A Completeness Optimized Algorithm for Closed Model Clone Detection

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

***Abstract*—The detection of model clone has been an active research area in recent years. The closed clone instances contain all the information of model clones so they can ensure the completeness of detection results essentially. In order to improve the degree of completeness in clone detection, a novel model clone detection algorithm named CL_MCD (Closed Model Clone Detection) is proposed. CL_MCD focuses on exactly matched clones and aims to find all the closed clone instances. The main innovation of CL_MCD is in the detection phase. Every time after finding a new node pair with the same label in the breadth-first search of model graph, CL_MCD transforms all the node pairs into a clone pair, and puts the clone pair into a set that contains all the candidate clone instances if its size is greater than or equal to the size of minimum clone. Then every candidate clone instance is compared with all the others in the set. If a candidate clone instance is one part of any other instance, it is deleted. After the filtering, redundant clone instances are removed and only the closed clone instances are kept in the set. Theoretical analysis and experimental studies demonstrate that CL_MCD has higher degree of completeness than CloneDetective.**

***Index Terms*—Simulink model, model clone, closed model clone, model clone detection**

I. Introduction

In recent years, model-driven software development (MDSD) has become a popular way of creating software systems [1,2]. Developers can define software systems on a higher level of abstraction by MDSD. Matlab/Simulink is a popular Model-Driven Engineering tool for designing and modeling software in many products from small electronic control software to large-scale flight control systems [3]. Models are the collection of logical entities which describe a system at multiple levels of abstraction and from a variety of perspectives. As models are used to generate code, they can be regarded as a higher level programming language.

Previous studies in [4] showed that most of the reasons leading to clones in code-based development are also valid for MDSD. Therefore, it is not surprising that simulink models often contain clones. Model clones are taken as the exactly or similar matched fragments in simulink models [5]. Similar to traditional code clones, model clones in simulink models require additional efforts for maintenance and management in most cases. For example, changes to one place must be carried out multiple times for all occurrences of clones. Thus, the identification and elimination of model clones is important to improve the maintainability of the system under development. Moreover, in the case of product lines construction, it is a core requirement to identify the reusable pieces of functionality and integrate them into a library for future reuse.

The detection of model clones has been an active area of research in recent years. There exist several approaches to detect clones in MDSD [4-9].Among them, CloneDetective [4] which is included in the open-source tool of ConQAT represents a classical clone detection algorithm. However, it has low degree of completeness in detection because the CloneDetective mainly focuses on the maximal clones and cannot reveal hidden clone instances which usually have smaller size and are covered by larger clone instances.

This paper presents a novel clone detection algorithm, i.e., CL_MCD (Closed Model Clone Detection), for exactly matched clones in Matlab/Simulink models. The core idea of CL_MCD is that it aims to find all of the closed clone instances in the phase of detection. Fundamentally, the closed clone instances can ensure the completeness of detection results because they contain all the information of exact model clones. Besides, theoretical and experimental studies also demonstrate that CL_MCD can find some hidden clones that CloneDetective has failed to detect. Therefore, its completeness is better than CloneDetective. Moreover, its running time is reasonable and acceptable.

The remainder of this paper is organized as follows. In section 2, we briefly describe MATLAB/Simulink graph and model clone representation. The process of model clone detection and clone detection problem are presented in section 3. We analyze limitations of CloneDetective and present CL_MCD in section 4. Section 5 performs practical evaluations of the detection algorithm CL_MCD and compares it against CloneDetective. Related works are discussed in section 6. Conclusion and future works appear at last section.

II. MATLAB/Simulink Graph and Model Clone Representation

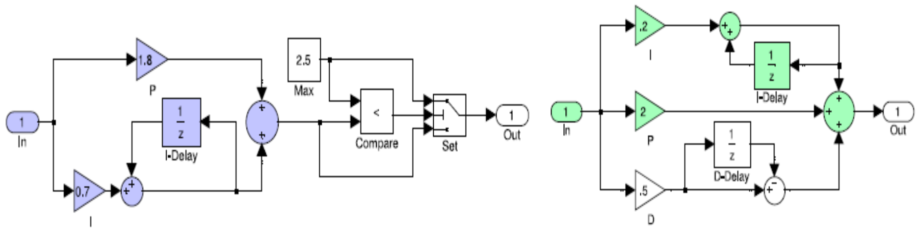

Figure 1. Clone between discrete saturated PI-controller and PID-controller [4]

MATLAB is a software package provided by The MathWorks Inc [10]. It can be extended by several toolboxes used for control systems, signal processing and optimization. Simulink is one of toolboxes that provide a graphical environment for designing, modeling and simulating dynamic systems via data flow graphs. Simulink also offers an extensive library of predefined function and parameter blocks for linear, non-linear, discrete and hybrid systems [11].

