Chapter 3

Investigation of Parameters for Semantic and Model Clone Detection

Semantic clone detection and model based clone detection techniques are challenging upcoming areas. Available techniques are explored in detail to identify the parameters which acts as the basis for our work in developing the proposed techniques of clone detection. Advantages and shortcomings of all the existing techniques are studied. Section 3.1 explores various semantic clone detection approaches. Various model clone detection approaches are detailed in sec 3.2. In section 3.3, various parameters affecting software clone detection are identified. Finally, in section 3.4, we list the motivation behind model based clone detection.

3.1 Semantic Clone Detection

Two program fragments differing in their concrete syntax may be semantically very close. Detecting semantic equivalence is very difficult. It needs deep semantic analysis. There are some studies which tried to detect semantic clones. They are mostly approximations to type-4 clones. Table 3.1 compares and details different semantic clone detection techniques.

Table 3.1 Semantic clone detection and comparative analysis

<table>
<thead>
<tr>
<th>Author &amp; Tool Name</th>
<th>Normalizations/Transformations</th>
<th>Source Code Representation</th>
<th>Clone Matching Technique</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jens Krinke [137] Duplix</td>
<td>PDG</td>
<td>Fine grained PDG</td>
<td>n-length patch matching (maximal similar subgraphs)</td>
<td>High precision and recall</td>
<td>Needs a PDG generator for different language, works for C language</td>
</tr>
<tr>
<td>Komondoor &amp; Horwitz [130] PDG-DUP</td>
<td>CodeSurfer to PDG</td>
<td>PDG</td>
<td>PDG Subgraph matching using program slicing</td>
<td>Mechanical refactoring can lead to procedure extraction</td>
<td>Needs a PDG generator, very slow for large code bases</td>
</tr>
<tr>
<td>Choi et al. [36]</td>
<td>programs to partite sets and functions to vertices</td>
<td>Birthmarks</td>
<td>maximum weighted bipartite matching</td>
<td>Efficient, highly resilient</td>
<td>Needs deobfuscation methods against attacks</td>
</tr>
<tr>
<td>Author &amp; Tool Name</td>
<td>Normalizations/Transformations</td>
<td>Source Code Representation</td>
<td>Clone Matching Technique</td>
<td>Advantages</td>
<td>Disadvantages</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------</td>
<td>---------------------------</td>
<td>--------------------------</td>
<td>------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Marcus &amp; Maletic [172]</td>
<td>Comment removal and token regularization</td>
<td>Text</td>
<td>Vector representation using LSI</td>
<td>Finds high level structural clones</td>
<td>Highly dependent on comments, low precision</td>
</tr>
<tr>
<td>Gabel et al. [60] Enhanced Deckard</td>
<td>CodeSurfer to PDG to AST</td>
<td>PDG</td>
<td>Characteristic vectors in Euclidean Space</td>
<td>Highly scalable</td>
<td>Needs a PDG generator, slow for large code bases</td>
</tr>
<tr>
<td>Jiang &amp; Su [97] EqMiner</td>
<td>program text to intermediate language</td>
<td>C Intermediate Language</td>
<td>Automated Random Testing</td>
<td>Scalable</td>
<td>Works for C programs only</td>
</tr>
<tr>
<td>Kim et al. [123] MeCC</td>
<td>Semantic based static program analysis tool</td>
<td>Abstract Memory States</td>
<td>Abstract Memory state comparison</td>
<td>Precise, can be used to identify bugs, inconsistencies, plagiarism</td>
<td>More false positives, semantic based static analyzer takes lot of time</td>
</tr>
<tr>
<td>Philipp Schugerl [205] DL_Clone</td>
<td>AST to description logic</td>
<td>Description logic</td>
<td>Semantic web reasoner</td>
<td>Scalable, can be parallelized</td>
<td>Fails to detect smaller clones and clones across methods</td>
</tr>
</tbody>
</table>

