Model Clone Detection in Practice

Florian Deissenboeck, Benjamin Hummel
Elmar Juergens, Michael Pfaehler
Technische Universität München
Garching b. München, Germany

Bernhard Schaetz
fortiss gGmbH
München, Germany

ABSTRACT

Cloned code is considered harmful for two reasons: (1) multiple, possibly unnecessary, duplicates of code increase maintenance costs and, (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior. Likewise, duplicated parts of models are problematic in model-based development. Recently, we and other authors proposed multiple approaches to automatically identify duplicates in graphical models. While it has been demonstrated that these approaches work in principal, a number of challenges remain for application in industrial practice. Based on an industrial case study undertaken with the BMW Group, this paper details on these challenges and presents solutions to the most pressing ones, namely scalability and relevance of the results. Moreover, we present tool support that eases the evaluation of detection results and thereby helps to make clone detection a standard technique in model-based quality assurance.

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, Experimentation

Keywords

Clone Detection, Data-Flow Models

1. INTRODUCTION

In many application domains for embedded software systems, model-based development—the specification of the functionality of the software using (graphical) models and the automatic generation of production code from these models—is a state-of-the-art technique. For applications focusing on control theoretic functionality, *e. g.*, in the domain of avionic or automotive systems, data-flow like models as used by Matlab/Simulink or ASCET SD are applied. Due to the size of these models (10,000 and more elements) and their longevity, problems like cloning commonly found with code-oriented development are also becoming relevant for these forms of models.

Cloning—code fragments that are similar *w. r. t.* to some definition of similarity—is known to hamper productivity of software maintenance in code-based development environments. This is because changes to cloned code are costly as they often need to be carried out multiple times for all (potentially unknown) instances of a clone [4, 12]. Moreover, inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior [2].

Obviously, the same problems also occur for duplicated parts of models in model-based development. Additionally, model cloning plays a specific and important role in this application domain since product line oriented domains like automotive software engineering excessively rely on model reuse. Hence, the identification and elimination of model clones is not only important to improve the maintainability of the system under development. Rather, the identification of reusable pieces of functionality and their integration in a library is a core requirement for the construction of product lines. Moreover, the identification of duplicated model elements can help to reduce program size which is beneficial in a domain where limiting the required hardware resources is still a central objective.

Research Problem. Previous experiences concerning the use of clone detection in a model-based development process have shown that the practical application of model-clone detection is not a straight-forward task [1]: besides dealing with large-scale models and providing a suitable notion of similarity that also covers model parts with slight modifications or variation, a pragmatic approach to deal with inevitable false positives is necessary. Since control-theoretic data-flow models are constructed from a small set of basic elements, the identification of similar parts within these models in general leads to a large number of clones. However, only a small fraction of these clones are relevant *w. r. t.* maintainability or reuse issues. Therefore, the treatment of large-scale models, the identification of sufficiently similar clones, and the extraction of relevant clones are decisive challenges for the practical application of model-based clone detection.

Contribution & Outline. Based on the experiences we made together with the BMW Group, this paper illustrates

IWSC'10, May 08, 2010, Cape Town, South Africa.

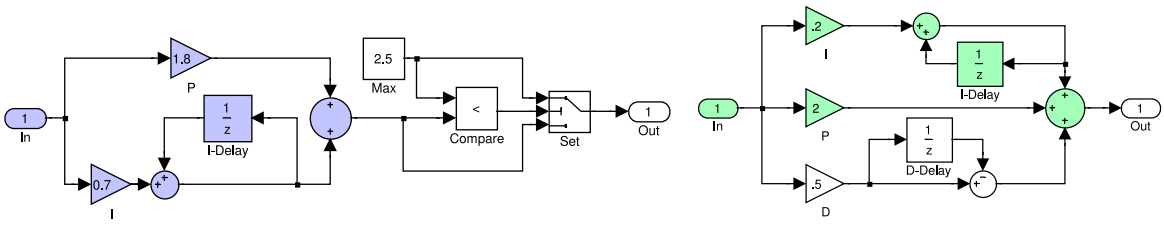

Figure 1: Example: Clone between discrete saturated PI-controller and PID-controller