In Simulink, users can construct systems models graphs by using instances of these function blocks that are connected to each other by signal flow lines. A block instance can be associated with a set of attributes depending on the block's type. With Real-Time Workshop (RTW) tools, systems models graphs can be used to generate source code of C or C++ etc. They can be regarded as a higher level programming language. Figure 1 shows an example of a Simulink model of two controllers [4]. The shapes (squares, triangles, circles) represent function blocks and the lines between those blocks represent the flow of data.

An important feature of Simulink is that users can combine sets of blocks and lines into subsystems and create higher-level domain abstractions like a PID controller [12]. This makes it possible to create very large and complex model without losing the overview. Models are partitioned into a layered hierarchy by using subsystems. If a subsystem shall be used at multiple locations, it should be externalized as a library element. To use the functionality provided by a library element, a model needs to reference the library element together with a set of input parameters.

Previous studies showed that with the nature of using graphical editors for models, it is not surprising that clones in simulink models often exist [4]. Similarly to the definition of traditional code clones, the exactly or similar matched fragments in simulink models are called model clones. For example, the two colored parts in Figure 1 are clones of each other, which are usually created by a sequence of copy, paste, and modify steps. Although sometimes clones are unavoidable and can't be eliminated completely, in most cases clones in simulink models require additional efforts for maintenance and management [5]. For example, changes to one place must be carried out multiple times for all occurrences of clones. Thus, in order to improve the maintainability of the system under development, it is useful to identify and eliminate model clones. Moreover, in the case of product lines construction, it is a core requirement to identify the reusable pieces of functionality and integrate them into a library.

III. Model Clone Detection

This section briefly describes the process of model clone detection at first, and then defines the clone detection problem for MATLAB/Simulink models.

A. The Process of Model Clone Detection

Generally clones are detected through three phases [4]: preprocessing and normalization, detection, and postprocessing. Figure2 shows the general process of model clone detection for MATLAB/Simulink models.

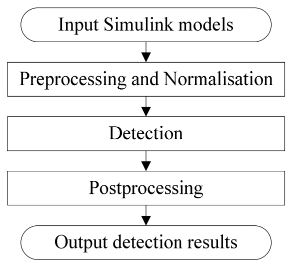

Figure 2. The process of model clone detection

In the process of model clone detection, simulink models must be preprocessed and normalized at first. The result of this step is a labeled and directed multi-graph with G= (*V*, *E*, *L*). The detection phase is the core content in the model clone detection. It works on the labeled graph produced during the previous phase. The main task of this phase is generating candidate clone instances, and then grouping them into clone groups. Postprocessing maps the clone results found at the detection phase with the simulink models. In order to analyze and manage the results of clone detection easily, we should generate readability clone detection report and show the results of clone detection in an intuitive, easy to accept and understandable way.

B. Formulation on the Clone Detection Problem

Simulink models can be represented as a labeled and directed multi-graph with $G= (V,E,L)$ after preprocessing and normalization. Now let us formulate the clone detection problem for MATLAB/Simulink models.

Definition 1(Clone Instance)

A clone instance is defined as a weakly connected sub-graph g_1 of the model graph $G= (V,E,L)$ that is isomorphic or approximate to at least one other sub-graph g_2 of G with regard to the labeling function L.

Definition 2(The Support of Clone Instance)

The support of clone instance g is denoted as *sup(g)*: $sup(g)=|M|$, and $g \in M$, where $|M|$ is the number of clone instances in the clone group M that contain the clone instance g.

Definition 3(Closed Clone Instance)

A clone instance g in clone group M is called closed clone instance if and only if there are not exist a clone instance g' in another clone group N, such that the following conditions hold: $g \subset g' \wedge sup(g) = sup(g')$.

The definition of closed clone instance is similar to the closed frequent sub-graph in graph mining. Closed clone instances contain all the information of model clones and they can ensure the completeness of detection result essentially [13].

Definition 4(Exact Clone Pair)

Two weakly connected and directed sub-graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ of $G=(V,E,L)$ are considered exact clone pair if they are isomorphic with regard to the labeling function L, and V_1, V_2 aren't overlapping. The clone in a exact clone pair is usually called exact clone.

Definition 5(Clone Group)

A clone group is a set that contains at least two clone instances and any two clone instances in the same clone group form a clone pair.

Definition 6(Covered Group)

A clone group M is said to be covered by another group N if and only if $\forall g_1 \in M$, $(\exists g_2 \in N) \wedge (g_1 \subseteq g_2)$. The g_1 is a clone instance in clone group M. The g_2 is a clone instance in clone group N.

If a clone group M is covered by another group N, M is redundant because the information of its member clones is also contained in the group N. In this case, M can be deduced from N.

