ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Volume 1, Issue 9, November 2013

Identification of Structural Clones Using Association Rule and Clustering

S.Brintha Rajakumari

Assistant professor, Department of CSE, Bharath Institute of Science and Technology, Chennai, TamilNadu, India.

ABSTRACT: Code clones are similar program structures of considerable size and significant similarity. Simple clone set formed by similar code fragments in software. The problem is the huge number of simple clones typically reported by clone detection tools. We observed that recurring patterns of simple clones – so-called structural clones - often indicate the presence of interesting design-level similarities. We propose a technique to detect some specific types of structural clones from the repeated combinations of co-located simple clone. We find the patterns of co-occurring clones in different files using the frequent item set mining (FIM) technique. Finally, we perform file clustering to detect those clusters of highly similar files that are likely to contribute to a design-level similarity pattern. We implement the structural clone detection technique in a tool called CCFinder. Detection of clones provides several benefits in terms of maintenance, program understanding, reengineering and reuse.

KEYWORDS: Design concepts, maintainability, reengineering and reusable software.

I. INTRODUCTION

CODE clones are similar program structures of considerable size and significant similarity. Several studies suggest that as much as 20-50 percent of large software systems consist of cloned code. Knowing the location of clones helps in program understanding and maintenance. The detection and subsequent resolution of clones by refactoring, Function calls, macros and templates etc., however, promises decrease in maintenance costs and code size.

In the past decade, clone detection and resolution has got considerable attention from the software engineering research community and many clone detection tools clone detection has been focused on detecting similar code fragments – so-called simple clones. We observed that recurring patterns of simple clones often indicate the presence of interesting higher-level similarities that we call Structural clones whose unification not only brings more size reduction, but also helps in understanding the design of the system for better maintenance and future enhancement. The limitation of considering only simple clones is known in the field. The main problem is the huge number of simple clones typically reported by clone detection tools. There have been a number of attempts to move beyond the raw data of simple clones. We observed that at the core of the structural clones, often there are simple clones that coexist and relate to each other in certain ways.

We proposed a technique to detect some specific types of structural clones from the repeated combinations of colocated simple clones. We implemented the structural clone detection technique in a tool called CCFinder, implemented in C++. It has its own token-based simple clone detector. Our structural clone detection technique works with the information of simple clones, which may come from any clone detection tool. It only requires the knowledge of simple clone sets and the location of their instances in programs. As structural clones often represent some domain or design concepts, their knowledge helps in program understanding, and their detection opens new options for design recovery that are both practical and scalable. Representing these repeated program structures of large granularity in a generic form also offers interesting opportunities for reuse and their detection becomes useful in the reengineering of legacy systems for better maintenance.

We can find clone patterns in different units of code, either methods or classes or components or modules, gaining useful insights into the cloning situation at different levels of abstraction. We have initially tried this approach at file level, by finding the frequently occurring clone patterns in different files and analyzing those patterns, with promising results. Detecting the frequently co-occurring clone classes in different files, we can isolate the groups of files that have strong similarity with each other. This is achieved by a clustering algorithm that we have devised for this particular problem. These clusters of highly similar files form basic structural clones.

The remainder of this paper is organized as follows: In Section 2, we define types of structural clones-higher level similarities in programs. Section 3 describes our detecting structural clones with data mining. Section 4 describes the implementation of CCfinder. Section 5 describes a mechanism to create generic representation of structural clones found in the system for better maintenance and reuse. Section 6 presents the related work in higher-level similarities and design recovery. Section 8 concludes the paper and presents future work.

II. STRUCTURAL CLONES- HIGHER LEVEL SIMILARITIES IN PROGRAMS

We describe in detail the phenomenon of higher level similarities, which we call structural clones. We define structural clones as similar program structures that can be analyzed hierarchically, at many levels of abstraction, with similar code fragments at the bottom of such hierarchy. Locating these higher level similarities can have significant value for program understanding, evolution, reuse, and reengineering.

2.1 FROM SIMPLE CLONES TO STRUCTURAL CLONES

We primarily focus on similarity patterns representing design concepts or solutions that can be of significant importance in the context of understanding, maintaining, reengineering or reusing programs. We use the term structural clone to mean similar program structures that are configurations of lower-level similar program entities. Therefore, our structural clones may form a hierarchy of clones, with cloned code fragments at the bottom level.

