Index-Based Model Clone Detection

Benjamin Hummel Elmar Juergens Daniela Steidl
Technische Universität München
Garching b. München, Germany
{hummelb,juergens,steidl}@in.tum.de

ABSTRACT

Existing algorithms for model clone detection operate in batch mode. Consequently, if a small part of a large model changes during maintenance, the entire detection needs to be recomputed to produce updated cloning information. Since this can take several hours, the lack of incremental detection algorithms hinders clone management, which requires up-to-date cloning information. In this paper we present an index-based algorithm for model clone detection that is incremental and distributable. We present a case study that demonstrates its capabilities, outline its current limitations and present directions for future work.

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, Measurement

Keywords

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

1. INTRODUCTION

Research in software maintenance has shown that many programs contain a significant amount of duplicated (cloned) code. Such cloned code can be problematic for two reasons: (1) multiple, possibly unnecessary, duplicates of code increase maintenance costs [17,24] and, (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior [1,14].

In response, a large number of clone detection approaches have been proposed that can uncover duplication in large scale software systems [17,24]. Their results are employed by clone visualization and management tools [4, 7, 12, 13, 15, 18, 23] to alleviate the negative consequences of cloning during maintenance of software that contains duplication.

Clone management tools rely on accurate cloning information to indicate cloning relationships in the IDE while developers maintain code. To remain useful, cloning information must be updated continuously as the software system under development evolves. To better support clone management, detection algorithms must operate incrementally to quickly produce results for large systems. Consequently, for *code*, several incremental clone detection algorithms have been proposed [3, 9, 11, 16, 20].

Cloning is not limited to source code but can occur in all software artifacts. In model-based development, models replace code as primary development and maintenance artifact. Previous work by us [5, 6] and others [21] has demonstrated that cloning also occurs—and needs to be managed—in Matlab/Simulink models.

In contrast to *code* clone detection, existing algorithms for *model* clone detection do not work incrementally. Instead, they operate in batch mode, *i. e.*, they read the entire software system and detect all clones in a single step. When the models change, the entire detection has to be performed again. In [5], we reported on detection times of more than one hour for a large model. If batch algorithms are used for clone management, they either have to be executed on-demand, or pre-computed results that are updated on a regular, *e. g.*, daily, basis have to be used. Both approaches cause substantial accidental complexity: on demand execution causes waiting times; pre-computed results can be outdated, causing invalid decisions or edits. This is complemented by the observation that a developer typically is not interested in knowing of all clones, but only of clones that affect the file or model part he is currently working on. These clones can be determined by extracting all clones and then filtering those that do not have at least one instance in the current model. However, for large models this can be inefficient. These limitations reduce the payoff achievable through the application of clone management tools and threatens their adoption by industry. The bigger the maintained model base—and thus the larger the need for automation of clone management—the greater the incurred accidental complexity, as detection times increase. We thus require incremental algorithms for model clone detection to better support model clone management.

Problem. Current model clone detection approaches are not incremental. Hence, they cannot be used for real-time detection in large systems, reducing their usefulness for clone management.

Contribution. This paper presents an algorithm for index-based model clone detection that is both incremental and distributable. The index can be updated incrementally, as the analyzed models evolve during maintenance. It can be combined with existing model clone detection algorithms to speed up their clone retrieval performance. We outline a case study that demonstrates its capabilities and limitations and shows its suitability for clone management.

2. TERMS

We define clones in data-flow models based on graph isomorphism. Two labeled graphs $G_1 = (V_1, E_1, L)$ and $G_2 = (V_2, E_2, L)$ with a labeling function L are *isomorphic* iff

1. there exists a bijection $f_V : V_1 \to V_2$ such that $(x, y) \in E_1 \iff (f_V(x), f_V(y)) \in E_2$ and for each $v \in V_1$ it holds $L(v) = L(f_V(v))$, and
2. there exists a bijection $f_E : E_1 \to E_2$ such that for each $e = (x, y) \in E_1$ it is both $L(e) = L(f_E(e))$ and $(f_V(x), f_V(y)) = f_E(e)$

For a graph, its *size* denotes the cardinality of the nodes set.

