Interface Variability in Family Model Mining

David Wille, Sönke Holthusen, Sandro Schulze and Ina Schaefer
Institute of Software Engineering and Automotive Informatics
Technische Universität Braunschweig, Germany
{d.wille, s.holthusen, sanschul, i.schaefer}@tu-braunschweig.de

ABSTRACT

Model-driven development of software gains more and more importance, especially in domains with high complexity. In order to develop differing but still similar model-based systems, these models are often copied and modified according to the changed requirements. As the variability between these different models is not documented, issues arise during maintenance. For example, applying patches becomes a tedious task because errors have to be fixed in all of the created models and no information about modified and unchanged parts exists. In this paper, we present an approach to analyze related models and determine the variability between them. This analysis provides crucial information about the variability (i.e., changed parts, additional parts, and parts without any modification) between the models in order to create family models. The particular focus is the analysis of models containing components with differing interfaces.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Algorithms, Design

Keywords

Family Model Mining, SPL, Variability, Analysis

1. INTRODUCTION

Model-based languages are often used in domains with high complexity where abstraction is useful to keep track of the development process [3]. Some of the applied languages allow to model continuous data flow between the different components of a system (e.g., between control units). Each of the components consists of a set of atomic function blocks (e.g., mathematical functions as in *MATLAB/Simulink*[1]) defining its functionality. Data is exchanged between the blocks with connections from outports to inports. The inports and outports define the interfaces of the blocks. Furthermore, some of the model-based languages allow to utilize arbitrarily nested hierarchies. Starting at a very basic level, the solution for the given task is decomposed in different hierarchy levels. With each added hierarchy the solution is refined further. Overall, the complexity is reduced, because the problems can be solved in a stepwise manner. A model-based language with the above linguistic elements, commonly used in the automotive domain is *MATLAB/Simulink*. Alternatives are *ASCET*[2] or *SCADE*[3], just to name some examples.

In practice, new variants of models are often created by copying an existing project and modifying it to meet new requirements [3, 4]. This way of reusing existing software is called *clone-and-own*. It includes deletions, additions, and modifications of the system's functionality. With a big number of variants evolving from an existing software system, several problems arise. For example, the maintenance of the variants becomes a difficult and tedious task. Bugfixes may cause increased effort, because errors have to be fixed in each of the variants. Moreover, updating parts of the variants to a new version becomes a problem since these parts could be altered during the initial clone-and-own phase of the project. Consequently, a lot of implementation steps need to be repeated to apply updates. Another issue is the increasing effort needed to test the different variants. For example, parts that are common to all related models have to be tested repeatedly.

Identifying the changed parts between models that evolved using a clone-and-own approach is a possible solution to solve the described problems. This approach is called *clone detection* and realized by different tools such as *CloneDetective* [3] and *ModelCD* [4]. These tools only identify cloned parts of the models. However, we are also interested in characterizing the differences between the models, as we want to analyze the variabilities and store the resulting information in *family models*. Family models allow modeling the variability of different model implementations using attributes for different types of variability. This information can be used later on to generate different model variants from the created family model.

In this paper, we describe a new technique to apply *family model mining* to block-oriented models, which do not use hierarchy (e.g., *MATLAB/Simulink* models without subsys-

[1]http://www.mathworks.de/

[2]http://www.etas.com/

[3]http://www.esterel-technologies.com/

tems). Family model mining not only aims at identifying commonalities of models, but also takes their differences (and *how* they differ) into account. We distinguish between mandatory parts (i.e., parts that were not changed between models), alternative parts (i.e., parts that were changed), and optional parts (i.e., parts that are not contained in each of the compared models). Our particular focus is on comparing model variants with changed interfaces. Approaches by other authors do not provide the information that we need to apply family model mining or do not take changed interfaces into account [3, 4, 7, 5, 6]. These approaches regard models with changed interfaces as models with changed functionality. This assumption is true in many cases, however in some cases a changed interface not necessarily results in changed functionality. For example, adding a logging functionality to a model does not change the results of any calculations, although it might change the interfaces of some blocks. Most of the approaches only allow to identify one aspect of the required information, either commonalities or differences between models, but not both at the same time. The approach by Ryssel et al. [7] allows to identify both information, but does not take changed interfaces into account. Furthermore, this approach is not fully automated as it needs user interaction. Thus, these approaches are not suitable for automated family model mining.

In order to be language-independent, we abstract block-oriented modeling languages, such as *MATLAB/Simulink*, *SCADE* and *ASCET*, to an *architecture description language (ADL)* providing all basic elements used in these languages. These elements include atomic blocks (i.e., components), ports, connectors, and subsystems. Complex elements such as behavioral models or buses are not included in our basic ADL. In the following, we use the introduced terms to describe block-oriented models language-independently.