2.1.1 FILE-LEVEL STRUCTURAL CLONE

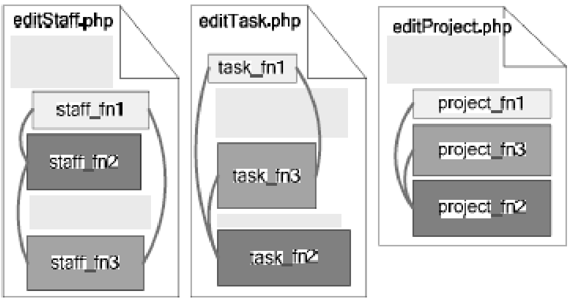


Fig 1- File level Structural clone

Functions shown in the same shade are clones of each other (e.g., staff_fn1, task_fn1, project_fn1). The Relationship between the functions is ‘same file’, which holds between fragments of the same file, regardless of the order in which they appear. The three host files editStaff.php, editTask.php and editProject.php perform similar tasks, but belong to three different modules (i.e., Staff module, Task module, and Project module). Provided these structural clones cover a substantial portion of the host files, we can consider the three files as abstract entities that are clones of each other, as discussed in the previous section. This illustrates how the concept of structural clones helps us to move from smaller entities (in this case, functions) to larger entities (in this case, files). These files can now be considered as entities in forming a higher level structure.

2.1.2 A MODULE LEVEL STRUCTURAL CLONE

According to the definition, structural clones of higher granularity can be made up of structural clones of lower granularity. For example, a module-level structural clone can consist of file-level structural clones. Such a situation is illustrated by the structural clone found in a web portal implementation, as shown in Figure 2. In this portal, files belonging to each module are stored in a separate folder. Each module contains a set of files providing module-specific implementation of certain common functionalities (e.g., create, display, edit, delete). When the module functionalities

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Volume 1, Issue 9, November 2013

are similar, each of these common files ends up being file-level structural clones of their counterparts in other modules. One such case was the basis for previous example. At a larger granularity, the modules Staff, Task and Project can be considered structural clones where each structure has four files create[M].php, display[M].php,edit[M].php,delete[M].php([M]=Staff, Task, Project) as entities and the relationship 'same folder' among the entities (relationships are not shown in figure). Note that the module Project does not carry a deleteProject.php file. Still, there was enough similarity among Project and other modules to consider all of them as structural clones of each other.

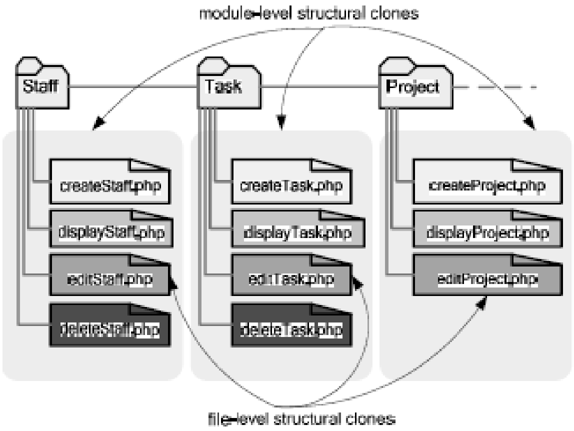


Fig 2 – Module Level Structural clone

2.1.3 MULTIPLE STRUCTURAL CLONES IN THE SAME FILE

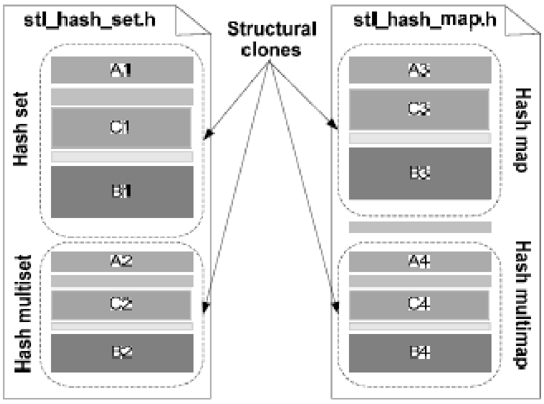


Fig 3 – Multiple Structural Clones in the same file

Multiple Structural clones can also exist in the single file fig 2 shown The four structural clones are structures of code fragments that are part of the different templates representing various hashed associative containers. Each structural clone covers a significant part of the template it belongs to; hence we can consider these templates as 'abstract entities' (ignoring their internal structure) and form a clone class of four clones at the next higher level. Furthermore, if the two templates present in one file are joined to each other with the 'same file' relationship, we have a structural clone class of two structures, one in each file. Raising the level of abstraction by one more step, we observed that the two templates present in each file cover the files significantly, so the files can also be considered as 'abstract entities', forming a clone class of two cloned files.

2.1.4 CROSSCUTTING STRUCTURAL CLONES

Structural clones can crosscut files (or classes, modules etc.), as the example in Figure 4 shows. This example involves three PHP files belonging to a Web portal module that supports two similar crosscutting features. This results in two structural clones, each consisting of code fragments belonging to one of the two crosscutting features.

