Complete and Accurate Clone Detection in Graph-based Models

Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Tien N. Nguyen
Electrical and Computer Engineering Department
Iowa State University
{nampham,hoan,tung,jafar,tien}@iastate.edu

Abstract

Model-Driven Engineering (MDE) has become an important development framework for many large-scale software. Previous research has reported that as in traditional code-based development, cloning also occurs in MDE. However, there has been little work on clone detection in models with the limitations on detection precision and completeness. This paper presents ModelCD*, a novel clone detection tool for Matlab/Simulink models, that is able to efficiently and accurately detect both exactly matched and approximate model clones. The core of ModelCD is two novel graph-based clone detection algorithms that are able to systematically and incrementally discover clones with a high degree of completeness, accuracy, and scalability. We have conducted an empirical evaluation with various experimental studies on many real-world systems to demonstrate the usefulness of our approach and to compare the performance of ModelCD with existing tools.*

1 Introduction

Model-Driven Engineering (MDE) has become an important development framework. Matlab/Simulink is a popular MDE tool for designing and modeling software in many products from small electronic control software to large-scale flight control systems. Models are the collection of logical entities which describe a system at multiple levels of abstraction and from a variety of perspectives.

Previous study by Deissenboeck *et al.* [8] showed that with the nature of using graphical editors for models, cloned fragments in Simulink models often appear. Cloned fragments are the exactly matched or similar fragments in Simulink models. Similar to traditional code clones, clones in Simulink models require additional efforts for maintenance and management. For example, changes to one place must be carried out multiple times for all occurrences of clones. Thus, detecting clones in models plays the same important role as in traditional software development [8].

Unfortunately, there have been very few work on detecting clones in models. CloneDetective represents the state-of-the-art of clone detection in MDE. However, it has several limitations. The important limitations are its inaccuracy and low degree of completeness in detection. The authors reported that several clones were not detected (e.g. small clones are covered in larger clone pairs) [8]. It was also reported that many detected clones by CloneDetective are not interesting to the developers even though they are clones according to CloneDetective's definition. Several detected clone groups are inaccurate and do not carry much meaning for developers. Another key limitation is that CloneDetective algorithm tends to find as large clones as possible. They are sometimes too large and not useful, and do not correspond well to copy-pasted fragments. Users are easily confused when CloneDetective reports such large clones in a graphical editor. Most importantly, CloneDetective could not detect *approximate* clones in which two parts of a model have slight differences. These cases occur often when users make a copy of a fragment and then modify it.

2 Approach Overview

In this paper, we introduce a novel clone detection tool for Matlab/Simulink models, named **ModelCD**, that is able to detect both *exactly matched* and *approximate* model clones. The core of ModelCD is two respective model clone detection algorithms: **eScan** and **aScan**. We develop different algorithms for exact-matched and approximate clone detection because by taking into consideration the nature of each kind of clones, we were able to design different optimization techniques for each algorithm to gain both efficiency and completeness.

The key ideas of our method are as follows. A Simulink model is represented as a sparse, labeled directed graph. Clones in that model are considered as its weakly connected and non-overlapping subgraphs. Two algorithms detect clones through three steps: generating, grouping, and filtering. They first generate candidate clones, then group them into clone groups, and finally filter those groups to remove the redundant ones. To efficiently generate candidate clones, ModelCD is based on an observation that if two subgraphs are cloned, they must contain two cloned subgraphs of a smaller size (size is measured by the number of edges). Thus, the algorithms generate candidate cloned subgraphs from the smallest to the largest size. By doing that way, the algorithms could systematically discover the clones with a high degree of completeness and precision. This also allows each algorithm to apply appropriate optimization and heuristic techniques to reduce the candidate sets. For example, to avoid combinatorial explosion, aScan applies a pruning technique that prohibits un-cloned subgraphs from being used in further generating of candidates.

In the grouping step, different techniques are used in two algorithms. Since eScan aims to detect exactly-matched clones, i.e. isomorphic non-overlapping weakly connected subgraphs, it uses *canonical labeling* [17], an advanced graph isomorphism technique to check the isomorphism of the subgraphs in a sparse graph. Then, it puts them into groups of isomorphic subgraphs, and uses those grouped subgraphs to generate larger isomorphic candidates with the extension of one edge. In contrast, aScan aims to detect approximate clones, i.e. non-overlapping weakly connected subgraphs that are similar in structure. It uses our novel vector-based technique, *Exas* [24], to approximate the structure of a subgraph by a counting vector of the sequences of nodes/edges' labels. Clone grouping is done by using hashing and maximal clique cover methods for those vectors.

ModelCD was integrated into ConQAT [8], an open-source software maintenance for programs and Simulink models. The front-end editor and visualization for models are provided by ConQAT. Users are able to specify the minimum and maximum desired clone sizes or a detecting time limit. An interesting feature of ModelCD is its ability of incremental operation. If users want more results (says, larger clones) after the first run, ModelCD is able to continue its execution without re-running the whole process. The storage cost for an incremental mode is reasonable.

We have performed an extensive empirical evaluation on several open-source Simulink systems and compared the performance of ModelCD with that of CloneDetective, the model clone detection tool within ConQAT. Experimental results show that both eScan and aScan are efficient and scalable to the very large models with reasonable time costs. Compare to CloneDetective, eScan has larger running time but produces more complete and accurate clone results with higher quality and much more quantity. Importantly, the running time of eScan for large models of several thousands of nodes and edges is only in the range of a few hundred seconds. aScan also has a high degree of completeness, precision, and time efficiency in clone detection.

Next section presents our formulation of the model clone detection problem. Sections 4 and 5 explain eScan and aScan algorithms. Additional improvements to both algorithms are in Section 6. Evaluation is in Section 7. Related work is discussed in Section 8. Conclusions appear last.

3 Graph Representation and Formulation

3.1 Representation of Simulink Models

To model a system with Matlab/Simulink, developers use basic Simulink blocks and then the system will be generated from the model. Simulink blocks can be instantiated from many basic types such as *gains*, *adders*, *comparisons*, *switches*, etc. Each block can be associated with *attributes*, depending on the block's type. Inputs and outputs of blocks are connected together via signal lines. Basic blocks can be combined to form a composite block or a subsystem. More details on Matlab/Simulink are in [20].

