Clone Detection in Automotive Model-Based Development

Florian Deissenboeck,
Benjamin Hummel, Elmar Juergens,
Bernhard Schätz, Stefan Wagner
Institut für Informatik
Technische Universität München
Garching b. München, Germany

Jean-François Girard, Stefan Teuchert
MAN Nutzfahrzeuge AG
Elektronik Regelungs- und Steuerungssysteme
München, Germany

ABSTRACT

Model-based development is becoming an increasingly common development methodology. In important domains like embedded systems already major parts of the code are generated from models specified with domain-specific modelling languages. Hence, such models are nowadays an integral part of the software development and maintenance process and therefore have a major economic and strategic value for the software-developing organisations. Nevertheless almost no work has been done on a quality defect that is known to seriously hamper maintenance productivity in classic code-based development: *Cloning*. This paper presents an approach for the automatic detection of clones in large models as they are used in model-based development of control systems. The approach is based on graph theory and hence can be applied to most graphical data-flow languages. An industrial case study demonstrates the applicability of our approach for the detection of clones in Matlab/Simulink models that are widely used in model-based development of embedded systems in the automotive domain.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries, Reuse models*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms*

General Terms

Algorithms, Design, Experimentation, Measurement

Keywords

Clone detection, model clone, Matlab/Simulink, data-flow

1. INTRODUCTION

Software in the embedded domain, and especially in the automotive sector, has reached considerable size: The current BMW 7 series, for instance, implements about 270 user functions distributed over up to 67 embedded control units, amounting to about 65 megabytes of binary code. The upcoming generation of high-end vehicles will incorporate one gigabyte of on-board software [27]. Due to the large number of variants in product-lines, high cost pressure, and decreasing length of innovation cycles, the development process in this domain demands a high rate of (software) reuse. This is typically achieved by the use of general-purpose domain-specific libraries with elements like PID-controllers as well as the identification and use of application-specific elements like sensor-data plausibilisation. As a consequence of this highly reuse-oriented approach, the identification of common elements in different parts of the software provides an important asset for the model-based development process.

A proposed solution for the increasing size and complexity as well as for managing product lines is to rely on model-based development methods, *i. e.,* software is not developed on the classical code level but with more abstract models specific to a particular domain. These models are then used to automatically generate production code[1]. Especially in the automotive domain today already up to 80% of the production code deployed on embedded control units can be generated from models specified using domain-specific formalisms like Matlab/Simulink [10]. The models employed here appear highly different from classic C or Java code. However, they share the common purpose of specifying executable programs and can therefore be understood as higher level programming languages. Hence, it does not surprise that they exhibit a number of quality defects that are well known from classic programming languages. One such defect is the presence of redundant program elements or *clones*.

Cloning is known to hamper productivity of software maintenance in classical code-based development environments [16, 25]. This is due to the fact that changes to cloned code are error-prone as they need to be carried out multiple times for all (potentially unknown) instances of a clone [1, 19, 20]. Hence, the software engineering community developed a multitude of approaches and powerful tools for the detection of code clones [1, 2, 9, 11, 15, 18]. However, there has been very little work on cloning in the context of model-based development. Consequently, current knowledge on cloning and its consequences for model-based development is limited to anecdotal evidence only. Naturally, this is also due to lack of clone detection tools for this purpose.

---

[1]The term "model-based" is often also used in the context of incomplete specifications that do mainly serve documentation purposes. Here however, we focus on models that are employed for full code generation.

Taking into account the economic importance of software developed in a model-based fashion and the well-known negative consequences of cloning for software maintenance, we consider this a precarious situation.

1.1 Problem

The reasons identified for cloning in code-based development suggest that cloning is expected to be as much of a problem concerning maintenance in model-based development as it is in code-based development. Additionally, especially in a production-line oriented development the detection of clones can support the identification of potential domain-specific library elements, easing the development of new variants. Unfortunately, there is currently neither empirical data on model cloning nor do tools exists that allow automatic detection of clones in models.

1.2 Contribution

This paper presents an approach for the automatic detection of clones in models. The approach is based on graph theory and hence applies to all models using data-flow graphs as their fundamental basis. It consists of three steps: pre-processing and normalisation of models, extraction of clone pairs (*i. e.*, parts of the models that are equivalent) and clustering of those pairs to also find substructures used more than twice in the models. Through the application of a suitable heuristic the approach overcomes the limits of algorithmic complexity and can be applied to large models ($>$ 10,000 model elements) as they are typically found in the embedded domain.

We demonstrate the applicability of our approach in a case study undertaken with MAN Nutzfahrzeuge, a German supplier of commercial vehicles and transport systems. Here we implemented our approach for the automatic detection of clones in Matlab/Simulink/TargetLink models as they are widely used in the automotive domain.

1.3 Results and Consequences

Our approach showed in the case study that there are clones in typical models used for code generation. In the analysed models with over 20,000 elements, 139 clone classes were found which affect over a third of the total model elements. By manual inspection a significant share of them were classified as relevant. Moreover, the case study shows that it is feasible to analyse models for clones. Our approach proved to be applicable to industry-relevant model sizes. Hence, it can be used to prevent the introduction of clones in models and to identify possible model parts that can be extracted into domain-specific intellectual-property library to support a product line-like development.

1.4 Outline

The next section summarises existing work about the impact of code cloning on software maintenance. Section 3 introduces Matlab/Simulink, which is widely used in the embedded domain, to illustrate cloning issues in model-based development. Then we explain our approach and present the detection algorithm in Section 4, followed by results for application of our approach in an industrial case study (Section 5). Sections 6 and 7 then outline possible directions for future research in the detection of model clones respectively differentiate our work from other publications on similar topics, before we summarise our current work in Section 8.

2. CLONING

In general, code clones are *code fragments that are similar w.r.t. to some definition of similarity* [16]. The employed notions of similarity are heavily influenced by the program representation on which clone detection is performed and the task for which it is used.

