A Novel Optimized Path-Based Algorithm for Model Clone Detection

Zhengping Liang, Yiqun Cheng, Jianyong Chen
College of Computer Science & Software Engineering, Shenzhen University,
Shenzhen Guangdong 518060, China
Email: jychen@szu.edu.cn

***Abstract*—According to previous reports software clones are considered harmful for software maintenance. Likewise, model clones are problematic in model-based development. It is significant to detect model clones in software models. In this paper, we present a novel optimized path-based model clone detection algorithm (OPMCD). It first builds paths from block graphs, and then identifies clone instances from the common subsequence of paths. Moreover, an experiment is designed to evaluate the algorithm through comparing with the state-of-the-art of model clone detection algorithm ConQAT model clone detection (CMCD). The experiment result illustrates that OPMCD has better performance in terms of efficiency, and it is practically suitable for large-scale MATLAB/Simulink models.**

***Index Terms*—Matlab/Simulink model, optimized path, model clone, clone detection**

I. Introduction

In classical programming languages, clone appears as duplicated code fragments. It is well known that most code clones are created by ad hoc reuse through frequent copy & paste, i.e., fragments being copied rather than being used with appropriate reuse mechanisms [1]. The cloning of code segments in code-based software development by copy & paste has adverse effects on maintainability [2-4], such as unnecessary duplicates of code which increase cost of maintenance, and inconsistent changes to cloned code which can create incorrect program behavior and lead to faults.

Code clone detection is very active area in software clone research [5]. A variety of code clone detection approaches have been proposed. In general, they can be divided into five types based on their source representations: text based, token based, metric based, abstract syntax tree based (AST), and program dependency graphs based (PDG). However, algorithms for code clone detection commonly make no sense to model clone detection as they using different representation except PDG.

Over the years, model-based development has become a promising approach for developing embedded software systems. It has many advantages over traditional development methodology [6]: independence of a target language; higher abstraction level than traditional programming languages; faster than traditional programming; higher automation degree, and possibility to detect errors earlier. MATLAB/Simulink is widely used for modeling in the embedded system domain [7] with which there is already up to 80% of the production codes in embedded control units that are generated from models [8].

Just as in code-based development, cloning also occurs in model-based development when a developer copies model elements instead of using an appropriate reuse mechanism [9]. In model-based software development, clone appears as redundant model elements. Since most of the reasons leading to clones in code-based development are also valid in model-based software development, it is not surprising that clones can also be found in models [10,11]. Cloned subgraphs in Simulink models often appear for different reasons. Most commonly, they are introduced by the habit of copy & paste — deliberate copy model elements with slight changes instead of using an appropriate reuse mechanism [12], or by the use of elements from specific libraries in general-purpose domain. Moreover, clones can also be unintentionally created when similar solutions are independently created [13].

Previous studies have proved that the existence of clone is likely to hinder the maintainability of the model in model-based development [10,11]. Besides the potentially increases of maintenance effort, clone is a potential source of bugs if not all impacted clone instances are changed consistently. Hence, it plays an important role for model-based development to identify duplicated model elements in different parts of the software model.

Existing algorithms for model clone detection have the bottleneck in detection phase among the whole clone detection pipeline. In this paper we propose a novel optimized path-based model clone detection algorithm (OPMCD), and compare it with ConQAT model clone detection (CMCD) [4,10]. The proposed OPMCD builds common subsequences from long paths which are extracted from model graphs. Clone instances are obtained by extending common subsequences. During the process of building paths and finding common subsequences, some optimized measures are introduced, i.e., those nodes without incoming edge are chosen as starting point to build path instead of an exhaustive search, and only the longest path is considered.

doi:10.4304/jsw.9.7.1810-1817

Experiments with MATLAB/Simulink show that the proposed OPMCD can significantly reduce detection time.

The remainder of this paper is organized as follows. Section 2 introduces the mainly processes of model clone detection. Section 3 describes heuristic clone detection algorithm of CMCD. Section 4 presents our approach and elaborates the OPMCD. In order to validate the effectiveness of OPMCD, section 5 implements a case study and compares OPMCD with CMCD. Finally, section 6 makes a conclusion.