This first phase of our tool is carried out in the same manner as in ConQAT [8]. Basically, it consists of three tasks: (1) *parsing* the model into a directed graph where a node represents a block and an edge represents a signal connection, (2) *flattening* all subsystems and converting them into graphs (this step is optional), and (3) *labeling* nodes and edges with the labels depending on their attributes. For example, the label for a node includes its block type, while other information are discarded. As in ConQAT, the label of an edge includes the labels of the source and target ports.

The output of this phase is a labeled, directed graph G in which the set of nodes V represents Simulink blocks, the set of directed edges E represents the signal lines, and the labeling function T assigns the labels to nodes and edges. G is a multi-graph because in a model, there might be multiple signal connections between two blocks.

3.2 Formulation

Given $G = (V, E, T)$ as the representation graph of a model M, let us formulate the clone detection problem.

Definition 1 (Fragment) *A fragment f is a set of edges of G which forms a weakly connected subgraph.*

A fragment f with k edges is called a *k-fragment* and is denoted by f_k, i.e. with the subscript as its size.

Definition 2 (Clone Pair) *Two fragments are called a* clone pair *if they are sufficiently similar with respect to a given* similarity measure.

We call them cloned fragments and say that they are clones of each other.

Definition 3 (Clone Group) *A* clone group *is a set of at least two fragments in which any two fragments form a clone pair.*

Thus, a clone group contains only cloned fragments. By definition, a clone pair is also a clone group. To model the non-redundancy in detected clone groups, we use the following concept:

Definition 4 (Covered Group) *A clone group P is said to be* covered *by another group Q if and only if each member of P is a subgraph of at least one member of Q.*

If a clone group is covered by another group, it is *redundant* because the information of its member clones is also contained in the group covering it.

Definition 5 (Clone Detection in Graph-based Models) *Given a graph G and a similarity measure. Find a set CG of* clone groups *satisfying:*

1. Any clone pair existing in G is covered by at least a group in the set CG.

2. CG has no covered group.

Condition 1 means the completeness of CG, because it contains clone information of all clones in the graph. That is, every clone pair of G is either contained in a clone group of CG or is "covered" by another pair in another clone group. Each fragment in a covered pair is a subgraph of a fragment in the covering pair. Condition 2 means that CG has no redundant group. Remember that by definition, all clone groups in CG contain only cloned fragments. It implies that CG is also fully precise.

4 Exact Model Clone Detection

Let us describe eScan algorithm, which aims to find exactly matched clones in models. In this case, the similarity measure for a pair of fragments is defined as follows:

Definition 6 (Exact Clone Pair) *Two fragments f_1 and f_2, with two corresponding subgraphs (V_1, E_1) and (V_2, E_2) in $G = (V, E, T)$ are a clone pair if and only if*

1. Non-overlapping*: $V_1 \cap V_2 = \emptyset$,*

2. Label-isomorphic*: there exist two bijections m: $V_1 \rightarrow V_2$ and $p : E_1 \rightarrow E_2$ such that $\forall v \in V_1 : T(v) = T(m(v))$ and $\forall e \in E(u,v) \cap E_1 : p(e) \in E(m(u), m(v)) \cap E_2$ and $T(e) = T(p(e))$. We use $E(u,v)$ to denote the set of edges between two nodes u and v in graph G.*

In other words, a clone pair is two non-overlapping subgraphs of G that are isomorphic regarding the labeling function T. Thus, to check if two fragments are a clone pair, one needs to solve the problem of labeled graph isomorphism. Currently, it is not known to be in P or NP-hard [17]. However, for the sparse graphs, we use an efficient technique, called *canonical labeling* [26], to solve that problem.

In brief, for each subgraph, a canonical label is computed based on its structure (topology) and the labels of its nodes and edges. This label is invariant with respect to isomorphism. In other words, all isomorphic labeled subgraphs have the same canonical label. Hence, to check whether two fragments are cloned or not, we only need to compare their canonical labels. More information about canonical labeling can be found in another document [26].

Therefore, in eScan, an **exact clone group** is a set of non-overlapping fragments having the same size and canonical label. Taking the union of all clone groups of size k, we have the set of all cloned k-fragments. This set is called **a clone layer** of size k, denoted by L_k.

Observation 1 In G and clone layers:

1. Every k-fragment can be generated from a $(k-1)$-fragment by adding a relevant edge.

2. If two k-fragments are a clone pair, there exists two cloned $(k-1)$-fragments (i.e. subgraphs) within them.

3. Every clone pair of L_k must be generated from a clone pair of L_{k-1}.

Fact 1 is easy to see. For fact 2, we remind that two isomorphic graphs must have two isomorphic subgraphs. If they are non-overlapping, so are their subgraphs. Fact 3 can be easily derived from the first two facts.

Those facts imply that L_k can be generated from L_{k-1} by extending all cloned fragments in L_{k-1} by one edge, collecting those resulting fragments into a candidate set, keeping only the cloned k-fragments, and then grouping them into clone groups. By gradually generating L_1, L_2,..., L_k, we could find all clone groups precisely and completely.

However, this generating strategy is in the breadth-first order, which requires much memory cost to maintain all the groups and candidates. To increase efficiency, eScan follows a depth-first order on a graph called *clone lattice*.

Clone lattice is a layered graph built on the clone layers L_1, L_2,..., L_k,... Each node of the clone lattice is a cloned fragment and the k^{th} layer of clone lattice contains the members of L_k. We use the subscript to a node to denote its layer index. If f_k is a subgraph of f_{k+1}, there will be an edge from f_k to f_{k+1} in the clone lattice. That edge represents the generating relationship between f_k and f_{k+1}. To find all cloned fragments of all possible sizes in G, is indeed to discover the nodes of the clone lattice by traversing from the nodes of the first level. The grouping and filtering process is applied to all cloned fragments (Section 4.2). The technique of using this kind of lattice is adapted from vSiGraM [17]. However, the details of each steps are different. The remaining of this section describes eScan in details.