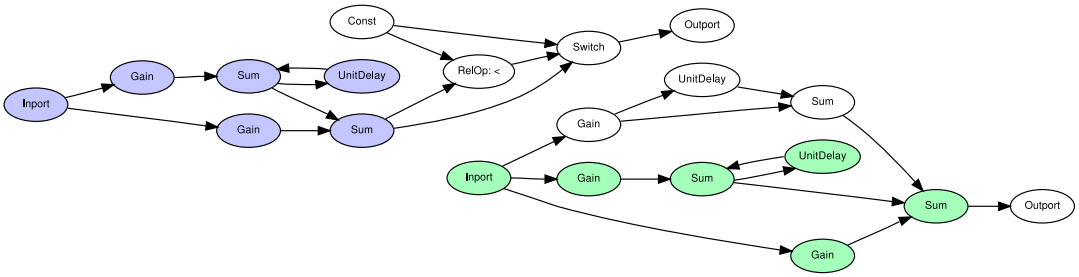

Figure 2: The model graph for our simple example model

which challenges arise when model clone detection is applied in an industrial context (Sec. 3) and explains how they can be addressed. To demonstrate that scalability is an utmost requirement for the application of model clone detection in practice, we compare the performance characteristics of existing detection algorithms in Sec. 4. In Sec. 5, we explain why detection approaches must satisfy more than a theoretical notion of completeness to be useful in practice. To address this, we present a set of metrics that helps to identify clones that are truly relevant for developers. In Sec. 6, we introduce adequate tool support that helps to inspect clone detection results. This is required to evaluate the relevance of detected clones and to deduce suitable measures.

2. MODEL CLONE DETECTION

Model clone detection deals with the identification of duplicated parts in models. More specific, in this paper we are interested in cloning within Matlab/Simulink models. An example of a very simple Simulink model of two controllers can be found in Figure 1. The shapes (squares, triangles, circles) are called *blocks* and are connected with *lines* to indicate the flow of data. Each block calculates a certain function on its inputs. The triangles (*gain*) multiply with a constant, the circles (*add*) sum up their inputs, and the squares have different meaning depending on their icon (constant value, unit delay, comparison, selection).

The two colored parts in Figure 1 are clones of each other. Such clones are usually created by a sequence of copy, paste, and modify steps, although the independent development of cloned parts may be expected in rare occasions. In the case of Simulink, instead of copying, a clone can also be introduced by inlining library elements. A more formal definition of a model clone in Simulink is *a connected submodel, which is structurally equivalent to another one, up to certain edit operations*. Structural equivalence means that the data-flow network is essentially the same, while the edit operations compensate for minor modifications of the copied submodel. In the example given before, the factors in the gain blocks are changed between the clones, which would be considered an edit operation. The actual detection process is usually organized in three major phases which are described next.

2.1 Preprocessing and Normalization

The first phases extracts the model data from the Simulink files, for which we use our own library[1]. The result is a labeled directed multigraph, which is a normalized representation of the model. The graph for the initial example is shown in Figure 2. Every block corresponds to a node and each line results in a directed edge in the graph. Each node and edge carries a label which contains an abstraction of the local information of the corresponding Simulink element. For example, we still need to know the type of each block, so the labels of the nodes contain this type. However, we are not interested in block parameters that can be substituted by a parameter in a library element, such as the factor used in a gain block. The same holds for layout information, which is not considered.

This normalization phase is important, as the detection algorithm works on this graph representation. If the normalization discards too much information, the precision will be low (*i. e.*, many false positives will be reported). Contrary, discarding too little information will make it impossible to find many relevant clones which differ only in minor details (low recall).

2.2 Detection

The detection phase works on the labeled graph produced during the previous phase. As normalization compensates for editing steps, only exact duplicates have to be considered. The graph theoretic problem to solve is the localization of isomorphic connected subgraphs of certain size. The normalization labels attached to the graph have to be respected for the isomorphism. As described in [1], since the underlying decision problem is NP-complete, we cannot expect to find an efficient and complete algorithm for detection. The area of graph mining provides a set of algorithms for

[1]Available as open source at `http://conqat.cs.tum.edu/index.php/Simulink_Library`

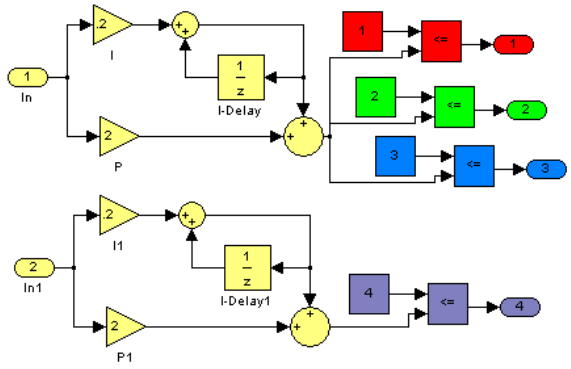

Figure 3: Ambiguous situation for clone definition

the problem, which is known as frequent subgraph mining there. However, as these general algorithms do not exploit specific properties of the normalized graph, they are usually too slow.

The algorithm proposed by us in the first paper on this topic [1] is a heuristic which is based on a depth first search for clone pairs. To avoid exponential running time, the search only inspects the single branch in the DFS which seems most promising. This trick reduces the worst-case running time to be quadratic in the size of the graph.