II. Related Work

The boom of model clone detection is accompanied with the rapid and wide practices of model-based development. Some people have studied model clone detection within the past years. In graph models, clone can be considered as isomorphic or similar subgraph. The problem of frequent subgraph mining might be the most similar problem to our work. An overview of algorithms for frequent subgraph mining is presented in [14]. Most of these algorithms focus on mining frequent item set among molecules [15,16]. However, some notable differences exist between Matlab/Simulink models and chemical molecules in terms of size and structure. Furthermore, frequent subgraph mining usually works with a higher required minimum pattern frequency. Thus, most of subgraph mining algorithms are not suitable for clone detection in real-world models.

Liu et al. [17] propose a clone detection algorithm for UML sequence diagrams. The approach firstly linearizes sequence diagram as an array and then detects clone by using tree-matching algorithm. The detection time is decreased by reducing duplicated subgraph identification to common substring identification. However, this approach is not appropriate for our work because a similarity representation cannot be created in Simulink models. Störrle [18] explores the problems and possibilities which associated with detecting clones in UML domain models, and designs a number of algorithms and heuristics to carry out clone detection. The basic idea of it is based on the observation that UML models are loosely connected to fat nodes rather than densely connected to graphs of lightweight nodes.

Deissenboeck et al [8,11] firstly proposed an algorithm for model clone detection in graph based models and developed a detection tool which called CMCD based on this algorithm. The core pair detection of CMCD routine performs iteration over all possible pairs of nodes in breadth first search (BFS) manner. On one hand, it can be solved in polynomial time and appropriate for large-scale models, but on the other hand it naturally causes certain clone instances that can't be found, as leads to a lower recall. It is heuristic because it only involves one potential mapping of nodes. Since CMCD is the state-of-the-art of model clone detection algorithm at present, our proposed algorithm will be compared with it in the following sections.

ModelCD (Model Clone Detection) was presented by Pham et al. [19]. It consists of two algorithms eScan and aScan. The core idea used by ModelCD to detect clones is to identify bigger clones through adding extension edges to already detected smaller clones. eScan is used to identify exact clones within a model graph routine through performing a depth first traversal of the clone lattice. It uses a generating parent technique to ensure each fragment and is processed only once. aScan is the first algorithm that can identifies approximate clones which uses a vector based approximation of the structure with a subgraph called Exas [20]. It is different from eScan that aScan traverses the clone lattice in a breadth first manner.

Hummel et al. [21] presents an index based algorithm for Matlab/Simulink model clone detection that is incremental and distributable. Their main purpose is to help developers who can quickly access all clones of a model element to consciously manage cloning during maintenance. To enable semantic clone detection of Matlab/Simulink model, Al-Batran et al.[22] proposes a pattern based approach with the concept of normal forms to identify clones which have identical behavior but different structure. In addition, Alalfi et al.[23] adapts NiCad code clone detector to find near miss clone of Matlab/Simulink by transforming graph-based models to normalized text form.

III. Model Clone Detection

In the context of Simulink models, model clone appears as a connected submodel. Two submodels are considered clone if they are isomorphic, non-overlapping, and connected [11].

There are two types of model clones, exact clone and approximate clone [24]. Exact clone is exactly matched to one another — two data-flow model graphs have exactly the same structure and corresponding labels. Approximate clone contains syntactic clone and semantic clones. Syntactic clone means that two data-flow model graphs have essentially the same structure and labels, allowing for minor adaptions like changes to the element names, attributes, and parameters, etc. In contrast to syntactic clones, semantic clones may exhibit a rather different structure but have the same behaviors.

Within the last few years, the detection of model clone has been an active area of software maintenance research. Theoretically, model clone detection is the problem of identifying all maximum common subgraphs within a graph, which is a NP-Complete problem. Just as mentioned in [25], one of most obvious source for improvement is the clone detection phase. There exist several algorithms to detect clones in models [11,19].

Generally, the model clone detection process includes three major phases: preprocessing and normalization, clone detection, postprocessing. To understand the process of model clone detection easily, we list out the flow-process diagram in figure 1.

In the preprocessing and normalization phase, the models are converted to a labeled, directed graph representation. Meanwhile, the nodes and edges are assigned with normalization labels. To facilitate following detection, all subsystems are flattened. Thus

the hierarchic structure of system model is eliminated. In detection phase, clone instances are identified by similarity compare algorithms. Several clone detection algorithms have been presented, such as heuristic graph based, vector based, index based, etc.