4.1 Cloned Fragments Generation

Pseudo-code of eScan (Figure 1) uses the followings:

- $Clones(f_k)$ is the set containing f_k and all of its clones (i.e. all fragments which are non-overlapping

```
1  function eScan (G = (V,E,T))
2    $L_1 \leftarrow \{all\ cloned\ 1{-}fragments\}$
3    for each $f_1 \in L_1$ do Discover($f_1$, $Clones(f_1)$)
4    for each $L_k$ do $CG \leftarrow CG \cup Group(L_k)$
5    Filter ($CG$)
6  return $CG$
7
8  function Discover($f_k$, $Clones(f_k)$)
9    for each $g_k \in Clones(f_k)$ do
10     $C_{k+1} \leftarrow C_{k+1} \cup \{g_k \oplus e \mid e \in E\}$
11   for each $c_{k+1} \in C_{k+1}$ do
12     if $GeneratingParent(c_{k+1}) = f_k$ then
13       Find($Clones(c_{k+1})$)
14       if $|Clones(c_{k+1})| > 1$ then
15         $L_{k+1} \leftarrow L_{k+1} \cup Clones(c_{k+1})$
16         Discover($c_{k+1}$, $Clones(c_{k+1})$)
```

Figure 1. Exact Clone Detection

with and are label-isomorphic to f_k). Section 4.4 will explain how to find this set (line 13).

- $\oplus$ is the extension operation: that is, $g \oplus e$ returns the fragment generated by adding an edge e to fragment g.

- $GeneratingParent(c_{k+1})$ returns the generating parent of c_{k+1}, i.e. the unique fragment that is used to generate c_{k+1}. This will be discussed later.

eScan first discovers L_1, the set of all cloned 1-fragments, i.e. all repeated edges of G (line 2). Then, eScan uses each fragments in L_1 as the starting point of a discovery process (function *Discover*) that traverses the clone lattice in the depth-first order (line 3). When visiting a cloned k-fragment f_k, eScan generates a candidate set C_{k+1} of $(k+1)$-fragments which can be obtained from a fragment in $Clones(f_k)$ (i.e. f_k and its clones) with an extension of only *one* edge (lines 9-10). For each candidate fragment $c_{k+1} \in C_{k+1}$, if it is a cloned fragment (line 14), its clones and itself are added into L_{k+1} (line 15) and it is used in the next iteration of discovery (line 16).

A candidate c_{k+1} can be generated by several cloned k-fragments (at most $k+1$). That is, c_{k+1} might be explored and processed many times. To avoid these redundant visits, eScan uses the generating parent technique to ensure that each cloned fragment is explored only once. The idea is to assign for each cloned fragment c_{k+1} a unique fragment f_k which is used to generate c_{k+1}. f_k is called the generating parent of c_{k+1} (Section 4.3). Then, while discovering f_k, a cloned fragment c_{k+1} will be used for next discovery if and only if f_k is the generating parent of c_{k+1} (line 12).

Since c_{k+1} has only one generating parent, it is discovered exactly once. The recursion terminates if the candidate set is empty, and the traversal will backtrack. After the traversal finishes, all cloned fragments are contained in the clone layers $L_1, L_2, ..., L_{max}$. They are grouped layer-by-layer into clone groups (line 4). Then, resulting groups are filtered to remove covered, i.e. redundant, groups (line 5).

4.2 Clone Grouping and Filtering

Remind that all the isomorphic fragments have the same canonical label. Therefore, the first step of grouping in eScan is to partition each clone layer into subsets of fragments having the same canonical label. The result of this step is a collection of smaller subsets of isomorphic fragments.

Despite of being isomorphic, the fragments in a subset might not be cloned to all others because of the non-overlapping condition. Thus, the next phase of grouping task is to find the groups of non-overlapping fragments. Let us give an example. Assume that a subset S has four fragments a, b, c and d isomorphic to one another. However, c overlaps with both b and d. By our definition, S is not a clone group. One can detect in S the following clone groups: $(a, b), (a, b, d), (a, c), (a, d), (b, d)$. Those groups cover all clone pairs in S. However, (a, b), (a, d), and (b, d) are redundant because they are covered by (a, b, d). The most desirable result is two groups (a, b, d) and (a, c).

Therefore, our goal is to find a set of non-redundant clone groups that cover all clone pairs of S and each group has a size as large as possible. To achieve this, eScan represents S as a graph in which nodes are fragments and two nodes have an edge if they are not overlapped. Each clone group is a clique of the graph, i.e. a set of nodes such that any two nodes are connected by an edge. Then, eScan applies Bron-Kerbosch, a maximal clique cover algorithm [7], on the graph to find the desired clone groups.

After grouping that way for all subsets of all layers, we have a set CG of all clone groups for the graph G. The filtering step is required to remove all covered groups in CG. A group is removed from CG if it is covered by another remaining group (see Definition 4 for covered groups).

4.3 Generating Parent Identification

We did not use the original technique of generating parent in vSiGraM [17] to avoid its expensive computational cost. Our procedure to identify the generating parent of a fragment c_{k+1} is as follows. After c_{k+1} is assigned a canonical label, the order of its nodes and edges are uniquely identified. Then, the last edge in that order which does not disconnect c_{k+1} is identified. If that edge is exactly the edge that was just added to f_k, then f_k is the generating parent of c_{k+1}. Figure 2 displays an example. Suppose that after canonical labeling, the order of edges in c_{k+1} is from 1 to 5. Assume that the fragment (e) is just used to create c_{k+1} by adding the edge 5. Thus, it is the generating parent of c_{k+1}.

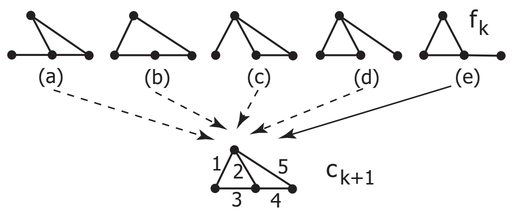

Figure 2. Generating Parent

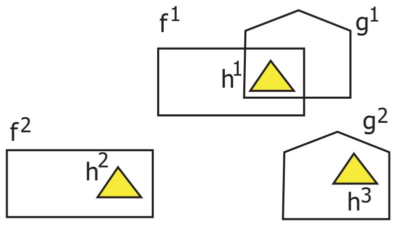

Figure 3. Detected Hidden Clones

4.4 Finding Clones of a Fragment

Our algorithm involves a step that requires the finding of the set $Clones(f_k)$ for a fragment f_k that includes itself and all of its clones in graph G. In general, it is the subgraph isomorphism, an NP-hard problem [11]. However, in eScan, this task requires a relatively inexpensive computational cost. Let us come back to the context of eScan (Figure 1) to show that $Clones(c_{k+1})$ is a subset of C_{k+1}.