IV. Closed Model Clone Detection Algorithm (CL_MCD)

This section provides an overview of the exact clone detection algorithm CloneDetective at first, then analyzes the limitations of CloneDetective and presents a closed model clone detection algorithm of CL_MCD that base on CloneDetective captions

A. CloneDetective

CloneDetective is a classical model clone detection algorithm of ConQAT tool [4] in MDSD. It was the first proposed exact model clone detection algorithm that enumerates all maximal exact clones in MATLAB/ Simulink models. The process of model clone detection in CloneDetective is the same with the description in section 3.1.

The result of the preprocessing and normalization phase is a labeled, directed multi-graph with $G=(V,E,L)$. After preprocessing and normalization, CloneDetective runs in two distinct steps in the phase of detection: firstly, all clone pairs are identified; secondly, clone pairs are clustered to form clone groups.

In the first step, the algorithm enumerates all pairs of clones, i.e., all pairs of sub-graphs that are isomorphic. To do that, the algorithm iterates over all possible pairs of nodes and proceeds in a breadth-first search manner from there. The authors do not use an exhaustive search. Instead, they use a heuristic to reduce the time complexity. While estimate the similarity of a pair of nodes, the heuristic reference the normalization labels as well as the structure of the neighborhood of both nodes. Moreover, the heuristic play a major role to quickly find other pairs of nodes that can be combined with the current pair of nodes to a clone pair in the course of the algorithm iterates over all possible pairings of nodes. In the second step, the CloneDetective provides a method to combine clone pairs to a clone class. It uses a union-find structure to build clone groups. More information about CloneDetective can be found in [4].

B. Closed Model Clone Detection Algorithm of CL_MCD

Although CloneDetective is the classical clone detection algorithm, it has several limitations [4,5]. The most important limitation is its low degree of completeness in detection. As CloneDetective always tries to build maximal clones, some hidden clone instances and clone groups are not reported. Those hidden clones are also valuable to construct reusable models library in the way of model-driven software development and the completeness of clone detection results in CloneDetective will decrease if without them. In this section, we present CL_MCD that can detect the hidden clone, as CloneDetective has failed to detect.

Firstly we present an example about some hidden clone groups and clone instances that CloneDetective has failed to detect in Figure 3. The clone instances are represented by geometric figures of rectangles, circles, and triangles. In Figure 3(a), CloneDetective does not find the star shaped clone group $S=(s_1,s_2,s_3)$ whose elements have smaller sizes and only reports three clone groups $R=(r_1,r_2)$, $T=(t_1,t_2)$ and $C=(c_1,c_2)$ (represented as shapes). Because CloneDetective always tries to build maximal clones, it only can find clone pairs with the sizes as large as possible, and smaller clone pairs are not reported if each clone pair is covered by a bigger clone pair. For example, the case starting from nodes in stars of s_1 and s_2 identifies the rectangles clone group $R=(r_1,r_2)$, the case starting from nodes in stars s_2 and s_3 identifies the circles clone group $C=(c_1,c_2)$, and the case starting from nodes in stars s_1 and s_3 identifies the triangle clone group $T(t_1,t_2)$.

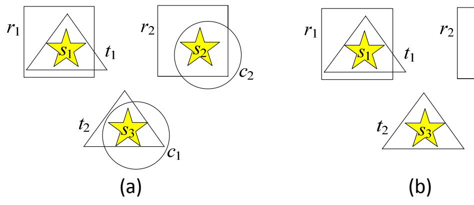

Figure 3. (a) Example that star shaped clones are not reported (b) Example that star shaped clones are reported partly

According to the definition 3 and 6 in section 3.2, it is easy to know that the clone instances of s_1, s_2, s_3 in clone group $S=(s_1,s_2,s_3)$ are closed clone instances, and the clone group $S=(s_1,s_2,s_3)$ is not covered by clone groups R, T, C. Therefore, they should also been reported as clone group $S=(s_1,s_2,s_3)$. However, the clone group $S=(s_1,s_2,s_3)$

is lost in the detection of CloneDetective. Evidently, the completeness of clone detection results in CloneDetective will decrease.

The same as the analysis of Figure 3(a), CloneDetective reports three clone groups $R=(r_1,r_2)$, $T=(t_1,t_2)$ and $S=(s_2,s_3)$ (represented as shapes) in Figure 3(b): the case starting from nodes in stars of s_1 and s_2 identifies the rectangle clone group $R=(r_1,r_2)$, the case starting from nodes in stars s_1 and s_3 identifies the triangle clone group $T=(t_1,t_2)$, and the case starting from nodes in star s_2 and s_3 identifies the star clone group $S=(s_2,s_3)$. It is found that the clone instance $s1$ is lost. However, the clone instance $s1$ also has clone relationship with clone instance s_2 and s_3. Thus we should report clone group $S=(s_1,s_2,s_3)$ instead of $S=(s_2,s_3)$. In this case, although there aren't clone groups be lost, some clone instances are still lost. It means that for CloneDetective, the completeness of clone detection will decrease.

