Comparing Multiple MATLAB/Simulink Models Using Static Connectivity Matrix Analysis

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

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

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

***Abstract*—Model-based languages such as MATLAB/Simulink are crucial for the development of embedded software systems. To adapt to changing requirements, engineers commonly copy and modify existing systems to create new variants. Commonly referred to as *clone-and-own*, this reuse strategy is easy to apply and beneficial in the short term, but it entails severe maintenance and consistency issues in the long term, leading to a huge amount of redundant and similar assets. Moreover, a later transition towards structured reuse such as with software product lines inevitably requires the comparison of all existing variants prior to the actual migration. However, current work mostly revolves around the comparison of only two systems and despite approaches proposed that can cope with more, such are not applicable to embedded software systems such as MATLAB/Simulink.**

In this paper, we bridge this gap and propose *Static Connectivity Matrix Analysis (SCMA)*, a novel comparison procedure that allows for the evaluation of multiple MATLAB/Simulink model variants at once. In particular, we transform models into a matrix form which is used to compare all models and to identify all similar structures between them, even with model parts being completely relocated during clone-and-own. We allow engineers to tailor results and to focus on any arbitrary variant subset, enabling individual reasoning prior to migration. We provide a feasibility study from the automotive domain, showing our matrix representation to be suitable and our technique to be fast and precise.

I. Introduction

Embedded systems are prevalent in various industrial domains such as factory automation, avionic, automotive and rail [1]. In such fields where large, complex, and safety-critical systems are developed, their reliability and maintainability are vital [2]. *Model-Driven Engineering (MDE)* is a paradigm commonly used in these domains and is known to improve reliability [3]. Instead of manually implementing functionality with imperative programming languages, MDE uses function-block-based designs [1]. To this end, modeling languages such as *MATLAB/Simulink*[1] are used, which enforce modularity and are considered to enhance maintainability [4], [5]. While they allow for functionality to be implemented on a more intuitive level for engineers, overall development and evolution of embedded software systems remains a challenging and time-intensive task. In order to adapt to new requirements, engineers commonly reduce both work and time effort by copying and subsequently modifying existing systems [6]. Denoted *clone-and-own* [7], this reuse approach is easy to use, as existing functionality is copied rather than reimplemented. However, in the long-run, this technique has severe implications on maintenance and evolution of the variant portfolio [8], [9]. Among others, redundancies emerge that need to be evolved and maintained separately as knowledge about commonalities and differences between variants is not present.

Recent approaches addressed this problem by exploiting modularity of modeling languages for variant composition, and thus, mitigate the drawbacks of clone-and-own [10], [11]. While this may help to fight the symptoms of clone-and-own, it does not allow for a comprehensive understanding of such variants. It also does not tackle the root cause, which is unstructured reuse. Thus, redundant artifacts across systems and the inherently increased maintenance effort are still pervasive [9].

In contrast, *Software Product Lines (SPLs)* [12], [13] facilitate strategic reuse and promote maintainability while preventing redundancies between related software systems [14]. In practice, however, it is hard to foresee the entire scope of functionality required upfront [15]. Thus, families of similar systems often emerge ad-hoc using clone-and-own, resulting in a proliferation of legacy systems without information about their relations [16], [17]. The need to transition from clone-and-own towards an SPL often becomes evident only after variant genesis and requires major migration [15], [18]–[20].

Unfortunately, reactively migrating variants to an SPL or taking other measures to instantiate structured reuse, and thus, improving maintainability, poses an enormous challenge to practitioners [8], [21]–[23]. In particular, current approaches incrementally compare pairs of related systems [24], which considerably impose efficiency and accuracy of the results [25], [26].

Hence, we argue that it is inevitable to consider *all* relevant systems at once for their strategic migration towards an SPL. To allow for their assessment and eventually, migration towards an SPL practice, it is indispensable to compare all system variants with each other rather than contenting with a restricted evaluation such as incremental pairwise approaches. However, in MDE, most work addressing SPL migration strategies and system maintainability are only applicable to two systems, which limits applicability in-the-wild [26], [27]. Although approaches have been proposed that evaluate multiple models [22], [26]–[28], they are not applicable to an entire family of complex *MATLAB/Simulink* model variants.

[1] MathWorks®- http://www.mathworks.com/simulink/ - July 2018

In this paper, we tackle the aforementioned problems and propose our *Static Connectivity Matrix Analysis* to compare multiple *MATLAB/Simulink* models. To cope with an entire portfolio of models, we introduce the *Connectivity Matrix*, an intermediate representation that exploits the data flow and modular construction inherent to *MATLAB/Simulink* models. With our technique, we provide a comprehensive overview of all variants. Furthermore, engineers can filter the produced results and focus on arbitrary variant combinations. Hence, engineers can tailor results, perform individual reasoning and by using their domain knowledge, facilitate strategic decisions. In particular, we make the following contributions:

- We introduce the *Connectivity Matrix*, a descriptor that transforms models into matrix form, allowing for their efficient comparison.
- We propose *Static Connectivity Matrix Analysis (SCMA)*, a procedure to compare multiple *MATLAB/Simulink* models evolved from clone-and-own regardless of input order. We identify all similar structures between all input models and allow for their preliminary assessment regarding a strategic migration towards an SPL practice.
- We evaluate our descriptor and approach using a feasibility study from the automotive domain and show our technique to be fast, precise and applicable to models of industrial size.

The remainder of this paper is structured as follows. We provide background information on *MATLAB/Simulink* models and properties we utilize for our technique and outline descriptors to abstract from complex systems (Sec. II). We introduce our descriptor, the *Connectivity Matrix (CM)* and propose our SCMA (Sec. III). We assess our technique using a feasibility study with models from the automotive domain (Sec. IV) and discuss the results produced (Sec. V). We state related work (Sec. VI), future work and conclude our paper (Sec. VII).

II. Preliminaries

In this section, we provide details on *MATLAB/Simulink* models, properties that we utilize for our proposed technique, and introduce *descriptors* [29] to abstract from complex systems.

A. MATLAB/Simulink

MATLAB/Simulink is a block-based behavioral modeling language that utilizes *functional blocks* and *signals* to specify certain software system functionality. It is vital for the development of embedded software systems in various industrial domains such as avionic and automotive engineering [30], [31]. Such models constitute the central development artifact and are used to generate code for operation on microcontrollers [32].

Each block of a *MATLAB/Simulink* model either represents a specific *functionality* or it is used to structure the model. Every block has a set of syntactical and semantical properties that allow for it to be identified and to be compared with other blocks. Focusing on *MATLAB/Simulink*, the following block properties are of interest for the remainder of this paper:

- *function:* Represented function, i.e., what the block is used for.
- *label:* Non-unique textural name of the block.
- *interfaces:* For incoming and outgoing data:
  - *in-ports:* A block contains an arbitrary number of in-ports. Each in-port connects to exactly *one* out-port.
  - *out-ports:* A block contains an arbitrary number of out-ports. Each out-port can connect to *one or more* in-ports.
- *signal:* A directed edge, connecting in-ports and out-ports.

Industrial *MATLAB/Simulink* models comprise thousands of blocks to capture complex system behavior [16], [30], [32]. Logically connected blocks are commonly grouped together and encapsulated within a *Subsystem (SM)* block. SMs can be nested and structure the model horizontally, constituting a *model hierarchy* [1]. Every SM resides on a specific *hierarchical layer* δ_j that corresponds with its nesting depth.

Complex models comprise numerous hierarchical layers and can exhibit a total *hierarchical depth* of ten and more [33]. In Figure 1, we depict a simple *MATLAB/Simulink* model M_0, and highlight in gray the contained SMs labeled 2, 3, and 6. For clarity, models throughout this paper contain unique block labels only. The corresponding graph representation includes an artificial SM named *root*, highlighted in gray respectively. It comprises all blocks residing on the first hierarchical layer δ_0 to illustrate M_0's model hierarchy being a tree structure. We refer to the SM labeled *6* on the second hierarchical layer δ_1 as the *child system* of its respective *parent system*, the SM labeled *2* in Figure 1. Every SM on any layer δ_j is the root of its respective subtree and, thus, every model structured with SMs can be represented as a tree (cf. Figure 1).