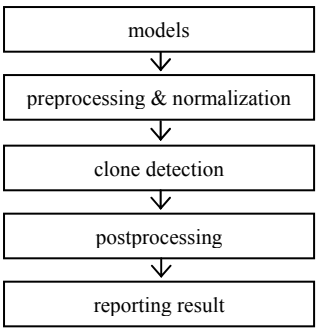


Figure 1. The processes of model clone detection

Here, we propose OPMCD at the detection phase. With our algorithm, clone instances can be found by extending common subsequences of long paths which is built from model graphs. In postprocessing phase, clone groups which have common clone instances are merged to a bigger group and the clone results are reported with visualization form.

CMCD is the state-of-the-art of model clone detection [10]. Our OPMCD algorithm implemented in Conqat framework, both preprocessing and postprocessing of OPMCD and CMCD are the same. But in the core clone detection phase the two algorithms are completely different.

To better illustrate and compare with the proposed OPMCD, we firstly analyze the heuristic clone pair detection algorithm of CMCD.

The algorithm of CMCD is depicted in Figure 2. The input is model graph which is preprocessed from models. At first, node pair set N is created. All node pairs are put into N if they have the same node labels. If node pair (u, v) has not been visited, put them into queue Q. If queue Q is not null, pair (m, n) will be dequeued from Q (line 6). Line 7 is the most important heuristic place as a list *P* of node pairs is constructed from the neighborhood of node pair with a high similarity value. More details of the algorithm can be found in [8].

```
1   function Heuristic model clone detection
2   Input: model graph G = (V, E, L)
3   create node pair set N containing all pairs, each pair has same
    node label, D := ∅
4   for each node pair (u, v) in N do
5     if {u, v } ∉ D then
6     Q := {(u, v)}, C := {(u, v)}
7     while Q≠∅ do
8       dequeue pair (m, n) from Q
9       P = build_list (m, n)
10      for each (x, y) ∈ P do
11        if (x, y) ∈ D
12          jump to the loop at line2
13        else if x ≠ y ∧ {x, y} has not been visited
14          C := C ∪ {(x, y)}
15          enqueue (x, y) in Q
16    export clone result C
17    D := D ∪ C
```

Figure 2. Heuristic algorithm of CMCD for detecting clone pairs

The core pair detection of CMCD routine performs iteration over all possible pairs of nodes in BFS manner. To improve time complexity, the algorithm uses heuristic search which is used to quickly extend new pairs of nodes that can be combined with the current pair of nodes to form a larger clone pair. Finally, clone pairs are combined to clone groups. Therefore, on the one hand it can be solved in polynomial time and appropriate for large-scale models, but on the other hand it naturally causes that certain clone instances can't be found, which leads to a lower recall.

IV. Optimized Path-Based Model Clone Detection Algorithm

Our proposed OPMCD focuses on diagram model with exact clone identification, and contributes to the detection phase, which is the bottleneck of performance and kernel phase in clone detection pipeline.

A. Definitions

Given $G = (V, E, L)$ as the representation graph of a model, we use the following definitions:

Definition 1 (Labeled Directed Graph) A labeled directed graph G is a pair $G = (V, E, L)$ consisting of a set V of nodes and a set $E \subseteq V \times V$ of directed edges, with an additional labeling function L: $V \cup E \rightarrow N$ which maps nodes and edges to labels from a set N.

Definition 2 (Graph Component) A graph component is a set of connected nodes of G which forms a weakly connected subgraph.

Definition 3 (Longest Path in Graph) The longest path is simple node path of maximum length in a graph component. This longest path is a simple path in which every node appears exactly once.

Definition 4 (Common Subsequence) A common subsequence in a pair of sequences is a node subsequence that appears in both sequences. The longest common subsequence is a common subsequence with maximal length.

Definition 5 (Clone Instance) A clone instance is the exactly matched or similar subgraph in model graph. All clone instances within a clone group have clone relationship with each other. Both graphs are called clone instances if they are isomorphic.

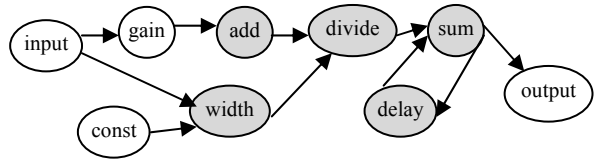
Definition 6 (Clone Group) A clone group is a set of clone instance, where any two instances are a clone pair.