The remainder of this paper is organized as follows: After an introduction to block-oriented architectures and family models in Section 2, we present a motivating example in Section 3, showing the benefits of our technique. In Section 4 we describe how we solved the problem to compare different models. In Section 5, we explain how we implemented a toolchain for this technique and in Section 6, we review related work. In Section 7 we summarize this paper with an outlook to future work.

2. BACKGROUND

In this section, we introduce block-oriented architecture descriptions and how they can be used to represent block-oriented modeling languages. Moreover, we present an XML-based architecture description language and reason why this solution allows us to reduce the overall complexity when comparing different block-oriented models. Furthermore, we explain family models and how they represent implementation-specific variability.

2.1 Block-oriented Architectures

Block-oriented architectures provide mechanisms to structure complex problems (e.g., data flows between different control units) with hierarchies. These *architecture description languages (ADLs)* consist of atomic *blocks*, which realize a certain functionality. The data flow between these blocks is modeled using *connectors* between the in- and outports of the blocks. Each of these *ports* is a connection point accepting only one connector at a time. In order to realize arbitrarily nested hierarchies, these ADLs allow to add blocks to other blocks.

Block-oriented modeling languages, such as *MATLAB/Simulink*, *ASCET* or *SCADE*, use the same basic elements (i.e., blocks, ports, and connectors) as the above mentioned ADLs. Consequently, these block-oriented models can be translated to ADLs, in order to have an abstract model without information that is structurally irrelevant (e.g., block colors or the block's position). These reduced representations of the models have a smaller set of elements and thus simplify any analysis.

We designed an ADL as introduced by Dashofy et al. [2]. We decided to use this XML-based approach to describe our problem domain because XML-based ADLs have well developed tool support. This advantage pays off during the implementation, because models can easily be imported into our toolchain using a simple XML-parser.

In Listing 1, we present the architecture description of Figure 1. This small example shows, all basic elements of our ADL. As we can see, models have a name (e.g., "Example for the ADL") and a number of subsystems (e.g., "Component"), which again can contain other subsystems. This way the hierarchy of block-oriented models can be mapped to our ADL. Each of the subsystems has a name and contains components (e.g., component `C0` and `C1`), implementing a part of the model's functionality. Each component has a unique id, a name, and a function (e.g., `Gain` or `Sum`). Each of the connectors has a unique id, a name, a datatype (e.g., integer), a source and a destination. The sources and destinations are defined by the component's unique id and one of it's port numbers. The components' ports can hold only one connector at a time, which is realized by using a composed key of a unique component id and a port number. By only using each of these composed keys once, we ensure that each port only accepts one incoming or outgoing connector. Adding the port numbers only to the connectors and not to the components, reduces redundant information.

```
<?xml version="1.0" encoding="utf-8"?>
<model>
    <name>Example for the ADL</name>
    <subsystem id="0">
        <name>Component</name>
        <component id="0">
            <name>C0</name>
            <function>Gain</function>
        </component>
        <component id="1">
            <name>C1</name>
            <function>Sum</function>
        </component>
    </subsystem>
    <connector id="0">
        <name>out</name>
        <datatype>integer</datatype>
        <source>
            <component_id>0</component_id>
            <port_id>0</port_id>
        </source>
        <destination>
            <component_id>1</component_id>
            <port_id>0</port_id>
        </destination>
    </connector>
</model>
```

Listing 1: A model in our block-oriented ADL

2.2 Family Models

Feature models [1] are used to hierarchically model the *problem domain* of a software family with all commonalities (i.e., mandatory parts) and variabilities (i.e., alternative and optional parts), without giving information about the concrete implementation. In contrast, *family models* allow to

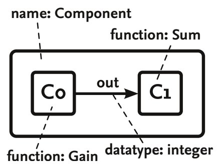


Figure 1: Model "Example for the ADL"

define the *solution space*, that is, the concrete design and the implementation of the software family. They store all possible configurations of a product and give a complete overview about the implementation-specific variability in a software family. Feature models and family models are both part of the *pure::variants*[4] framework for developing and managing software product lines (SPLs). The pure::variants software is a plugin created for *Eclipse*[5] and allows to create both model types using its API.

In Figure 2, we present a small example for a family model. It consists of two mandatory elements `C0` and `C1` (indicated by the "!"), an optional element `C4_m2` (indicated by the "?") and a mandatory VariantSubsystem containing the two alternatives `C2_m1` and `C3_m2` (indicated by the "⇔").

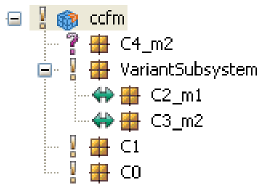


Figure 2: Example for a family model

These different elements can contain further implementation-specific information (e.g., the block type for block-oriented languages) and consequently show how a specific problem could be solved. By selecting and deselecting any of the non-mandatory elements, it is possible to create different variants of a product.

3. MOTIVATING EXAMPLE

Deciding whether two (partial) models are similar enough in order to be interchangeable is usually easy when these models have the same interfaces (i.e., the same number of in- and outports). In Figure 3, we show an example consisting of two models that could be exchanged because both have the same interface (one inport and one outport). The only difference between these models are the two blocks `C0` (`Delay`) and `C1` (`Integrator`).