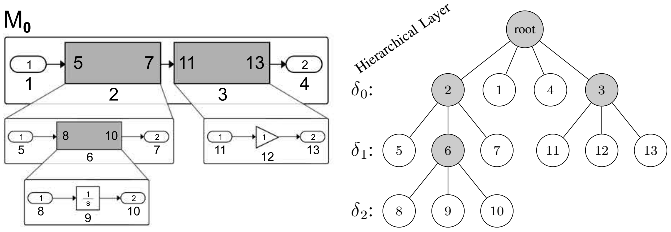


Fig. 1: *MATLAB/Simulink Model and its Graph Representation*

B. Model Abstraction Using Descriptors

To allow for the analysis of multiple *MATLAB/Simulink* models in their entirety, their complexity needs to be reduced first. Extensively applied in various fields such as robotics, image processing, network and electrical circuit design [34]–[37], *descriptors* abstract from complex systems by *describing* salient system information and representing them in a simpler format. Descriptors can be compared efficiently, and thus, even allow for the comparison of large quantities of data [38]. A descriptor should exhibit the following key characteristics [39]:

- The descriptor should be easy to extract from the original model.
- There should be a low probability of mismatch, that is, two distinct models should not result in the same descriptor.

Given their numerical efficiency, *matrices* have prevailed as one of the most widely utilized descriptor formats [29], [40], [38]. Furthermore, matrices are a generally accepted representation for graph structures and regarding their numerical efficiency, are intrinsically suitable for large-scale graph transformation and analysis procedures [41]. As shown in Figure 1, function block diagrams such as *MATLAB/Simulink* inherently constitute such graph structure. Consequently, we utilize matrices in our technique to derive a descriptor, the *Connectivity Matrix*, to abstract from *MATLAB/Simulink* models.

III. Static Connectivity Matrix Analysis

In this section, we propose *Static Connectivity Matrix Analysis* to identify and group all similar SM structures across all *MATLAB/Simulink* model variants, regardless of their location or their input order. Given SCMAs' workflow in Figure 4, we illustrate its four sequentially processed phases using the three models shown in Figure 5 for the remainder of this section.

A. Descriptor Creation - The Connectivity Matrix

We introduce our descriptor, the Connectivity Matrix, to abstract *MATLAB/Simulink* SMs into a matrix representation. Inherently characteristic of function block diagrams, *MATLAB/Simulink* models and, thus, SMs, are the composition of directly connected blocks, each of them having a specific *function*.

The CM exploits this property to approximate a given SM. Precisely, a CM represents which two block functions directly connect and how often they connect within the evaluated SM. For the models *M₁* and *M₂* from Figure 2, we show the corresponding CMs created by SCMA, *CM₁* and *CM₂*, in Figure 3. The depicted models contain six signals as well as eight blocks with a total of four distinct functions. For each block, its specific function is given in Figure 2 and pointed out to using arrows. The CMs from Figure 3 highlight in gray the single connection present in both abstracted models *M₁* and *M₂* from Figure 2, connecting the functional block types *Gain* and *Outport*. For readability reasons, non-present connections are left blank.

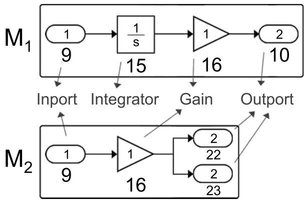
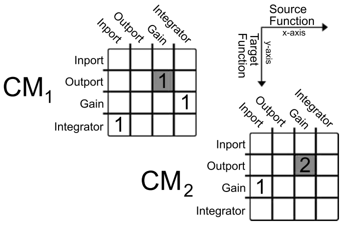


Fig. 2: Models *M₁* & *M₂*



$$CM_1: \begin{array}{c|cccc} & \text{Inport} & \text{Outport} & \text{Gain} & \text{Integrator} \\ \hline \text{Inport} & & & & \\ \text{Outport} & & & 1 & \\ \text{Gain} & & & & 1 \\ \text{Integrator} & 1 & & & \end{array}$$

$$CM_2: \begin{array}{c|cccc} & \text{Inport} & \text{Outport} & \text{Gain} & \text{Integrator} \\ \hline \text{Inport} & & & & \\ \text{Outport} & & & 2 & \\ \text{Gain} & 1 & & & \\ \text{Integrator} & & & & \end{array}$$

Fig. 3: CMs for *M₁* & *M₂*

For the remainder of this paper, we refer to a *connection* as both functions of a specific signals' source- and target block. Within any CM, the source of a connection always resides on the x-axis whereas its target resides on the y-axis (cf. Figure 3). Connections can be present multiple times in a certain SM. If, for instance, blocks with the functions *Gain* and *Outport* connect *n* times, the corresponding entry in the CM would be *n*. We refer to the CM as *static*, meaning that upon instantiation, every single CM has the same dimensions and is identical in its row and column construction. Only the entries of the CMs vary, depending on the individual connections present within the specific SM abstracted by the CM (cf. Figures 2 & 3).

To ensure all CMs to have the same dimensions, SCMA preprocesses all input models and generates a dictionary of all distinct block functions. The size of the dictionary then determines the dimensions of every single CM. For instance, *CM₂* in Figure 3 contains the block *Integrator* as a row and column entry, although this block function is not present in the corresponding model *M₂* in Figure 2. However, this function is present within the input model *M₁*. It is, therefore, part of the dictionary and used to construct the CMs. The retrieved dictionary is ordered and constitutes both CM axes, resulting in CMs being *quadratic* (cf. Figure 3). Moreover, when preprocessing SMs to create the dictionary, we already store all connections between any block functions within the corresponding CM. The order of the dictionary's entities can be chosen arbitrarily after preprocessing but it must be fixed, and thus, *static* for subsequent CM creation. As illustrated in Figure 4, every input model is processed separately and for each SM, a corresponding CM is created. For a *MATLAB/Simulink* model comprising k SMs, k CMs are generated by SCMA plus one additional CM to represent connections present on the top hierarchical layer δ_0 (cf. Sec. II-A). Each CM holds a reference to the specific SM it represents.

In Table I, we list all entities required to transform any SM from a *MATLAB/Simulink* model into its respective CM. We explicate the applied transformation procedure in Algorithm 1.

TABLE I: Entities Required for Creating the CM

M_i	A *MATLAB/Simulink* model
SM_j	A SM from the model M_i
A_{ij}	A CM representing the SM_j from the model M_i
CM_i	Set of all CMs for the model M_i
Φ	Set of all sets CM_i for all models M_i

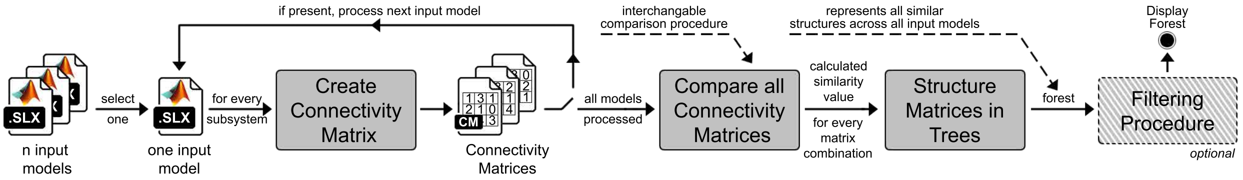
Algorithm 1: Creating the Connectivity Matrices

```
Input: $\Phi$, $M_i$
Output: $\Phi$
$CM_i \leftarrow \emptyset$
forall $SM_j \in M_i$ do
    $A_{ij}^{(n \times n)} \leftarrow \emptyset$
    foreach Block $b \in SM_j$ do
        forall Signals $\varphi_{out} \in b$ do
            $x \leftarrow function(b)$
            $y \leftarrow function(target(\varphi_{out}))$
            $A_{ij}(x, y) \leftarrow (A_{ij}(x, y) + 1)$
        end
    end
    $CM_i \leftarrow CM_i \cup \{A_{ij}\}$
end
$\Phi \leftarrow \Phi \cup \{CM_i\}$
return $\Phi$
```

Iterate through all SMs of the model and create a new connectivity matrix A_{ij}

Store every connection in the corresponding entry of the matrix A_{ij}

Store the matrix A_{ij} in the set of matrices for the current model M_i



if present, process next input model
.SLX
n input models
select one
.SLX
one input model
for every subsystem
Create Connectivity Matrix
CM
Connectivity Matrices
all models processed
interchangable comparison procedure
Compare all Connectivity Matrices
calculated similarity value
for every matrix combination
represents all similar structures across all input models
Structure Matrices in Trees
forest
Display Forest
Filtering Procedure
optional

Fig. 4: Workflow of the Static Connectivity Matrix Analysis (SCMA)