B. Optimized Path-Based Model Clone Detection Algorithm

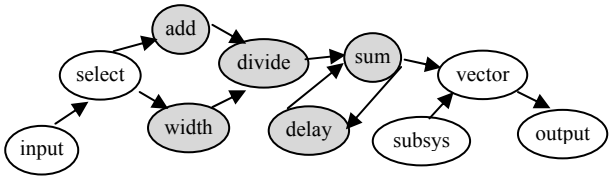
In this paper, we focus on the detection algorithm with graph based data-flow models.

The kernel idea of the detection algorithm is based on an observation that the longest path in a data-flow model contains majority nodes of the model. For instance, the longest path in model shown in figure 3 contains 7 nodes, which cover 77.8% nodes of total. Basically, our algorithm consists of three main steps: building path, finding common subsequences, and identifying clone instances. The procedure to identify clone instance is as follows.

First, all connected components are exacted from model graph. Then, the proposed OPMCD uses those nodes which have no incoming edge as the starting point to build paths. If the length of one path is long enough, it will be selected for subsequent processing. Figure 3 displays an example. From nodes "input" without incoming edge, two paths with (input, gain, add, divide, sum, output) and (input, select, add, divide, sum, vector, output) are constructed separately from the components of figure 3 (a) and figure 3 (b). Meanwhile, those paths whose length is smaller than a desired value will not be considered (e.g., the path (subsys, vector, output) in the figure 3 (b)). Next, the common subsequences (add, divide, sum) of long paths are created between two paths. Finally, all common subsequences are expanded by the extension with nodes with the same label in a BFS manner. All nodes contained in clone instances which are marked with gray color are identified.



(a)



(b)

Figure 3. The illustration of finding clone instance with OPMCD

The pseudo-code of OPMCD is described in figure 4. *G* denotes set of graph representations transformed from system model by preprocessing and normalization. *C* denotes set of connected components. Container stack *S* is used to store nodes temporarily. *T* denotes set of nodes which have been visited.

```
1   function OPMCD
2   input: labeled directed graph G = (V, E, L)
3     extract connected components from model graph G, and put
    into the set C
4     path_list := ∅, seq_list := ∅
5     for each component x in C
6       T := ∅
7       for each node n in component x do
8         S := ∅
9         if(getIncomingEdge(n) = 0)
10          buildPath (n)
11    for each p_i, p_j ∈ path_list and p_i≠p_j do
12      findSubseq(p_i, p_j)
13      for each subSeq s in seq_list do
14        clones := clones ∪ findClones(s)
15    perform cluster on clones and report the results
16
17  function buildPath(n)
18    push n into S, T := T ∪ {n}
19    nodeCluster := getNodeCluster(n);
20    if nodeCluster ≠ ∅
21      for each node y in nodeCluster
22        if y ∉ T
23          buildPath(y)
24    else
25      if (S.size > ∂1)
26        path_list := path_list ∪ constructPath(S)
27    pop n from S
28
29  function findSubseq(p, q)
30    m = p.size+1, n= q.size+1
31    create an m×n all zero matrix M
32    for each element M[i][j] in M
33      if node p[i-1] and q[j-1] have same label
34        M[i][j] = M[i - 1][j - 1] + 1
35    for each element M[i][j]
36      if(M[i][j] >∂2)
37        seq_list :=seq_list∪p.substring(i-M[i][j], i-1)
```

Figure 4. Pseudo-code of OPMCD

First, model is represented as a sparse, labeled directed graph $G = (V, E, L)$. In line 3, all labeled directed graph are enumerated, and then all connected components are extracted to construct set *C*. If the size of a component is small, it would be eliminated.

Since the first node of a path has no incoming edge, we use those nodes without incoming edges as a starting point to build path instead of enumerating all nodes, as shown in line 9. It can effectively prevent the construction of vast redundant paths, which helps to reduce time consumption of the algorithm.

Line 10 calls function buildPath that is detailed in lines 17-27. A depth first search (DFS) backtracking algorithm is used in the construction of path within graph components. At first, the starting node *n* is put into *S* and *T* in line 18, and then getNodeCluster performs a BFS traversal from *n* using forward edges to get node cluster in line 19. If *nodeCluster* is not empty, the function buildPath is repeatedly called for each node which has not been visited in *nodeCluster* (lines 20-23). In line 22, OPMCD guarantees that each node be visited at most one time and no repeated nodes existed in different paths. When a node without forward edge has been reached, a path is constructed from node sequence in *S* supposing the size of *S* is big enough (lines 23-25). Next, the top node *n* of *S* is popped in line 26.