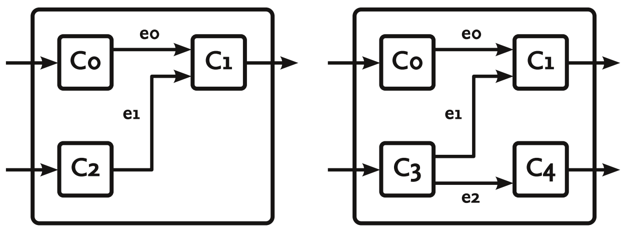
However, there are certain cases where making this decision is non-trivial. For example, some subsystems are different according to their interfaces, but their main functionality is basically the same. In Figure 4, we show an example with two models, differing in their number of outports (one outport vs. two outports) and components (three components vs. four components). Assuming component `C0` and `C1` are the same in both models and the only difference between component `C2` and `C3` is the additional outport, we argue that these models are similar. Three out of four components have the same functionality and only the additional component `C4` differentiates the models. Hence, the two models basically realize the same functionality. A possible use case for this example may be an additional value that needs to be calculated in component `C4` or a logging functionality realized by component `C4`. Consequently, the two models are fairly similar although their interfaces differ.

[4]http://www.pure-systems.com/
[5]http://www.eclipse.org/

Delay
Co
Integrator
C1

(a) `Model 1` (b) `Model 2`

Figure 3: Interchangeable models



(a) Model 1 (b) Model 2

Figure 4: Motivating example

We need to find an approach, which allows to compare different models stored in our XML-based ADL and generate family models based on the results. The approach should analyze the models by comparing all their elements (i.e., components and connectors) and determine their commonalities (i.e., mandatory parts) and differences (i.e., alternative and optional parts). The results should be represented by a family model. Furthermore, the approach should pay attention to the models' interfaces, in order to allow us to apply family model mining on models, which realize a similar functionality, but have differing interfaces.

4. DETERMINE MODEL VARIABILITY

In order to compare two models with each other, a metric is needed to calculate the overall similarity value of the compared models. This metric weights the different parts of the models according to their importance for the functionality. But most importantly, the metric is utilized to get the required information to generate family models. In this section we explain our metric and how we apply it to compare models with varied interfaces to create family models.

4.1 Measuring Model Similarity

In order to ease the comparison of two models m_0 and m_1 and to calculate their similarity values, we first define the overall structure of the considered models:

Definition 1. *A model M = (name, S, CP, CN) is defined by a name, a set of subsystems S, a set of components CP, and a set of connectors CN (including the connectors of the subsystems). Each subsystem* $S_i = (name_i, S_s, CP_s) \in S$ *has a name and consists of other subsystems* $S_s \subseteq S$ *and components* $CP_s \subseteq CP$. *A component* $CP_i = (name_i, function_i, I_i, O_i) \in CP$ *has a name, a function (e.g., sum or gain), and two sets* I_i *and* O_i*, which define the component's in- and outports. A connector* $CN_i = (name_i, datatype_i, src_i, dst_i) \in CN$ *is defined by its name, datatype (e.g. integer) and* $src, dst \in CP$*, where src and dst are the components that are connected by the connector.*

$$\begin{aligned}\text{similarity} &= \frac{2}{3} \cdot \text{similarity}_{components} \\ &+ \frac{1}{3} \cdot \text{similarity}_{connectors}\end{aligned} \tag{1}$$

$$\begin{aligned}\text{similarity}\left(cp_0,\ cp_1\right) &= 0.05 \cdot name_{eq} + 0.75 \cdot fnc_{eq} \\ &+ 0.1 \cdot \frac{\#\text{similar inports}}{\max\left(|I_0|, |I_1|\right)} \\ &+ 0.1 \cdot \frac{\#\text{similar outports}}{\max\left(|O_0|, |O_1|\right)}, \\ &\text{where } name_{eq}, fnc_{eq} \in \{0,\ 1\}\end{aligned} \tag{2}$$

$$\begin{aligned}\text{similarity}\left(cn_0,\ cn_1\right) &= 0.05 \cdot name_{eq} + 0.75 \cdot fnc_{eq} \\ &+ 0.1 \cdot \text{similarity}\left(src_0, src_1\right) \\ &+ 0.1 \cdot \text{similarity}\left(dst_0, dst_1\right), \\ &\text{where } name_{eq}, fnc_{eq} \in \{0,\ 1\}\end{aligned} \tag{3}$$

For developing the metric, we analyzed, which parts of a model primarily contribute to its functionality. Models consist of components and connectors, so we only have to distinguish between these two elements. Since subsystems are only used for structural purposes (e.g., to use hierarchies or to allow reuse of a certain functionality) and do not influence the functionality directly, we do not need to take them into account separately. The components define what functions are applied and consequently, how data is processed. In contrast, connectors only can exchange data between the components, but still affect the functionality, because additional connectors add new data flow and can change the way data is processed. As a result, we weight the importance of components to connectors with a ratio of 2:1.