Through above analysis, we can see the hidden clones of s_1, s_2, s_3 in Figure 3(a) and the hidden clone $s1$ in Figure 3(b) are closed clone instances and they are also useful for showing all the information of model clones. However, those valuable hidden clones are discarded by CloneDetective because they are not maximal clones. So closed clone instance is better than maximal clone in the aspect of completeness of model clone detection. Therefore, we had better find all the closed clone instances instead of maximal clones in model clone detection.

Based on the above findings, we present CL_MCD, a model clone detection algorithm that has better performance in the aspect of completeness than CloneDetective. Figure 4 presents the process of clone detection with algorithm CL_MCD.

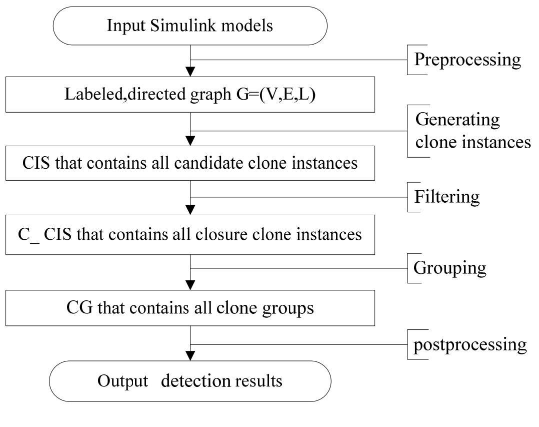
Input Simulink models
Preprocessing
Labeled,directed graph G=(V,E,L)
Generating clone instances
CIS that contains all candidate clone instances
Filtering
C_ CIS that contains all closure clone instances
Grouping
CG that contains all clone groups
postprocessing
Output detection results

Figure 4. The process of clone detection with CL_MCD

The core of innovation in CL_MCD is at the step of generating all the candidate clone instances in Figure 4. In order to find all the closed clone instances, the breadth-first search of model graph is used. Every time after finding a new node pair with the same label, CL_MCD transforms all the current node pairs into a clone pair, and puts this clone pair into a set that contains all the candidate clone instances if its size is greater than or equal to the size of minimum clone. CL_MCD aims to find closed clone instance and can find the valuable hidden clone that CloneDetective has failed to detect, such as the clone group $S=(s1,s2,s3)$ in Figure 3(a), and the clone instance $s1$ in Figure 3(b). CL_MCD has higher degree of completeness than CloneDetective in clone detection.

The Pseudo code of CL_MCD is presented in Figure5. *D* denotes a set of already visited node pairs. S is a set of nodes seen in the current breadth-first search. Set *C* contains all node pairs of the current clone. *CIS* is a set containing all candidate clone instances. *C_CIS* is a set containing all closed clone instances. CG denotes the set of clone groups.

CL_MCD also detects clones through three phases: preprocessing, detection and postprocessing. Let's analyze each phase of CL_MCD now. Generally, simulink models must be preprocessed and normalized at first in the process of model clone detection, and the result of the preprocessing phase is a labeled, directed multi-graph $G=(V,E,L)$ where a node represents a block and an edge represents a signal connection line. This preprocessing phase is carried out in the same manner as CloneDetective.

```
    Input: Model graph $G=(V,E,L)$
    Output: Clone Group Set $CG$
1.    $D := \varnothing$ , $CIS := \varnothing$ , $C\_CIS := \varnothing$
2.   for each $(u,v) \in V \times V$ with $u \neq v \wedge L(u) = L(v)$ do
       if $\{u,v\} \notin D$ then
3.       Queue $Q := \{(u,v)\}$ , $C := \{(u,v)\}$ , $S := \{u,v\}$
4.       while $Q \neq \varnothing$ do
5.         dequeue pair $(w, z)$ from $Q$
6.         from the neighborhood of $(w, z)$, build a list of node
     pairs $P$ with equivalent labels
7.           for each $(x,y) \in P$ do
8.             if $(x,y) \in D$ then continue with loop at line 2
9.             if $x \neq y \wedge \{x,y\} \cap S = \varnothing$ then
10.              $C := C \cup \{(x,y)\}$ , $S := S \cup \{x,y\}$
11.              enqueue $(x, y)$ in $Q$
12.              report node pairs in $C$ as clone pair $\{p_1,p_2\}$
13.              $CIS := CIS \cup \{p_1, p_2\}$
14.      $D := D \cup C$
15.   $C\_CIS := Filter(CIS)$
16.   $CG = Group(C\_CIS)$
17.  Return $CG$

18.  Function $Filter(CIS)$
19.    for each $c \in CIS$ do
20.      if $c \subseteq c' \wedge sup(c) = sup(c')$ the $c' \in CIS$ then
21.        $CIS := CIS - clones(c)$
22.    $C\_CIS := CIS$
```

Figure 5. The pseudo code of CL_MCD

The phase of detection is the main innovation of CL_MCD. Instead of only finding maximal clone instances as CloneDetective, CL_MCD aims to find all of the closed clone instances, on account of closed clone instances are better than maximal clone instances at the side of completeness of detection results. The definition of closed clone instance is presented in section3.2. According to the previous analysis, some hidden clone instances that can't be detected by CloneDetective are also closed clone instances and all the information of model clones are contained in closed clone instances, so the closed clone instances can ensure the completeness of detection results essentially.