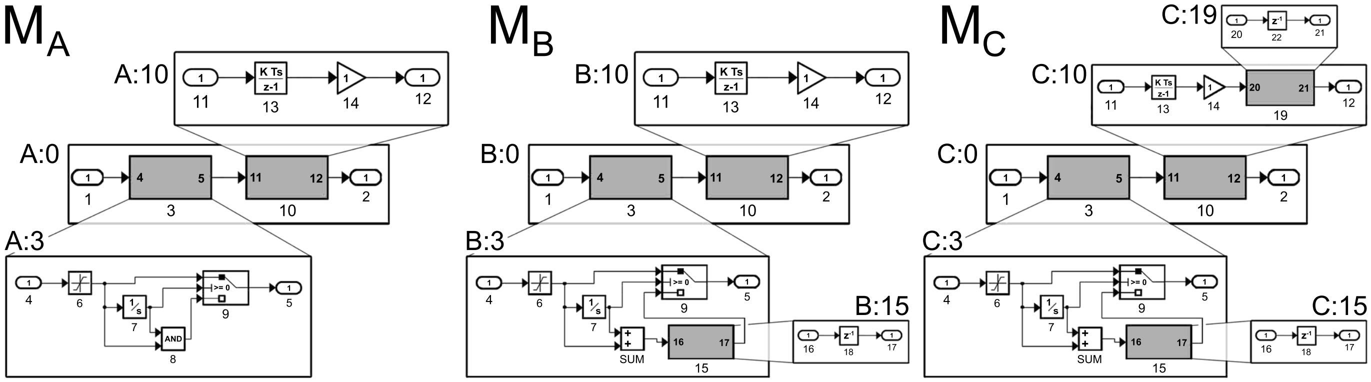


Fig. 5: *MATLAB/Simulink* Models Used as a Running Example for SCMA

Preprocessing the models from Figure 5 yields eleven distinct block functions. The model M_A comprises nine functions, M_B introduces the *Summation* and *Inverse* function located in its SMs *B:3* &*B:15* (cf. blocks *SUM* &*18*) and M_C does not exhibit new functions. The order in which models are processed to retrieve all distinct functions for the dictionary can be arbitrary.

Using the model M_B from Figure 5 as input for Algorithm 1, an empty set CM_B is created (cf. Line 1). For every SM_j present within the evaluated model M_B (cf. Line 2), a new CM A_{Bj} is created that holds references to both, the specific SM_j it abstracts as well as the corresponding *MATLAB/Simulink* model (cf. Line 3). The dimensions of any CM correspond to the size of the retrieved dictionary, hence *n=11* for the three models shown in Figure 5, and therefore, $A_{Bj}^{(11\times 11)}$. For every block within the current SM, all outgoing signals φ_{out} are evaluated and the connection established between the source and the target blocks' function is stored in the corresponding field of the CM (cf. Lines 4-8 & Figure 3). An existing entry is simply incremented by one when processing a connection that is already present within the CM. Every CM A_{Bj} is stored in the set CM_B prior to processing further SMs (cf. Line 11). Once all SMs of M_B have been processed, the resulting set CM_B is stored in Φ (cf. Line 13). For the models M_A, M_B and M_C from Figure 5, SCMA yields:

$$\Phi = \{CM_A\} \cup \{CM_B\} \cup \{CM_C\}, \quad |CM_A| = 3, |CM_B| = 4, |CM_C| = 5$$

For the models from Figure 5, all CMs generated by SCMA are shown in Figure 6, illustrating that they preserve the *parent-child* relation that exists between the associated SMs. Hence, CMs exhibit a hierarchical depth δ_j and fully resemble the entire *model hierarchy* (cf. Figures 5 & 6). Illustrated in Figure 6, we refer to the CM labeled *B:3* as the *parent* P of its respective *child*, the CM labeled *B:15* (cf. Sec. II-A).

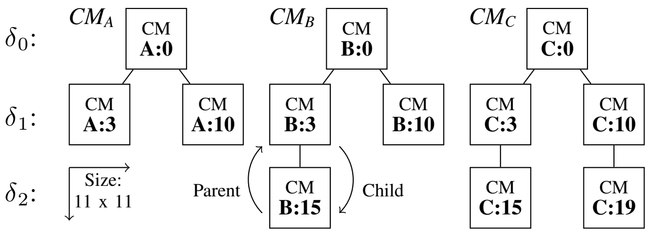


Fig. 6: CMs created by SCMA for the Models from Figure 5

B. Comparing Connectivity Matrices

To identify all similar CMs across all models despite relocation, SCMA compares all generated CMs with each other.

More precisely, any CM from a model M_X is compared with all other CMs except those from M_X. For instance, the CM *A:0* from Figure 6 is, therefore, not compared with itself or another CM (e.g. *A:3*) associated with the model M_A, but with each CM associated with another model (here: M_B and M_C). With SCMA, the focus is not to analyze a single but multiple model variants to support their migration towards an SPL. However, even the former can be achieved by SCMA using a single model and its copy only. For each CM comparison, a similarity value $0 \le \omega \le 1$ is calculated that represents the normalized distance between all two entries with the same x- and y-coordinates. Thus, ω reflects to what extent the respective SMs exhibit the same connections between certain functions.

Algorithm 2 details the procedure applied with SCMA to compare any two CMs. The entities required for Algorithm 2 are provided in Table II and complement those listed in Table I.

TABLE II: Entities Required for Comparing CMs

$\omega(A_{ij}, B_{xy})$	Compared CMs A and B with their similarity value ω
$A(x, y)$	Entry of the CM A, given its x and y coordinates
Ω	Set of all CM comparisons

Algorithm 2: Comparison of the Connectivity Matrices

```
Input: $\Phi$
Output: $\Omega$
$\Omega \leftarrow \emptyset$
foreach $CM_i \in \Phi$ do
    $j \leftarrow (i+1)$
    while $j < |\Phi|$ do
        foreach Matrix $A \in CM_i$ do
            forall Matrix $B \in CM_j$ do
                $\omega(A,B) = \frac{\sum_{x=1}^{n}\sum_{y=1}^{n}\begin{cases}0 & A(x,y) = B(x,y) = 0\\ \frac{min(A(x,y), B(x,y))}{max(A(x,y), B(x,y))} & else\end{cases}}{\sum_{x=1}^{n}\sum_{y=1}^{n}\begin{cases}0 & A(x,y) = B(x,y) = 0\\ 1 & else\end{cases}}$
                $\Omega \leftarrow \Omega \cup \{\omega(A,B)\}$
            end
        end
    end
end
return $\Omega$
```

Retrieve the set CM_i for the respecive model M_i

Retrieve another set CM_j and compare all comprised CMs

Compare all entries of the matrices with the same indices x and y and store the normalized value ω

Prior comparison, Ω is initialized to store every calculated similarity value ω along with its associated CMs *A* & *B* (cf. Line 1). Every set of CMs within Φ (cf. Line 2) is compared with all remaining sets representing different models (cf. Lines 2 to 4). For instance, given the sets *CM*$_{A\text{-}C}$ from Figure 6, Algorithm 2 compares all CMs present within *CM*$_A$ with all CMs contained in *CM*$_B$ and *CM*$_C$. Consequently, for *CM*$_B$, only *CM*$_C$ remains and for *CM*$_C$ itself, no further comparisons are necessary. This is because the similarity value calculation (cf. Line 7) is based on the *minimum* and *maximum* of the specific matrix entries, and thus, is *commutative*. Entries that are non-present in both CMs and by that, indicating non-existing connections in the SMs, are disregarded. Taking such connections into account would wrongfully increase the similarity value, and thus, adversely affect its soundness. Within *CM*$_1$ and *CM*$_2$ from Figure 3, a total of five entries exist that are non-zero in either of both CMs. For instance, for the connection *Gain - Outport*, the calculated similarity value would be $\frac{1}{2} = 0.5$. Comparing the CMs from Figure 3 in their entirety yields a similarity value of $\frac{(\frac{1}{2})+3*(\frac{0}{1})+(12*0)}{(4*1)+(12*0)} = \frac{0.5}{4} = 0.125$. For the CM sets *CM*$_{A\text{-}C}$ from Figure 6, representing the *MATLAB/Simulink* models *M*$_{A\text{-}C}$ from Figure 5, Algorithm 2 yields a total of

$$|\Omega| = |CM_A| * \big(|CM_B| + |CM_C|\big) + \big(|CM_B| * |CM_C|\big) = 47$$

distinct CM comparisons. In Table III, we list all similarity values ω calculated by Algorithm 2 for the CMs illustrated in Figure 6. Gray entries represent comparisons between CMs for which the associated SMs originate from the same *MATLAB/Simulink* model (cf. Figure 5). Blank entries depict comparisons that are obsolete because of the commutativity of ω.