Follow-up work from other authors [7, 9] proposes the *eScan* algorithm that is based on graph mining algorithms and includes some tricks specific to Simulink models (detailed in Sec. 4). This way, potentially more clones can be identified. Interestingly, this work only attempts to improve the detection phase.

While it seems plausible that isomorphic subgraphs should be the result of this stage, there is no agreement yet on whether *all* isomorphic subgraphs or just a subset of them should be reported. A part of the problem is illustrated by the artificial model from Figure 3. The left (yellow) parts are obviously copied, as are the four block triplets (constant, comparator, output) on the right. For the complete lower part of the model there are actually three possible matches in the upper part of the model (including either the red, green, or blue triplet). One way to deal with this, is to report all four clone instances (each consisting of the *base clone* and one of the triplets) as one clone group. This, however, could confuse a developer as it includes overlapping clones which are often not meaningful to a developer. Another approach chosen by [9] is to report these clones as three individual groups, each consisting of two clones (the bottom one and one of the top ones). In this case a developer might be confused by seeing the (seemingly) same clone reported multiple times. A third solution is used in the algorithm from [1], where due to the branching limits only one clone pair would be reported consisting from the bottom part and one instance (non-deterministically chosen) from the upper part. While this way some information is lost, the most crucial information for the developer is still present.

2.3 Postprocessing

Postprocessing maps the subgraphs found during the detection phase back to the Simulink model. In the simplest case, all clones are just reported to the user in some way. However, this phase can also be used to order clones or discard some of them. The latter may be necessary, as detection is based purely on graph isomorphism without considering the meaning of a clone in the original model. While the detection phase only affects recall, the postprocessing phase cannot find new clones and thus only affects precision.

3. CHALLENGES IN PRACTICE

Our initial publication [1] as well as subsequent work by other authors showed that automatically detecting clones in models is possible in principle. Applying model clone detection in practice, however, poses the following challenges:

3.1 Scalability

Real-world Simulink models as they are, *e. g.*, used in the automotive domain are usually of significant size. For example, the model used in the case-study presented in [1] has about 20,000 blocks, the model that served the case study discussed in Sec. 5 of this paper consists of almost 100,000 blocks. As these numbers may appear exceptionally large, it must be noted that models of this size are often not stored within a single model file but spread over multiple units. As the case study in [1], however, showed that clones often occur across model files, clone detection must include multiple model files to be effective. Consequently, clone detection approaches must be capable of processing such models within reasonable time and memory limits. Sec. 4 demonstrates that scalability is, indeed, a challenge in practice as it shows that some existing algorithms are not capable of processing models of realistic size.

3.2 Relevance

From code clone detection we know that the relevance of an identified clone strongly depends on the task one intends to perform [13]. For example, an identified clone may be relevant for the task of change propagation, *i. e.*, the identification of duplicated models elements that need to be changed synchronously. However, the same clone may be irrelevant for the task of clone removal because the modeling language provides no suitable abstraction mechanisms to remove the clone. Hence, the relevance of clone detection results can only be judged if the intended task is known.

Notwithstanding the intended task, we found that the purely theoretical notion of similarity, that is based on the localization of isomorphic connected subgraphs, is not strong enough to identify clones that are relevant to the developers. Consequently, a large number of clones identified by all model clone detection approaches was considered irrelevant by developers. Put differently, many clones identified by the clone detection algorithms are considered *false positives* by the developers, *i. e.*, the algorithms exhibit a low detection *precision*. To improve precision, filtering or prioritization of the detected clones is paramount. Ideally, the applied mechanisms should be configurable to accommodate the task-dependency of clone relevance judgments.

3.3 Clone Inspection

Theoretically, a simple of list of found clones is enough to allow developers to locate the clones in the analyzed models. However, to apply model clone detection in practice, this form of output is too crude as opening the affected models and navigating to the cloned parts of the models is tedious and time-consuming. Moreover, the development environments (IDEs) used to create the models are not equally appropriate for inspecting clone detection results. This is mainly because the IDEs display hierarchical model layers

Model	Original Size # Nodes	# Edges	After Reduction # Nodes	# Edges
SIM	428	415	387	379
MUL	475	576	450	432
SEM	1741	2029	710	704
ECW	2312	2274	1315	984
MPC	369	395	294	317
AUT	98251	90056	66944	66026

Table 1: Sizes of the Simulink models before and after the removal of duplicate subsystems

in separate windows whereas clones often consist of model parts that span multiple hierarchical layers and should be viewed in their entirety. As a result, we found that developers spent considerable time only for inspecting the clone detection results before they could even start to remove the clones. We need tools that support the developer in inspecting clones by automatically opening the affected model files and presenting the clone in a way that allows them to quickly judge the relevance of a clone. In particular, such tools must highlight the commonalities and differences between the individual clone instances.