Proof. Assume that c'_{k+1} is a clone of c_{k+1}. Because c_{k+1} is generated from f_k, there must exist a cloned fragment f'_k of f_k that can generate c'_{k+1} (see Observation 1 above). Remind that C_{k+1} is the set of all of fragments extended from a fragment in $Clones(f_k)$ (i.e. f_k and its clones) with one edge. Therefore, $c'_{k+1} \in C_{k+1}$.

From this result, to find $Clones(c_{k+1})$, we search in C_{k+1} all fragments c'_{k+1} having the same canonical label and non-overlapping with c_{k+1}. This search can be done efficiently by storing C_{k+1} as a chaining hash table using canonical labels as keys. Since calculating canonical labels for fragments takes time, eScan uses a cache mechanism.

4.5 Detecting Hidden Clones

One interesting feature of eScan over CloneDetective [8] is the ability to detect smaller size clones in the cases that the larger clone pairs hide smaller ones. Figure 3 shows an example that CloneDetective has failed to detect because it has two separate phases: clone pair detection and pair-to-group conversion. In Figure 3, it detects two pairs of model clones (f^1, f^2) and (g^1, g^2) (represented as shapes). However, because it finds clone pairs with the sizes as large as possible, the clone group of (h^1, h^2, h^3) is missed. In contrast, eScan is able to detect that clone group (h^1, h^2, h^3) first, whose elements have smaller sizes. Then, from cloned fragments h^1, h^2, or h^3, the fragments f^1, f^2 and g^1,g^2 are extended, thus, the other two groups are discovered.

5 Approximate Model Clone Detection

As in code clones, clones in model should be considered not only as exactly but also similarly matched. That is, a fragment of the model is copied from one place and pasted in another place with small changes of replacing, adding or removing blocks. In this case, it still maintains almost the same structure but is no longer isomorphic to the original. Therefore, the similarity measure that uses the isomorphism relation and eScan algorithm are not applicable.

To define a new and more appropriate similarity measure in such cases, we develop *Exas* [24], a vector-based representation and feature extraction method that can approximate the structure within a (sub)graph. A (sub)graph is characterized by a vector whose elements are the occurrence counts of the selected structural features within the (sub)graph. By doing this way, the changes of the vector, which can be measured by an appropriate distance function, can approximately capture the changes to a fragment. If the distance is sufficiently small (i.e. smaller than a specific threshold δ), the respective fragments could be considered as clones. Next, we will discuss about Exas vectors.

5.1 Exas Characteristic Vectors

Exas focuses on two kinds of *structural patterns* in a (sub)graph, called *(p,q)-node* and *n-path*. A (p, q)-node is a node having p incoming and q outgoing edges. An n-path is a directed path of n nodes, i.e. a sequence of n nodes in which any two consecutive nodes are connected by a directed edge. A special case is an 1-path, which contains only one node. *Structural feature* of a (p, q)-node is the label of the node and two numbers p and q. For an n-path, it is a sequence of labels of nodes and edges along the path.

Figure 4 shows an illustrated example of a graph and its two cloned fragments A and B [24]. Table 1 lists all patterns and features extracted from fragment A. It could be easy to check that fragment B, which is isomorphic to fragment A, has the same set of features as fragment A.

To efficiently describe the feature set of a fragment, Exas uses the occurrence-count vector of the features extracted from that fragment as its characteristic vector. That is, each position in the vector is indexed for a feature and the value at that position is the number of occurrences of that feature in the fragment. Table 2 shows the indexes of the features, which are global across all vectors, and their occurrence counts in fragment A. The vectors for both A and B are the same. It is (2,1,1,1,1,2,1,1,1,2,1,1,1,1,1).

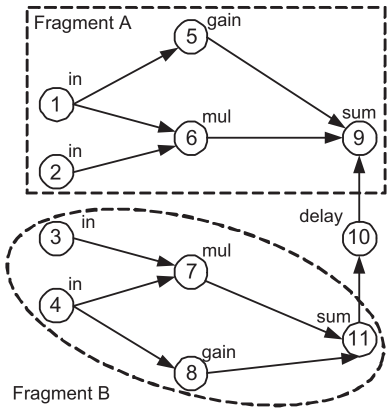

Figure 4. Example of Fragments

Pattern	Features of fragment A				
1-path	1 in	2 in	5 gain	6 mul	9 sum
2-path	1-5 in-gain	1-6 in-mul	2-6 in-mul	6-9 mul-sum	5-9 gain-sum
3-path	1-5-9 in-gain-sum		1-6-9 in-mul-sum		2-6-9 in-mul-sum
(p,q)-node	1 in-0-2		2 in-0-1		5 gain-1-1
(p,q)-node (cont-)	6 mul-2-1		9 sum-2-0		

Table 1. Example of Patterns and Features

In general, it is easy to verify that two isomorphic fragments have the same feature set, thus, have the same vector. Moreover, in [24], we proved a more generic property.

Theorem 1 *If graph edit distance of G_1 and G_2 is λ, then $\|v_1-v_2\| \leq \|v_1-v_2\|_1 \leq (2P+4)\lambda$ with $P = \sum_{l=1}^{N} l.b^{l-1}$.*

G_1 and G_2 are two subgraphs of G. b is the maximum degree of nodes in G (i.e. branching factor), and N is the maximum size of n-paths of interest. (Since there might exist an infinite number of n-paths of all sizes, Exas is interested only in the n-paths of certain limited sizes.)

This result means that, the vector distance of two fragments is bounded by their edit distance, i.e. similar fragments (having small edit distance) will have small vector distance. Therefore, vector distance could be used as a similarity measure of fragments. To normalize the vector distance with respect to the vectors of different lengths, in aScan, we use this measure: $d(v_1, v_2) = \frac{\|v_1-v_2\|}{(\|v_1\|+\|v_2\|)/2}$.