TABLE III: Similarity Values for CMs from Figure 6

ω	A:0	A:3	A:10	B:0	B:3	B:10	B:15	C:0	C:3	C:10	C:15	C:19
B:0	**1.0**	0	0									
B:3	0	**0.42**	0									
B:10	0	0	**1.0**									
B:15	0	0	0									
C:0	**1.0**	0	0	**1.0**	0	0	0					
C:3	0	**0.42**	0	0	**1.0**	0	0					
C:10	**0.16**	0	**0.4**	**0.16**	0	**0.4**	0					
C:15	0	0	0	0	0	0	**1.0**					
C:19	0	0	0	0	0	0	**1.0**					

CM comparisons such as *ω(A:0,B:0)* reflect identical SMs (cf. Figure 5), and thus, exhibit a similarity value of *ω=1.0*. This holds for other comparisons as well, such as for those representing the SMs *A:10* and *B:10* or *B:3* and *C:3*. Looking at *A:0* and *C:10* in both, Table III and Figure 5, there is one common connection, *Subsystem* to *Outport* (cf. blocks *10* to *2* & *19* to *12*). Considering all connections, this accounts for a minute, but not necessarily negligible similarity value ω.

Furthermore, the information provided in Table III permits a preliminary assessment of more than two input models. For instance, *A:0* is not only identical to *B:0* but also to *C:0*.

C. Structuring Connectivity Matrices - The Forest Creation

SCMA utilizes the CM comparisons to group together *similar* CMs within *nodes*. We regard CMs *similar* if their comparison result exceeds a threshold ω_{min}. We do not preset this value, but given their domain knowledge, allow practitioners to define it either prior to SCMA or afterwards as part of an optional filtering step (cf. Sec. III-D). The nodes get connected if the contained CMs exhibit a parent-child relation (cf. Sec. II-A & Figure 6). As a result, nodes form *trees*. Since each CM corresponds to a SM, a tree represents a similar hierarchical SM structure between multiple models. Regardless of their hierarchical depth or structural location across models, SCMA creates all trees, and thus, identifies all similar SM structures between all input models. We store all trees within a *forest*.

In Table IV, we list the entities required for the forest creation, complementing those listed in Tables I & II. We detail the procedure utilized for the forest creation in Algorithm 3.

TABLE IV: Entities Required for Creating the Forest

$G_t = (V_t, E_t)$	Tree with index t and nodes V and edges E
Ψ	Forest containing all generated trees
P_X	Parent CM of the CM X (cf. Figure 6)
$\tilde{\omega}$	Average of all similarity values ω for a node

Algorithm 3: Forest Creation Using Connectivity Matrices

```
Input: $\Omega$
Output: $\Psi$
$\Psi \leftarrow \{G = (V, E)\}$
foreach $\omega(A, B) \in \Omega$ do
    insertionPossible ← false
    TreeLoop:
    foreach $(V_t, E_t) \in \Psi$ do
        if $\exists k \in V_t : \{P_A, P_B\} \in k$ then
            if $\exists v \in V_t : \{A, B\} \cap v \neq \emptyset$ then
                if $\forall x \in v : \omega(x, y) > 0,\ y \in \{A, B\}$ then
                    $v \leftarrow v \cup \{A, B\}$
                    $\tilde{\omega}(v) \leftarrow \frac{\sum_{\{x,y\} \in v} \omega(x,y),\ x \neq y}{|v|}$
                    insertionPossible ← true
                    break TreeLoop
                end
            end
            else
                $z \leftarrow \{A, B\}$
                $\tilde{\omega}(z) \leftarrow \omega(A, B)$
                $V_t \leftarrow V_t \cup \{z\}$
                $E_t \leftarrow \{e(k, z)\}$
                insertionPossible ← true
                break TreeLoop
            end
        end
    end
    if !insertionPossible then
        $G_{|\Psi|+1} \leftarrow (V_{|\Psi|+1}, E_{|\Psi|+1})$
        $z \leftarrow \{A, B\}$
        $\tilde{\omega}(z) \leftarrow \omega(A, B)$
        $V_{|\Psi|+1} \leftarrow V_{|\Psi|+1} \cup \{z\}$
        $\Psi \leftarrow \Psi \cup G_{|\Psi|+1}$
    end
end
return $\Psi$
```

All CMs within each set CM_i (cf. Table I) are sorted in a descending order with respect to their hierarchical depth δ_j. We thus ensure that for any CM comparison, the respective *parent* CMs *P* (cf. Sec. III-A), have already been processed. This way, Algorithm 3 can at all times establish the *parent-child* relation between compared CMs if such is present. With each set CM_i being sorted, the order in which they are processed by Algorithm 3 can be arbitrary. In other words, the input order for *MATLAB/Simulink* model variants, reflected by the sets CM_i, is irrelevant for the forest produced by SCMA.

Setting the similarity threshold ω_{min} to *0* causes Algorithm 3 to process all comparisons provided in Table III exceeding that value. Every ω is evaluated separately (cf. Line 2) and the associated CMs are then either inserted into an existing *node* within a tree (cf. Lines 8-12), used to create a new node within an existing tree (cf. Lines 15-22) or utilized to start a new tree (cf. Lines 25-31). For every ω and its associated CMs, every tree is evaluated separately (cf. Line 5) and a new tree is created only if no insertion in any existing tree is possible (cf. Lines 3 & 25). If an insertion is possible, no further trees are evaluated (cf. Lines 12 & 21). Consequently, no two trees can exist that exhibit the same CM comparison, resulting in the forest to be duplicate-free. Line 6 specifically returns *true* when comparing top level elements (i.e. *A:0 & B:0*) that do not have *parent CMs*. However, no node exists within the current tree G_t that either contains *A:0* or *B:0* (cf. Line 7). Hence, the CMs cannot be grouped within an existing node but are stored within a new node (cf. Line 16). Followed by *A:0* & *C:0*, Line 7 now holds because *A:0* has already been processed. Within a node, we only group together those CMs that for all possible combinations exhibit a similarity value greater zero (cf. Line 8). This way, a node being part of a larger structure only contains CMs that are at all similar to each other.

Consequently, grouping *A:0* & *C:0* together with *A:0* & *B:0* requires *ω(B:0,C:0)* to be greater than zero. This holds (cf. Table III), and thus, the CMs are grouped together and the overall similarity value for that node $\tilde{\omega}$ is recalculated as the average of all comprised ω values. When evaluating *ω(A:0,C:10)*, no tree exists that contains a node comprising both parent CMs (cf. Figure 6). Hence, a new tree and a node are created for that comparison (cf. Lines 26 & 27) and added to the forest (cf. Line 30) so that they can be used when processing further comparisons. For the comparisons from Table III, we provide all three trees generated by SCMA in Figure 7.

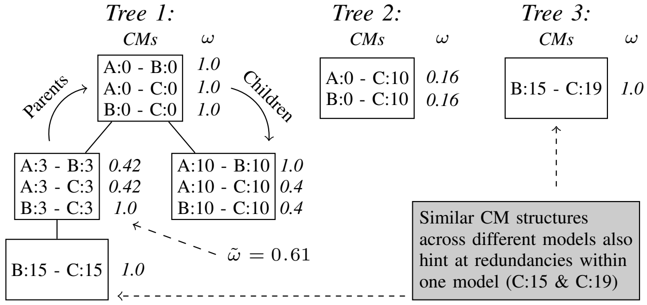


Fig. 7: Forest based on the Comparisons from Table III

Overall, SCMA identifies three trees for the models from our example (cf. Figure 5) and their respective CMs (cf. Figure 6 & Table III). *Tree 1* reflects the largest similar structure with three nodes containing CMs from all input models. *Tree 1* also indicates that the models M_B and M_C exhibit a stronger similarity for parts of the structure than, for instance, M_A and M_B or M_B and M_C respectively. Additionally, such tree representation reveals CMs on the third hierarchical layer δ_2 to be part of the structure but only between models M_B and M_C. Finally, *Tree 3* reveals similar structures that, within their original models M_B and M_C, reside at entirely different locations.

D. Filtering the Forest

The forest can be filtered to provide tailored information depending on individual demands. For instance, the similarity threshold ω_{min} is customizable and can even be set after forest creation. All CM comparisons not exceeding the specified value are then removed from the *trees*. Therefore, stakeholders must not necessarily set thresholds in advance, which bears the risk of losing information, especially without precise domain knowledge. For instance, setting ω_{min}=*0.2* would remove *Tree 2* from the forest in Figure 7 while ω_{min}=*0.5* would additionally remove parts of *Tree 1*.