4. SCALING TO REAL-WORLD MODELS

As discussed before, finding all model clones is NP-hard and thus we do not expect to find an efficient algorithm that solves the model clone detection problem exactly while still scaling well to models of realistic sizes. This is confirmed by our experiments with existing algorithms from graph mining. An existing implementation of the *gSpan* algorithm taken from [14] either timed out or ran out of memory for even small model sizes. The reason might be that graph mining usually attempts to find very frequent subgraphs (not just two occurrences) in a large set of disconnected graphs (and not in a single graph).

In the next sections we will discuss techniques which can be used to speed up clone detection, possibly at the cost of reduced recall. This is mostly a comparison of the approach presented by Deissenboeck et al. [1] and Pham et al.'s *eScan* algorithm [9][2]. The main contribution here is a comparison of performance improvements and especially the evaluation of their performance on a real world model.

For evaluation, we used 5 models, which are publicly accessible on MATLAB Central[3]: A Simulink model for a communications lab (SIM), a simulation of multiple unmanned air vehicles (MUL), a video surveillance system (SEM), an echo canceler model (ECW) used to introduce beginners to modeling in Simulink, and a gas separation plant model (MPC). To ease comparison of our performance measurements, the first four models are the ones also used by Pham et al. in [9]. Additionally, we included a model being developed at BMW from the automotive domain (AUT) as an example of a real-world model. The sizes of these models are summarized in the left columns on Table 1.

[2]These two algorithms were chosen as they represent the state-of-the-art in model clone detection. Another algorithm proposed by Pham et al., *aScan*, is not included here as its principal characteristics w.r.t. scalability are comparable

[3]`http://www.mathworks.com/matlabcentral/`

4.1 Removal of Cloned Subsystems

A typical approach to make an algorithm work on large inputs, is to reduce the input size in a preprocessing step. For clone detection, one possible preprocessing is the removal of "obvious" clones. For example, in code-based clone detection, completely identical files obviously are clones of each other, and as checking for duplicate files initially can be performed efficiently, these duplicates could be resolved.

Simulink models are hierarchically structured into so called *subsystems*, which are blocks constructed from other blocks. As subsystems are likely candidates for copy&paste operations, preprocessing looks for duplicated subsystems first. As in this case the subgraph is already fixed, instead of subgraph isomorphism the slightly easier graph isomorphism problem has to be solved. Duplicated subsystems are then reported as clones and all instances of the cloned subsystem are removed but one. This was first proposed by Pham et al. in [9], where a subsystem's name is used to identify cloned subsystems. As identical names do not guarantee identical content, we instead identify duplicate subsystems by structural comparison based on a canonical graph labeling.

In our experiments, model sizes could be substantially reduced if only one instance of a cloned subsystem is considered during clone detection. The model sizes after removal are listed on the right columns of Table 1. A drawback of this approach is that clones spanning these removed subsystems are no longer detected, so the recall is affected negatively. However, we expect reported clones to be manually inspected prior to removal, so if the subsystems are also embedded in a similar context, this should be noted.

4.2 Removal of High-Degree Nodes

Pham et al. present another strategy in [9]: As the performance of eScan deteriorates especially in the presence of nodes with high degree, they remove all nodes with a high degree from the model graph before clone detection starts. After the clone detection step is finished, the algorithm tries to assemble bigger clones by connecting smaller clones that are connected to each other over high degree nodes. While the clone detection phase benefits from this step, this new assembly phase becomes substantially more complex.

This two-phase approach may miss certain clones if they are composed from special patterns of low- and high-degree nodes (examples are provided in [8]). These patterns, however, are rather artificial and not expected to occur often in real-world models. Still, the overall speed-up obtained by the removal of the high-degree nodes is counteracted by the second assembly phase. In extreme cases, a model might consist mostly of such high-degree nodes, causing the assembly phase to be not easier at all. In the model of our industry partner (AUT), about $\frac{1}{5}$ of all nodes have a degree higher than 6. These nodes are connected to $\frac{1}{3}$ of all edges. While the two phase approach discussed here may help in reducing the required detection time for this model, the runtimes given later indicate that this alone does not make the model solvable.

4.3 Reducing Branching

One of the standard approaches for solving NP-hard problems is to use a branching strategy which explores the entire solution space. Usually, the solution space is traversed in a tree-like fashion by constructing larger solutions from smaller ones. This tree is typically searched depth first to minimize memory consumption. The run-time of this

search depends on both the depth of the tree and its average branching factor, but is exponential in any case where the branching factor is more than 1. One way to speed up a branching algorithm is to bound the branching factor.