What must be emphasized is that we choose all non-overlapping candidate paths which are greater than a given threshold besides the longest path in line 25. This is due to the fact that some large graph components contain hundreds of blocks. If only the longest path is considered, we may lost the chance of finding potential candidate clones resulted from other path. At the same time, short paths are not being considered because clones that resulted from those paths often tend to duplicate with clone result from long paths. Consequently, it can guarantee that our algorithm has high recall and fast detection.

After that, the common subsequence set *seq_list* is obtained by function findSubseq in line 12. Function findSubseq identifies all kinds of common subsequences with a trick of matrix in lines 33-34. The primary character of it is introducing a threshold in line 36 to eliminate trivial common subsequences from which meaningful clone instances generally can't be generated.

Later, clone instances in model graph are identified by function findClones through the extension of common subsequences via BFS search in line 14. Finally, all clone groups are clustered and the results are reported.

Totally, the optimized measures are introduced in lines 3, 22, 25 and 36 respectively. The heuristic mechanism in line 9 not only helps to significantly reduce total time for detection, but also ensure that OPMCD has excellent detection performance in precision and recall.

C. Time Complexity

Assume that d is the maximum number of the nodes without incoming edge in components, m is the average size of components, n is the number of connected components, and N is the total number of nodes in model system. The OPMCD mainly consist of three functions: building path, finding common subsequence, extending from common subsequences.

For building path function, all nodes in a connected component are traveled only once from a given starting point (nodes without incoming edge) by DFS. Building path function in our algorithm runs $O(d*m*n)$ times, in which $m*n$ equals to N. Moreover, in practice, nodes without incoming edge are small proportion of all nodes of a component, i.e. d is small. Thus, $O(d*N)$ is trivial.

During the extension of common subsequences, there are $N^2 = n^2*m^2$ compactions in worst case. In summary, the time complexity of novel optimized path-based algorithm can be solved in $O(N^2)$ time.

V. EXPERIMENTS

In order to evaluate the performance of OPMCD, ConQAT is used as a common framework to integrate OPMCD for simulation. The ConQAT is an integrated toolkit for creating quality dashboards that allow to continuously monitor quality characteristics of software systems. The experiments are performed on personal computer with a duo Intel core of a 2.4GHz CPU, 3GB of main memory, and Windows 7 operating system. In these experiments, the weight of minimum clone instance is set as 5.

ModelCD developed by Pham et al. [18] operates in roughly the same way as ConQAT, has little improvement and is not publicly available, so we excluded it in our comparison.

A. Analyzed Model

We choose four open source Simulink system models which are available from MATLAB Center [7]: A Simulink model for a communications lab (SIM), a simulation of multiple unmanned air vehicles (MUL), a video surveillance system (SEM), and an echo canceller model (ECW). These Simulink model-based systems are also chosen as experiment objects in [11,19]. Table I shows the sizes of these models where system denotes model name, files denotes the number of model file of system, nodes denotes the total number of blocks, and edges denotes the total number of lines.

TABLE I.

SIZES OF MODELS

system	files	nodes	edges
SIM	49	452	422
MUL	2	475	576
SEM	16	1558	2029
ECW	31	2312	2274

As above table shown, the models contain hundreds to thousands of blocks. The largest system has 2312 blocks distributed over 31 files. What should be noted is that all blocks of the MUL model contain just in one .mdl file while another file does not contain any block. Unfortunately, industrial-scale models are not available. These analyzed model are typical models. They have been used to verify the effectiveness of algorithm in many papers [8,11,19].

Since we are not interested in small clone instances, graph components which are less than 5 will be removed in the process of extracting connected components. The results of extracted connected components are show in table II. There are 6 columns: model name (system), the number of graph components which are kept after extraction processing (#CKeep), the number of graph components which are skipped after extraction processing (#CSkip), the total number of nodes in kept components (#NKeep), the total number of nodes which are skipped (#NSkip), and the average size of kept components (#Asize).

TABLE II.