Specifically, we provide two filtering procedures χ_r and χ_s that can be applied to the forest Ψ and filter it given a customizable subset Δ of all initial input models M. Hence, χ_r and χ_s constitute a projection [42] of the forest to a refined subset, allowing practitioners to tailor the forest to perform further reasoning. Applying χ_r results in the same output as for the forest creation λ but only for that subset Δ alone.

$$\Psi = \lambda(M)$$
$$\chi_r\Big(\Psi, \Delta\Big) = \lambda(\Delta), \ \ \Delta \subset M \wedge |M| \geq |\Delta| + 2$$

More precisely, χ_r removes all CM comparisons that contain a CM with the corresponding model not contained within Δ. For instance, Δ=$\{M_A$,$M_B\}$ would remove *Tree 3* from the forest in Figure 7 because the corresponding model for the CM *C:22* is not in Δ, and thus, the only node of *Tree 3* is removed. The second filtering procedure χ_s only retains nodes that contain CM comparisons encompassing *all* models within Δ. For instance, given Δ=$\{M_A$,M_B,$M_C\}$, χ_s would remove the bottom node from *Tree 1* because it does not comprise a CM associated with M_A. Applying this Δ but with procedure χ_r would retain that bottom node because *B:15* and *C:15* associate to M_A and M_B. Thus, χ_s produces a subset of χ_r's results.

$$\chi_s\Big(\Psi, \Delta\Big) = \chi_s\Big(\chi_r(\Psi, \Delta)\Big) \quad \neq \quad \lambda(\Delta)$$

Filtering can be reverted and reapplied with different settings. Thus, results can be tailored to specific demands while SCMA needs to process input models only once. With SCMA, we allow engineers to compare *all* input variants. Afterwards, filtering enables them to explicitly focus on a specific subset of these variants. For instance, engineers can perform additional analysis which targets certain SMs to identify more fine-grained variability [43] and to obtain profound understanding.

IV. Evaluation

In this section, we provide our objectives, information about the analyzed models, and the data analysis guidelines [44] we used.

A. Research Questions

With SCMA, we compare all system variants and identify all similar structures between them. For our empirical evaluation, we use *F-measure*, an approach widely used in software engineering that combines *precision* and *recall* [45]. We focus on the following research questions:

RQ1: *Can we regard CMs suitable to abstract MATLAB/Simulink models?* For our proposed SCMA, CMs are crucial and only suitable if they fulfill the characteristics defined in Sec. II. Hence, we evaluate if they are easy to extract and have a low probability of mismatch.

RQ2: *What level of precision and recall can we achieve with SCMA?* Precision and recall are vital for engineers to accept our technique. We refer to *precision* as the extent to which each generated tree reflects a similar hierarchical CM structure between all analyzed models. We refer to *recall* as the extent to which each tree only contains CMs that are similar and that exhibit such hierarchical relation.

RQ3: *Is SCMA's performance reasonable when scaling up?* Especially in an industrial environment, an acceptable runtime is essential for our proposed technique to be applicable in practice. We refer to performance as the total runtime required and its distribution over SCMA's three mandatory phases: CM creation, CM comparison, and forest creation. (cf. Sec. III).

B. Setup

To assess the feasibility of our proposed technique and our descriptor, the CM, we conducted a case study with real-world models from the automotive domain. Using an exemplary *driver assistance system (DAS)* from the publicly available SPES_XT[2] project, we artificially generated a set of model variants by identifying self-contained parts within the *DAS* model and extracting them. The extracted parts we used for the composition of model variants are listed in Table V, along with information on their overall size and structural complexity.

TABLE V: *DAS* Model Parts used for Variant Creation

Model name & *Abbreviation*	#blocks	#SMs	H_D
EmergencyBreak *'EB'*	409	43	7
FollowToStop ($req.$ CC) *'FTS'*	699	77	11
SpeedLimiter *'SL'*	497	57	10
CruiseControl *'CC'*	671	74	11
Distronic ($.req$ CC) *'DT'*	728	78	11

SMs – subsystem blocks, H_D – max. hierarchical depth, $req.$ - requires

Using the project documentation, we identified dependencies for *FTS* & *DT* that prohibit using them in isolation. Respecting the identified dependencies given in Table V, we combined the listed *DAS* model parts and created a total of 19 different variants that explicitly address a clone-and-own scenario. For instance, the largest model variant created contains all *DAS* model parts listed in Table V. Other variants contain only one *DAS* model part, e.g., *FTS* or two parts respectively, e.g., *FTS* and *EB*. From a clone-and-own standpoint, functionality was copied to the new variant and then extended by adding *EB*. The *DAS* model contains a total of 37 distinct block *functions*.

C. Data Analysis Guidelines

For the suitability of CMs, we evaluate their compliance with the characteristics defined for descriptors (cf. Sec. II). Hence, we first assess algorithmic complexity of the CM creation. Second, we evaluate whether (a) distinct *MATLAB/Simulink* SMs correctly result in distinct CMs and (b) whether distinguishable SMs exist that wrongfully result in identical CMs. To assess the feasibility of our proposed technique, manual evaluation of all possible 524.268 combinations[3] is infeasible. For precision and recall, we, therefore, focus on 18 comparisons, ranging from the smallest including only two systems to the largest possible comparison that includes all 19 model variants. The corresponding *trees*, generated by SCMA, were evaluated by an expert well familiar with the *DAS* model. Results were assessed directly within the *MATLAB/Simulink* environment. For performance, we state the algorithmic complexity and examine the actual runtime and its distribution over SCMAs mandatory phases. Each comparison was performed 10 times and the average was calculated to account for runtime deviations inherently present in a non-closed system. We implemented our technique in Java[4] using Eclipse[5] and its Modeling Framework[6].

V. Results

The SPES_XT case study was evaluated on a Dual-Core i7 processor with 12 GB of RAM, running Windows™ 7 on 64bit. We can only show aggregated data in this section, but detailed results as well as a screencast on SCMA can be found online[7].

RQ1: Suitability of the CM as a Descriptor

According to Sec. II-B, our descriptor is considered suitable if it is (a) easy to extract given a SM and (b) exhibits a low probability of mismatch for multiple SMs. For (a), creation of any CM inevitably requires a dictionary to be built in advance (cf. Sec. III-A). Hence, preprocessing input models to retrieve such a dictionary is mandatory for the CM creation, and thus, part of the overall process of deriving our descriptor. The dictionary is build by evaluating all blocks of all input models with every block and its *function* retrieved only once.

To this end, every SM is retrieved separately and all connections are stored accordingly within the associated CM. For each block within any SM, its own function as well as, for any of its outgoing signals, the respective target blocks' function are retrieved once (cf. Sec. II-A, III-A & Algorithm 1). Consequently, the CM creation exhibits a linear complexity of:

$$\mathcal{O}(n) \text{ - Complexity of CM Creation}$$

Figure 8 depicts the runtime required to create all CMs for a certain subset of model variants (cf. Sec. IV-B) given their size in the total number of contained blocks, provided on the x-axis.

[2] Software Platform Embedded Systems 'XT', TU München - spes2020.informatik.tu-muenchen.de/spes_xt-home.html - July 2018

[3] $\sum_{k=2}^{n} \binom{n}{k}$ because comparing no or only one variant can be omitted.

[4] Oracle Systems® - https://www.java.com/en/ - July 2018

[5] Eclipse Foundation®- https://eclipse.org/ - July 2018

[6] Eclipse Foundation®- https://eclipse.org/modeling/emf - July 2018

[7] Supplemental Material - http://www.vmsoftworks.com/reseach/SCMA

Each data point represents a comparison, ranging from the smallest with two models ($\approx$ 1.500 blocks) to the largest with all 19 models ($\approx$ 14.300 blocks). For the latter, 1528 CMs are created in $\approx$ 19 milliseconds, a runtime we consider acceptable.

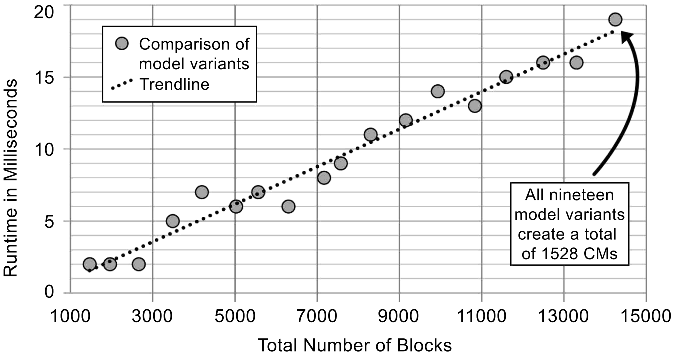


Fig. 8: Runtime of the Overall CM Creation

To check for mismatches, we first analyzed the dissemination of the similarity values for all $\approx$ 1.1 million CM comparisons performed when using all 19 model variants as input to SCMA. In Figure 9, we show the results with similarity values ω on the x-axis and their occurrence in percentage on the y-axis.