More details on how Exas can efficiently compute and store the vectors for subgraphs can be found in [24].

5.2 Design Strategies

Due to the nature of similar clones, two cloned fragments have to share at least some isomorphic core part. Those above insights are formalized in our definition for approximate (similarly matched) model clones as follows:

Feature	Index	Counts	Feature	Index	Counts
in	1	2	in-gain-sum	9	1
gain	2	1	in-mul-sum	10	2
mul	3	1	in-0-1	11	1
sum	4	1	in-0-2	12	1
in-gain	5	1	gain-1-1	13	1
in-mul	6	2	mul-2-1	14	1
gain-sum	7	1	sum-2-0	15	1
mul-sum	8	1			

Table 2. Vector Indexing and Counting

Definition 7 (Similar Clone Pair) *Two fragments f_k^1 and f_h^2 represented by two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with two corresponding Exas vectors v_1 and v_2 are cloned if and only if: (1) $V_1 \cap V_2 = \emptyset$, (2) $d(v_1, v_2) \leq \delta$, and (3) there exists a pair of subgraphs G_1^o of G_1 and G_2^o of G_2 such that G_1^o and G_2^o are exact clones of the same size s where $\frac{s}{k} \geq \sigma$ and $\frac{s}{h} \geq \sigma$, for two given thresholds δ and σ.*

Condition 1 means that two cloned fragments are non-overlapping. Condition 2 requires them to have similar characteristic vectors. Condition 3 implies that they have an isomorphic core common part. The ratio between the size of the core part and that of each fragment is at least σ.

Observation 2 Because the sizes of those two fragments are larger than that of the core part, $k \geq s \geq h\sigma$ and $h \geq s \geq k\sigma$. This implies $k\sigma \leq h \leq k/\sigma$. That means a cloned h-fragment of a k-fragment must have its size h in the range $[l(k), r(k)]$ where $l(k) = \lceil k\sigma \rceil$ and $r(k) = \lfloor \frac{k}{\sigma} \rfloor$.

Based on those aforementioned observations, we use the following strategies for our aScan detection algorithm:

Breadth-First Traversal. aScan discovers the candidate fragments for clones by traversing the clone lattice in the breadth-first traversal order, rather than depth-first order in eScan. This allows aScan to efficiently consider candidate fragments with different sizes to form clone groups.

By Definition 7, two cloned fragments must contain two isomorphic subgraphs. Checking subgraph isomorphism is NP-hard. Therefore, in aScan, two fragments are considered a clone pair if they satisfy those three conditions in which (1) and (2) mean non-overlapping and small vector distance. Thus, we sacrifice precision for efficiency.

Candidate Window. Based on Observation 2, we use the following heuristic to increase precision and performance. Since any cloned fragment of a k-fragment has its size of at least $l(k)$. Therefore, at the layer k in the lattice, to find clones of k-fragments, aScan considers not only the k-fragment candidates but also the cloned fragments in previous layers (from layer $l(k)$ to $k-1$). *Candidate window* is the set of all those layers.

```
function aScan($G = (V, E, T)$)
  $k \leftarrow 1, L_k \leftarrow E, CG \leftarrow L_k$
  repeat
    $k \leftarrow k + 1$
    $C_k \leftarrow C_k \cup \{f_{k-1} \oplus e | f_{k-1} \in L_{k-1}, e \in L_1\}$
    for $i = l(k)$ to $k - 1$ do $C_k \leftarrow C_k \cup L_i$
    $CG \leftarrow CG \cup Group(C_k)$
    Filter($CG$)
    $L_k \leftarrow$ {all detected cloned k-fragments}
  until $L_k = \emptyset$
return $CG$
```

Figure 5. Approximate Clone Detection

Pruning techniques. The number of candidates for approximate clones could be very large. We apply a pruning technique that prohibits *un-cloned* fragments from being used in further generating of candidates. Our heuristic is that "the cloned fragments at a small size that can form clone groups are likely to be sub-fragments of cloned fragments at some larger sizes".

From Observation 2, a k-fragment f_k cannot be a clone of the fragments of size larger than $r(k)$. Thus, at step $r(k) + 1$, if f_k is not a clone of any generated fragments, it could be removed without reducing the completeness. However, aScan removes it from consideration right at level k to prohibit the generating of larger fragments from it. Therefore, only cloned fragments detected at step k are used to generate the candidate set of the next step.

Another pruning strategy for aScan is applied as follows. At a step, the edges whose all the connections to smaller fragments produce no cloned fragments will not be used in the next iteration because it will not lead to any clones. This greatly reduces the number of candidate fragments because this number is proportional to the number of edges used in each extension.

5.3 Detailed Algorithm

Details of aScan are given in Figure 5. Firstly, aScan collects all the edges of G into the clone layer L_1 (line 2). Then, at a step $k > 1$, it generates clone layer L_k (lines 3-10). At this step, all clone layers from L_1 to L_{k-1} have been generated. Now, aScan includes into the candidate set C_k all k-fragments generated by extending a fragment in L_{k-1} by one edge (line 5). The vectors of those fragments are computed when they are generated. Then, all cloned fragments in the candidate window, i.e. from layer $l(k)$ to layer $(k-1)$ are included in C_k (line 6) since the generated k-fragments can be clones of those smaller fragments.

After collecting members for the candidate set, aScan does grouping on them to detect and add new clone groups into the set of resulting clone groups CG (line 7). Such grouping gives the resulting clone groups at the level k. Since the new groups might cover some detected groups in CG, a filtering process is required (Section 5.4) to remove the redundant groups (line 8). At last, all detected cloned k-fragments are added to L_k. If L_k is not empty, the process continues and L_k is used to generate the candidates of size $(k + 1)$ in the next iteration. Otherwise, since no further level in the lattice could be explored, aScan stops and returns the final clone groups.

Note that, a same k-fragment might be generated many times. To avoid the redundancy, aScan stores candidate set C_k as a hash set and removes all duplicately generated ones.