Graph isomorphism can be determined based on *canonical labeling*: A canonical label of a graph is a unique code that is invariant to the ordering of vertices and edges. Thereby, graphs that are not isomorphic, have different canonical labels. It follows directly from this definition that two graphs are isomorphic, iff they have the same canonical label.

We define a *clone pair* to consist of two weakly connected, directed, labeled multi-graphs G and G' with $G \neq G'$ that are isomorphic. Both graphs G and G' are called *clone instance*. The *size* of a clone instance, also referred to as *clone size*, denotes the cardinality of the vertices set.

A *clone group* is a set of graphs, in which any two graphs are a clone pair. The *clone group cardinality* denotes the cardinality of the graph set. A clone group is *disjunctive* if the node sets of any two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are disjunctive ($V_1 \cap V_2 = \emptyset$). A clone group M is *maximal* if there is no other clone group N such that each graph of M is subgraph of at least one graph in N.

Given a directed, labeled Graph $G = (V, E, L)$, the problem of *model clone detection* for a given, constant $k \in \mathbb{N}$ is equal to enumerating all maximal disjunctive clone groups where each clone instance has at least size k. For solution algorithms the size of k is crucial.

The problem of clone detection is NP-complete: the largest common subgraph isomorphism decision problem, which is known to be NP-complete [8], can be reduced to detecting clones. Hence we cannot expect to find the perfect solution (*i. e.*, all maximal clones) in polynomial time.

3. STATE OF THE ART

In this section we discuss the current state of the art of clone detection in graph-based environments. In contrast to existing algorithms, our algorithm is the first one which is able to perform incremental updates. Furthermore, we discuss parallels in our approach and an existing approach for clone detection in code-based environments.

3.1 Graph-based Clone Detection

While there are lots of papers on the general problem of identifying common or frequent subgraphs in general graphs, such as [2, 10, 22], only two approaches are published on the specifics of clone detection in Simulink models, which we outline next.

Heuristic Approach. In [6], we presented a heuristic approach that enumerates maximal clones in a graph extracted from Matlab/Simulink models. To obtain this graph, we use preprocessing and normalization, which are also used in this approach. The algorithm enumerates all disjunctive clone pairs by iterating over all pairings of nodes and proceeds in a breadth-first-search manner from there. Instead of an exhaustive search, a heuristic is provided to reduce the runtime. The heuristic gives an estimation of the similarity of a pair of nodes that includes not only the normalization labels but also the structure of the neighborhood of both nodes. During the iteration over all pairings of nodes, the heuristic is used to quickly find other pairs of nodes that can be combined with the current pair of nodes to a clone pair. In a second step, clone pairs are combined to a clone class.

In contrast to our new algorithm, this algorithm does not work index-based. Therefore it is not able to perform incremental updates. If parts of the models change, the entire clone detection must be performed again. Furthermore, the clone detection cannot be distributed, but must be performed on one single machine.

Exact and Approximate Clone Detection. In [21], two different algorithms for clone detection in graphs are presented: eScan enumerates exact clones with the same definition of a clone pair as in Section 2; aScan focuses on detecting *approximate* clones. A definition of similarity is given by Exas, a vector-based representation and feature extraction method that can approximate the structure of a graph. Therefore each graph is assigned a characteristic vector. Similarity of graphs is measured by a distance function of the characteristic vectors.

Both algorithms are based on the observation that each clone pair with k edges can be derived from a clone pair with $k - 1$ edges by adding a single edge. eScan does this in a depth-first-search manner. A method of parent identification ensures that no subgraph of size k is created multiple times from different subgraphs of size $k - 1$. Cloned subgraphs are contained in layers of different subgraph sizes and grouped layer-by-layer into clone groups. In contrast to eScan, aScan performs a breadth first search. To reduce complexity of the algorithm, pruning techniques are applied.

Both algorithms start with subgraphs of size 1 and continuously create larger subgraphs. If parts of the model change, both algorithms need to be run again, because they cannot be updated incrementally. Furthermore, calculations of both algorithms cannot be distributed over several machines neither. In contrast, our algorithm is incremental and can be easily distributed.