A conservative way is used in [9], where the so called *generating parent test* ensures that no subgraph is encountered multiple times in the search tree. This is conservative, as no solutions are lost, however, the overall complexity still remains exponential. A more effective strategy is used in ConQAT's algorithm [1], where in any case only a single branch is explored. The only exception is the root (or seed) clone pair consisting of two identical nodes, for which all possible combinations are tried. The selection of the branch to be explored is based on a similarity metric of the graph's nodes which also incorporates the structure and similarity of its neighborhood. With this branching factor of 1 a polynomial complexity can be achieved which allows the algorithm to scale to large model sizes. However, the heuristic nature of the branch selection also causes certain clones to be missed and, hence, possibly leads to a lower recall.

4.4 Run-Time Comparison

To evaluate the effectiveness of the different optimization strategies, we applied both the eScan algorithm from [9] and ConQAT's algorithm [1] to the instances from Table 1 and combined them with the optimizations suggested in Sections 4.1 to 4.3. For eScan we had to use our own implementation of the algorithm, as the authors of [9] did not provide their implementation upon request. While this slightly threatens our comparison as our implementation might be flawed, we are confident in the reimplementation as we could reproduce the numbers of clones reported in their paper and have even lower run-times, which could be due to a more efficient calculation of canonical labels. For ConQAT's algorithm we used our own implementation, which is also available as Open Source[4].

The run-times are listed in Table 2. The optimization from Sec. 4.3 (branching reduction) refers to the heuristic used in ConQAT and not the generating parent scheme which is part of eScan. Removal of high-degree nodes (4.2) is only listed for eScan, as an inclusion into ConQAT is not possible without serious changes to the algorithm. The last row reproduces the numbers from [9] for the first 4 models, which corresponds to an eScan with optimizations 4.1 and 4.2, for comparison. The table shows that especially for eScan, the removal of duplicated subsystems (4.1) is essential to reduce the models to a solvable size. However, even with all optimizations eScan can solve only 3 of our 6 models[5]. So it seems that for models of realistic size, the heuristic used in ConQAT is essential, as an approach with exponential complexity does not scale well. Still, the ideas from [9] helped in reducing the running times of ConQAT's algorithm nearly by a factor of 2.

[4] `http://www.conqat.org/`

[5] The reason that contrary to [9] the model SEM could not be solved within time is likely that [9] applies subsystem removal based on naming equivalence. While we consider this an invalid optimization, as subsystems with the same name may have different content, we rerun this model using *name-based* subsystem removal. With both name-based subsystem removal and removal of high-degree nodes, SEM could be processed within 25 seconds. This is probably because the graph's size after subsystem removal then is 569 nodes (compared to 710 nodes for content-based removal).

4.5 Discussion

The performance-related results show that it is unavoidable to resort to heuristic approaches, that are necessarily incomplete, when large models need to be processed. It remains unclear, however, which consequences the incompleteness of these approaches has for the detection recall, *i. e.*, how many relevant clones are identified. Obviously, knowledge of the total set of clones is required to calculate the recall of a detection algorithm. As explained in Sec. 2, however, this set is unknown as it is unclear what the "correct" result of a model clone detection algorithm is. Based on the experiences made in the context of code clone detection we, furthermore, have to assume that there is no single correct result as developers judge the relevance of a clone depending on the task they intend to perform [13]. Consequently, it is not possible to quantitatively compare the recall of the detection algorithms discussed above. Moreover, our experience is that the insufficient precision (high number of false positives) of existing approaches is the greater obstacle for an application of model clone detection in practice. Hence, the following sections presents metrics that help to increase detection precision by filtering irrelevant clones.

5. IMPROVING RELEVANCE OF CLONES

To reduce the number of false positives, metrics that prioritize clones *w. r. t.* their relevance can be applied for filtering. In [1] we used two simple metrics to sort clones, namely the size of the clone in nodes (denoted by S) and its weight (denoted by W). The weight of a clone is the sum of the weights of the blocks it contains, where the block weight can be configured based on the block type. This is used to prefer clones with complex blocks (*e. g.*, integrators) over those consisting mostly from rather simple blocks (*e. g.*, terminators or multiplexers) by assigning a higher relative weight to them. Both of these measures can be used for filtering, by requiring a certain threshold for each clone. While this can help in removing extremely small and uninteresting clones, many clones still remain, which are considered irrelevant by developers.

5.1 Metrics for Clone Relevance

In concordance with [13], interviews with the developers revealed that the false positives were considered irrelevant as they were not suited for specific task the developers had in mind. As in our case the removal of clones was the task that was important to our industrial partners, the following discussions of relevance assumes that a clone is relevant if it is worthwhile to remove it by consolidating it as a library block. An example of a clone that is considered a false positive *w. r. t.* this definition of relevance is a clone that almost exclusively consist of building blocks of low semantic significance, *e. g.*, *mux/demux* blocks, in comparison to "weightier" blocks like a *discrete integrator*. In the remainder of this section we propose a set of metrics which can be used to rank model clones according to their relevance, and summarize a small case study which compares the obtained rankings with those of two developers.