5.4 Clone Grouping and Filtering

In principle, to do grouping, all the pairwise comparisons between fragments' vectors must be done. With the number of fragments from tens thousands to hundreds thousands, the computation cost is high. In aScan, we reduce this computation by partitioning fragments into subsets and clustering only within these subsets. The partitioning must guarantee that any two fragments having similar vectors will belong to at least one subset. Locality Sensitive Hashing (LSH) [1] is a scheme satisfying this requirement.

aScan performs clustering on each subset S in the same manner as in eScan (Section 4.2). A relation graph is created in which nodes represent for fragments of S and two nodes have an edge if the corresponding fragments do not overlap and have the distance of their vectors no larger than the threshold δ. Then, Bron-Kerbosch clique detection algorithm is run on that group to find all the cliques. Each clique corresponds to a clone group.

From Observation 2, when $k \le k_o = \lfloor \frac{\sigma}{1-\sigma} \rfloor$, we have $r(k) < k + 1$ and $l(k) > k - 1$. It implies that all cloned k-fragments must be exact clones when $k \le k_o$. Therefore, aScan generates all clone layers from 1 to k_o as in the exactly matched clone detection, by requiring all the clones of a group to have the same vector. This improves precision and efficiency of aScan since hashing into subsets just needs a normal hashing function and produces smaller subsets.

Filtering process is applied to remove the redundant groups. Because the detected groups increase in term of the size of their members, aScan performs filtering in an efficient manner in which at level k, it needs to check redundancy only between the groups created at that level and the ones at level $(k - 1)$.

6 Additional Improvements

An ideal goal would be to find all the clones and get the detection completeness (or recall) of 100% while still maintaining high precision. However, this leads to the problem of generating all subgraphs to find all candidate fragments.

In theory, the number of generated subgraphs can be exponential. Thus, that generation is impossible due to both time and storage costs. After investigating Simulink, we have found and applied some improvements using the semantics and nature of Simulink models to achieve a very high degree of detection completeness in short time.

6.1 Subsystems in Simulink

In Simulink, developers often use subsystems and the reuse of subsystems produces model clones. The flattening strategy integrates the subgraphs corresponding to subsystems into the parent system's graph. A detection tool would discover the cloned subgraphs corresponding to the re-used subsystems. However, the detection of re-used subsystems will be more efficient if it is performed without flattening because re-used subsystems will have nodes with the same label. In this case, the re-used subsystems, rather than their graphical structures, should also be reported as clones.

In ModelCD, the detection is carried out to detect (1) all cloned subsystems in each (sub)system hierarchical layer, (2) all clones within each subsystem and between subsystems, and (3) all clones across subsystem hierarchical layers. Firstly, ModelCD parses a Simulink model and all of its subsystems. A subsystem is kept as a node with its name as the label in the representation graph. Then, the structure of a subsystem is flattened into a subgraph and added into the representation graph as a disconnected component. If the subsystem contains within itself another subsystem, ModelCD processes that subsystem in the same way. However, to avoid the aforementioned problem, ModelCD does not flatten a subsystem that was already expanded. Finally, the algorithm is carried out as normal. This approach allows ModelCD to significantly reduce the amount of considered subgraphs. In addition, it avoids the detection of large cloned subgraphs of re-used subsystems, which sometimes are too cumbersome to display in a meaningful way.

6.2 Occurrences of Switches

Another observation is that in a Simulink system, there are blocks with very high degrees, i.e. having many incoming (e.g. Multiplex) and/or outgoing (e.g. De-multiplex) edges. Let us call them *switches*. When the degrees of switches increase, the number of generated subgraphs increases exponentially. Moreover, when two or more cloned switches appear, the number of cloned fragments also increases exponentially since for any subgraph in one switch, there always exists its clone(s) in the other switch(es).

We solve this by a divide-and-conquer approach. Firstly, we separate from the graph the switches having degrees higher than some threshold, and find all clones in the remaining graph. Then, switches are joint to those cloned fragments to form a new set of fragments in which the connectivity between fragments' subgraphs is only the connectivity between the switches. These new fragments are considered the atomic fragments and used to build bigger candidate fragments. Then, the detection from these fragments is carried out in a normal way as in algorithms.

6.3 Incrementality

When aScan explores the lattice layer-by-layer, it has the advantage to detect clones incrementally and stop at any desired size of clones. This is useful in cases where the developers are interested only in certain sizes of clones or have limited time of detection. The detection tool could be temporarily stopped, and then resume its operations to detect larger clones without restarting entire process. ModelCD enables this by allowing users to specify the maximum size of clones. The tool would continue to run until all clones smaller than or equal to the specified size limit are detected or the users decide to temporarily stop it. The tool reports the clone groups and stores the information about the current state of the process. It needs to store only the information about the immediately preceding iteration. With eScan, ModelCD also allows the incremental operation with respect to time. That is, users are able to set the time limit for detection and resume the execution to find larger clones without restarting it. All information required to store for an incremental operation is an array of currently found groups and edges/branches that eScan has not visited.

7 Implementation and Empirical Evaluation

We have implemented ModelCD with two aforementioned algorithms for detecting clones in Simulink models. ModelCD partially re-uses the front-end editor and the Simulink parser from ConQAT [8].

7.1 Experiment Settings

This section presents an empirical evaluation of ModelCD. We compare it against the state-of-the-art Simulink model clone detection tool CloneDetective in ConQAT [8]. In all experiments, we used WindowsXP, Intel Pentium 4 2Ghz, 2GB RAM. We evaluate the performance of ModelCD with regard to both algorithms in term of detection *precision*, *completeness*, *scalability* and *incrementality*.

We chose several open-source Simulink model-based systems (Table 3) ranging from small to large-scale systems in term of total number of blocks (*#blk.*), connections (*#conn.*), used block types (*#bt.*), the maximum size of connected components (*mCC*), and the number of connected components (*#CC*). The systems are available from SourceForge and MATLAB Center. We use the default setting for

System	#blk.	#conn.	#bt.	mCC	#CC
simulink_labs (SIM)	428	415	39	16	47
multiuav (MUL)	475	576	52	123	24
seminar_designs (SEM)	1741	2029	83	283	22
ecwf (ECW)	2312	2274	68	120	151

Table 3. Subject Systems

	eScan		CloneDetective	
System	%Clones	%Groups	%Clones	%Groups
SIM	183/183	60/60	6/6	3/3
MUL	117/117	41/41	7/7	3/3
SEM	180/180	46/46	101/121	29/31
ECW	435/435	64/64	354/428	59/64

Table 4. Precision of eScan

CloneDetective (*minSize* = 5) for good performance. For comparison, we also set the minimum clone size for both eScan and aScan to 5. For aScan, the thresholds are δ = 0.05, σ = 0.9, and the maximum size of n-paths is 4.