As shown in Figure 4, the detection phase of CL_MCD mainly contains three steps: Firstly, generating the set CIS that contains all candidate clone instances. Secondly, filtering *CIS* to obtain *C_CIS* that contains all the closed clone instances. Thirdly, grouping the *C_CIS* into clone groups. Through these three steps, CL_MCD can find all of the closed clone instances, each step of detection phase in CL_MCD is analyzed as follows.

In the first step of detection, the CL_MCD generates the set CIS that contains all candidate clone instances. CL_MCD iterates over all possible node pairs and uses the pairs with the same label as the starting point of a breadth-first search to find other node pairs that can be combined with the current node pairs to form a candidate clone pair(lines 2-14). In this step, in order to generate the set *CIS* that contains all of the candidate clone instances, every time after putting a new node pair into the queue of *Q*, CL_MCD transforms all the current node pairs in set C into a clone pair, and put this clone pair into the set *CIS* if its size is greater than or equal to the size of minimum clone(lines 13-14).Through this way, all of the candidate clone instances can be detected by CL_MCD, and of course the hidden clone instances of s1, s2, s3 in Figure3 (a),and the clone instance s1 in Figure3 (b) can also be found. Therefore, CL_MCD can ensure the completeness of detection results essentially. However, CloneDetective does that until the queue of *Q* is empty, so it can only find maximal clone instance and all of the hidden clone instances which usually have smaller size and are covered by larger clone instances are lost. Such as the hidden clone instances s1, s2, s3 shown in Figure 3(a), and the clone instance s1 in Figure3 (b) are lost.

Analysis showed that in the first step of detection some clone instances incompletely covered groups are also put into CIS. These completely covered clone instances are redundant because all the information of them are also contained in the bigger clone instances covering them. By this means, we should remove them in the filtering step (line 16). In the filtering step, we remove the redundant clone instances in CIS and only keep all the closed clone instances to form *C_CIS*. It is carried out as follows. Firstly it sorts all the clone instances of *CIS* in the order of increasing clone sizes, and then for every two different size clone instances of c and in *CIS*, if, the clone instance isn't closed clone instance but a redundant clone instance. Therefore, *C* and all the clone instances which have clone relationship with clone instance c should be removed from *CIS*. The result of filtering step is that the set of *C_CIS* only contain all of the closed clone instances. In the third step of detection phase, we obtain the set of clone groups *CG* by grouping *C_CIS* (line 17).In this step, all closed clone instances in *C_CIS* are inserted into a hash table with their label as a key. The hash table consists of several lists, where each list contains the closed clone instances that have the same key. Because the closed clone instances that have the same key are isomorphic, each list represents a clone group.

The postprocessing phase maps the results of clone detection with the original simulink models. Generally, this phase can also be used to order clones or discard some of them. In the simplest case, all the results of clone are just reported to the user in the order of increasing clone sizes. Postprocessing phase should show the clones in an intuitive, easy to accept and understandable way, so that users can analyze and manage the results of clone detection easily.

V. Empirical Evaluation

To evaluate performance of CL_MCD, we do several experiments with both CL_MCD and CloneDetective and compare their performance each other.

A. Experiment Settings

All experiments were performed on a desktop computer running Windows XP with an Intel Pentium 4 CPU 2.1 GHz and 3GB RAM. We evaluate the performance of CL_MCD in term of completeness and running time.

For comparison, We choose four public simulink model-based systems in table 1 as experimental case that were also used by Deissenboeck et al. [4] and by Pham et al. [5]. The systems are publicly available from Matlab Central.Table1 shows the sizes of these systems where #Bl denotes the total number of blocks, #Li denotes the total number of connection lines, #Ty denotes the total number of used block types. The minimum clone size for both CloneDetective and CL_MCD is 5.

B. Completeness

We conduct experiments to compare the clone detection results between the CL_MCD and CloneDetective. In our experiment, the level of clone detection results is evaluated from completeness and running time.