In order to determine the similarity of components with varying interfaces, we developed a metric, which computes a numeric value for predicting whether two models are similar. In Equation (1) we present the resulting equation used to calculate the similarity of two compared models, taking the previous considerations into account. Basically, it consists of the weighted similarity for the components and connectors.

For determining the similarity between components, we analyze, which parts define their functionality. According to Definition 1, a component consists of a *name*, a defined *function*, and in- and outports. As the *function* obviously defines how data is processed, it has a high impact on the functionality. In contrast, the *name* does not affect the functionality at all. Additionally, it does not need to be unique, and two components may have the same *name* although they have different *functions*. The in- and outports have a higher impact on the similarity of components than the *name*, because they define how data is exchanged with other components. However, they do not directly change the functionality of the components, when they are added or removed. When analyzing the ports of two components, it is important not only to check if the interfaces are the same, but to determine their similarity by calculating the ratio of equal ports to the overall number of ports.

We take all the considerations above into account, when creating Equation (2) to calculate the similarity of two components $cp_0 = (name_0, function_0, I_0, O_0) \in CP$ and $cp_1 = (name_1, function_1, I_1, O_1) \in CP$ from model m_0 and m_1, respectively. Because of its high importance for the component's functionality, the *function* is weighted with 75%. Inports and outports are weighted with 10% each and the *name* is only considered with 5%. The similarity of the *function* and the *name* are only added to the similarity if they are equal, as we cannot calculate partial similarity for these parameters. This is represented in the equation by $name_{eq} \in \{0, 1\}$ and $fnc_{eq} \in \{0, 1\}$ (not equal or equal).

Attribute	Percentage	Multiplier
name	5%	$\{0, 1\}$
function	75%	$\{0, 1\}$
inports	10%	$\frac{\#\text{similar inports}}{\max(\|I_0\|, \|I_1\|)}$
outports	10%	$\frac{\#\text{similar outports}}{\max(\|O_0\|, \|O_1\|)}$

Table 1: Metric for components

In order to calculate the ratio of equal inports to the overall number of inports, we determine the ratio between the number of similar inports and the maximum number of inports (i.e., $\max\left(|I_0|, |I_1|\right)$). Two inports in component cp_0 and cp_1 are considered similar if the *functions* of the components connected to these ports are similar, as it is the most important information about neighboring components. As the order of the connected components could have changed because of a different processing order when storing the models to a file, we keep track of all ports that were not matched to ports from the other model and create all possible permutations of them. The permutations are used to compare these ports and reduce the number of unmatched ports (i.e. ports that have not been matched to a port from the other model). After comparing all inports, we use the resulting ratio in the above mentioned equation. The same ideas apply when calculating the similarity for the outports, except that we have to compare the destinations instead of the sources. We summarized all the aforementioned parameters for the similarity of components in Table 1.

Finally, we have to compute the similarity of connectors. Similarly to components, connectors consists of different attributes that differ in their importance. When looking at Definition 1, we see that a connector is defined by its *name*, *datatype* and its source and destination component. The *datatype* is the only part of a connector that influences its functionality directly, because it defines what kind of data can be exchanged with the corresponding connection. The *name* is a weak measure to compare connectors, since it does not need to be unique. The source and destination component need to be considered with a higher weight, since they indirectly influence the overall functionality of the model. But as they are already considered when comparing two components, they should not get a very high weight. Based on these considerations, we suggest the formula in Equation (3) to compare two connectors $cn_0 = (name_0, datatype_0, src_0, dst_0) \in CN$ and $cn_1 = (name_1, datatype_1, src_1, dst_1) \in CN$ from model m_0 and m_1, respectively.

As the connector's *datatype* has a high impact on its functionality, we weight it with 75%. The *name* has the lowest impact and thus is considered with only 5%. Both values are only added to the similarity if they are equal, because partial similarities cannot be calculated. This is represented in the equation by $name_{eq} \in \{0, 1\}$ and $fnc_{eq} \in \{0, 1\}$ (not equal or equal). The similarities of the source and destination components are weighted with 10% each, whereby we use

Attribute	Percentage	Multiplier
name	5%	$\{0, 1\}$
datatype	75%	$\{0, 1\}$
src	10%	Similarity of the sources
dst	10%	Similarity of the destinations

Table 2: Metric for connectors

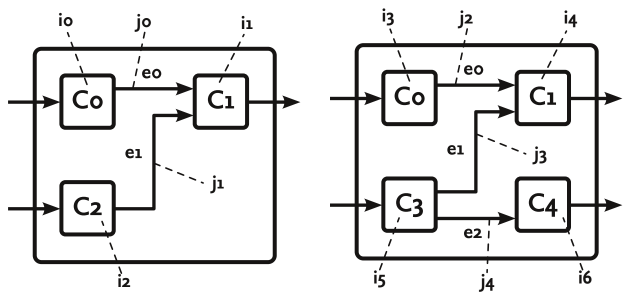
the similarity of src_0 compared to src_1 and dst_0 compared to dst_1 as factors. Consequently, it is useful to calculate the similarity of all components before starting to compare the connectors. We summarized all the above mentioned parts of the equation for the similarity of connectors in Table 2.