3.2 Index-based Code Clone Detection

In this section we discuss an algorithm for clone detection in code-based environments as presented in [11], whose architecture is similar to our approach. This algorithm is both incremental and scalable to very large code bases. The central data structure of the detection phase is the *clone index* which contains a mapping from sequences of normalized statements to their occurrences. Each statement is assigned a sequence hash, a hash code using MD5 hashing for the next n normalized statements in the file starting from the statement index. If there are two entries in the index with the same hash sequence, a clone pair of size at least n was found. After index construction, a second phase, namely clone retrieval, reports only maximal clones.

This approach inspired the work in this paper. In section 4, we show that our algorithm also consists of the two phases index construction and clone retrieval. However, both algorithms are working on different underlying environments. Our algorithm, in contrast to code-based environments, works on graph-based data-flow models.

4. APPROACH

This section introduces a novel algorithm for clone detection in graph-based data-flow models, which is both scalable and incremental.

4.1 Architecture

We first give an overview of the architecture of the algorithm and then an outline of our approach.

Clone detection pipeline. The clone detection process comprises three phases, which are executed in a pipeline fashion. Each phase builds on the results of the previous phase [6].

Preprocessing reads files containing Simulink models. The hierarchical structure is flattened by inlining all subsystems and removing all unconnected lines. The normalization assigns a label to each block and line based on attributes relevant for their differentiation. This removes differences between blocks and lines that are not relevant for clone detection. The information included in the label depends on the type of clone that should be found. Preprocessing results in a directed, labeled multi-graph, where vertices correspond to nodes of the Simulink models and edges to lines.

Detection searches in the labeled graph for isomorphic subgraphs, clone groups. The result of detection is hence cloning information on the level of subgraphs.

Postprocessing creates cloning information on the level of Matlab/Simulink files from cloning information on the level of subgraphs. Furthermore, depending on the clone detection use case, detected clones can be filtered, presented in project dashboards or written to a report for interactive analysis in an IDE, or for use by clone management tools.

Our algorithm contributes only to the detection phase, which is also the bottleneck of the performance of the clone detection pipeline. Both preprocessing and postprocessing are not novel, but reused from our clone detector ConQAT[1].

Detection. Similar to [11], the detection phase consists of two parts: creation of the *clone index* and *clone retrieval*, as shown in Figure 1. The clone index is the central data-structure of the algorithm and allows easy look-up of clone pairs of clone size k. The index is a list of those subgraphs that were extracted from the Matlab/Simulink graph. All subgraphs in the clone index have size k. For each graph in the index, we compute the canonical label. If there are two graphs with the same canonical label, clone instances of size k were found. After the creation of the clone index, the clone retrieval process merges clone pairs with clone size k to maximal and disjunctive clone groups.

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

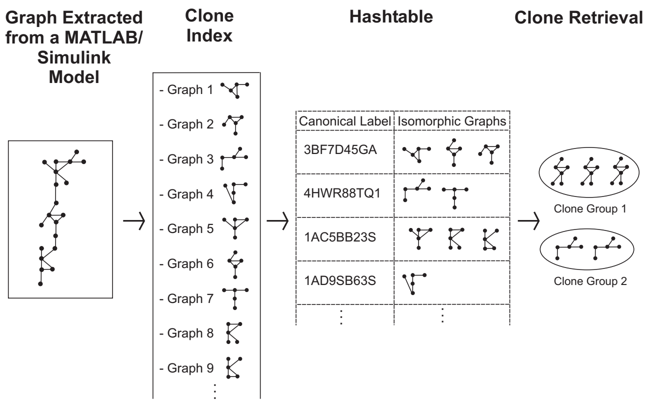

Figure 1: Overview of the detection phase: Creation of clone index and clone retrieval process

4.2 Creation of Clone Index

In this section, we explain how the central data structure of the algorithm, the *clone index* is created and maintained.

Enumerating subgraphs. To create the clone index, all subgraphs of size k of the input graph are enumerated. The algorithm works by starting from an arbitrary vertex v. For this vertex, all adjacent vertices are determined. For each possible subset of this neighborhood we recurse by again determining the adjacent vertices of the new subset, but thereby skipping all vertices that have been considered before (as either part of the graph or neighborhood). Each recursion repeats until the subgraph's size reaches k. After all recursions for v have finished, v is removed from the graph and we repeat the process for another vertex. A more detailed explanation of this algorithm is provided in [19].