PDG is a directed attributed graph representation for several program analyses and transformations. The edges in the PDG represent the control dependencies and flow dependencies. The clone detection lies in finding isomorphic subgraphs of PDG that represent clones. There are techniques by Krinke [137], Komondoor and Horwitz [130], Gabel et al. [60] who transformed the source code to PDG for clone detection. Krinke [137] used k-length patch matching to detect maximal induced subgraphs. The technique worked well with reasonable precision and recall on sample software systems. Komondoor and Horwitz [130] used program slicing to detect isomorphic subgraphs in the PDG. The technique uses backward and forward slicing and able to detect good clone candidates for procedure extraction. It can detect non-contiguous clones. Gabel et al. [60] presented a scalable technique to detect semantic clones from the PDG representation of the source code. The key element of the algorithm is to map the NP-complete graph isomorphism problem to tree similarity. The tree similarity is based upon comparing characteristic vectors. Upon empirical evaluation, the tool has good execution times on large code bases.
Marcus and Maletic [172] applied latent semantic indexing (LSI) on the textual representation of source code to identify semantic similarities across functions/files/programs. LSI is a vector-based statistical method to represent meanings of comments and identifiers of source code. They tried to detect similar high-level concept clones e.g. abstract data types.

Software birthmarks have been used successfully to detect copied programs and software theft. Choi et al. [36] used a set of API calls to detect similar programs. The similarity technique depended upon maximum weighted bipartite matching. In this way, the method is useful in detecting semantic equivalences duplications in case of software thefts. However, the technique is vulnerable to deobfuscation attacks.

There is only one study by Jiang and Su [97] which identified functionally equivalent code fragments of arbitrary size depending on the input–output behavior of a piece of code. They detected two pieces of code that always produce the same output on random inputs although they are syntactically different. They defined functional equivalence as a special case of semantic equivalence. The results were validated by applying random tests. The tool was scalable and able to work on million lines of code finding that 58% of functionally equivalent code was syntactically different. The technique worked for C programs only.

Kim et al. [123] proposed MeCC, a semantic clone detector based on a path-sensitive semantic-based static analyzer. The analyzer was used to estimate the memory states at each procedure's exit point; then memory states were compared to determine clones. The authors compared their findings with CCFinder and Deckard. MeCC is able to detect a larger number of procedural clones, to trace inconsistencies, identify refactoring candidates and understand software evolution related to semantic clones.

Philipp Schugerl [205] presented a novel technique to detect global clones. An abstract syntax tree representation of the source code is normalized in the form of description logic. Then a semantic web reasoner is applied to trace similar source code based on control-blocks and used data types. The author compared the technique with state-of-the-art clone detection tools but only on Java source code. The technique is highly scalable with the use of semantic web reasoner for match detection.

Several authors suggested areas for further research. Marcus and Maletic [172]
proposed the combination of multiple detection algorithms in future. A number of hybrid clone detection algorithms were developed as a result. We noticed that PDG based approaches of detecting semantic clones are quite slow. Future work [63] lies in developing the framework to help in fast generation of PDG from source code. So, approximate solutions of mapping PDG to trees as by Gabel et al. [60] by applying tree similarity technique are more scalable. Choi et al. [36] proposed both extending birthmarks with more information and making technique robust against attacks. Philip Schugerl [205]’s technique can be parallelized using a cluster of computers. Empirical evaluation across more subject systems would help identify future extensions of the tool which currently does not detect small clones and clones across method. Jiang and Su [97] proposed exploring future research on functionality-equivalent code refactoring and reuse. A general method for different programming languages should be developed to detect functionally equivalent but syntactically different code fragments. *MeCC* [123] can be extended by adapting a static analyzer to collect memory states for any arbitrary code blocks to make clone detection possible at finer granularity.

### 3.2 Model based Clone Detection

With the rise in abstraction, model driven development has turned to be an emerging area. Large models are developed using UML, Matlab/Simulink, domain specific modeling languages, etc. The presence of duplicated sub structures in different types of models cannot be ruled out. Thus, detecting clones in models is an emerging area. But, model based clone detection techniques are still in their infancy. So this classification includes studies and tools related to detection of duplication in different diagrams and models as summarized in Table 3.2.

Liu et al. [158] detected duplications in sequence diagrams by converting the 2-dimensional sequence diagram to a 1-dimensional array. Then, the 1-dimensional array is used to build suffix tree. Common prefixes are identified from the suffix tree in the form of reusable sequence diagram as refactoring candidates. The study confirmed the presence of 14% duplication in sequence diagrams of sample industrial projects.