The central observation motivating clone detection research is that code clones normally implement a common concept. A change to this concept hence typically requires modification of all code fragments that implement it, and therefore modifications of all clones. In a software system with cloning, a single conceptual change (*e. g.*, a bug fix) can thus potentially require modification in multiple places, if the affected source code (or model) parts have been cloned. Since the localisation and consistent modification of all duplicates of a code (or model) fragment in a large software system can be very costly, cloning potentially increases the maintenance effort. Additionally, clones increase program volume and thus further increase maintenance efforts, since several maintenance-related activities are influenced by program size.

In [25], Monden et. al. analyse the change history of a large COBOL legacy software system. They report that modules containing clones have suffered significantly more modifications than modules without cloning, giving empirical indication of the negative impact of cloning on maintainability. Furthermore, bugs can be introduced, if not all impacted clones are changed consistently. In [20] Jiang et. al. report the discovery of numerous bugs uncovered by analysing inconsistencies between code clones in open source projects.

Despite the mentioned negative consequences of cloning, the analysis of industrial and open-source software projects shows that developers frequently copy-and-paste code [1, 19]. Different factors can influence a programmer's choice to copy-and-paste existing code instead of using less problematic reuse mechanisms: Language limitations are often the source of duplication, if programmers cannot employ other reuse mechanisms, as Kim et. al. report in [14]. In [16], Koschke lists time pressure, insufficient knowledge of consequences of cloning, badly organised reuse processes or questionable productivity metrics (lines of code per day) as possible process-related issues. Kapser and Godfrey [12] describe situations (*e. g.*, experimental validation of design variations) in which cloning of source code, despite its known drawbacks, can be argued to have some advantages over alternative solutions. But even if cloning is applied on purpose, as rarely as it seems to be the case, the ability to identify and track clones in evolving software is crucial during maintenance.

Since neither the reasons nor the consequences for cloning are rooted in the use of textual programming languages as opposed to model-based approaches for software development, we expect cloning to also impact model-based development.

3. MODELS FOR CONTROL SYSTEMS

The models used in the development of embedded systems are taken from control engineering. Block diagrams – similar to data-flow diagrams – consisting of blocks and lines are used in this domain as structured description of these systems. Thus, blocks correspond to functions (*e. g.*, integrators, filters) transforming input signals to output sig-
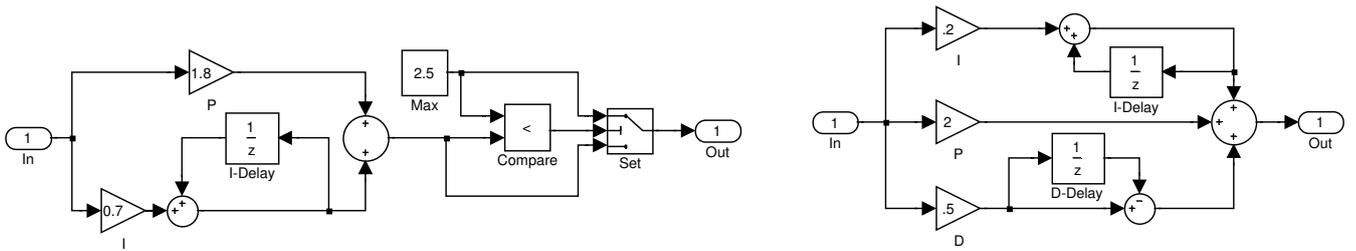
Figure 1: Examples: Discrete saturated PI-controller and PID-controller

nals, lines to signals exchanged between blocks. The description techniques specifically addressing data-flow systems are targeting the modelling of complex stereotypical repetitive computations, with computation schemes largely independent of the computed data and thus containing little or no aspects of control flow. Typical applications of those models are, *e. g.,* signal processing algorithms.

Recently, tools for this domain – with Matlab/Simulink [22] or ASCET-SD as prominent examples – are used for the generation of embedded software from models of systems under development. To that end, these block diagrams are interpreted as descriptions of time- (and value-)discrete control algorithms. By using tools like TargetLink [7], these descriptions are translated into the computational part of a task description; by adding scheduling information, these descriptions are then combined – often using a real-time operating system – to implement an embedded application.

Figure 1 shows two examples of simple data-flow systems using the Simulink notation. Both models transform a time- and value-discrete input signal In into an output signal Out, using different types of basic function blocks: *gains* (indicated by triangles, *e. g.,* P and I), *adders* (indicated by circles, with + and − signs stating the addition or subtraction of the corresponding signal value), one-unit delays (indicated by boxes with $\frac{1}{z}$, *e. g.,* I-Delay), *constants* (indicated by boxes with numerical values, *e. g.,* Max), *comparisons* (indicated by boxes with relations, *e. g.,* Compare), and *switches* (indicated by boxes with forks, *e. g.,* Set).

Systems are constructed by using instances of these types of basic blocks. When instantiating basic blocks, depending on the block type different attributes are defined; *e. g.,* constants get assigned a value, or comparisons are assigned a relation. For some blocks, even the possible input signals are declared. For example, for an adder, the number of added signals is defined, as well as the corresponding signs.

By connecting them via signal lines, (basic) blocks can be combined to form more complex blocks, allowing the hierarchic decomposition of large systems into smaller subsystems. Because of this simple mechanism of composition, block diagrams are ideally suited for a modular development process, supporting the reuse of general-purpose control functions as well as application-domain specific IP-blocks. However, it also eases a copy-and-paste approach which – combined with the evolution of product lines typically found with embedded systems and large block libraries – potentially lead to a substantial number of clones.