Distribution. Clone index creation can be distributed over several machines. In our experience (*c.f.*, [6]) large industrial Simulink models consist of many files. The enumeration of subgraphs for each file is independent of the other files and can be executed on different machines, which operate on the same clone index.

Incremental updates. The clone index can also be updated incrementally. Since large models consist of several files, a typical change by the software developer affects only few files. Let each subgraph in the clone index have a reference to the file it was extracted from. Then only those subgraphs, belonging to the changed files are removed from the index. Subgraphs of the changed files are recalculated and added to the index. Thereby all other subgraphs remain unchanged. This incremental update saves waiting time compared to a scenario where the entire clone detection needs to be performed again.

Hashing with canonical labels. To find clone pairs in the clone index, the *canonical label* of each subgraph is calculated. We use a method to calculate canonical labels that can be found in [25]. While computation of canonical labels is expensive in general, we only apply it to graphs of size k, which allows us to keep the time complexity low. As the size of these labels can be large, we only store an MD5-hash to save space and comparison time. As there may be collisions (*i. e.*, different labels mapped to the same hash value), we have to check isomorphism after retrieving all subgraphs with the same hash value. Isomorphic subgraphs with the same hash value form a clone group. Hence clone groups of at least cardinality 2, with clone size k, can be extracted from the clone index.

4.3 Clone Retrieval

Clone retrieval returns all clones affecting a given model file from a clone index (of subgraph size k). With the clone index described before, this can be performed as follows. First, we extract all subgraphs and their hashes (of the canonical label) for the given file. Then, all subgraphs having the same label as one of the subgraphs returned in the first step are taken from the index. Any clone with a size of at least k must be contained in the graph formed by the nodes and edges of the extracted subgraphs. So, by performing model clone detection on the reduced graph, we can find exactly the clones affecting the given file.

For the detection of the clones in the extracted graph, we can use any detection algorithm, such as those from [6] or [21]. The rationale is that the graph formed by the subgraphs of the current model file and its duplicated subgraphs is significantly smaller than the overall model graph so that detection time is reduced.

4.4 Clone Merging

Instead of running a detection algorithm, we also can exploit the fact that the clone index already provides groups of isomorphic subgraphs (clones of size k) and construct larger clones from these. In this section we show how clone groups with a clone size k can be merged in order to obtain maximal clone groups with large clone sizes as defined in Section 2. Experimental results provided in Section 5, however, show room for improvement of the merging approach.

Clone group keys. To decide, if two clone groups can be merged to one group with a greater clone size, we introduce the concept of a *key*: two clone groups have a common key, if they can be merged. We call this the *merging property*.

For the definition of a key, we use *isomorphic node lists*: let c be a clone group of cardinality l and isomorphic graphs $G_1 = (V_1, E_1, L), G_2 = (V_2, E_2, L), \ldots, G_l = (V_l, E_l, L)$ where each graph has k nodes. Then a list of isomorphic nodes consists of l nodes $n_1, n_2, \ldots, n_l$ and $\forall i = 1 \ldots l : n_i \in V_i$ and all nodes are pairwise isomorphic. Thereby two nodes n_i and n_j are isomorphic under the bijection f_V (defining the isomorphism between the graphs) if $f_V(L(n_i)) = L(n_j)$ or $f_V(L(n_j)) = L(n_i)$. Hence clone group c defines k different isomorphic node lists, $list_1 \ldots list_k$.

A key K for a clone group c consists of l lists where each list contains $k-1$ nodes $n_1, \ldots, n_{k-1}$ and $\forall i = 1 \ldots k-1 : n_i \in list_i$. The size of a key denotes the number of nodes in each list, in this case $k-1$. It follows that there exists $\binom{k}{k-1} = k$ distinct keys for each clone group c. One possible key is visualized in Figure 2. If two clone groups have one common key, they can be merged[2]. The visualization of this process can be found in Figure 3.

[2]There are cases, where only one of the induced subgraphs/clone instances contains additional edges after merging. As we consider such pairs still relevant for a practitioner, we include this case in our clone definition.