The presented weights might have to be fine-tuned for some applications, but they address all relevant parts of a model regarding the similarity of their functionality.

4.2 Model Comparison

We now show, how the metric is used to compare two models in order to find a matching between them, which is used to generate a family model. In Figure 5, we present two models, which we use as an example in this section. When parsing the models for the comparison, we assign a unique identifier to each component and connector. We depict these identifiers for our exemplary models by the dashed lines and the connected identifiers (i.e., `i0`, `i1`, ..., `i6` and `j0`, `j1`, ..., `j4`, respectively).

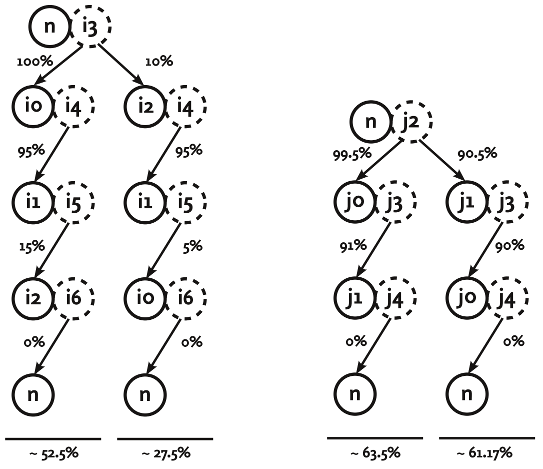
After labeling both models, we select the model with the highest number of components as our base model (i.e., in our example `Model 2`). This way, we increase the chance to match all components from the base model to components in the compared model. Besides, we assure that potential optional components are not ignored, which could be possible when selecting the model with the lowest number. The next step is to create two lists with the *start components* of both models (i.e., `i0`, `i2` and `i3`, `i5`). These start components are the first components after the models' inports and provide initial access to the model. Thus, we can guarantee that we walk through the whole data-flow when selecting them as a start for our comparison. Next, we select one of the start components from the base model (e.g., component `i3`) and compare it with all other start components from the other model (i.e., component `i0` and `i2`).



(a) `Model 1` (b) `Model 2` (base model)

Figure 5: Example models used to explain the approach

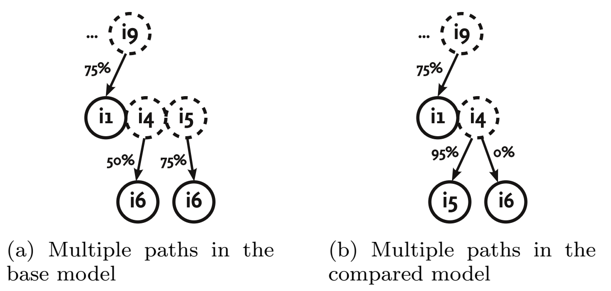
In Figure 6a, we show how the comparisons for components are documented in a *compare tree*. A compare tree is a special data structure, which we use to represent the comparisons and store the calculated similarity values. Each compare tree can contain different paths, which represent different matchings. The nodes of the tree are linked with directed edges, whereas the edges always represent exactly one comparison of a component from the base model (i.e., in our example `Model 2`) with a component from the compared model (i.e., in our example `Model 1`) and store their similarity. A node consists of multiple circles. Dashed circles represent components from the base model, whereas solid circles represent components from the compared model. The node's solid circles are always part of the previous comparison $n-1$ and the connected dashed circles represent the components of the next comparison n (e.g., in the left branch in Figure 6a, component `i0` is part of a different comparison than component `i4`). As the node's solid circle is always part of the last comparison, the root node's solid circle is `null`, because there cannot be any previous comparisons. A comparison of two models can generate multiple trees, because in certain cases (explained further below) a generated tree does not necessarily contain the best matching between two models.



(a) Compare tree for the components (b) Compare tree for the connectors

Figure 6: Examples for the compare trees

In order to select the components for the next comparison, we take a look at the last compared component from the base model and analyze its in- and outgoing connections. Every linked component is a possible candidate for a new comparison. For example, after starting in component `i3` we are able to reach component `i4` via connector `j2`. The same analysis is applied for the last component from the compared model. This way, we get possible candidates for the comparison with the components from the base model. For example, we can reach component `i1` via the connection `j0` from component `i0`. With the newly discovered components, we can create a new comparison (e.g., component `i4` is compared with `i1`). The newly discovered components are only candidates, as they could already be used in a previous comparison. For example, the analysis of possible candidates in component `i4` returns the components `i3` and `i5`. As component `i3` is already used in a previous comparison, we do not take it into account during following comparisons. When comparing models with a different number of components it is possible, that there are no new components discovered in the



(a) Multiple paths in the base model

(b) Multiple paths in the compared model

Figure 7: Examples for possible branches

compared model. Consequently, the remaining components from the base model are compared with `null`, resulting in a similarity of 0%.