7.2 Precision

For the clone quality, we use the *precision* of clone detection. We inspect each of the resulting clone groups returned from a tool, and check two criteria (1) whether the group contains at least a pair of cloned fragments, and (2) whether all fragments in the group are clones of one another. If one criteria is not met, we count the group as incorrect.

Firstly, we want to compare the precision of eScan and CloneDetective. To check the above correctness criteria, we wrote a simple tool to compare and check the detected clones and groups. The result is displayed in Table 4.

The result shows that the precision of eScan is 100% in term of the numbers of correctly detected clones (*%Clones*) and correctly detected groups (*%Groups*) for all subject systems. Although CloneDetective achieves good precision on small systems, it finds incorrect clones and groups in large systems (SEM and ECW). The incorrectly detected clone groups contain *overlapping* fragments. This is due to the CloneDetective's clustering strategy via union finding algorithm, which does not check the overlapping condition to avoid the computational cost of pairwise comparison. eScan is able to use the maximal clique cover algorithm because it works on small sets of isomorphic fragments (Section 4.2).

Secondly, we evaluate the precision of aScan. To avoid completely manual checking, we run aScan on the project SEM, and divide the detected clone groups into two sets: one having the groups with clone sizes from 5-9 and one having groups with clone sizes greater than 9. With σ = 0.9 (percentage of the common core part) and the conditions of a similar clone pair, all the clones in the former set must actually be exact clones, i.e. they must be isomorphic. This set is automatically checked via our isomorphic checking tool. The latter set is checked manually. The result is that among 59 groups in the former set, 4 groups are incorrect, and all 8 groups in the latter set are correct. Thus, the precision is 94%. aScan could not achieve full precision as in eScan due to the use of Exas. It provides an approximate way to measure the structural similarity but can not solve the problem of finding two isomorphic subgraphs in two given graphs.

7.3 Completeness

It is impractical to know the total number of existing clones and groups in large projects. Therefore, in our experiment, the level of completeness is determined by the numbers of correctly detected clones (*#Cl*) and groups (*#Gr*), and the maximum sizes of clones (*mCl*) and groups (*mGr*). We also wrote a tool to check the correctness of groups with respect to our criteria in Section 3.

We conduct an experiment to compare the degrees of *completeness* in exact-matched clone detection of CloneDetective and eScan. In theory, eScan can be proved to be fully complete in detection. However, our optimization techniques for gaining time efficiency could make it less perfect.

In Table 5, in most subject systems, the numbers of clones and groups (*#Cl* and *#Gr*) correctly detected by eScan are much larger than those found by CloneDetective in reasonable running time. Although detection time is longer, it is in the range of few hundred seconds for large systems. CloneDetective is able to find clones and groups with larger sizes (*mCl* and *mGr*). However, many of its large clones actually correspond to the subgraphs of re-used subsystems after being flattened. ModelCD reported them as cloned subsystems, which makes more sense to developers.

We also conducted a similar experiment to evaluate the completeness in detection of aScan. Among three algorithms, aScan detected the most in the shortest time. This conforms to the fact that the similar clone relation implies the exact one and the similar clones should include all exact clones. The other observation is that the maximum clone sizes (*mCl*) are smaller. This is because aScan focuses mainly on the small clones while the big ones are left for the detection of cloned subsystems. In this experiment, the maximum clone size is set to 20. In the rows of SEM and ECW, the *mCl* values are the sizes of cloned subsystems.

7.4 Scalability

Figure 6 shows the running time of eScan on systems increasing in size (total number of blocks and connections). When the size of a subject system increases, the running time also increases. However, the increasing rate of running time is smaller than that of a system's size. For example,

	eScan					CloneDetective					aScan				
System	T(s)	#Cl	mCl	#Gr	mGr	T(s)	#Cl	mCl	#Gr	mGr	T(s)	#Cl	mCl	#Gr	mGr
SIM	120	183	13	60	8	6	6	13	3	2	1.4	272	10	105	8
MUL	132	117	20	41	3	3	7	34	3	3	3	562	18	210	18
SEM	300	180	74	46	8	77	101	253	29	8	6.8	424	74	67	36
ECW	612	435	55	64	16	40	354	109	59	20	6.2	455	55	169	14

Table 5. Completeness

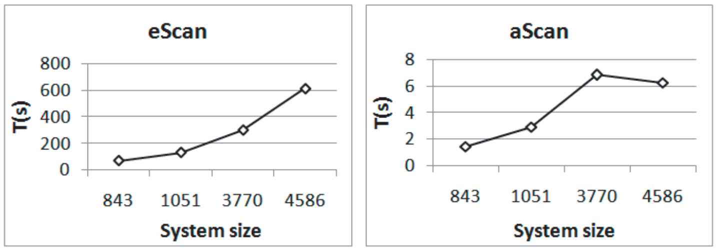

Figure 6. Scalability

T(s)	#Cl	mCl	#Gr	mGr
70	16	6	1	16
131	70	10	5	16
300	178	17	25	16
612	435	55	64	16

Table 6. Incrementality

ECW is 6 times bigger than SIM (4586 versus 843 blocks and connections), but running time is only 5 times longer. This shows that eScan is scalable and able to process large-scale systems of thousands blocks in reasonable time.

aScan is also able to scale to large systems. It is much affected by the size of the maximum connected component. This explains why the time to run the largest system ECW (size of 4586 blocks and connections) is shorter than that of SEM (size of 3770) since ECW has a smaller maximum connected component (size of 120 versus 283). The reason is from the nature of aScan's breath-first traversal. The optimization of separating "switches" helps in this case because it breaks the large connected components into smaller ones.