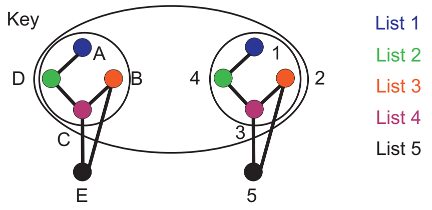

Figure 2: A clone group of cardinality 2, with clone size 5, and one out of 5 keys. The coloring of the nodes indicates the isomorphic node lists.

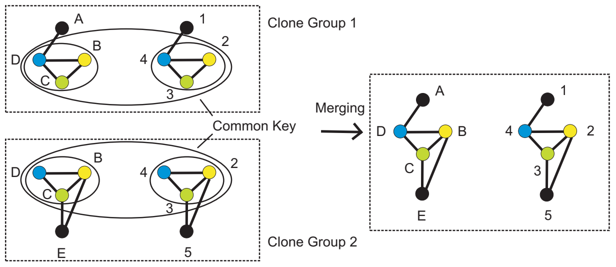

Figure 3: Two clone groups of cardinality 2, with clone size 4, and a common key can be merged to one clone group with clone size 5.

Defining a key of a clone group with any key size smaller than $k-1$ does not fulfill the merging property, because the graphs of the new clone group do not necessarily have the same size and are therefore not isomorphic. A specific example can be found in Figure 4. There the size of the subgraphs is $k = 4$, and the key size 2, which is smaller than $k-1$.

With the definition of a key as shown above, we are able to guarantee that two clone groups with clone size k can be merged, if they have a common key of size $k-1$. However, we can not guarantee that if two clone groups can be merged, the algorithm computes a common key. An example can be found in Figure 5. It shows two clone groups, where one clone group consists of regular graphs. Therefore neither the isomorphism bijection between the graphs is unique nor are the isomorphic node lists. Hence the necessary common key is not computed. The problem mainly occurs in regular graphs, but we expect graphs in real-world models to be sparse graphs due to the structure of Matlab/Simulink models. Hence the problem will only occur in few cases in practical applications.

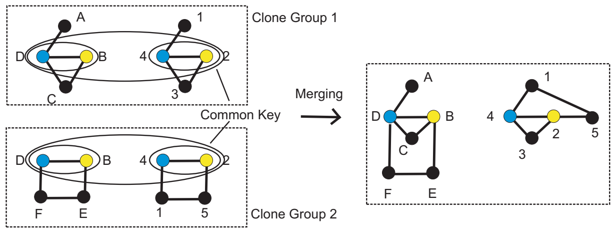

Figure 4: Merging two clone groups with a common key and a key size smaller than $k-1$ leads to a failure in the merging process.

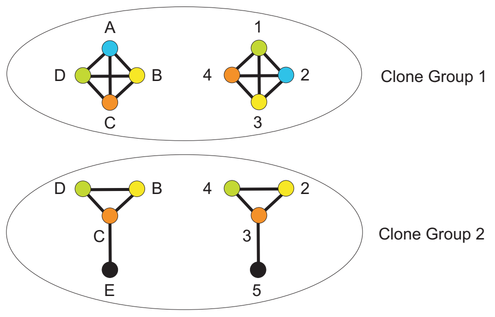

Figure 5: Two clone groups that can be merged, but no common key is found.

Merging process. After the keys are created, each clone group of clone size k, extracted from the clone index, is inserted into a hash table with each of its $k-1$ keys. The hash table provides easy access to clone groups that have a common key. Among all clone groups with a common key, we can only guarantee to merge the first two clone groups. After the merge, the clone size is at least $k+1$ and therefore any further merging process with the new clone group could fail due to the key size being too small.

To find maximal clone groups, we try to merge as many clone groups as possible. For this we iterate over the keys and for all clone groups with this key attempt to merge them. Merging a list of clone groups with a common key is performed by an attempt to merge the first group with each of the remaining ones, thus merging as many as possible. Then the (now possibly larger) clone group is finished and merging is repeated on the remaining list. After merging two clones, the size is larger than k. For performance reasons, we can not recalculate the keys. Instead the merged clone "inherits" the keys (of size $k-1$) of both clones. A slightly different strategy for the merging order is described in [25].