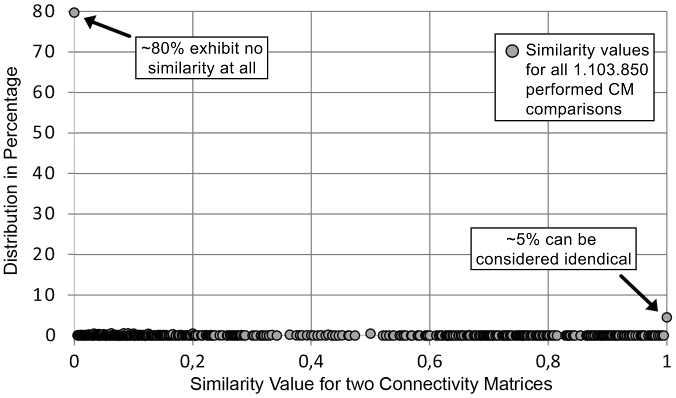


Fig. 9: Distribution of Similarity Values for CMs

Our data reveal that only two peaks appear within the similarity value distribution. First, with $\approx$ 80%, the majority of CM comparisons exhibit a similarity of $\omega = 0$, and thus, indicate distinct SMs. Secondly, $\approx$ 5% of all CM comparisons exhibit a similarity value of $\omega = 1$, and thus, indicate identical SMs. Moreover, the remaining $\approx$ 15% distribute almost equally with a slight separation of values for $\approx 0.3 \geq \omega \geq \approx 0.6$ (cf. Figure 9). For the four similarity value increments from $\omega = 0.2$ to $\omega = 0.8$ shown on the x-axis in Figure 9, we manually evaluated 100 CM comparisons with their associated SMs varying in size, yielding a total of 400 comparisons. Additionally, we evaluated 2000 comparisons with a similarity value of $\omega = 0$ as well as another 2000 for those with $\omega = 1$. For the former, all comparisons (100%) correctly represented *MATLAB/Simulink* SMs that were distinct. For the latter, we found 1863 (93.15%) to correctly represent identical SMs while for the remaining 137 (6.85%), associated SMs were not identical. For these 137 comparisons, we identified all respective 274 SMs to be very small and, thus, to comprise only a few blocks. We found their *connections* to be identical but rearranged. For example, two SMs, each with *Inport* blocks connecting to a *Summation* block, but at different *interfaces* (cf. Sec. II-A). Given the linear runtime required for their extraction and our manual evaluation on a subset of 4400 comparisons, we consider the CM to be easy to derive from a given system and for such, to exhibit a low probability of mismatch. As a result, we argue that the CM is suitable to abstract *MATLAB/Simulink* models (and contained SMs), evolved from clone-and-own.

RQ2: Precision and Recall of our Technique

SCMA may generate a vast amount of trees, depending on the number of input models and their structural diversity. Furthermore, practitioners can define a similarity threshold to exclude those CM comparisons not exceeding it from the forest creation. Depending on their settings, the forest size can drastically vary as similarity values distribute almost equally (cf. Figure 9). In Table VI, we provide details about the trees we evaluated for our 18 model comparisons, which is an excerpt from all trees generated by SCMA. We give more information online[7].

TABLE VI: Trees Evaluated For Precision & Recall

Compared Model Variants	Tree Size in Terms of Contained Nodes						
	1	2	3	4	21	30	44
1-2	10/152	2/2			1/1	1/1	
1-3	10/361	2/2			1/1		1/1
1-4	10/718	2/2	2/2		2/3		1/1
1-5	10/1369	2/2	3/5		2/5	1/1	1/1
1-6	10/1974	2/2	3/7		2/7	1/1	1/1
1-7	10/2985	3/3	3/9		2/9	1/1	1/1
1-8	10/3670	3/3	3/9		2/9	1/1	1/1
1-9	10/4632	3/3	3/13		2/13	1/1	1/1
1-10	10/6143	3/3	3/17		2/17	1/1	1/1
1-11	10/6586	3/3	3/17		2/17	1/1	1/1
1-12	10/7831	3/3	3/21		2/21	1/1	1/1
1-13	10/9763	3/3	3/26		2/25	1/1	1/1
1-14	10/11287	3/3	3/32		2/31	1/1	1/1
1-15	10/13609	3/3	3/38	1/1	2/37	1/1	1/1
1-16	10/15381	3/3	3/44	1/1	2/43	1/1	1/1
1-17	10/18063	3/3	3/50	1/1	2/49	1/1	1/1
1-18	10/20113	3/3	3/58	1/1	2/57	1/1	1/1
1-19	10/23185	3/3	3/66	1/1	2/65	1/1	1/1
Total:	180	49	47	5	34	16	17

Due to the sheer number of generated trees[8], we set the similarity threshold for CMs used for the forest creation to $\omega = 1$. In Table VI, we list the model variants included in the comparison on the y-axis. For instance, *1-10* refers to the comparison of 10 model variants (cf. Sec. IV-B). Moreover, SCMA generates trees of seven different sizes in terms of the number of comprised nodes and we list them at the top of Table VI. For each of the 18 comparisons and the tree sizes listed in Table VI, we provide the total number of generated trees and the number of manually evaluated trees. For instance, for the comparison *1-10*, SCMA generated 17 trees of size three, for which we manually assessed three trees regarding precision and recall (3/17).

[8] SCMA produces $\approx$ 38 thousand trees for all 19 variants and $\omega_{min} = 0$

In Table VI, fields left blank indicate tree sizes not created by SCMA for that specific comparison. Stated at the bottom of Table VI, we manually evaluated a total of 348 trees of various sizes. For instance, we evaluated 180 trees of size one (i.e., comprising only one node) while we evaluated a total of 16 trees with size thirty. We assessed precision and recall directly within the *MATLAB/Simulink* working environment and we provide more detailed information on our website[7].

For precision, all evaluated trees and their comprised nodes respectively, correctly represented a similar SM structure between the compared model variants. Furthermore, CMs contained within such nodes at all times correctly indicated a parent-child relation (cf. Sec. II & Figure 6). In other words, evaluated trees always did reflect a hierarchical structure that was similar between all analyzed models. For recall, we found all trees to be complete and by that, no CM comparison to be erroneously missing. Hence, each tree reflects a similar and complete hierarchical structure between all variants used as input.

Consequently, we argue that our proposed technique is precise and exhibits a high recall for the evaluated comparisons.

RQ3: Performance of our Technique

SCMA comprises three sequentially processed phases that determine its overall performance (cf. Sec. III-A-III-C & Figure 4). For the first phase, CM creation, we showed it to be of linear complexity (cf. Figure 8). The second phase, CM comparison, requires all created CMs to be compared with each other. Given their *commutativity* (cf. Sec. III-B), $\frac{n^2-n}{2}$ distinct comparisons can be performed for n CMs. Thus, their comparison depicted in Algorithm 2 exhibits a computational complexity that is quadratic in the total number of CMs:

$$\mathcal{O}(n^2) \text{ - Complexity of CM Comparisons}$$

The third phase, forest creation (cf. Sec. III-C), inserts every generated CM comparison either into an existing tree or utilizes it to found a new tree. To this end, each tree must be assessed and the comprised nodes must be checked for their compatibility (cf. Lines 6-8 in Algorithm 3). In the worst case, a new tree is created for every single CM comparison. For n CMs, a total of $\sum_{n=0}^{n-1} n = \frac{n^2-n}{2}$ trees can be constructed and, thus, need to be evaluated. Hence, the computational complexity for the forest creation is quadratic in the number of CM comparisons:

$$\mathcal{O}(n^2) \text{ - Complexity of the Forest Creation}$$

Combining all stated complexities for the three mandatory phases, SCMA exhibits a quadratic computational complexity. Supporting that assessment, Figure 10 illustrates the overall runtime required by SCMA to process a given number of CMs. In particular, we illustrate SCMA's runtime with regard to the number of used CMs, defined by the similarity threshold ω_{min}. Each data point in Figure 10 represents a model comparison (cf. Table VI). We provide information on the combined size of all models, included in the comparison by the number of blocks, on the x-axis and the respective runtime on the y-axis.