TABLE I.
THE SIMULINK MODEL-BASED SYSTEMS AS CASE STUDY

System	*#Bl*	*#Li*	*#Ty*
SIM	428	415	47
MUL	475	576	44
SEM	1741	2029	86
ECW	2312	2274	68

Table 2 shows the clone detection results of both CL_MCD and CloneDetective. *#Cl* is the numbers of correctly detected clone instances after reviewing by the definition of clone instance. *#CG* is the numbers of correctly detected clone groups after reviewing by the definition of clone group. *#T* is running time. The running time in the table is the only the times taken by the clone detection algorithms. The time for preprocessing and postprocessing is not included in the displayed numbers. *#hid-Cl* is the numbers of hidden clones that CL_MCD can find but CloneDetective has failed to detect.

TABLE II.
CLONE DETECTION RESULTS OF CL_MCD AND CLONEDETECTIVE

	CloneDetective			CL_MCD				
System	*#Cl*	*#CG*	*#T*(ms)	*#Cl*	*#hid-Cl*	*#CG*	*#hid-CG*	*#T*(ms)
SIM	30	10	391	30	3	10	0	421
MUL	21	7	250	21	6	9	2	281
SEM	151	38	1890	206	55	47	9	2187
ECW	405	82	3453	545	140	82	0	3850

#hid-CG is the numbers of hidden clone groups that CL_MCD can find but CloneDetective has failed to detect.

Generally, the completeness of clone detection results is determined by the numbers of correctly detected clone instances (*#Cl*) and clone groups (*#CG*). As shown in table 2, CL_MCD can find all the clones that CloneDetective finds. Moreover, it also can find some hidden clones（*#hid-Cl*） that CloneDetective has failed to find. So in all subject systems, the number of clone instances correctly detected by CL_MCD is larger than that found by CloneDetective in reasonable running time. The rate of increased clones found by CL_MCD is measured. That in SEM is 36.4% and that in ECW is 34.6%. Thus the new algorithm CL_MCD yields a higher completeness than CloneDetective.

C. Running Time

The running time in the table is the only time taken by the clone detection algorithms. The time for preprocessing and postprocessing is not included. The running time is the average value of multiple running results (twenty times).Because the randomness of initial expansion in algorithm and the influence of computer cache, the running time has a certain variation amplitude

From the table 2, we can see that the running time of CL_MCD is longer than CloneDetective. This is not surprising because CL_MCD needs to filter the CIS. The more clones found the more time need. Although the running time of CL_MCD is longer, it's in the range of a few hundred milliseconds for large simulink systems. Therefore, it is acceptable. Furthermore, from the table 2 we can know that from SIM to ECW the rate of increased clones is greater than the rate of increased running time. For example, in SEM the rate of increased clones is 36.4%. However, the rate of increased running time is only 11.5%. In a word, the running time of CL_MCD is reasonable and acceptable.

In summary, CL_MCD is better than CloneDetective in the side of completeness, and the running time of it is reasonable and acceptable. Furthermore, CL_MCD has a certain of scalable as it can process large-scale case like ECW in reasonable time.

VI. RELATED WORKS

In this section, we summarize existing related works in the area of clone detection on models and in source code. We also give a short overview on frequent sub-graph mining.

A. Clone Detection in Models

The detection of model clone has been an active area of research within the last years. There exist several approaches to detect clones in Simulink models.

In 2008 Deissenboeck et al. published the first exact model clone detection algorithm CloneDetective that enumerates all maximal exact clones in MATLAB/Simulink models [4]. CloneDetective first detects all clone pairs and then performs the grouping process. In 2009 Pham et al. proposed a clone detection framework for Simulink models called ModelCD [5]. The framework consists of two algorithms, eScan and aScan. The eScan algorithm is designed to find exact clone, while the aScan algorithm can find approximate clone. Pham et al. attempts to improve CloneDetective by providing eScan and aScan together in order to detect both exact and near-miss clones.

Their improvements utilize graph mining work and Simulink specific properties.

In 2011 Hummel et al. presented an approach to clone detection by storing indices of fragments of a model in a database [8]. The approached is based on the idea that most of the times only small parts of a model are altered between clone detection runs. In their approach only newly changed parts of the model are taken into consideration in consecutive algorithm runs.

In 2012 Alalfi et al. made an improvement based on the code clone detection tool NiCad and applied it to clone detection for Matlab/Simulink models [6]. After improvement, NiCad can find exact clone and approximate model clone as well. In addition, in 2012 Stephan et al. choose some public simulink model-based systems as experiment case, and made a comparison of the existing simulink model clone detection approaches [7]. Furthermore, Stephan et al. presented a new approach for evaluating and comparing model clone detectors that is based on mutation analysis and also clone representation transformation in 2013, which helps to address the challenges of manual comparison and to provide a standard and extendable way of evaluating and comparing model-clone detectors [14].

B. Code-based Clone Detection

Code-Based clone detection research starts much earlier than model clone detection. There are a large number of code clone detection approaches and a survey can be found in [15].