Generally, there could be more than one component linked with the analyzed components. Hence, there are two possible types of branches in compare trees. If there is more than one possible base model component for the next comparison, we denote the results as shown in Figure 7a. For each of the discovered components a new dashed circle is created and each of the discovered components from the other model is compared with them (cf. the comparisons `i4` ⇔ `i6` and `i5` ⇔ `i6` in Figure 7a). Likewise, there could be more than one candidate component in the compared model. In this case, we connect all of them to the compared base model component (cf. the comparisons `i4` ⇔ `i5` and `i4` ⇔ `i6` in Figure 7b). This type of branch is often used, if there is more than one start component in the compared model (cf. the comparisons `i3` ⇔ `i0` and `i3` ⇔ `i2` in Figure 6a). Of course the two types of branches can be combined, as there could be more than one candidate in both models at the same time.

All steps are repeated until every newly discovered component is already contained in each of the tree's branches. If the termination condition is fulfilled, but not every of the base model's start components was visited, we have to create another compare tree, as the other start components might provide a better matching. We also have to do this, if one of the start components is already included in the compare tree, but was compared with `null` (e.g., if the two models have a different number of components).

The described technique is also used to compare the connectors, where we use the metric for connectors. In this case the connectors, which are connected to the previously mentioned start components, represent the *start connectors*. For example, in Figure 5b the connector `j2` is connected to start component `i3`. Consequently, the list of all start connectors includes the connectors `j0` and `j1` for `Model 1` and the connectors `j2`, `j3`, and `j4` for `Model 2`. The candidates for the next comparisons are the connectors linked with the components, which could be visited with the connectors from the previous comparison. The connector compare tree for the two models in Figure 5 is presented in Figure 6b.

In order to calculate the complete similarity value of the two compared models, we start at the root node of a compare tree and walk through every branch, sum up the calculated similarity values and divide them by the number of components contained in the base model. Of course, we have to take all created compare trees into account when calculating the overall similarity, as the different trees represent different matchings. The branch with the highest similarity represents the best matching between the two models. The highest values for the components and connectors can be used afterwards to calculate the overall similarity of the two models according to Equation (1). The overall similarity of the components and connectors in Figure 5 is 52.5% and 63.5%, respectively. As a result, the overall similarity of the models, calculated using Equation (1), is 56.2%. Although, the overall similarity value is not directly used to generate family models, we have to find the best matching between the models, because it allows us to generate the family model for the comparison. We have to consider special branches, such as in Figure 7a and Figure 7b, when calculating the overall similarity. During the calculation of the overall similarity value, each of these branches results in a separate similarity value.

The approach explained in this section always returns a matching between models, because block-oriented models must have start components to get inputs for their calculation. Consequently, our approach is applicable to all well-formed block-oriented models with respect to the considered set of modeling elements (i.e., as mentioned before, we do not consider buses or behavioral models). Furthermore, the approach always stops at some point, because block-oriented models cannot have an infinite number of components. Thus, every component has to be included in every branch of the compare tree, which results in the termination of the comparison.

4.3 Creating the Family Model

After calculating the best matching, we use the obtained information to determine, which components of the compared models are mandatory, alternative, or optional. The recognized variability is exported to a family model by walking through the best matching for the components and analyzing its comparisons. As mandatory components cannot change between compared models and the *name* (i.e., 5% of the similarity value) is the only parameter that does not effect their functionality, we define that we add a mandatory element to our family model for comparisons with a similarity $\geq 95\%$ (e.g., for the comparison of component `i4` with component `i1`). For all other components with a similarity between $> 0\%$ and $< 95\%$ we add alternative elements to the family model (e.g., for the comparison of component `i5` with component `i2`), because they are not similar enough to be mandatory. Alternative components are stored in *VariantSubsystems*, which are subsystems containing the recognized variability. Components compared with `null` and a resulting similarity of 0% (e.g., component `i6`) are considered to be optional elements in the family model, because the comparison with `null` implies that it is an additional component. For these alternative and optional features, we have to annotate their respective source model (e.g., shown above `_m1` and `_m2`) in order to distinguish them. In Figure 2, we present the resulting family model for our example. The best matching of the connectors is not considered when creating the family models as it is only used to calculate the overall similarity of the models.