Relative weight RW**.** From an economical perspective the time a developer spends on the inspection of a clone is a valuable asset. We assume that the time required for a developer to understand the meaning of a cloned model graph depends on the number of nodes within the clone. Therefore, if two clones have the same weight, the clone

Base Algorithm	Optimization			Run-time for Instance					
	4.1	4.2	4.3	SIM	MUL	SEM	ECW	MPC	AUT
eScan	–	–	–	5	–	–	–	–	–
eScan	✓	–	–	4	6	–	784	–	–
eScan	–	✓	–	9	–	–	–	–	–
eScan	✓	✓	–	8	9	–	72	–	–
ConQAT's algorithm	–	–	✓	3	4	14	20	5	3283
ConQAT's algorithm	✓	–	✓	3	3	7	8	6	1888
Run-times of eScan published in [9]				120	132	300	612	n.a.	n.a.

Table 2: Run-time comparisons of different detection algorithms. Run-times are in seconds, the dash indicates that the algorithm either ran out of memory or did not terminate within 24 hours.

consisting of fewer nodes should be ranked higher than the clone that achieved the weight with more nodes. Therefore, the relative weight metric RW relates the weight W of a clone to its node size S, so $RW = \frac{W}{S}$. This value is the same for all clones in a clone group. For the relative weight, larger values are considered more relevant.

Interface size metrics I^W and I^S. While the relative weight was motivated by the required inspection time, the interface size metric is motivated mostly by the clone removal itself. The effort required to externalize a part of a model into a library is affected by its interface complexity. In case of Simulink, the interface complexity is captured quite well by the number of lines connecting the submodel with its surrounding model. Potentially, clones having a smaller interface size are better candidates to form a library element, as their externalization is likely to require less effort.

We refer to the number of edges in the model graph that connect nodes within a clone instance to nodes outside the clone instance as the *interface size* of a clone instance. The interface size is not necessarily equal for all clone instances within a clone group. Thus, for a clone group CG (*i. e.*, a set of clones), we use the *average interface size* $\text{aifs}(CG)$ defined as $\frac{\sum_{c \in CG} \text{ifs}(c)}{|CG|}$, where $\text{ifs}(c)$ denotes the interface size of a given clone.

As larger submodels can be expected to have a larger interface, we do not use the interface size directly as a metric, but set it in relation to the size of the clone. This results in two metrics depending on whether size is measured in number of nodes S or clone weight W. The interface weight metric I^W relates the average number of interface edges to the weight of the clone $I^W = \frac{\text{aifs}(CG)}{W}$, while the interface node size metric I^S relates the average number of interface edges to the number of nodes within a clone $I^S = \frac{\text{aifs}(CG)}{S}$. Contrary to the relative weight, the interface size metrics should be minimized to support clone removal.

5.2 Case Study

In order to evaluate the described ranking metrics, we performed a small case study with our automotive industry partner BMW.

Design. We selected a set of 21 clone groups, each consisting of two clone instances, from the clones of the model AUT introduced before. These clone groups were chosen manually to cover a large spectrum of structural features and especially cover a large range of different values for the metrics presented before. For this set, two domain experts (developers) from BMW were asked to rank the clone groups *w. r. t.* their appropriateness for extraction as a library element. We did not require a total order among the groups, such that the developers could assign the same position within the ranking to two or more clone groups.

Clone group metric	Deviation to Developer 1	Deviation to Developer 2
Node size S	139	163
Clone weight W	157	183
Relative weight RW	110	90
Interface weight I^W	86	90
Interface node size I^S	123	123

Table 3: Ranking deviations between different metrics and the developer rankings

To analyze the appropriateness of existing and our new clone metrics, we compared the developers' ranking to the ranking based on the clone metrics. For each developer and metric, we calculated for each clone group the difference between the developer's rank and the rank based on the metric and summed up the absolute value. Let $R^d(CG)$ denote the rank that developer d assigned for clone group CG and $R^m(CG)$ the rank a clone group CG has in the ranking that is based on the metric m. We then calculated for each developer and metric the deviation as

$$D_{d,m} = \sum_{CG \in \text{CloneGroups}} |R^d(CG) - R^m(CG)| \ .$$

We used a fractional ranking for equal values, *i. e.*, the rank of clone groups that compare equal is the average of their ordinal rankings. The value $D_{d,m}$ then is used to assess the similarity of the rankings, where a value of 0 would indicate two identical rankings. For 21 clones the worst possible value is 220, while the average over all possible permutations is about 146.