4. APPROACH

In this section we formalise the problem of clone detection in graph-based models and describe an algorithmic approach for solving it. Basically our approach consists of three steps. First we preprocess and normalise the Simulink model, then we extract clone pairs (*i. e.,* parts of the model that are equivalent), and finally we cluster those pairs to also find substructures used more than twice in the model.

Since the functionality of data-flow models as used in Simulink only depends on the link structure and the functionality of the basic blocks, they can be seen as labelled graphs from a functional point-of-view. Therefore our approach is described in a graph-based fashion.

4.1 Preprocessing and Normalisation

The preprocessing phase consists of reading the models, flattening them (*i. e.,* inlining all subsystems; the inverse function of *Create Subsystem*), and removing unconnected lines. This is followed by the normalisation which assigns to each block and line a label consisting of those attributes we consider relevant for differentiating them. Two blocks or lines are considered equivalent, if they have the same label.

Which information to include in the normalisation labels depends on which kind of clones should be found. For blocks usually at least the type of the block is included, while semantically irrelevant information, such as the name, the colour, or the layout position, are excluded. Additionally some of the block attributes are taken into account, *e. g.,* for the *RelationalOperator* block the value of the *Operator* attribute is included, as this decides whether the block performs a greater or less than comparison. For the lines we store the indices of the source and destination ports in the label, with some exceptions as, *e. g.,* for a *product* block the input ports do not have to be differentiated.

The result of these steps is a labelled model graph $G = (V, E, L)$ with the set of vertices (or nodes) V corresponding to the blocks, the directed edges $E \subset V \times V$ corresponding to the lines, and a labelling function $L : V \cup E \to N$ mapping nodes and edges to normalisation labels from some set N. As a Simulink block can have multiple ports, each of which can be connected to a line, G is a multi-graph. The ports are not modelled here but implicitly included in the normalisation labels of the lines.

For the simple model shown in Figure 1 the model graph would be the one in Figure 2. The nodes are labelled according to our normalisation function and the grey portions of the graph mark the part we would consider a clone.
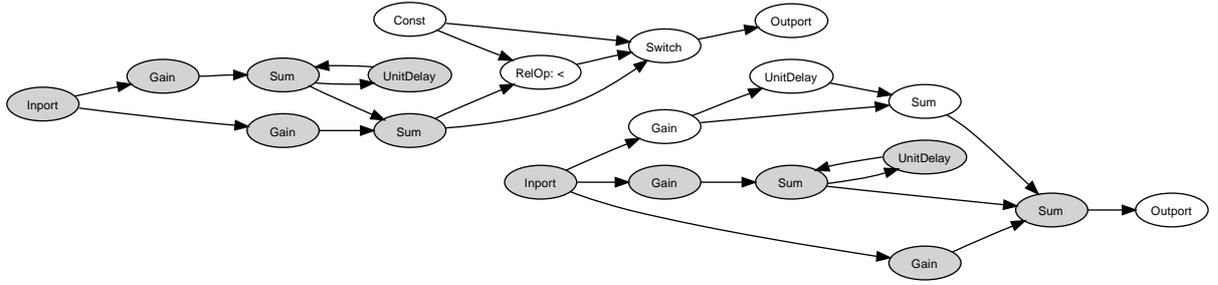
Figure 2: The model graph for our simple example model

4.2 Problem Definition

Having the normalised model graph $G = (V, E, L)$ from the previous section, we can now define what a clone pair is in our context. A *clone pair* is a pair of subgraphs (V_1, E_1), (V_2, E_2) with $V_1, V_2 \subset V$ and $E_1, E_2 \subset E$, such that the following conditions hold:

1. There are bijections $\iota_V : V_1 \to V_2$ and $\iota_E : E_1 \to E_2$, such that for each $v \in V_1$ it holds $L(v) = L(\iota_V(v))$ and for each $e = (x, y) \in E_1$ it is both $L(e) = L(\iota_E(e))$ and $(\iota_V(x), \iota_V(y)) = \iota_E(e)$.

2. $V_1 \cap V_2 = \emptyset$

3. The graph (V_1, E_1) is connected.

For $V_1, V_2 \subset V$, we say that they are in a *cloning relationship*, iff there are $E_1, E_2 \subset E$ such that $(V_1, E_1), (V_2, E_2)$ is a clone pair.

The first condition of the definition just states that those subgraphs must be isomorphic regarding to the labels L, the second one rules out overlapping clones, and the last one ensures we are not finding only unconnected blocks distributed arbitrarily through the model. Note that we do not require them to be complete subgraphs (*i. e.,* contain all induced lines).

We denote by the *size* of the clone pair the number of nodes in V_1. Then the goal is to find all maximal clone pairs, *i. e.,* all such pairs which are not contained in any other pair of greater size.

While this problem seems to be similar to the well-known NP-complete *Maximum Common Subgraph (MCS)* problem (also called *Largest Common Subgraph* in [8]), it is slightly different in that we only deal with one graph (while MCS looks for subgraphs in two different graphs) and we do not only want to find the largest subgraph, but all maximal ones. However, as the problem is structurally similar, the algorithms used for this problem (see, *e. g.,* [4]) are a good starting point.

4.3 Detecting Clone Pairs

As already discussed, the problem of finding the largest clone pair is NP-complete[2], *i. e.,* we cannot expect to find an efficient (polynomial time) algorithm for enumerating all maximal ones which we could use for models containing thousands of blocks. So we developed a heuristic approach for finding clone pairs which is presented next.

---

[2]We omit the simple reduction from the MCS problem for brevity here.

We give an outline of our algorithm in Figure 3. It basically consists of iterating over all possible pairings of nodes and proceeding in a breadth-first-search (BFS) manner from there (lines 4-12). During this we manage the sets C of current node pairs in the clone, S of nodes seen in the current BFS, and D of node pairs we are completely done with.

Line 9, which is optional, skips the currently built clone pair, if we find a pair of nodes we have already seen before. This was introduced as we found that clones reported this way are often quite similar to others already found (although with different "extensions") and thus rather tend to clutter the output.