According to Section 3, we could consider components with the same *function* but changed interfaces as equal, because their functionality is not changed. Given that the blocks' *names* can also change, we would regard blocks with a similarity $\geq 75\%$ as mandatory. Nevertheless, we choose $< 95\%$ for the upper threshold of alternative blocks, because we have to add the changed interfaces to the family model. Otherwise, we would loose the information about the changed interface, when merging two blocks with changed interfaces into one mandatory block. For example, in Figure 5 we would loose the information about the changed interface of the alternative components C2 and C3. Thus, we would not be able to generate all variants of the software family, because only one interface type would be known.

4.4 Matching Multiple Models

The technique described above is also applicable to compare more than two models. In this case, we have to define a base model and compare all other models with it. The created compare trees can be used to merge the components into a family model. This is done by merging the base model with the first compared model and afterwards merging all other models step by step into the generated family model. In Figure 8 we show an example with three models. On the left there is the compare tree for `Model 2` compared to `Model 1` and on the right the compare tree for `Model 3` compared to `Model 1`, respectively. Comparing `Model 1` with `Model 2` results in a family model with a mandatory component `i1` (the same component as `i3`) and an optional component `i2`. After merging `Model 3` into the previously merged models, the final family model in Figure 9 has an alternative between component `i1` and `i4`, whereby component `i1` is the same component as component `i3` from `Model 2`. Besides, component `i5` is an alternative to component `i2`. Consequently, both components are an optional alternative, because component `i2` was compared to `null` in the compare tree for `Model 2` and thus is optional.

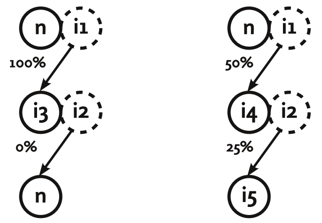


Figure 8: Comparison of `Model 1` with `Model 2/Model 3`

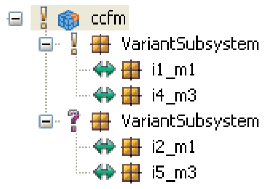


Figure 9: Example family model for multiple merged models

5. IMPLEMENTATION

We implemented our approach to show its feasibility. In Figure 10, we present the toolchain, together with the basic workflow, which we created for our approach. The toolchain is *Eclipse*-based and consists of multiple plugins. The first plugin parses the architecture descriptions stored in XML-files. These files can be created by parsing all necessary information (i.e., all elements mentioned in Definition 1) from a block-oriented model (e.g., a *MATLAB/Simulink* model) and storing it to our XML-based ADL.

When running our toolchain, the information parsed from such an XML-file is transferred into an internal representation using an EMF-Meta-Model developed for our architecture format. This EMF-Meta-Model allows us to create

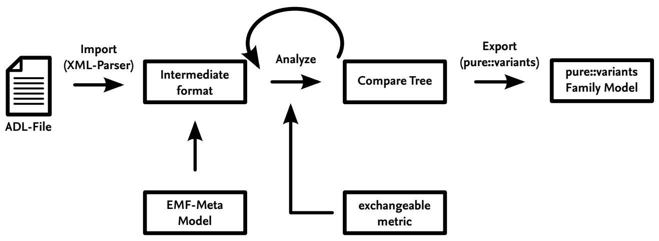


Figure 10: The basic workflow

object-oriented representations of ADLs and work on these objects in order to apply our approach. The model is analyzed by applying the previously described technique and the metric explained in Section 4. Our approach and the described toolchain allow to exchange our metric with any other metric, for example, to pay more attention to other parts in the models. After creating the compare tree for the compared models, the exporter plugin is used to export a family model of the found variability to *pure::variants*.

6. RELATED WORK

There are several approaches to detect clones between different models. Deissenboeck et al. [3] developed an approach, which is used in the *CloneDetective* tool to detect clones in *MATLAB/Simulink* models. First, the model is "flattened" by moving all blocks from subhierarchies to the highest level. Afterwards, the model's blocks and connectors are labeled with a special labeling function, which adds all relevant information for the detection (e.g., name and function) to the labels and drops irrelevant information (e.g., color and position). This labeled model is transferred to a graph representation, whose nodes and edges represent blocks and connectors. All nodes having the same label, but representing different nodes are compared using a breadth-first search. The order is defined by the node's neighborhood. In order to find a clone, the neighboring nodes and the connecting edges have to use the same label. The *CloneDetective* approach is integrated into the ConQAT tool[6].

Pham et al. [4] use a similar approach for their *ModelCD* tool, which is based on labeled graphs and consists of two algorithms *eScan* and *aScan*. The *eScan* algorithm is similar to the approach by Deissenboeck et al. and detects direct clones. The *aScan* algorithm detects slightly varied nodes (e.g., nodes that were copied and modified afterwards). Both approaches only detect commonalities between models and lack information about the differences. Hence, in contrast to our approach they cannot predict if the blocks are mandatory, alternative or optional. Consequently, these approaches are not applicable for family model mining.

Furthermore, there are tools to detect differences between models. These tools are mostly commercial such as SiDiff [7] and SimDiff [8]. In contrast to the clone detection tools, they only allow to analyze the differences and consequently are not suitable for family model mining either, because no information about possible commonalities is provided.