As the problem of clone detection is NP-complete, we know that our approach can not be expected to reliably find the best solution (*i. e.*, all maximal clones). So, our approach, just as the other published algorithms, is only a heuristic for providing a solution for the cloning problem.

5. EVALUATION

We provide results from a set of experiments. The goal is to better understand (1) the impact of the subgraph size k, (2) performance improvement gained by our index-based retrieval, and (3) the performance and results of the merging phase. All experiments are performed using our implementation in ConQAT, which provides the normalization pipeline and heuristic detection algorithm from [6].

5.1 Models

We used 5 models, available on Matlab Central[3], listed in the first 5 rows of Table 1. These models were also used in [5] and four of them are known from [21]. The columns of the table provide the number of Simulink blocks contained in the models (including subsystems) and the size of the graph resulting from preprocessing. As preprocessing inlines subsystems and their ports, the number of nodes can be significantly lower than the number of blocks.

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

		normalized graph	
model	# blocks	# nodes	#edges
SIM	480	474	441
MUL	695	475	576
SEM	2.631	1.560	2.029
ECW	3.707	2.312	2.274
MPC	440	356	422
BIG	18.262	11.523	10.877

Table 1: The models used in the evaluation

subgraph	Number of subgraph for model				
size	SIM	MUL	SEM	ECW	MPC
1	474	475	1560	2312	356
2	437	552	1929	2229	421
3	673	1196	5097	3620	1188
4	1021	3634	22320	6810	4190
5	1436	14369	128380	13383	14915
6	1821	67585	792303	26492	52265
7	2071	337166	4812052	54657	188600

Table 2: Number of connected subgraphs extracted

To have a larger model available, we also added the model called BIG in Table 1. This artificial model consists of all Simulink-Models we had access to at the time of writing this paper. Besides the 5 models mentioned before, it contains several models from academic and industrial partners, which we may not list for nondisclosure reasons. As we reported in [5] there are models with more than 100.000 blocks used in practice, but due to limited access, the model *BIG* is the best approximation we can offer.

5.2 Impact of Subgraph Size k on Number of Subgraphs

We determined the number of connected subgraphs that are stored in our clone index. This number depends both on the number of nodes of the input graph n and the size of the extracted subgraphs k. There are two theoretical bounds on the number of subgraphs of a connected graph. For a fully connected graph (a clique), we get $\binom{n}{k}$ connected subgraphs, which is asymptotically $O(n^k)$ if k is significantly smaller then n. On the other hand, if the graph is a single path (*i. e.*, has no loop and all but two nodes have degree 2) we get exactly $n-k+1$ connected subgraphs. Our goal here is to empirically find where typical models are in the spectrum between linear and exponential number of subgraphs.

The result for our models is shown in Table 2. Even for these rather small models the number of subgraphs explodes for small values of k. The same numbers are shown as a plot in Figure 6 with the y-axis scaled logarithmically. This plot shows that the number of subgraphs still grows exponentially even for our sparse graphs from Simulink. However, the numbers are significantly smaller than the worst case for a fully connected graph. Still, for larger industrial models with up to one million blocks, subgraphs larger than 4 are clearly not manageable for an index-based approach in terms of computation time and especially in terms of memory required to store the data.

5.3 Index-Based Clone Retrieval

To get an estimate of the efficiency of the retrieval process, we evaluated the performance when combining index-based

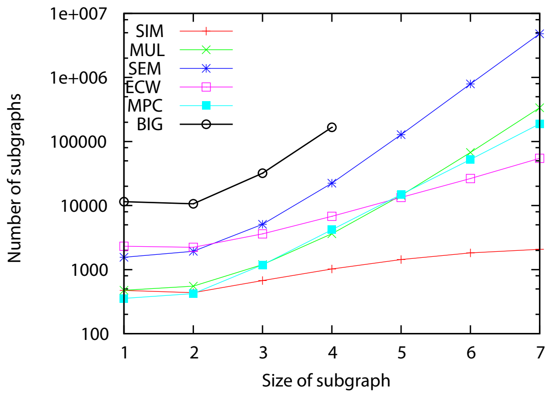

Figure 6: Number of connected subgraphs extracted

	remain. blocks	size reduction	index constr.	detec. time
No retrieval	11523	—	—	25s
Retrieval $k = 2$	663	17.4	4s	2s
Retrieval $k = 3$	457	25.2	11s	2s
Retrieval $k = 4$	298	38.7	60s	2s