CONNECTED COMPONENT EXTRACTION

system	#CKeep	#CSkip	#NKeep	# NSkip	# Asize
SIM	42	28	411	41	9.8
MUL	15	14	448	27	29.9
SEM	18	91	1457	101	85.9
ECW	74	192	1957	355	26.4

We can see that almost 83.5% components have been eliminated in SEM from Table II. For the largest model ECW, 192 components were removed and just 74 connected components are kept. This shows that the filter of graph components is effective, especially for large-scale model.

B. Run-Time Comparison

To compare the run-time between the proposed OPMCD and CMCD, we performed both of them with the models in Table I. The results of run-time comparison are illustrated in figure 5. The initial parsing phase of OPMCD and CMCD are the same and not taken into account. Thus, the time shown in vertical direction of figure 5 which does not involve time for loading and parsing Simulink files, and displaying clone results.

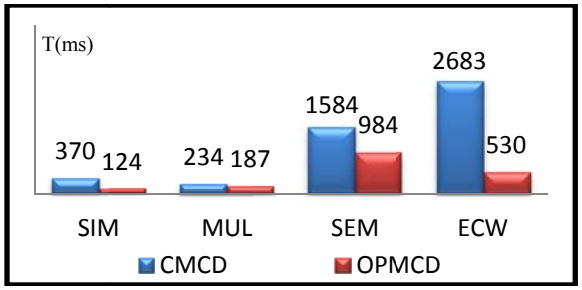


Figure 5. Time used in model clone detection.

The average size of kept components of SIM is the smallest. Therefore, the detection phase for SIM requires least time. OPMCD requires 984ms for model of SEM, while it only requires 530 for ECW whose model is larger than SEM. This is because the average size of kept components of SEM is the 85.9 while the average size of kept components of ECW is 26.4. For all the models, OPMCD performs much faster than CMCD as shown in figure 5. Especially for the largest model ECW, the detection time of CMCD is almost 5 times longer than that of OPMCD. The comparison of the run-time of both algorithms shows that the OPMCD has better performance in terms of speed.

C. *Precision and Recall*

For clone detection quality, it must consider the precision and recall. We inspect each clone group resulted from both OPMCD and CMCD manually.

The precision of OPMCD is 100% from the point of theory. To validated practical precision of OPMCD, we inspect the detection results from following two aspects: each clone instance exists in original models, and each clone instance in the same clone group are clone of one another. After exhausted manual checking, we find that all clone instances and clone groups reported with our algorithm satisfy above conditions, which confirms that the practical precision of detection results is also 100%.

To validate the recall of our algorithm, we firstly analyze the cover relationship among clone groups and find that all clone groups identified with OPMCD is independent of each other. Figure 6 shows the numbers of clone groups identified with both OPMCD and CMCD for models of table I.

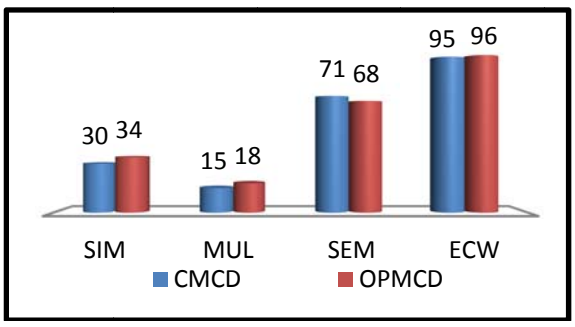


Figure 6. The number of identified clone groups

By one to one comparison, we find that OPMCD can identify all the 30 clone groups with SIM and all the 15 clone groups with MUL that CMCD identifies. Moreover, extra clone groups are identified with OPMCD, i.e., 4 extra clone groups with SIM and 3 clone groups with MUL. It means that OPMCD has better recall than CMCD to SIM and MUL.

For SEM, CMCD identified 3 more clone groups than OPMCD in total as shown in figure 6. However, we found that there are 8 different clone groups existing isomorphic clone instances in results from CMCD. Since those 8 clone groups should be taken as one clone group, CMCD actually identified 64 clone groups. Although there are 2 clone groups existing isomorphic clone instances in results from OPMCD, it still has 67 clone groups. Therefore, for SEM, OPMCD also has better recall than CMCD.

ECW is similar to SEM. CMCD identifies 95 clone groups, and OPMCD identifies 96 clone groups. There are 6 clone groups containing isomorphic clone instances in CMCD, which can be merged into 3 groups. And there are also 4 groups in OPMCD which can be merged into 2 groups. That is to say, 92 clone groups are identified by CMCD and 94 clone groups are identified by OPMCD in fact.