Ryssel et al. [7] describe an approach fairly close to ours, but as the authors follow a different goal some differences are recognizable. The approach aims at creating a library of varying components (i.e., subsystems with variant-specific

[6]https://www.conqat.org
[7]http://pi.informatik.uni-siegen.de/Projekte/sidiff/
[8]http://www.ensoftcorp.com/simdiff/

blocks), in order to ease reuse in different projects. The usability of these libraries often is hampered by a growing number of variants as developers have to choose the right components from this pool. Ryssel et al. try to overcome this complexity by creating *implementation component configuration language (ICCL)* files. These special XML files store variants of the same model. Before these ICCL files can be created the variations between two models have to be analyzed. For this analysis, two similarity criteria are considered. First, the *local criteria* of compared components are used to calculate their similarity. Thereby, the types, names, structural parameters (e.g., the interface) and behavioral parameters are considered. Similar to our approach, Ryssel et al. regard names as a weak criterion for similarity. In contrast to our metric, they do not compare blocks with differing function types to reduce the number of comparisons, as these blocks are not relevant for their approach. Overall, Ryssel et al. consider structural parameters (e.g., the number of in- and outports) to have a high influence on the interaction and the semantics of components. Second, the *neighborhood criteria* are used to consider the neighborhood of the compared components such as changes of the source and destination ports. In order to identify the nearest neighbors, the compared models are transferred to directed graphs. The above mentioned criteria are used to calculate all local similarities by comparing components of a certain type in the first graph with all components of the same type in the other graph. These graphs are clustered by merging them if they are close enough according to the metric. The created clusters can be transferred to an ICCL and allow to store the found variation points language-independently. In order to review the generated clusters, this approach needs user interaction at the end of the algorithm. Consequently, it is not suitable in our case, as we try to automatically compare models, find the variability, and create a family model for the compared models.

Based on this work, Ryssel et al. [5] describe how *variation points* (i.e., points with differences between models) can be found in a set of models. Furthermore, the authors determine dependencies between these variations points and create feature models with the gathered information. A third article by Ryssel et al. [6] shows how feature models can be created with tables, containing information about the variation points in models. Both approaches do not take interface similarity into account and thus are not applicable for family model mining of model variants with varying interfaces.

Besides, several approaches exist to generate feature models from development artifacts. For example, Weston et al. [9] process natural-language requirement documents into candidate feature models. Zhang et al. [10] use EMF Compare to determine the variability between different models and transfer them to the Common Variability Language (CVL) in order to express feature-based variability. She et al. [8] describe an approach for reverse engineering feature models from product maps. In contrast, we generate family models by comparing model variants.

7. CONCLUSION

In this paper, we developed a technique, which automatically analyzes different models with varied interfaces. These models are stored in a special ADL following the approach by Dashofy et al. [2]. In order to get a better understanding of the similarity of the compared models, a numeric value representing the overall similarity is computed according to the proposed similarity metric. After generating a compare tree between two model variants, we can use the most similar matching of the components to export a family model. A family model stores the common, alternative and optional parts of the compared model variants.

We argue that our approach to compare models could be applied for different block-oriented modeling languages, such as *MATLAB/Simulink*, *SCADE* and *ASCET*, by developing an importer plugin for these modeling languages converting them into the EMF-Meta-Model for our ADL. The imported information could be used in our toolchain to analyze the models and create the corresponding family models.

In future work, we want to evaluate our approach with realistic industrial-scale models (e.g., *MATLAB/Simulink* models). So far, we created artificial models and compared the results of our similarity metric with expectations of people with experience in model-driven development. This allowed to see that our approach generates similarity values, which are close to the expectations. As we only considered models without hierarchies, we want to analyze the effects of hierarchical elements, such as subsystems, and extend our approach in order to be able to perform a case study with realistic models. Besides, we argue that our approach could easily be modified to create feature models instead of family models.

8. REFERENCES

[1] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications.* Addison-Wesley, 2000.

[2] E. Dashofy, A. van der Hoek, and R. Taylor. A comprehensive approach for the development of modular software architecture description languages. *TOSEM*, 14(2):199–245, 2005.

[3] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE '08*, pages 603–612, 2008.

[4] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, and T. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE '09*, pages 276–286, 2009.

[5] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic variation-point identification in function-block-based models. In *GPCE '10*, pages 23–32, 2010.

[6] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Extraction of feature models from formal contexts. In *SPLC '11*, pages 4:1–4:8, 2011.

[7] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic library migration for the generation of hardware-in-the-loop models. *SCP*, 77(2):83–95, 2012.

[8] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE '11*, pages 461–470, 2011.

[9] N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *SPLC '09*, pages 211–220, 2009.

[10] X. Zhang, O. Haugen, and B. Møller-Pedersen. Model comparison to synthesize a model-driven software product line. In *SPLC '11*, pages 90–99, 2011.