Table 3: Results for index-based clone retrieval

retrieval with the branching heuristic from [6]. We selected one submodel M of *BIG* with 209 blocks, which is slightly larger than the average model size of 170 blocks and about the size of the model files encountered in the MAN study of [6]. For this model, we implemented clone retrieval by building the clone index and extracting only those groups of duplicated subgraphs with at least one subgraph being a part of M. On this reduced graph, we executed the branching heuristic from [6]. This setup ensures that all clones in the overall system that have one instance in M are reported.

The results for this setup are shown in Table 3 for different sizes of k. The first line provides the numbers for detection of the entire model for reference. The detection times reported here are the raw algorithm time without loading and parsing of the Simulink files and thus are slightly lower than the numbers reported before. The results show that using the index for retrieval of the relevant subgraph reduces the model size by a factor of 17 to 39 (calculated as overall block count divided by remaining block count). This is also reflected by the detection time that is reduced to about 2 seconds[4]. For an index of subgraph size $k = 2$ or $k = 3$ the retrieval-based preprocessing is faster than the detection on the full model even if the index has to be created from scratch. The ultimate goal of a clone detection tool for Simulink, however, is to keep the clone index in memory all the time and only update the changed model parts. In such a setup, the index creation time can be neglected.

5.4 Merging

In this section we evaluate the performance and result quality of our merging algorithm from Section 4.4. We compare the setup of creating a clone index for the entire model and merging these clone groups with the branching heuristic introduced in [6] by executing both on the model *BIG*. The index/merge-based approach was executed with different values for k (the size of subgraphs in the index). For the minimal clone size we chose 6 nodes and the minimal weight[5] was chosen as 9. In Table 4 we report the overall executions time, the coverage (*i. e.*, the relative amount of nodes covered by the reported clones), the average clone instance size (ACIS) in nodes, and the average clone group cardinality (ACGC) in clone instances.

	time	coverage	ACIS	ACGC
Branching heuristic	34s	42.5%	32.6	4.1
Index-based ($k = 2$)	11s	3.5%	11.2	6.3
Index-based ($k = 3$)	22s	16.5%	18.6	4.5
Index-based ($k = 4$)	189s	31.7%	20.3	5.3

Table 4: Direct comparison of detection algorithms

The index/merge approach is faster than the branching heuristic for $k = 2, 3$. However, the running time deteriorates for $k = 4$, and for $k = 5$ we aborted the detection run after several hours. It turns out that the coverage and ACIS is always lower for the index-based approach, although they improve with increasing values of k. This indicates that our greedy merging strategy does not lead to clones of at least the minimal size of 6 often enough[6]. Contrary, the values for ACGC are much higher for the index-based approach. An explanation is that the branching heuristic first looks for large clone pairs which are merged, while the index-based approach first creates groups of high cardinality that are merged to become large. Thus, the index-based approach favors group cardinality over instance size.

As a side note, for $k = 4$ the index creation phase alone took 67 seconds for generating 166,803 subgraphs. Thus, even if we could speed up merging infinitely, the algorithm would still take twice the time of the merging heuristic.

6. DISCUSSION

Our experiments show that the clone retrieval scenario, which is the original motivation for the index-based approach, can reduce detection times significantly. The measured times suggest that integration of index update and clone retrieval into a modeling tool as a frequent operation (*e. g.*, at saving) would be feasible. How this scales to even larger models is unknown, but we expect a large speed up also for those models. If k is 2 or 3, the index-based retrieval of the clones with a single model file is faster than the calculation of all clones even if we have to calculate the index as a first step. In a clone detection tool, however, the index should be either managed in memory or in a central server (similar to [16]) and thus index creation time can be neglected. Keeping the index up to date is possible, as after changing a file, only subgraphs belonging to that file have to be updated. Thus, refreshing the index is relatively cheap.