The main difference between our heuristic and an exhaustive search (such as the backtracking approach given in [24]) is that in line 7 we only inspect one possible mapping of the nodes' neighbourhoods to each other. To find all clone pairs we would have to inspect all possible mappings and perform backtracking. Obviously even using only two different mappings quickly leads to an exponential time algorithm, which will not be capable of handling thousands of nodes.

Thus for a pair of nodes (u, v) we only consider one mapping P of their adjacent blocks. Obviously all block pairs (x, y) of P must fulfil the following two conditions:

$$L(x) = L(y) \tag{1}$$

$$\begin{array}{c} (u, x), (v, y) \in E \quad \text{and} \quad L((u, x)) = L((v, y)) \\ \textbf{or} \\ (x, u), (y, v) \in E \quad \text{and} \quad L((x, u)) = L((y, v)) \end{array} \tag{2}$$

As we are only looking at a single assignment out of many, it is important to choose the "right" one. This is accomplished by a *similarity function* which is described in the following section.

4.4 The Similarity Function

The idea of the *similarity function* $\sigma : V \times V \to [0, 1]$ is to have a measure for the structural similarity of two nodes which not only looks at the normalisation labels, but also the neighbourhood of the nodes. We use the similarity at two places. First we visit the node pairs in the main loop in the order of decreasing similarity, as a high σ value is more likely to yield a "good" clone. The more important place is in line 7, where we try to build pairs with a high similarity value. This is a weighted bipartite matching with σ as weight, which can be solved in polynomial time [26].

For this we define for nodes u, v a function $s_i(u, v)$, which intuitively captures the structural similarity of all nodes

Input: Model graph $G = (V, E, L)$

```
1   D := ∅
2   for each (u, v) ∈ V × V with u ≠ v ∧ L(u) = L(v) do
3      if {u, v} ∉ D then
4         Queue Q := {(u, v)}, C := {(u, v)}, S := {u, v}
5         while Q ≠ ∅ do
6            dequeue pair (w, z) from Q
7            from the neighbourhood of (w, z) build a list of
             node pairs P for which the conditions (1,2) hold
8            for each (x, y) ∈ P do
9               if (x, y) ∈ D then continue with loop at line 2
10              if x ≠ y ∧ {x, y} ∩ S = ∅ then
11                 C := C ∪ {(x, y)},  S := S ∪ {x, y}
12                 enqueue (x, y) in Q
13         report node pairs in C as clone pair
14         D := D ∪ C
```

Figure 3: Heuristic for detecting clone pairs

which are reachable in exactly i steps, by

$$s_0(u,v) = \begin{cases} 1 & \text{if } L(u) = L(v) \\ 0 & \text{otherwise} \end{cases}$$

and

$$s_{i+1}(u,v) = \begin{cases} \frac{M_i(u,v)}{\max\{|N(u)|, |N(v)|\}} & \text{if } L(u) = L(v) \\ 0 & \text{otherwise} \end{cases}$$

where $N(u)$ denotes the set of nodes adjacent to u (its *neighbourhood*) and $M_i(u, v)$ is the weight of a maximal weighted matching between $N(u)$ and $N(v)$ using the weights provided by s_i and respecting conditions (1) and (2).

It is easily seen by induction that for every i and pair (u, v) it holds by induction that $0 \leq s_i(u, v) \leq 1$ and thus defining

$$\sigma(u,v) := \sum_{i=0}^{\infty} \frac{1}{2^i} s_i(u,v)$$

is valid as the expression converges to a value between 0 and 1. The weighting with $\frac{1}{2^i}$ makes nodes near to the pair (u, v) more relevant for the similarity. For practical applications only the first few terms of the sum have to be considered and the similarity for all pairs can be calculated using dynamic programming.

4.5   Clustering Clones

So far we only find clone pairs, thus a subgraph which is repeated n times will result in $n(n-1)/2$ clone pairs being reported. The clustering phase described in this section has the purpose of aggregating those pairs into a single clone class.

While it seems straightforward to generalise the definition of a clone pair to n pairs of nodes and edges to get the definition of a clone class, we felt this definition to be too restrictive. For an example consider clone pairs $(V_1, E_1), (V_2, E_2)$ and $(V_3, E_3), (V_2, E_4)$. Although there is a bijection between the nodes of V_1 and V_3 they are not necessarily clones of each other, as there might not be the required edges. However, we consider this relationship to be still relevant to be reported, as when looking for parts of the model to be included in a library the blocks corresponding to V_2 might be a good candidate, as it could potentially replace two other parts.

So instead of clustering clones by exact identity (including edges) which would miss many interesting cases differing
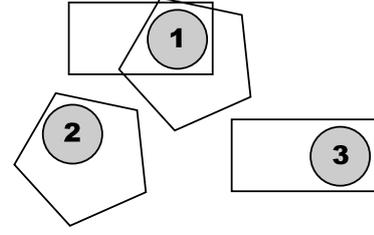

Figure 4: A partially hidden clone of cardinality 3

only in one or two edges, we perform clustering only on the sets of nodes. Obviously this is an overapproximation which can lead to clusters containing clones that are only weakly related. However, as we consider manual inspection of clones to be important for deciding how to deal with them, those cases (which are rare in practise) can be dealt with there.

Thus for a model graph $G = (V, E, L)$ we define a *clone class* of *cardinality* n to be a set $\{V_1, \ldots V_n\}$, such that for every $1 \leq i < j \leq n$ it is $V_i \subset V$ and there is a sequence $k_1, \ldots, k_m$ with $k_1 = i$, $k_m = j$, and V_{k_l} and $V_{k_{l+1}}$ are in a clone relationship for all $1 \leq l < m$ (*i.e.*, there is a *clone path* between any two clones). The *size* of the clone class is the size of the set V_1, *i.e.*, the number of nodes duplicated.