Results. For each of the metrics discussed before, we ranked the clones accordingly (increasing or decreasing depends on the metric) and calculated the deviation to both of the developers' rankings. These deviations are listed in Table 3. For the interpretation of these numbers it is worth noting that the rankings of both developers deviated from each other by a value of 74. This matter is related to reports of code clone detection studies, mentioned in [13], in which developers could rarely agree upon which clones ought to be removed.

The first two rows of the table show that ordering the clones by node size or weight is not better on average that just using a random ordering of clones. Obviously the relative weight and both interface metrics offer a better approx-

imation of the developers' rankings than the metrics used beforehand. With a deviation of 86 respectively 90, the interface weight metric even comes close to the deviations in between the developers' rankings.

Threats to validity. In the study we only used clone groups containing exactly two clone instances. Potentially, the developers' rankings might be different for clone groups containing three or more instances. However, in the model of our industry partner, more than 70% of the identified clones contain only two clone instances, so the considered case seems to be the most common.

The manual selection of the clone groups as well as the limitation to clones from one single industrial model can bias our results. Additionally, the number of developers interviewed and the number of clones studied may be too small to generalize our results. Thus, these results should only be seen as a first step towards a better understanding of clone relevance in model-based development.

6. CLONE INSPECTION

In the simplest case, a model clone detection algorithm reports its detection results as a list of fully qualified names of the affected model elements. To inspect a clone group and assess its relevance and opportunities for consolidation, a developer needs to inspect all involved clone instances and understand their commonalities and differences.

To do so, a developer first needs to open the models that contain the clone instances. Second, the cloned model parts need to be located in their models. In an industrial-size model, this can involve a significant number of individual navigation steps. Moreover, it is hard to understand the extent of a clone without dedicated visualization as clone detection algorithms report the individual affected model parts. In practice, clone location is further challenged since cloned model parts can be layouted differently in the involved models and can cross subsystem boundaries. In these cases, the involved navigation steps between parent and child subsystems further complicate clone location, since the full extent of a clone cannot be viewed in a single diagram.

This manual process of clone comparison requires substantial mental effort. Consequently, from our experience, concentration noticeably drops after around one hour of clone inspection. Furthermore, inspection of a single clone group in this fashion can take up to several minutes, due to the large amount of required navigation steps and switches between models. Considering the hundreds or even thousands of clone groups typically detected for industrial size systems [1], this is prohibitively expensive. Furthermore, this tediousness of clone inspection can cause failure of adoption of clone detection by developers, since payoffs of clone detection and removal occur too late or are at least perceived as such.

A lot of the effort involved in clone inspection is of accidental nature. It can thus substantially be reduced with suitable tool support. Our tool named *Model Quality Assessor* provides such support. It allows to execute the model clone detection within an integrated environment that is also used for visualizing the detection results. When a developer inspects a clone group, the Model Quality Assessor automatically opens the involved models. Within them, it highlights the clone instances, thus making their location and extent immediately visible—the number of required navigation steps and model switches is thus substantially reduced. Furthermore, the Model Quality Assessor implements a so called fish-eye visualization technique [10, 11] that enables the display of clones that cross subsystem boundaries in a single diagram, as depicted in Figure 4. From our experiences from model clone detection in industry, the use of the Model Quality Assessor increases the productivity of clone inspection significantly. We consider the availability of such tool support to be a crucial factor in the application of model clone detection in industry.

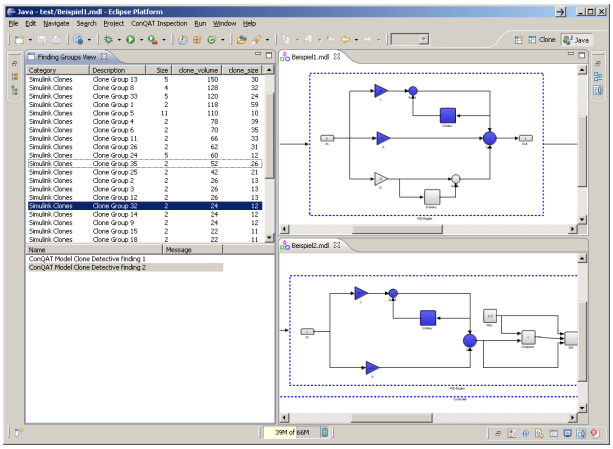

Figure 4: Screenshot of the tool used for model clone inspection

7. RELATED WORK

This section summarizes existing work dealing with the detection of clones in data-flow models, as well as with the more general topics of finding clones in general models and identifying isomorphic subgraphs. This paper itself is based on results from [8].