The results also show that the subgraph size for an index should be 2 or 3. Already for a value of 4 the number of subgraphs is too large, increasing both processing time and memory consumption too much to be useful for large models.

[4]For these small graph sizes our measuring setup was not precise enough to differentiate the time measures below one second.

[5]Node weight is determined in normalization. Details: [6].

[6]If we choose k the same as the minimal clone size, we trivially reach the maximal clone coverage possible, even if merging is not performed at all. However, typical values for the minimal clone size are too large to be used as k.

The algorithm comparison between our merge-based algorithm and older branching heuristic shows that the current implementation of the merging algorithm can not match the older heuristic in terms of performance and clone recall. Currently, it is better to apply an existing model clone detection algorithm on the reduced graph, although it can not reuse information from the preclustered subgraphs. The perfect solution of trying all possible merging orders is infeasible due to exponential running time. To remedy this, one could investigate heuristics that find a better merging order.

The index can also be the basis for parallelizing the detection algorithm. We could iterate over all model files and for each perform the index-based retrieval steps. Combining the results of these smaller detections returns the set of all clones (after filtering clone groups reported by more than one retrieval step). While the overall detection time would surely increase, the individual retrieval/detection steps are completely independent and can be distributed over multiple threads or even separate machines. As our current models are too small to benefit from such a parallel approach, we did not investigate this further, but leave it as future work.

7. CONCLUSION

In this paper, we presented the first index-based approach for model clone detection. It can be combined with existing algorithms to speed up clone retrieval. This way, developers can quickly access all clones of a model element to consciously manage cloning during maintenance.

Since the index can be updated incrementally—as the analyzed models evolve during software maintenance—it can provide up-to-date cloning information. Consequently, the presented clone index improves the suitability of existing algorithms for model clone management, as it speeds up clone retrieval times. Our evaluation demonstrates that the performance speedup of index-based clone retrieval (if compared to batch-detection) is significant: for the analyzed models, it was reduced to 2s. We consider this fast enough for clone management in practice.

For future work, we want to investigate better clone merging heuristics to make index-based clone detection independent of other detection algorithms. Furthermore, we plan a more extensive empirical evaluation on real-world models. Most importantly, however, we plan to develop a tool for model clone management that exploits the clone retrieval capabilities of the index-based approach.

8. REFERENCES

[1] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *ICSM'07*, 2007.

[2] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and M. Vento. A comparison of algorithms for maximum common subgraph on randomly connected graphs. In *SSPR/SPR'02*, 2002.

[3] M. Chilowicz, É. Duris, and G. Roussel. Syntax tree fingerprinting for source code similarity detection. In *ICPC'09*, 2009.

[4] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *ICSM'09*, 2009.

[5] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model clone detection in practice. In *IWSC'10*, 2010.

[6] 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.

[7] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *ICSE'07*, 2007.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[9] N. Göde and R. Koschke. Incremental clone detection. In *CSMR'09*, 2009.

[10] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM'03*, 2003.

[11] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-Based Code Clone Detection: Incremental, Distributed, Scalable. In *ICSM'10*, 2010.

[12] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Eclipse'07*, 2007.

[13] E. Juergens and F. Deissenboeck. How much is a clone? In *SQM'10*, 2010.

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

[15] C. J. Kapser and M. W. Godfre. Improved tool support for the investigation of duplication in software. In *ICSM'05*, 2005.

[16] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: A Tool for Automatic Code Clone Detection in the IDE. In *WCRE'09*, 2009.

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

[18] R. Koschke. Frontiers of software clone management. In *FoSM'08*, 2008.

[19] T. Neumann. Join ordering – dynamic programming – connected subgraphs. `http://www-db.in.tum.de/teaching/ss10/qo/lecture7.pdf`, 2010. Lecture slides for *Query Optimization*, Lecture 7.

[20] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen. Scalable and incremental clone detection for evolving software. *ICSM'09*, 2009.

[21] 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.

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

[23] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *WCRE'04*, 2004.

[24] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University, Kingston, Canada, 2007.

[25] D. Steidl. Index-based model clone detection. `http://www4.in.tum.de/~hummelb/theses/2010_steidl.pdf`, 2010. Bachelor Thesis, Technische Universität München.