What this boils down to, is to have a graph whose vertices are the node sets of the clone pairs and the edges are induced by the cloning relationship between them. The clone classes are then the connected components, which are easily found using standard graph traversal algorithms, or a union-find structure (see, *e.g.,* [5]) which allows the connected components to be built on-line, *i.e.*, while clone pairs are being reported, without building an explicit graph representation.

Though this seems simple enough, there are still two issues to be considered. One is that while we defined clone pairs to be non-overlapping, clone classes can potentially contain overlapping block sets. This does not have to be a problem, especially as examples for this are rather artificial, but should be kept in mind. The other issue is the existence of clone classes which are not completely found due to larger clone pairs hiding some of the smaller ones. An example of this can be found in Figure 4, where equal parts of the model (and their overlapping) are indicated by geometric figures. The goal would be to find the clone class of cardinality 3 shown as circles. However, as the clone pair detection finds maximal clones, when starting from nodes in circles 1 and 2, actually the clone pairs consisting of the pentagons will be found. Similarly the circle pair 1 and 3 is hidden by the rectangle. So our pair detection would report the rectangle pair, the pentagon pair, and the circles 2 and 3.

We deal with this issue by checking the inclusion relationship between the clone pairs reported as a final step. In the example this would discover that the nodes from circle 2 are completely contained in one of the pentagons and thus there has to be a clone of this circle in the other pentagon, too. Using this information (which analogously holds for the rectangle) we can also find the third circle to get a clone class of cardinality 3. Obviously, if there was an additional clone overlapping circles 2 and 3, we had no single clone pair of the circle clone class and thus this approach does not work for this case. However, we consider this case to be unlikely enough to ignore it.

4.6 Scalability

The time and space requirements for the clone pair detection are depending quadratically on the overall number of blocks in the model(s). While for the running time this might be acceptable (though not optimal) as we can just let the program work in batch mode, the amount of required memory can be too much to even handle some thousand blocks.

A simple solution to this is to split the model graph into its connected components. Then we can independently find clone pairs within each such component and between each pair of connected components, which clearly still allows us to find all clone pairs we would find without this technique. While this does not help anything in terms of running time, as still each pair of blocks is looked at (although we might gain something by filtering out components being smaller than the minimal clone size we are interested in), the amount of memory needed now depends quadratically only on the size of the largest connected component. So if the model is composed of unconnected sub models (as is the case with the model used in Section 5) or we can split the model into smaller parts by some other heuristic (*e. g.,* separating subsystems on the topmost level), memory is no longer the limiting factor.

5. CASE STUDY

This section describes the case study, that was carried out with a German truck and bus manufacturer to evaluate the applicability and usefulness of our approach.

5.1 Analysed Model

We performed our experiments on a model provided by MAN Nutzfahrzeuge Group, which is a German-based international supplier of commercial vehicles and transport systems, mainly trucks and buses. It has over 34,000 employees world-wide of which 150 work on electronics and software development. Hence, the focus is on embedded systems in the automotive domain.

The organisation's development process is supported by an integrated data backbone developed on the eASEE framework from Vector Consulting GmbH. On top of this backbone, a complete model-based development approach has been established using the tool chain of Matlab/Simulink and Stateflow as modelling and simulation environment and TargetLink of dSpace as C-code generator.

The analysed model implements the major part of the functionality of the power train management system, deployed to one ECU. It is heavily parametrised to allow its adaption to different variants of trucks and buses. The model consists of more than 20,000 TargetLink blocks, which are distributed over 71 Simulink files. Such a file is the typical development/modelling unit for Simulink/TargetLink models.

5.2 Implementation

For performing practical evaluations of the detection algorithm, we implemented it as a part of the quality analysis framework ConQAT [6] which is publicly available as open source software[3]. This includes a Java-based parser for the Simulink model file format which makes our tool independent of the Simulink application. Additionally, we developed
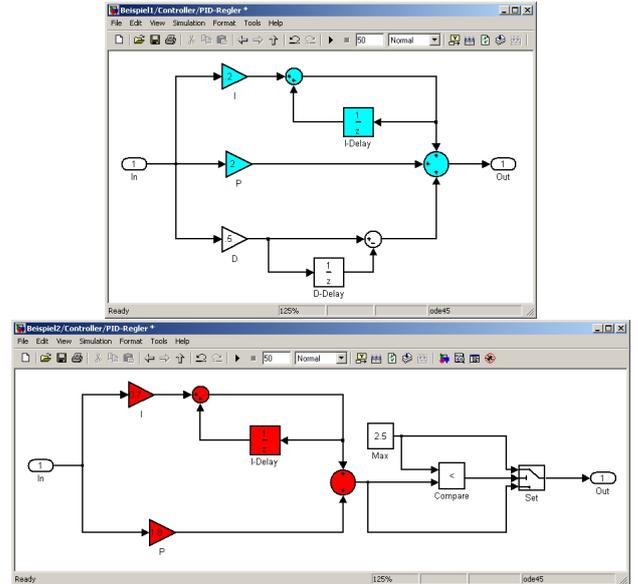
---

[3] http://conqat.cs.tum.edu/

Figure 5: An example for the visualisation of clones within Matlab

preprocessing facilities for the TargetLink-specific information and for *flattening* the Simulink models by removing *subsystems* that induce the models' hierarchy. To review the detection results we extended ConQAT with functionality to layout the detected clones and to visualise them within the Simulink environment that developers are familiar with. The later is done by generating a Matlab script which assigns different colours to the blocks of each clone. An example of this is shown in Figure 5.

5.3 Application

To be applicable to real-world models, the general approach described in Section 4 had to be slightly adapted and extended. In this section we provide the details that have been omitted until now.