The automatic detection of clones in models leads to identification of potential domain specific library elements [45]. Deissenboeck et al. [45] used *ConQAT* [100] as an integrated framework to detect clones in Matlab/Simulink models especially in
automotive domain. The tool *CloneDetective*, which is part of the *ConQAT* framework, works by representing the model as a normalised multi-graph where labels are assigned to relevant blocks. Similarity between blocks is checked by a heuristic which performs a depth first search based looking for matched pairs. After detection, clones are clustered based on set of nodes using union function.

Table 3.2 Model based clone detection and findings

<table>
<thead>
<tr>
<th></th>
<th>Liu et al. [158]</th>
<th>Pham et al. [187]</th>
<th>Deissenboeck et al. [45]</th>
<th>Herald Storrle [216]</th>
<th>Hummel et al. [85]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preprocessing/ Normalization</strong></td>
<td>Two dimensional sequence diagrams into one dimensional array</td>
<td>Transformation of models to graphs, assigning labels to relevant blocks</td>
<td>Transformation of models to graphs, assigning labels to relevant blocks</td>
<td>XMI files from UML domain models</td>
<td>Transformation of models to graphs, assigning labels to relevant blocks</td>
</tr>
<tr>
<td><strong>Source Representation</strong></td>
<td>One dimensional array</td>
<td>Sparse, labeled directed graph</td>
<td>Labeled multigraph</td>
<td>Prolog code</td>
<td>Directed, labeled multigraph</td>
</tr>
<tr>
<td><strong>Clone Matching Technique</strong></td>
<td>Suffix Tree</td>
<td>Canonical matching, vector based approach</td>
<td>Maximum Weighted Bipartite Matching</td>
<td>Model matching</td>
<td>Canonical matching, clone index based hashing</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>High precision and recall</td>
<td>algorithm is able to detect model fragments with modifications, incremental</td>
<td>Scalable</td>
<td>Supports refactoring</td>
<td>Incremental, distributed, fast detection time</td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td>Works only for sequence diagrams</td>
<td>Lower precision</td>
<td>Large number of false positives</td>
<td>Complex Java Implementation</td>
<td>Infeasible for large subgraphs</td>
</tr>
<tr>
<td><strong>Application Area</strong></td>
<td>Sequence Diagrams</td>
<td>Matlab/Simulink models</td>
<td>Matlab/ Simulink/ Targetlink models</td>
<td>UML Domain models</td>
<td>Matlab/ Simulink models</td>
</tr>
<tr>
<td><strong>Model Clone Granularity</strong></td>
<td>Extractable fragment of a sequence diagram</td>
<td>Number of blocks</td>
<td>Number of blocks</td>
<td>Sub models</td>
<td>Sub models</td>
</tr>
<tr>
<td><strong>Tools</strong></td>
<td><em>Duplication Detector</em></td>
<td><em>ModelCD</em></td>
<td><em>CloneDetective</em></td>
<td><em>MQclone</em></td>
<td>Integrated in <em>ConQAT</em></td>
</tr>
</tbody>
</table>
Pham et al. [187] presented two algorithms namely escan and ascan to detect clones in models. These were incorporated in the tool ModelCD. Firstly, the model was preprocessed to be represented as a parsed, labeled directed graph. escan was used to detect exact matching using an advanced graph matching technique called canonical labeling and ascan was used to detect approximate matching by counting vector of sequence of nodes and edges’ labels. The technique is incremental in the way it generates candidate cloned sub graphs. Their tool ModelCD was compared with CloneDetective which is included in the ConQAT framework [45]. For the same clone granularity and subject systems, both the tools were compared based on 4 parameters: Precision, completeness, scalability, incrementality. ModelCD performed better. In a subsequent paper, Deissenboeck et al. [47] discussed the presence of a large number of false positives in Matlab/Simulink models, pointing out that it is important to identify relevant clones. Model clones suffer from the problem of scalability, clone inspection and relevance. Their study provided useful insights in addressing these problems in real time industrial context. The authors proposed reducing the size of models to make clone detection speedier. Firstly, removal of obvious cloned sub-systems is carried out. After which, as proposed by Pham et al. [187], all nodes with high degree are removed. After the detection process is complete, the algorithm tries to connect smaller nodes that are connected to each other over high degree nodes. Deissenboeck et al. [47] then compared their enhanced version of ConQAT and escan showing ConQAT has a faster execution times than escan.