7.5 Incrementality

In this experiment, we aim to evaluate ModelCD's capability of running in an incremental mode in term of running time with eScan. The chosen subject system is ECW, the largest system in our experiment (2312 blocks and 2274 connections). Table 6 shows that (1) eScan can run in the incremental mode with the time limit set by users (70, 131, 300, and 612s), and (2) the quality and quantity of clones and groups detected by eScan is better when the time limit increases. Specifically, when running time in the same project is increased about four times (70s to 300s), the number of detected clones increases about 11 times (16 to 178).

8 Related Work

The state-of-the-art tool for clone detection in models is CloneDetective [8]. There are several significant differences between ModelCD and CloneDetective. The key difference is that ModelCD systematically detects the clone groups with various sizes of clones from the smallest to largest size. In contrast, CloneDetective first detects all clone pairs and then performs the grouping process. Its clone pair detection is more lightweight in which its heuristic method inspects only the first possible mapping of the nodes' neighborhoods to one another without backtracking. Thus, not all clones could be detected. ModelCD addresses the scalability with its optimization techniques, rather than sacrificing detection completeness as in CloneDetective. Moreover, ModelCD could work incrementally, allowing the completeness level to be improved with more running time. Importantly, aScan is capable of handling approximate clones. Our vector-based approach, Exas, is lightweight, scalable, and enables ModelCD to deal with large graphs and similar clones.

The work by Liu *et al.* [19] aims to find duplications in UML sequence diagrams. They represent a sequence diagram as an array and then build a suffix tree for it. Detecting clones becomes finding common prefixes of suffixes. Ren *et al.* [27] propose to detect clones in sequence diagrams and then to refactor them. However, the detection is not fully automated. In addition, there are several approaches to support model evolution including the detection of differences between models [22, 25, 29], the merging of different models or different versions [22, 23], and the management of consistent model changes [30]. The approaches in [25, 29] represent a UML diagram as a tree. The similarity measure is based on the matching elements in each tree level.

Many approaches for code clone detection have been proposed and a survey can be found in [6]. Generally, they can be classified based on the representation of features extracted from source code. Text-based approaches [2, 13] consider two code fragments as clones if their constituent texts match. Token-based approaches [3, 14, 18] view a code fragment as a sequence of program tokens. Similar or exactly matched sequences of tokens signify clones. Tree-based approaches [4, 5, 9, 12, 16, 21, 28] represent a code fragment as a subtree in the program's AST. Subtrees with similar features are detected as clones. Deckard [12], a tree-based approach, extracts characteristic vectors from AST's subtrees by counting AST node types. Deckard clusters the vectors, i.e. the fragments into clone groups. Existing graph-based code clone detection approaches are not general to be applied to models because they rely on structure and semantics of a program. Komondoor and Horwitz [15] use program dependence graphs (PDGs) and program slicing, and isomorphic subgraphs signify code clones. To detect semantic clones, Gabel *et al.* [10] map PDG subgraphs to related structured syntax and use Deckard approach.

9 Conclusions

As in code-based development, cloning in models creates many difficulties in software maintenance. However, existing clone detection tools for models have limitations on accuracy and completeness. In this paper, we present two algorithms that are able to systematically detect clones and clone groups in the graph-based Simulink models with a high degree of completeness and accuracy. The core ideas include the systematic generation of the candidate clones with the optimization techniques, and the precise structural feature extraction for candidate subgraphs. They have been implemented into ModelCD. Our empirical evaluation on large-scale Simulink systems showed that it is able to handle both exact-matched and similar clones. Compared to CloneDetective, the state-of-the-art detection tool for models, ModelCD gives detection results with a higher quality and much more quantity in reasonable running time. Our algorithms are also general for any graph-based models.

Acknowledgment. This project is partially funded by a Vietnam Education Foundation (VEF) grant for the third author.

References

[1] A. Andoni and P. Indyk. LSH 0.1 User Manual. http://web.mit.edu/andoni/www/LSH/manual.pdf, 2005.

[2] B. S. Baker. Parameterized pattern matching: algorithms and applications. *J. Comp. Sys. Sciences*, 52(1):28–42, 1996.

[3] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal of Computing*, 26(5):1343–1362, October, 1997.

[4] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program transformations for practical scalable software evolution. In *ICSE'04*, pages 625–634. IEEE CS, 2004.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM'98*, pages 368-377. IEEE Computer Society, 1998.

[6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.*, 33(9):577-591, 2007.

[7] C. Bron and J. Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *CACM*, 16(9):575-577, 1973.

[8] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE'08: Int. Conference on Software Engineering*, pages 603–612. ACM Press, 2008.

[9] W. Evans, C. Fraser, and F. Ma. Clone detection via structural abstraction. In *WCRE'07*, pp. 150–159. IEEE CS, 2007.

[10] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE'08*, pp. 321–330. ACM Press, 2008.

[11] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE'07*, pages 96–105. IEEE Computer Society, 2007.

[13] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON'93*, pp. 171–183. IBM Press, 1993.

[14] T. Kamiya, S. Kusumoto, K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large-scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

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

[16] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1): 77–108, 1996.

[17] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11(3):243–271, Kluwer Academic Publishers, 2005.

[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[19] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *APSEC'06*, pages 269–276. IEEE CS, 2006.

[20] The MathWorks Inc. *SIMULINK Model-based and System-based Design - Using Simulink*. 2002.

[21] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM'96*, pages 244-253. IEEE CS, 1996.

[22] A. Mehra, J. Grundy, and J. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE'05*, pages 204–213. ACM Press, 2005.

[23] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *ICSE'07*, pages 54–64. IEEE CS, 2007.

[24] 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*, pages 440–455. Springer-Verlag, 2009.

[25] D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. *FSE'03*, pp. 227-236, ACM, 2003.

[26] R. Read and D. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363, 1977.

[27] S. Ren, K. Rui, and G. Butler. Refactoring the scenario specification: A message sequence chart approach. In *9th Conf. on Object-Oriented Inf. System*, pp. 294–298. Springer, 2003.

[28] V. Wahler, D. Seipel, J. Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM'04*, pages 128–135. IEEE CS, 2004.

[29] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. *ASE'05*,pp. 54–65. ACM,2005.

[30] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *ASE'07*, pages 164–173. ACM Press, 2007.