For the normalisation labels we basically used the type, and for some of the blocks which implement several similar functions added the value of the attribute which distinguishes these functions (*e. g.,* for the *Trigonometry* block this would be an attribute deciding between sine, cosine, and tangent). Numeric values, such as the multiplicative constant for *gain*, were not included in the normalisation, *i. e.,* were ignored for clone detection, as we were interested in partial models which could be extracted as library blocks where such constants could be made parameters of the new library block. Overall we rather included less information into the normalisation labels to avoid loosing potentially interesting clones.

From the clones found, we discarded all those consisting of less than 5 blocks, as this is the smallest amount we still consider to be relevant at least in some cases. As this still yielded many clones consisting solely of "infrastructure blocks", such as *terminators* and *multiplexers*, we implemented a weighting scheme. This assigned each block type a weight, with a default of 1. Those infrastructure blocks were assigned a weight of 0, while blocks having actually a functional meaning (*e. g.,* integration or delay blocks) were
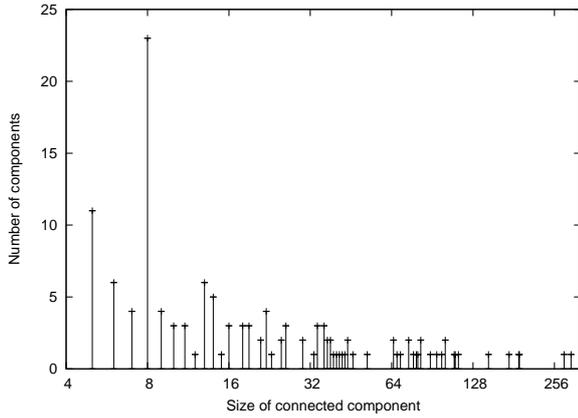
Figure 6: Size distribution of connected components in the model (logarithmically scaled in the x-axis)

Table 1: Number of Files/Modelling Units the Clone Classes were Affecting

Number of models	Number of clone classes
1	43
2	81
3	12
4	3

Table 2: Number of Clone Classes for Clone Class Cardinality

Cardinality of clone class	Number of clone classes
2	108
3	20
4	10
5	1

weighted with 3. The weight of a clone then is defined as the sum of the weights of its blocks. Clones with a weight less than 8 also were discarded, which ensures that at least small clones are considered only, if their functional portion is large enough.

5.4 Results

Our implementation of the detection algorithm needed 50 seconds on an Intel Pentium 4 3.0 GHz workstation with 1 GB of main memory for parsing the models from Section 5.1 and performing the detection and clustering of clones. About 125 MB of memory was used by the Java virtual machine during this process, from which the major amount was used for our rather verbose in-memory representation of the Simulink model and the infrastructure code of ConQAT.

From the 20454 blocks nearly 9000 were removed during flattening the model (4314 Inports, 3199 Outports, 1412 Subsystems). The model then consisted of 3538 connected components of which 3403 could be skipped as they consisted of less than 5 blocks[4]. Finally the clone detection was run on 4762 blocks contained in 135 connected components. The distribution of component size is shown in Figure 6, with the largest component having less than 300 blocks. This shows that processing these components separately definitely can help in handling large models, by keeping the memory consumption low as outlined in Section 4.6. Additionally, it seems that a size of 300 is about the number of blocks required for modelling a single function, while those smaller fragments are a side effect of splitting the system across multiple models. When the models are connected, these smaller parts are likely to be connected to other parts of the model.

We found 166 clone pairs in the models which resulted in 139 clone classes after clustering and resolving inclusion structures. Of the 4762 blocks used for the clone detection 1780 were included in at least one clone (coverage of about 37%). As shown in Table 1, only about 25% of the clones were within one modelling unit (*i. e.,* a single Simulink file), which was to be expected as such clones are more likely to be found in a manual review process as opposed to clones be-

---

[4]The large quantity of small components is due to comments in the model (which are themselves blocks) and buttons containing macros for printing and similar functionality, which were included in many subsystems.

tween modelling units, which would require both units to be reviewed by the same person within a small time frame. Table 3 gives an overview on the cardinality of the clone classes found. As mostly pairs were found, this indicates that the clustering phase is not (yet) so important. However, from our experience with source code clones and legacy systems, we would expect these numbers to slightly shift when the model grows larger and older.

Table 3 shows how many clones have been found for some size ranges. The largest clone had a size of 101 and a weight of 70. Smaller clones are obviously more frequent, which is because smaller parts of the model with a more limited functionality are more likely to be useful at other places.

5.5 Discussion

Our results clearly indicate that our approach is capable of detecting clones and a manual inspection of the clones showed, that many of the clones are actually relevant for practical purposes. Besides the "normal" clones, which at least should be documented to make sure that bugs are always fixed in both places, we also found two models which were nearly entirely identical. Additionally some of the clones are candidates for the project's library, as they included functionality that is likely to be useful elsewhere. We even found clones in the library (which was included for the analysis), indicating that developers rebuilt functionality contained in the library they were not aware of. Another source of clones is the limitation of TargetLink that scaling (*i. e.,* the mapping to concrete data types) cannot be parameterised, which leaves duplication as the only way for obtaining different scalings.

The main problem we encountered is the large number of *false positives* as more than half of the clones found are obviously clones according to our definition but would not be considered relevant by a developer (*e. g.,* large Mux/Demux constructs). While weighting the clones was a major step

Table 3: Number of Clone Classes for Clone Size

Clone size	Number of clones
5 − 10	76
11 − 15	35
16 − 20	17
> 20	11

in improving this ratio (without weighting there were about five times as many clones, but mostly consisting of irrelevant constructs) this still is a major area of potential improvement for the usability of our approach.

6. FUTURE WORK

As the application of clone detection on model-based development has not been studied before, there is a wide field of possible further research questions. One major direction consists of improving the algorithm and the general techniques and ideas involved. The other area complementing this is to have larger case studies or to apply the algorithm to related problems to get a better understanding of its strengths and weaknesses and its general usefulness.