Störrle [216] pioneered the detection of clones in UML domain models. The technique was based on model querying. Using any of the UML case tools, XMI files are generated from UML domain models. These files are transformed into Prolog files. A model is input in the query and using model matching, the output is generated. We observed that the tool is still to be verified for heterogeneous subject systems. Hummel et al. [85] introduced an incremental algorithm for model clone detection. A Matlab/Simulink model is pre-processed by flattening the model into a directed multigraph. Then, relevant edges and blocks are labeled. A clone index is created for all subgraphs of same size. Canonical label of all subgraphs in the clone index is calculated and similar labels are hashed. Clone retrieval and index update are integrated for fast retrieval. It has not been verified on large models.
It is still to be verified whether canonical matching and vector based approach can be applied on other graph based models like UML models. We observed one study by Deissenboeck et al. [47] highlighting the practical issues to be resolved in model clone detection. The study pointed out that ranking of clones may help in improving scalability and relevance of model clones. A comprehensive model clone detection tool for UML models and other data flow languages is missing. Different forms of models have individual features which need to be exploited for clone detection [177]. We noticed vagueness in the definition of model clones which hinders understanding of the topic area [69].

3.3 Parameters for software semantic clone detection

From the comparative analysis of various semantic clone detection approaches, we concluded that the large number of techniques transform the source code to PDG. Krinke [137], Komondoor and Horwitz [130], Gabel et al. [60] are some of these techniques. We observed that these techniques are not scalable to run for large code bases primarily due to slow generation of clone pairs. After the transformation to PDG, source code is represented as a graph. Then, various researchers tend to apply graph mining techniques upon that to mine patterns. The application of subgraph isomorphism to this graphical representation of source code makes the technique computationally tough as subgraph isomorphism is NP-complete. Scalability of the PDG based technique is another challenge.

Due to the difficulty and cost associated in detection of semantic clones, we tend to orient towards methods of clone detection which are more close to implementation i.e. the use of object oriented models. Thus, we carried out a comprehensive analysis of sample forward designed and reverse engineered models. The main objective of this analysis is to identify key parameters for the proposed model clone detection technique. UML class models are created by reverse engineering using standard modeling tools. The UML class models are exported to intermediate representation i.e. .mdl and .xml. XMI file of sample open source subject system eclipse-ant is shown in Fig. 3.1. From the intermediate representation, we were able to identify key parameters which affect software clone detection and acts as the basis for our model clone detection approach.
We used following UML tools for comprehensive analysis of UML class models:

- Magic Draw
- Altova Spy
- MS Visual Studio
- Visual Paradigm for UML
- ConQAT
- Eclipse and various plug-ins

![Image of XMI file for eclipse-ant](image.png)

Fig. 3.1 XMI file for eclipse-ant

Our analysis of intermediate files of object oriented models revealed following parameters which affect software semantic clone detection:

- Name of the class
- Name of the fields, methods
- Return type of methods
- Number of arguments in methods
• Data type of arguments in methods
• Inheritance
• Association
• Dependency
• Realization

Thus our technique of clone detection for object oriented models is based on extracting the above said parameters from the models.

By studying the existing techniques of semantic and model clone detection, we identified following parameters to evaluate the effectiveness of software clone detection technique:

• Scalability
• Clone Coverage
• Frequency
• Precision
• Recall
• F-Measure

Above attributes are defined in chapter 4 and chapter 5 at the time of evaluation of proposed techniques.

3.4 Motivations behind Model Clone Detection

Models are being used as a way to express design. We are motivated by following reasons:

• Increasing size and complexity of software systems promotes the use of models for comprehensibility and better understanding.

• Due to the increase in use of Unified Modeling Language and Model Drive Architecture, models are replacing code as core artifacts.

• Models are independent of programming languages. Various software modeling tools permit automatic code generation in different languages.
- Models give architectural details which help in identifying parameters which are close to semantics [169].

- Model based system development simplifies design and offers better compatibility between subsystems, thereby promoting communication between individuals and teams.

3.5 Summary

In this chapter, we have discussed various semantic clone detection and model based clone detection techniques. These techniques are compared based upon different characteristics. Advantages and shortcomings of all the existing techniques are mentioned. We have mentioned the process of analysing the forward designed and reverse engineered models to identify the parameters which act as the basis for our work. Finally, we have listed various factors which acted as the motivation for our work to devise model clone detection techniques.