7.1 Clone Detection in Data-Flow Models

The first work describing clone detection in data-flow models is [1], where the problem and a first heuristic detection algorithm are presented. Furthermore, the full detection pipeline and a case study for the special case of MATLAB/Simulink models are presented. The paper also noted the need for further improvements in the precision, as more than half of the clones found where false positives, although they were clones according to the graph-theoretic definition.

Based on this work, Pham et al. [7, 9] propose improved algorithms for the detection phase based on existing graph theoretic algorithms. While these algorithms improve the recall of the detector, the papers do not contribute to the problem of low precision.

7.2 Clone Detection in General Models

The promising results in clone detection for data-flow models raise the question to what extent cloning does exist and can be detected in other kinds of models. This becomes increasingly relevant due to the gradual shift from writing code to drawing models in the context of model-based development and (partial) code generation. Relevant models include control-flow models (*i. e.*, variants of state machines) and structural models, such as class diagrams. However, so far there is not even a suitable definition of what constitutes a clone in such models.

The only application to other models is [6], where an algorithm for clone detection in UML sequence diagrams is proposed. The approach works by linearizing parallelism-free sequence diagrams, which can then be searched by algorithms for common sub-string detection.

7.3 Code-Based Clone Detection

Algorithms from code clone detection are only of minor interest for model clone detection, as they usually work on either a linear text or token stream or on the tree structured AST of the code, which both are not transferable to general directed graphs. The only exception are algorithms working on a graph representation of the code, based on the control-flow and/or data-flow relation between code statements.

In [3] a combination of forward and backward program slicing is used to identify isomorphic subgraphs in a program dependence graph (PDG). However, their strong exploitation of PDGs makes the approach hard to transfer to general (directed) graphs. Contrary, the algorithm from [5] does not have such limitations. However, the rather relaxed notion of similarity used in this approach that is not sensitive to topological differences between subgraphs, makes the application to data-flow models doubtful, since topology plays a crucial role there.

The problem of eliminating clones, which adhere to the clone definition but are not relevant to a certain task or developer is also highly relevant in code clone detection, as it is a major ingredient in increasing precision. As this task depends to a large extent on the semantics of the cloned portion, results in this area are hard to transfer from code to data-flow models.

7.4 Graph Mining

The problem of finding clones in data-flow models can be reduced to mining frequent subgraphs after the model has been normalized to a labeled graph. An overview and comparison of algorithms for subgraph mining is given by [14]. As described before, these algorithms strive for an exact solution and often target either smaller graphs or subgraphs with much higher frequency than 2. Thus, most of these algorithms do not scale to the graph sizes required for clone detection in real-world models. Additionally, as these algorithms work on the normalized graph, they can not contribute to the improvement in precision we are looking for. However, they provide a valuable foundation in the development of novel algorithms for model clone detection.

8. CONCLUSION

The principle applicability of clone-detection in model-based development has already been shown in earlier work. However, for successful application in industry, practical questions (*e. g.*, detecting reusable clones in large-scale models) rather than fundamental issues (*e. g.*, detecting only exact or maximal clones) have to be addressed for successful adoption. In this paper, techniques have been presented to improve *scalability* by an adapted subsystem detection, to improve *relevance* of detected by clones by providing use-case specific rankings, and finally tool-support to ease *inspection* of the instances of the detected clones.

While the presented approach has demonstrated the practical relevance of clone-detection in model-based development, there are several issues requiring research to further enhance the successful detection of model-clones; e.g., the use of semantic normalization to detect models with equivalent behavior but different syntactical structure, the distributed detection of clones to improve scalability, or the detection of inconsistent clones supporting the inclusion of cloned model-parts with small local changes. Finally, the detection approach can also be extended to other, less data-flow oriented forms of models, e.g., state-oriented models like State Charts or StateFlow, or process-oriented models like BPEL or ARIS, however, requiring adapted definitions of similarity as well as means of normalizing models and ranking clones.

9. REFERENCES

[1] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE'08*, 2008.

[2] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE'09*, 2009.

[3] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS'01*, 2001.

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

[5] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE'01*, 2001.

[6] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *APSEC'06*, 2006.

[7] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *FASE'09*, 2009.

[8] M. Pfähler. Improving clone detection for models. Master's thesis, Technische Universität München, 2009.

[9] 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 *ICSE'09*, 2009.

[10] T. Reinhard, S. Meier, and M. Glinz. An improved fisheye zoom algorithm for visualizing and editing hierarchical models. In *REV'07*, 2007.

[11] T. Reinhard, S. Meier, R. Stoiber, C. Cramer, and M. Glinz. Tool support for the navigation in graphical models. In *ICSE '08*, 2008.

[12] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical report, Queen's University, Canada, 2007.

[13] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *WCRE'03*, 2003.

[14] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In *PKDD'05*, 2005.