Figure 7 is a clone instance originated from MUL which OPMCD can detect but CMCD can't find. It consists of 17 blocks with three layers. Here, blocks with different colors represent different layers.

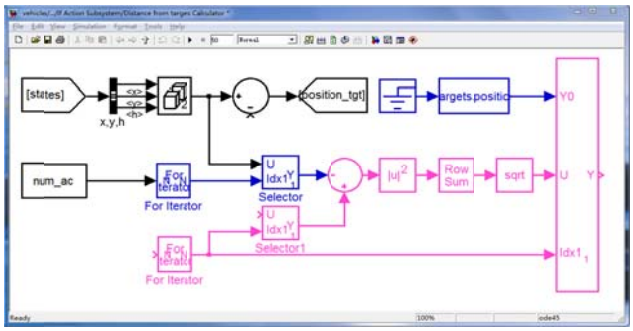


Figure 7. Clone instance that OPMCD finds, but CMCD cannot find

To sum up, the experiment results on models of Table III demonstrate that our algorithm has prominent advantage in time performance, which means that OPMCD has better scalability and practical suitability for large-scale MATLAB/Simulink models. At the same time, the precision and recall of OPMCD is not worse than CMCD.

VI. CONCLUSION

There are lots of reasons leading to numerous of clones in model-based development. Therefore, model clone detection plays an important role in software maintenance domain.

In this paper, a novel optimized path based model clone detection algorithm named OPMCD has been presented. This method initially introduced path-based algorithm into the model clone detection. The OPMCD identifies common subsequences from the long paths which are extracted from model graphs. Then clone instances are obtained by extending common subsequences. During the

process of paths building and common subsequences finding, those nodes without incoming edge are chosen as starting point to build path instead of an exhaustive search, and only the longest path is considered. The experimental evaluation demonstrates that OPMCD can significantly speed up detection time. At the same time, it can preserve high quality in precision and recall which has advantage in clone detection for large-scale model systems. Future works include further improvement in detection time and applying our algorithm to industrial model systems. Moreover, according to specific objectives and scenarios of model clone detection, it is interesting to study on clone groups filter and cluster to improve performance of OPMCD.

ACKNOWLEDGMENT

This work was supported by the Science &Technology Fund of Shenzhen under Grant JCYJ20120613114918935, JCYJ20120616135936123and JCYJ20130326112033984, National High-Technology Research and Development Program ("863" Program) of China under Grand 2013AA01A212, Ministry of Education in the New Century Excellent Talents Support Program of China under Grand NCET-12-0649, and National Nature Science Foundation of China under Grant 61170283.

REFERENCES

[1]. D. Yael, R. Julia, B. Thorsten, et al. "An Exploratory Study of Cloning in Industrial Software Product Lines", *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, pp. 25-34, 2013.

[2]. D. Rattan, R. Bhatia, M. Singh. "Software Clone Detection: a Systematic Review", *Information and Software Technology*, volume 55, issue 7, pp. 1165-1199, 2013.

[3]. C. Debarshi, C. Jeffrey, K. Nicholas, "Cloning: The Need to Understand Developer Intent", *Proceedings of the 7th International Workshop on Software Clones*, pp. 14-15, 2013.

[4]. S. A. Mohammad, M. Yasuhiko, A. S. Mohammad. "Baenpd: a Bilingual Plagiarism Detector", *Journal of Computers*, volume 8, issue 5, pp. 1145-1156, 2013.

[5]. J. Elmar, D. Florian, H. Benjamin, W. Stefan, "Do Code Clones Matter?", *Proceedings of the 31st International Conference on Software Engineering*, pp. 485-495, 2009.

[6]. K. Rajeev. "Business Rules Modeling for Business Process Events: An Oracle Prototype", *Journal of Computers*, volume 7, issue 9, pp. 2099-2106, 2012.

[7]. http://www.mathworks.com/, Accessed May 10, 2013.

[8]. F. Deissenboeck, B. Hummel, E. Jurgens, B. Schätz , S.Wagner, J.-F. Girard, and S. Teuchert. "Clone Detection in Automotive Model-Based Development", *Proceedings of the 30th International Conference on Software Engineering*, pp. 603-612, 2008.