6.1 Algorithmic Improvements

The most obvious source for improvement are the algorithms for detecting clone pairs and for clustering them to clone classes. While we doubt that there is an efficient exact algorithm for the former problem, the heuristic might still have room for improvement. Since the implementation is publicly available, a comparison of new heuristics with ours should be easy.

Maybe there are also radically different approaches to clone detection in models. For example the separation into a pair detection and a clustering could be avoided. At least for the MCS problem there are algorithms for finding the MCS for n graphs [3], although the transfer to our problem of finding clones *efficiently* is not obvious.

Another interesting problem would be to find *approximate clones, i. e.,* parts of the model which are slightly different. A major part of this would include to define reasonable similarity metrics on clones (*e. g.,* number of edge insertions).

However, the main line of improvement in our opinion is in the area of processing and categorising the detected clones. Currently many of the clones found are not interesting for the developer as they do not carry enough semantical meaning, although they are of course clones according to our definition. While some of this can be handled by fine-tuning the normalisation and weighting, it might be helpful to include some graph theoretic ideas. So a clone might be more relevant if it contains a cycle, which usually indicates some kind of control loop. This will be part of some larger case study as indicated in the next section.

6.2 Applications of Clone Detection

Although we consider our case study to be large enough to evaluate our approach and prove its usefulness, we would like to apply it to even larger models, maybe from other vendors in the automotive domain, which would allow some degree of comparison, or maybe from other domains. With more practical experience with the algorithm we expect to be able to answer many of the questions from the previous section more precisely.

Another interesting direction is the application of clone detection with a specific goal in mind, such as finding candidates for a library or finding clones in a library, where a developer rebuilt existing functionality. One could also use it the other way round and build a library of *anti-patterns* which includes common but discouraged model constructs (such as cascading switch blocks). Clones into these patterns then could indicate potential defects in the model.

A different application would be to aid in building product lines. Both product lines and model-based development are commonly used or introduced in the industry. Using clone detection on models of different products could help in deciding whether making a product line out of them is a good idea, and in identifying the common parts of these models.

Finally we would like to apply clone detection not only to Simulink models, but other kinds of models, too. As the algorithm itself is only based on graph theory, most of the adjustments for adaptation to other models are in the parsing and preprocessing phase, including normalisation. Especially for other data-flow based models, as found for example in the development of digital signal processors, or similar formalisms, such as models of business processes, the transfer should be easy to make. However, a long-term goal of course is the application to completely different models (*e. g.,* state machines) which are likely to also need a somewhat different algorithm.

7. RELATED WORK

In this section we discuss related work starting with other approaches in the area of clone detection on models. Then we cover the more thoroughly studied problem of clone detection in source code, to understand which of the methods proposed in this field could be applied to arbitrary models. We conclude with a short overview on related graph theoretic problems.

7.1 Model-Based Clone Detection

Up to now, little work has been done on clone detection in model-based development. In [21], Liu et. al. propose a suffix-tree based algorithm for clone detection in UML sequence diagrams. They exploit the fact that parallelism-free sequence diagrams can be linearised in a canonical fashion, since a unique topological order for them exists. This way, they effectively reduce the problem of finding common subgraphs to the simpler problem of finding common substrings. However, since a unique, similarity preserving topological order cannot be established for Matlab/Simulink models, their approach is not applicable to our case.

A problem which could be considered as the dual of the clone detection problem is described by Kelter et. al. in [13] where they try to identify the differences between UML models (usually applied to different versions of a single model). In their approach they rely on calculating pairs of matching elements (*i. e.,* classes, operations, etc.) based on heuristics including the similarity of names, and exploiting the fact that UML is represented as a rooted tree in the XMI used as storage format, making it inappropriate for our context.

7.2 Code-Based Clone Detection

A large body of research targets the detection of clones in source code. For the sake of brevity and comparison with graph-based clone detection as proposed in this paper, we classify them by the type of algorithm used to find similar code fragments. Please refer to [16] for a comprehensive survey of code clone detection.

Sequence-based clone detection transforms the source code into a sequence of normalised units. Clone detection is then performed by searching for common substrings, which can be efficiently done using suffix trees. Baker [1] and Kamiya et al. [11] proposed approaches employing tokens as units, Koschke et al. [17] serialise abstract syntax trees by pre-order traversal. In general, sequence-based tech-

niques require a similarity-preserving serialisation, which is not known for general graphs, such as Matlab/Simulink models.

By *characteristic-vector-based* clone detection approaches we summarise those that partition source code into atomic code fragments, assign vectors to them by some characterising function and use a distance metric in vector space to determine their similarity. Baxter et al. [2] partition the abstract syntax tree into subtrees and compute characteristic values for them using a suitable hash function. Mayrand et al. [23] use functions[5] as atomic units and compute several software metrics to yield the characteristic vectors. Jiang et al. [9] compute characteristic vectors on abstract syntax trees based on tree patterns. A crucial factor influencing the results of characteristic-vector-based approaches is the choice of the partitioning, since these approaches do not find clones that start in one partition and end in another partition. A suitable partition for Matlab/Simulink models would be to compare only single subsystems with each other. However, as we are also interested in clones distributed over multiple subsystems or covering only parts of a subsystem, we could not choose this approach. Also other graph theoretic partitions of the model, such as strongly connected components, will not find all clones considered important.