Generally, based on the representation of features extracted from source code, these approaches can be classified as text-based [16,17], token-based [18,19], tree-based [20,21], graph-based [22,23] and metric-based [24] approaches. Among these approaches, graph-based approach is most similarly with the approaches used in model clone detection. In the approach of program dependence graphs (PDGs) which is used by Komondoor and Horwitz firstly [25], the isomorphic subgraphs represent code clones. However, existing code clone detection algorithms can't be applied to model clone detection directly because the fundamentally different between code and model.

6.3 Frequent sub-graph mining

Graph-based clone detection can be regarded as a specialization of frequent sub-graph mining problem within a single graph and a minimum required pattern frequency of two after the model has been normalized to a labeled graph.

Frequent sub-graph mining deals with the extraction of interesting structures from graphs [26,27]. An overview and comparison of algorithms for sub-graph mining is given by [28]. These algorithms strive for an exact solution and usually work with a much higher required minimum pattern frequency than 2. Thus, they may not be appropriate for our purpose. However, in order to develop novel algorithms for model clone detection, those sub-graph mining algorithms offer a certain reference value.

VII. Conclusion and Future Works

In this section we summarize our findings and give an overview of potential future works to improve clone detection for models.

A. Conclusion

MDSD has become a popular approach for creating software systems. Most of the reasons leading to clones in code-based development are also valid for MDSD. In simulink models clones are the exactly or similar matched fragments, and clones in simulink models require additional efforts for maintenance and management.

This paper presents CL_MCD (Closed Model Clone Detection), a clone detection algorithm for Matlab/ Simulink models, which aims to find closed clone instance. Experimental results show that CL_MCD has better performance than CloneDetective. It detects all of the closed clone instances in the detection phase through three steps: generating all the candidate clone instances at first, then filtering redundant clone instances to obtain all the closed clone instances, and grouping all the closed clone instances into clone groups at last. The process of generating all the candidate clone instances is the main innovation of CL_MCD. CL_MCD iterates over all possible node pairs and uses the pairs with the same label as the starting point of a breadth-first search to find other node pairs that can be combined with the current node pairs to form a candidate clone pair. In order to find all the closed clone instances, every time after finding a new node pair with the same label, CL_MCD transforms all the node pairs into a clone pair, and puts this clone pair into the set CIS that contains all the candidate clone instances if its size is greater than or equal to the size of minimum clone. If a candidate clone instance is one part of any other instance, it is deleted. Therefore, redundant clone instances in CIS are removed and can obtain the set C_CIS that contains all the closed clone instances. CL_MCD can find all the clones that CloneDetective found. Moreover, it also can find some hidden clones that CloneDetective has failed to detect. These hidden clones are included in closed model clone instances and they are also useful for constructing reusable models library in the way of model-driven software development.

B. Future Works

Model clone detection has been studied only in recent years. There are still many problems remain to solve. The most obvious direction for improvement is the clone detection algorithm which should has higher degree of completeness in reasonable and acceptable running time. Moreover, model clone detection algorithms must be capable of processing larger-scale models within reasonable time and memory limits because Simulink models usually have significant size in real-world. We can learn some techniques and ideas from frequent sub-graph mining and code clone detection.

Another interesting research problem is to find approximate model clone in which two parts of a model have slight differences. CL_MCD can only find exact model clone at present, we will improve CL_MCD and make it can detect approximate model clone as well in the future. Furthermore, improving the relevance of clone detection is also an important research direction. Deissenboeck etc. have shown that currently many of the clones found are not interesting for the developer, although they are of course clones according to clone definition [29]. In order to make the clone detection more targeted and personalized, in the phase of preprocessing and normalization, it is interesting to study how to remove the blocks and its adjacent lines whose types are not cared by users, and only keep the blocks and its adjacent lines whose types are more valuable to construct reusable models library. It is also interesting to analyze the factors that affect clone relevance and study the scheme of clone ranking.

Acknowledges

This work was supported by the Science &Technology Fund of Shenzhen under Grant JCYJ20120613114918935, JCYJ20120616135936123 and 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.C. Schmidt, “Guest Editor’s Introduction: Model-Driven Engineering,” *Journal of IEEE Computer*, volume 39, issue 2, pp. 25-31, 2006.

[2] K. Rajeev. “Business Rules Modeling for Business Process Events: An Oracle Prototype”, *Journal of Computers*, volume 7, issue 2, pp. 2099-2106, 2012.

[3] A. Angermann, M. Beuschel, M. Rau, et al. “Matlab Simulink Stateflow, Grundlagen, Toolboxen, Beispiele,” *Oldenbourg*, 2007.

[4] F. Deissenboeck, B. Hummel, E. Juergens, et al. “Clone Detection in AutomotiveModel-Based Development,” *Proc. 30th International Conference on Software Engineering(ICSE)*, pp. 603-612, 2008.