[9]. M. Broy, S. Kirstan, H. Krcmar, and B. Schätz. "What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?", *Emerging Technologies for the Evolution and Maintenance of Software Models*, pp. 343-369, 2012. DOI: 10.4018/978-1-61350-438-3.ch013

[10]. E. Juergens. "Research in Cloning Beyond Code: a First Roadmap", *Proceedings of the 5th International Workshop on Software Clones*, pp. 67-68, 2011.

[11]. F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. "Model Clone Detection in Practice", *Proceedings of the 32th ACM/IEEE International Conference on Software Engineering*, pp. 499-500, 2010.

[12]. W. Hu, , J. Wegener, I. Stürmer, R. Reicherdt, E. Salecker, S. Glesner, "MeMo – Methods of Model Quality", *Proceedings of Dagstuhl-Workshop: Model-Based Engineering of Embedded Systems*, pp. 127-132, 2011.

[13]. http://www.mathworks.com/help/toolbox/slcontrol/gs/bsp4o3g.html, Accessed May 10, 2013.

[14]. H. J. Patel, R. Prajapati, M. Panchal, et al. "A Survey of Graph Pattern Mining Algorithm and Techniques", *International Journal of Application or Innovation in Engineering & Management*, volume 2, issue 1, pp. 125-129, 2013.

[15]. L. Chen, Y. Chen, L. Tu. "A Fast and Efficient Algorithm for Finding Frequent Items over Data Stream", *Journal of Computers*, volume 7, issue 7, pp. 1545-1554, 2012.

[16]. K.M. Tang, C. Y. Dai, L. Chen, "A Novel Strategy for Mining Frequent Closed Itemsets in Data Streams", *Journal of Computers* , volume 7, issue 7, pp. 1564-1573, 2012.

[17]. H. Liu, Z. Ma, L. Zhang, and W. Shao. "Detecting Duplications in Sequence Diagrams Based on Sufftrees", *Proceedings of the 13th Asia Pacific Conference on Software Engineering*, pp. 269-276, 2006.

[18]. H. Störrle. "Towards Clone Detection in UML Domain Models", *Software and Systems Modeling*, volume 12, issue 2, pp. 307-329. 2013.

[19]. 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", *Proceedings of the 31th International Conference on Software Engineering*, pp. 276–286, 2009.

[20]. H.A. Nguyen, T.T. Nguyen, N.H. Pham, J.M. Al-Kofahi, T.N. Nguyen. "Accurate and Efficient Sructural Characteristic Feature Extraction for Clone Detection", *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering,* pp. 440–455. 2009.

[21]. B. Hummel, E. Juergens, D. Steidl. "Index-Based Model Clone Detection", *Proceedings of the 5th International Workshop on Software Clones*, 21-27, 2011.

[22]. B. Al-Batran, B. Schatz, and B. Hummel. "Semantic Clone Detection for Model-Based Development of Embedded Systems", *Lecture Notes in Computer Science*, issue 6981 pp. 258-272, 2011.

[23]. M. Alalfi, J. R. Cordy, T. Dean, M. Stephan, and A. Stevenson. "Near-miss Model Clone Detection for Simulink Models", *Proceedings of the 6th International Workshop on Software Clones*, pp. 78-79, 2012.

[24]. M. Stephan, M. Alalfi, A. Stevenson, and J. R. Cordy. "Towards Qualitative Comparison of Simulink Model Clone Detection Approaches", *Proceedings of the 6th International Workshop on Software Clones*, pp. 84-85, 2012.

[25]. S. Matthew, H. A. Manar, S. Andrew, R. C. James, "Using Mutation Analysis for a Model-Clone Detector Comparison Framework", *Proceedings of the 35th International Conference on Software Engineering.* pp. 1261-1264, 2013.

Zhengping Liang received his Ph.D. degree on computer software and theory from the School of Computer, Wuhan University, Wuhan, China in 2006. Now he is an associate professor in the College of Computer Science & Software Engineering, Shenzhen University, Shenzhen, China. His current research interests include software analysis, requirements engineering and computational intelligence, etc.

Yiqun Cheng received his BS.c. from the Jiangxi Normal University in 2010. Now he is an MS.c. student at the College of Computer Science & Software Engineering, Shenzhen University, Shenzhen, China. His research interests include model clone detection and analysis.