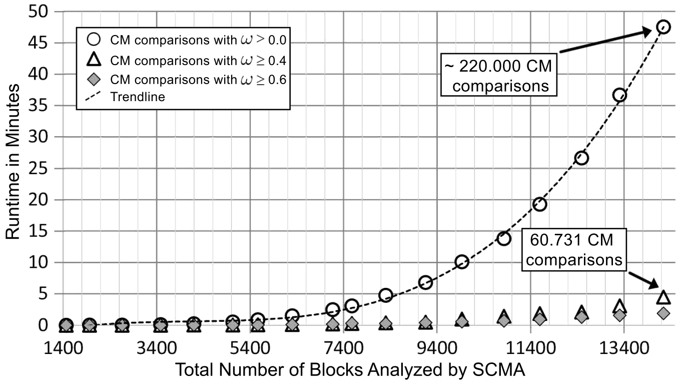


Fig. 10: SCMAs Runtime in Relation to the Number of CMs

For the largest comparison that includes all 19 model variants, comprising ≈ 14.300 blocks and ≈ 220.000 CMs (cf. Figure 10), SCMA required ≈ 47 minutes. Only considering comparisons with a similarity value of $\omega = 0.4$ or greater drastically reduces the runtime to under 5 minutes for all 19 variants. For $\omega = 0.6$, SCMA terminates in just ≈ 2 minutes for all 19 variants. The overall runtime distributes as follows. CM creation accounts for ≈ 3.4%, CM comparisons require ≈ 0.8% and ≈ 95.8% are needed for the forest creation. With the analysis and structuring of an entire model family of 19 model variants but without expertise of domain engineers, we conservatively argue that SCMA scales very well, and thus, is applicable to *MATLAB/Simulink* models of industrial size.

With the similarity threshold set to $\omega = 1$, SCMA identifies all identical structures and, thus, *type-1* and *type-2* clones [46]. However, by adjusting the threshold, SCMA can also capture *type-3* and, thus, *near-miss* clones that contain relocated model fragments and small additions and removals. Details on how SCMA copes with different clone types can be found online[7].

Threats to Validity

For the evaluation, we utilized models from the automotive domain only and assessed feasibility using a single study. We developed SCMA independently and prevented ourselves from being biased to that specific domain. The results may not be generalizable, however, we argue that models from the automotive domain are of considerable complexity, which makes us confident that our technique is applicable to other domains as well. Nevertheless, we acknowledge that other domains may exhibit peculiarities we did not consider and which may adversely affect our technique. We furthermore acknowledge that clone-and-own sceneries present in industry may not be fully reflected by the SPES_XT models we evaluated.

We abstract models using descriptors and compare them to calculate a similarity value. Our descriptor only approximates syntactical but not semantical equality. Thus, practitioners might judge differently on the suitability of the descriptor. They may also question the procedure we use to compare them. We argue that we have shown our descriptor to be suitable in a clone-and-own scenario and that we allow practitioners using their domain knowledge to adjust or replace the comparison procedure accordingly to meet their specific demands.

VI. Related Work

Current model comparison techniques mainly focus on the analysis of just two models [27]. In [47], the authors propose an approach to derive the differences of *MATLAB/Simulink* models and semantically lift them to ease comprehensibility. While we also aim to reinstate maintainability by enhancing system comprehensibility, the approach proposed in [47] is applicable to only two models. Respectively, model clone detection approaches are proposed in [6] and their applicability is shown. However, our work fundamentally differs from most clone-detection approaches as our technique is specifically designed to cope with multiple models whereas work such as [6], [47] is limited to comparing only two models. Although clone-detection approaches exist that can cope with more [48], applicability is shown only on small case studies and found clones are not aggregated. Our work aggregates similar structures to provide an extensive overview of multiple models while work such as [48] does not. In [1], the authors identify variation points within multiple *MATLAB/Simulink* models, also using a matrix representation for their approximation. Their approach facilitates on finding cliques of maximum size, an *NP-hard* problem. Hence, applicability is shown only using small models while our approach is designed to cope with an entire variant portfolio. In [22], the authors identify similarities between *Unified Modeling Language (UML)* model variants, mapping them to features to allow for their SPL migration. With our work, we do not identify features. The approach that was first proposed in [27] and extended upon by visualizing results in [22], [49], however, indispensably requires domain knowledge to map system artifacts to features. Our approach was motivated by the fact that such domain knowledge is not present in a *clone-and-own* scenario. In [28], the authors utilize *EMF Compare* to identify variability by means of the *Common Variability Language (CVL)*. Similar to the approach proposed in [27], domain knowledge is required in the process. Much like [28], authors propose to use CVL in [50] to identify commonalities and differences within model variants for their migration towards an SPL practice. Their work also relies on domain knowledge and the definition of a *base model*. Our work differs by that no domain knowledge is required upfront and no specific starting point has to be defined. Utilizing CVL to achieve the identification of feature locations with a model family, authors in [51] evaluate different search algorithms. We do not focus on features and, in contrast to [51], do not require domain knowledge upfront. Proposing the *ECCO tool* in [10], [52] the authors identify, much like our work, similar structures between system variants. Their approach targets source code while our approach targets model-based languages. Furthermore, domain knowledge is required for their analysis. In [53], the authors *slice MATLAB/Simulink* models. Although they utilize trees for their approach, much like our technique, their work focuses on a single model only. Our approach, however, targets multiple models. In [54], compatibility between *MATLAB/Simulink* model versions and variants is evaluated. Models are transformed into state machines and compared with each other. However, comparison is only performed on two models at a time and precise domain knowledge is required to determine which model versions and variants are to be compared. Our work focuses on a larger data set and does not require such knowledge. An approach to extractively generate a SPL from a set of system variants is proposed in [55]. Unlike our approach, their work focuses on source code rather than model-based languages. Furthermore, evaluation is done using only three variants while we utilized a portfolio comprising nineteen variants. Assessing source code in [56], domain knowledge is again required for the analysis proposed by the authors. Extending upon *NICAD* [57], a code clone detection tool, authors leverage it to cope with *MATLAB/Simulink* models in [58] and introduce *SIMONE*. Using their textural form rather than a descriptor for comparison, *SIMONE*, much like our approach, identifies *type-3* and, thus, *near-miss clones* [46] within model subsystems. However, SCMA furthermore aggregates results with respect to hierarchical structures. An approach to compare multiple UML models is proposed in [26]. Classes are compared based on their *attributes*, assuming them to vary between different models. With *MATLAB/Simulink*, however, blocks always have the same attributes (i.e. *name* & *function*) and it is rather their *value* that differs. With the computational complexity of their approach, we argue that the work in [26] is not applicable to *MATLAB/Simulink*.

VII. Conclusion and Future Work

To adapt to new requirements, clone-and-own, describing the process of copying and modifying existing systems in an unstructured and undocumented fashion, prevails. It eases development efforts in the short-term but unfortunately, results in a proliferation of almost-alike redundant copies of assets with information on their relations rendered incomprehensible. As a result, system maintainability is severely impeded and sustainable development may be at risk.

Consequently, it is of superior interest for engineers to regain comprehensibility of the emerged model portfolio. In practice, such models are of considerable complexity and size. Hence, techniques must scale to deal with an entire model family, which is a limitation for current techniques. We address this problem by proposing a new technique for the comparison of an arbitrary number of *MATLAB/Simulink* models, called *Static Connectivity Matrix Analysis (SCMA)*. In its core, our technique creates a descriptor, called *Connectivity Matrix (CM)*, that abstracts salient information of actual models, and thus, allows for an efficient comparison of an entire model portfolio at large scale. Moreover, based on CMs, we are able to create trees that resemble structural similarities of the original models. Our evaluation shows that creating CMs is fast and that SCMA is precise and scales even for a vast amount of models. Thus, our technique can be applied even to real-world scenarios to recreate information about commonalities across several models.

For future work, we plan to extend our evaluation with further industrial case studies. We also intend to discuss our technique with engineers to identify and tackle limitations they may see. Ultimately, we plan to apply clustering to our results to quantify redundancies within models, and thus, their reduction potential.

Acknowledgments

Supported by the DFG (German Research Foundation) (SCHA 1635/12-1). We thank Remo Lachmann, Tina Murse and Alexander Knüppel for their strong support and guidance on this paper.

References

[1] U. Ryssel, J. Ploennigs, and K. Kabitzsch, "Automatic Variation-point Identification in Function-block-based Models," in *Proc. of the Intl. Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2010, pp. 23–32.

[2] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert, "Clone Detection in Automotive Model-based Development," in *Proc. of the Intl. Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 603–612.

[3] L. Cretu and F. Dumitriu, *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice*. Apple Academic Press, 2014.

[4] R. Pressman, *Software Engineering: A Practitioner's Approach*, ser. McGraw-Hill Higher Education. Boston, 2005.

[5] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, "The Structure and Value of Modularity in Software Design," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2001, pp. 99–108.

[6] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Complete and Accurate Clone Detection in Graph-based Models," in *Proc. of the Intl. Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 276–286.