[5] N.H. Pham, H.A. Nguyen, T.T. Nguyen, et al. “Complete and Accurate CloneDetection in Graph-based Models,” *Proc. 31th International Conference on Software Engineering(ICSE)*, pp. 276-286, 2009.

[6] H. Alalfi, R. Cordy, T.R. Dean, et al. “Near-miss Model Clone Detection for Simulink Models,” *Proc. 6th International Workshop on Software Clones(IWSC)*, pp. 78-79, 2012.

[7] M. Stephan, M. Alafi, A. Stevenson, and J. Cordy, “Towards Qualitative Comparison of SimulinkModel Clone Detection Approaches,” *Proc. 6th International Workshop on Software Clones(IWSC)*, pp. 84–85, 2012.

[8] B. Hummel, E. Juergens, D. Steidl, “Index-Based Model Clone Detection,” *Proc. 5th International Workshop on Software Clones(IWSC)*, pp. 21-27, 2011.

[9] Bakr Al-Batran, Hummel B, Bernhard Schätz, “Semantic Clone Detection for Model-Based Development of Embedded Systems,” *Proc. 14th International Conference on Model Driven Engineering Languages and Systems (MODELS)* , pp. 258-272, 2011.

[10] The MathWorks Inc. Matlab Product Website. http://www.mathworks.com/products/ma- tlab/.

[11] The MathWorks Inc. Simulink Product Website. http://www.mathworks. com/products/sim- ulink/.

[12] The MathWorks Inc. “Simulink Model-Based and System-Based Design - Using Simulink,” 2002.

[13] D.J. Cook and L.B. Holder, Mining Graph Data, *John Wiley & Sons*, 2006.

[14] M. Stephan, M. Alafi, A. Stevenson, and J. Cordy, “Using Mutation Analysis for a Model-Clone Detector Comparison Framework,” *Proc. 35th International Conference on Software Engineering(ICSE)*, pp. 1261–1264, 2013.

[15] C. Roy, J. Cordy, and R. Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: a Qualitative Approach,” *Science of Computer Programming*, volume 74, issue 7, pp. 470–495, 2009.

[16] R. Wettel, R. Marinescu, “Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments,” *Proc. 6th Symbolic and Numeric Algorithms for Scientific Computing(SYNASC)*, pages 11–15, 2005.

[17] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic, “An Information Retrieval Approach to Concept Location in Source Code,” *Proc. 11th Working Conference on Reverse Engineering (WCRE)*, pp. 214–223, 2004.

[18] Y. Yuan, Y. Guo, “Boreas: an Accurate and Scalable Token-based Approach to Code Clone Detection,” *Proc. 27th International Conference on Automated Software Engineering (ASE)*, pp. 286-289, 2012.

[19] H. Murakami, K. Hotta, Y. Higo, et al. “Folding Repeated-instructions for Improving Token-based Code Clone Detection,” *Proc. 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 64–73, 2012.

[20] A. Corazza, S.Di Martino, V. Maggio, and G. Scanniello, “A Tree KernelBased Approach for Clone Detection,” *Proc. 26th International Conference on Software Maintenance (ICSM)*, pp. 1–5, 2010.

[21] R. Koschke, “Large-scale Intersystem Clone Detection Using Suffix Trees and Hashing,” *Journal of Software: Evolution and Process*, Feb 2013, doi: 10.1002/smr.1592.

[22] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto, “Incremental Code Clone Detection: A PDG-based Approach,” *Proc. 18th Working Conference on Reverse Engineering (WCRE)*, pp. 3–12, 2011.

[23] Y. Higo, S. Kusumoto, “Code Clone Detection on Specialized PDGs with Heuristics,” *Proc. 15th European Conference on Software Maintenance and Reengineering(CSMR)*, pp. 75–84, 2011.

[24] A. Goto, N. Yoshida, M. Ioka, E. Choi, K. Inoue, “How to Extract Differences from Similar Programs? A Cohesion Metric Approach,” *Proc. 7th International Workshop on Software Clones(IWSC)*, pp. 23-29, 2013.

[25] R. Komondoor and S. Horwitz, “Using Slicing to Identify Duplicationin Source Code,” *Proc. 8th International Symposium on Static Analysis(SAS)*, pp. 40-56, 2001.

[26] 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.

[27] 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.

[28] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen, “a Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston,” *Proc. 9th Principles and Practice of Knowledge Discovery in Databases(PKDD)*, Vol. 3721, pp. 392-403, 2005.

[29] F. Deissenboeck, B. Hummel, E. Juergens et al. “Model Clone Detection in Practice,” *Proc. 4th International Workshop on Software Clones(IWSC)*, pp. 57-64, 2010.

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.