approaches to Software Engineering*, pages 285–300. Springer, 2012.

[23] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: A framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 101–110, New York, NY, USA, 2013. ACM.

[24] B. Rumpe, C. Schulze, J. Richenhagen, and A. Schloßer. Agile Synchronization between a Software Product Line and its Products. In *Informatik 2015*, volume P-246 of *LNI*, pages 1687–1698. Bonner Köllen Verlag, 2015.

[25] B. Rumpe, C. Schulze, M. v. Wenckstern, J. O. Ringert, and P. Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *International Conference on Software Product Line (SPLC)*, pages 141–150, Nashville, Tennessee, 2015. ACM New York.

[26] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic library migration for the generation of hardware-in-the-loop models. *Science of Computer Programming*, 77(2):83 – 95, 2012. Special Issue on Automatic Program Generation for Embedded Systems.

[27] B. N. Taylor and A. Thompson. The international system of units (si). 2001.

[28] O. Tibermacine, C. Tibermacine, and F. Cherif. A practical approach to the measurement of similarity between wsdl-based web services. *Revue des Nouvelles Technologies de l'Information*, pages 03–18, 2014.

[29] H. Venkitachalam, J. Richenhagen, and S. Pischinger. A generic control software architecture for battery management systems. Technical report, SAE Technical Paper, 2015.

[30] Y. Wang and E. Stroulia. Flexible Interface Matching for Web-Service Discovery. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 147–156. IEEE, 2003.

[31] A. M. Zaremski and J. M. Wing. Signature Matching: A Tool For Using Software Libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(2):146–170, 1995.

[32] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):333–369, 1997.