[7] C. Riva and C. D. Rosso, "Experiences with Software Product Family Evolution," in *Proc. of the International Workshop on Principles of Software Evolution*, 2003, pp. 161–169.

[8] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An Exploratory Study of Cloning in Industrial Software Product Lines," in *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 25–34.

[9] R. Lapeña, M. Ballarin, and C. Cetina, "Towards Clone-and-own Support: Locating Relevant Methods in Legacy Products," in *Proceedings of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2016, pp. 194–203.

[10] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants," in *Proc. of the International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 391–400.

[11] D. Rabiser, P. Grünbacher, H. Prähofer, and F. Angerer, "A Prototype-based Approach for Managing Clones in Clone-and-own Product Lines," in *Proc. of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2016, pp. 35–44.

[12] K. Pohl, G. Böckle, and Linden, F. J. van der, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[13] F. J. v. d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[14] J. A. Kim, "Case Study of Software Product Line Engineering in Insurance Product," in *Proc. of the International Systems and Software Product Line Conference (SPLC)*. Springer, 2010, pp. 495–495.

[15] J. Rubin and M. Chechik, "Quality of Merge-Refactorings for Product Lines," in *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2013, pp. 83–98.

[16] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz, "Model Clone Detection in Practice," in *Proceedings of the International Workshop on Software Clones (IWSC)*, 2010, pp. 57–64.

[17] J. Rubin and M. Chechik, "Combining Related Products into Product Lines," in *Proc. of the Intl. Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2012, pp. 285–300.

[18] S. Duszynski, J. Knodel, and M. Becker, "Analyzing the Source Code of Multiple Software Variants for Reuse Potential," in *Proc. of the Working Conference on Reverse Engineering (WCRE)*. IEEE, 2011, pp. 303–307.

[19] S. Apel, A. v. Rhein, P. Wendler, A. Grsslinger, and D. Beyer, "Strategies for Product-line Verification: Case Studies and Experiments," in *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 482–491.

[20] L. Linsbauer, E. R. Lopez-Herrejon, and A. Egyed, "Recovering Traceability Between Features and Code in Product Variants," in *Proc. of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2013, pp. 131–140.

[21] C. Kastner, A. Dreiling, and K. Ostermann, "Variability Mining: Consistent Semi-automatic Detection of Product-Line Features," *IEEE Transactions on Software Engineering*, vol. 40, pp. 67–82, 2014.

[22] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. l. Traon, "Automating the Extraction of Model-Based Software Product Lines from Model Variants," in *Proc. of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 396–406.

[23] J. Font, L. Arcega, Ø. Haugen, and C. Cetina, "Building Software Product Lines from Conceptualized Model Patterns," in *Proc. of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2015, pp. 46–55.

[24] *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[25] S. Lity, M. Al-Hajjaji, T. Thüm, and I. Schaefer, "Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-line Analysis," in *Proc. of the International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*. ACM, 2017, pp. 60–67.

[26] J. Rubin and M. Chechik, "N-way Model Merging," in *Proc. of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*. ACM, 2013, pp. 301–311.

[27] J. Martinez, T. Ziadi, J. Klein, and Y. le Traon, "Identifying and Visualising Commonality and Variability in Model Variants," in *Proc. of the European Conference on Modeling Foundations and Applications (ECMFA)*. Springer, 2014, pp. 117–131.

[28] X. Zhang, Ø. Haugen, and B. Møller-Pedersen, "Model Comparison to Synthesize a Model-Driven Software Product Line," in *Proc. of the International Systems and Software Product Line Conference (SPLC)*. IEEE, 2011, pp. 90–99.

[29] V. Olshevsky, *Structured Matrices in Mathematics*, ser. Contemporary mathematics - American Mathematical Society. American Mathematical Society, 2001, no. Bd. 1.

[30] R. Reicherdt and S. Glesner, "Slicing MATLAB Simulink Models," in *Proc. of the International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 551–561.

[31] M. W. Whalen, A. Murugesan, S. Rayadurgam, and M. P. E. Heimdahl, "Structuring Simulink Models for Verification and Reuse," in *Proc. of the International Workshop on Modeling in Software Engineering (MiSE)*. ACM, 2014, pp. 19–24.

[32] D. Merschen, A. Polzer, G. Botterweck, and S. Kowalewski, "Experiences of Applying Model-based Analysis to Support the Development of Automotive Software Product Lines," in *Proc. of the Workshop on Variability Modeling of Software-Intensive Systems (VAMOS)*. ACM, 2011, pp. 141–150.

[33] A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer, "First-class Variability Modeling in Matlab/Simulink," in *Proc. of the International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*. ACM, 2013, pp. 4:1–4:8.

[34] D. Zhang and G. Lu, "A Comparative Study of Curvature Scale Space and Fourier Descriptors for shape-based Image Retrieval," *Journal of Visual Communication and Image Representation*, vol. 14, no. 1, pp. 39 – 57, 2003.

[35] H. Jeong, M. Obaidat, N. Yen, and J. Park, *Advances in Computer Science and its Applications (CSA)*. Springer, 2013.

[36] G. Duan, *Analysis and Design of Descriptor Linear Systems*, ser. Advances in Mechanics and Mathematics. Springer, 2010.

[37] H. Tan, *Knowledge Discovery and Data Mining*, ser. Advances in Intelligent and Soft Computing. Springer, 2012.

[38] P. Benner, V. Mehrmann, and D. Sorensen, *Dimension Reduction of Large-Scale Systems*, ser. Lecture Notes in Computational Science and Engineering. Springer, 2005.

[39] T. Tuytelaars and K. Mikolajczyk, "Local Invariant Feature Detectors: A Survey," *Foundations and Trends in Computer Graphics and Vision*, vol. 3, no. 3, pp. 177–280, 2008.

[40] A. Antoulas, *Approximation of Large-scale Dynamical Systems*, ser. Advances in Design and Control. Society for Industrial and Applied Mathematics, 2005.

[41] *Graph Theory in Modern Engineering: Computer Aided Design, Control, Optimization, Reliability Analysis*, ser. Mathematics in Science and Engineering. Elsevier Science, 1973.

[42] C. Meyer, *Matrix Analysis and Applied Linear Algebra*, ser. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM), 2000.

[43] A. Schlie, D. Wille, S. Schulze, L. Cleophas, and I. Schaefer, "Detecting Variability in MATLAB/Simulink Models: An Industry-Inspired Technique and Its Evaluation," in *Proc. of the International Systems and Software Product Line Conference (SPLC)*, 2017, pp. 215–224.

[44] P. Runeson and M. Höst, “Guidelines for Conducting and Reporting Case Study Research in Software Engineering,” *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[45] R. Malhotra, *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*. CRC Press, 2016.

[46] 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*, vol. 74, no. 7, pp. 470–495, 2009.

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

[48] B. Al-Batran, B. Schätz, and B. Hummel, “Semantic Clone Detection for Model-Based Development of Embedded Systems,” in *Proc. of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 2011, pp. 258–272.

[49] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, “Bottom-up Adoption of Software Product Lines: A Generic and Extensible Approach,” in *Proc. of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2015, pp. 101–110.

[50] J. Font, M. Ballarín, Ø. Haugen, and C. Cetina, “Automating the Variability Formalization of a Model Family by Means of Common Variability Language,” in *Proc. of the International Systems and Software Product Line Conference (SPLC)*. ACM, 2015, pp. 411–418.

[51] J. Font, L. Arcega, . Haugen, and C. Cetina, “Achieving Feature Location in Families of Models through the use of Search-Based Software Engineering,” *IEEE Transactions on Evolutionary Computation*, vol. PP, no. 99, pp. 1–13, 2017.

[52] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, “The ECCO Tool: Extraction and Composition for Clone-and-own,” in *Proc. of the International Conference on Software Engineering (ICSE)*, 2015, pp. 665–668.

[53] N. E. Gold, D. Binkley, M. Harman, S. Islam, J. Krinke, and S. Yoo, “Generalized Observational Slicing for Tree-represented Modelling Languages,” in *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 547–558.

[54] B. Rumpe, C. Schulze, M. von Wenckstern, J. Ringert, and P. Manhart, “Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution,” in *Proc. of the International Systems and Software Product Line Conference (SPLC)*, 2015, pp. 141–150.

[55] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho, *Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming*, 2007, pp. 117–142.

[56] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, “Variability Extraction and Modeling for Product Variants,” *Software & Systems Modeling*, vol. 16, pp. 1179–1199, 2017.

[57] C. K. Roy and J. R. Cordy, “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization,” in *16th IEEE International Conference on Program Comprehension*, 2008, pp. 172–181.

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