Graph based clone detection uses graphs as program representation containing data- and control-flow information. Clone detection is then performed by searching for similar subgraphs. Among the code-based clone detection approaches, these bear the closest resemblance to the model-based clone detection approach presented in this paper. In [15], Komondoor and Horwitz propose a combination of forward and backward program slicing to identify isomorphic subgraphs in a program dependence graph. Their approach is difficult to adapt to Matlab/Simulink models, since their application of slicing to identify similar subgraphs is very specific to program dependence graphs. In [18], Krinke also proposes an approach that searches for similar subgraphs in program dependence graphs. Since the search algorithm does not rely on any program dependence graph specific properties, it is in principle also applicable to model-based clone detection. However, Krinke employs a rather relaxed notion of similarity that is not sensitive to topological differences between subgraphs. Since topology plays a crucial role in data-flow languages, we consider this approach to be sub-optimal for Matlab/Simulink models.

7.3 Graph Theory

As we boiled down the problem of detecting clones to a purely graph theoretic problem in Section 4.2, we will also briefly cover related work in graph theory here. Probably the most similar problem to ours, as discussed before, is the well known NP-complete *Maximum Common Subgraph* problem. An overview of algorithms is presented by Bunke et. al. [4]. Most practical applications of this problem seem to be studied in chemoinformatics [28], where it is used to find similarities between molecules. However, while typical molecules considered there have up to about 100 atoms, many Matlab/Simulink models consist of thousands of blocks and thus make the application of exact algorithms as applied in chemoinformatics infeasible. Furthermore our problem is slightly different as discussed in Section 4.2.

---

[5]in the sense of programming language constructs

8. CONCLUSIONS

Model-based development is more and more becoming a routinely used technique in the engineering of software in embedded systems. Especially in some parts of the automotive domain, generating production code from domain-specific models is a common practise. As these models are more abstract than previously used C code, they provide various advantages in productivity and quality. However, also such models, especially when employed to generate production code, grow large and complex just like classical code. Therefore, classical quality issues can also appear in model-based development. A highly problematic and well-recognised problem is that of clones, i.e., redundant code elements.

So far, no approach or tool for clone analysis of models has been developed. Considering the massive impact of clones on quality and maintenance productivity, this is an unsatisfying situation. Moreover, we are in a unique position w.r.t. model-based development. We have the opportunity to introduce clone detection early in the development of large systems and product lines. In systems developed using classical code approaches, clone analysis often results in tremendously high amounts of clones so that there are economical and psychological barriers preventing their significant reduction. In model-based development, these large systems – and especially product-lines – are now emerging. Clone detection for models, if applied early, can prevent the development of these overwhelming numbers of clones. This shows also the two main uses of a clone detection approach for models: (1) redundant parts can be identified that might have to be changed accordingly when one of them is changed and (2) common parts can be identified in order to place them in a library and for product-line development.

We propose an approach containing an algorithm and a corresponding tool that can identify clones on models based on graph structures, including a weight-based filtering heuristic that allows to reduce the output to relevant clones. The algorithm is scalable enough to be able to handle models that are common in industry now. The approach was applied in an industrial case study with MAN Nutzfahrzeuge using their Matlab/Simulink/TargetLink models. In the case study, we are able to show that the approach can analyse industrial size models (20,000 elements) and clones can be found in such models for both purposes described above. 139 clone classes were found that showed in manual inspection of a sample of those clones, that a significant share of them are relevant for the MAN engineers. These clones can now be more closely inspected to decide how to deal with them in the future.

We see this approach and the application demonstration at MAN as an important first step for clone detection in models. The results can obviously be improved by fine-tuning the tools and algorithms based on further case studies. Nevertheless, the results should be encouraging to take advantage of the current situation in model-based development where a lot of redundancy can be avoided by using this kind of approach early on.

9. REFERENCES

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering.* IEEE Computer Society, 1995.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 1998.

[3] D. M. Bayada, R. W. Simpson, A. P. Johnson, and C. Laurenço. An algorithm for the multiple common subgraph problem. *Journal of Chemical Information and Computer Sciences*, 32:680–685, 1992.

[4] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and M. Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In *Joint IAPR International Workshops SSPR 2002 and SPR 2002*, volume 2396 of *Lecture Notes in Computer Science*, pages 123–132. Springer, 2002.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company, 2nd edition, 2001.

[6] F. Deissenboeck, M. Pizka, and T. Seifert. Tool support for continuous quality assessment. In *Proc. 13th IEEE Int. Workshop on Software Technology and Engineering Practice*. IEEE Computer Society, 2005.

[7] dSpace GmbH. TargetLink Production Code Generation. www.dspace.de.

[8] M. R. Garey and D. S. Johnson. *Computers and intractability. A guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th international conference on Software engineering*, 2007.

[10] M. Jungmann, R. Otterbach, and M. Beine. Development of Safety-Critical Software Using Automatic Code Generation. In *Proceedings of SAE World Congress*, 2004.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28:654–670, 2002.

[12] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28. IEEE Computer Society, 2006.

[13] U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for UML models. In *Software Engineering*, volume 64 of *LNI*, pages 105–116, 2005.

[14] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92. IEEE Computer Society, 2004.

[15] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. Springer, 2001.

[16] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings, 2007.

[17] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 253–262, 2006.

[18] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301. IEEE Computer Society, 2001.

[19] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance*, 1997.

[20] E. C. Lingxiao Jiang, Zhendong Su. Context-based detection of clone-related bugs. In *ESEC/FSE 2007*, 2007.

[21] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 269–276. IEEE Computer Society, 2006.

[22] The MathWorks Inc. *SIMULINK Model-Based and System-Based Design - Using Simulink*, 2002.

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

[24] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software – Practice and Experience*, 12:23–34, 1982.

[25] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *METRICS '02: Proceedings of the 8th International Symposium on Software Metrics*, page 87. IEEE Computer Society, 2002.

[26] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: Algorithms and complexity.* Prentice-Hall, 1982.

[27] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner. Software Engineering for Automotive Systems: A Roadmap. In L. Briand and A. Wolf, editors, *Future of Software Engineering 2007*, pages 55–71. IEEE Computer Society, 2007.

[28] J. W. Raymond and P. Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002.