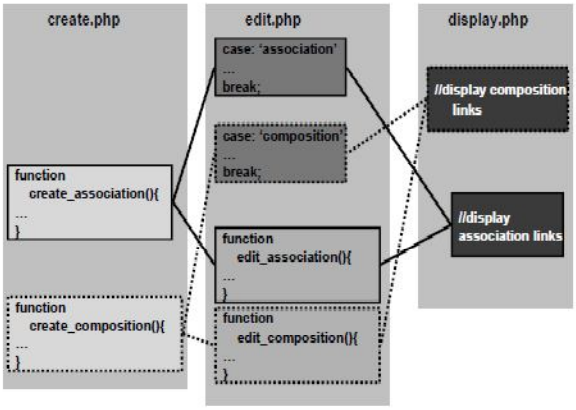


Fig 4 – Two crosscutting structural clones

2.1.5 STRUCTURAL CLONES BASED ON INHERITANCE HIERARCHY

The relationship(s) between entities of a structural clone can vary widely. In object-oriented systems, a set of entities related by inheritance can be used to define a structural clone. We found such a case in the Buffer library (java.nio.*) Figure 5 shows two instances (out of seven) of the structural clone, each consisting of seven Java classes. More information on the structure of the Buffer library, and how 'feature combinatorics' problem gave rise to this structure.

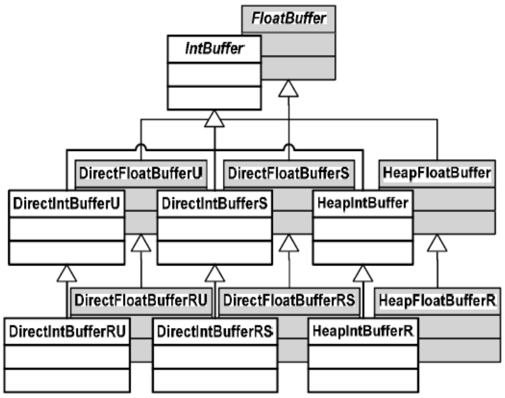


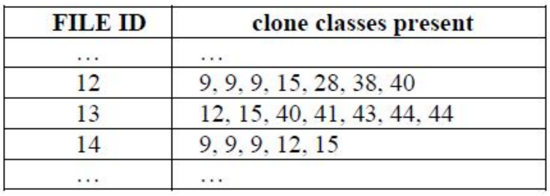
Fig 5 – Structural Clones based on hierarchy

III. DETECTING STRUCTURAL CLONES WITH DATA MINING

This section focuses on our approach to detect some specific types of structural clones from the bottom up analysis of similarities using a data mining technique similar to the well-known market basked analysis. This analysis builds on the detection of simple clones discussed in the previous chapter. We introduce an iterative approach for the detection of structural clones by moving from the low-level similarities to the higher level similarities. To raise the level of analysis, we use abstraction of entities based on clone coverage. The higher level entities that have significant low-level cloning are grouped together. These groups of entities form the basic similarity blocks for the next higher level analysis.

3.1 FINDING RECURRING PATTERNS OF SIMPLE CLONE CLASSES

Here we describe the detection of patterns of simple clones in a file - the first level of structural clones. The same technique can be applied to detect structural clones at other levels, as will be described later. An example of this format is shown in Figure 6. After detecting simple clone classes in a system, the data of simple clone classes is organized in terms of files represented by their IDs. The first data row says that the file with file ID 12 contains three clone instances belonging to clone class 9 and one instance from each of the clone classes 15, 28, 38, and 40. The interpretation is likewise for the other data rows.

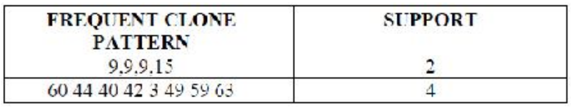


FILE ID	clone classes present
…	…
12	9, 9, 9, 15, 28, 38, 40
13	12, 15, 40, 41, 43, 44, 44
14	9, 9, 9, 12, 15
…	…

Fig 6 - Simple clone classes listed per file

To detect the recurring patterns of simple clones in different files, we apply the “market basket analysis” technique from the data mining domain. The idea behind this technique is to find the items that are usually purchased together by different customers from a departmental store. These patterns of clone classes will act as the unique representation for a group of files, and depending upon its significance in terms of files’ coverage, will lead to identifying groups of highly similar files. This will be the next level of structural clones. Market basket analysis is done with “frequent itemset mining (FIM)”. The difference between our problem and the standard problem for frequent itemset mining is that in FIM, the items in a transaction are considered unique, whereas in our data, one file may contain multiple instances of the same clone class. We can normalize the data by removing by doing so, we miss out the important information, as multiple occurrences of the instances of same clone class in different files is a valid pattern of clones. For example, 9, 9, 9, 15 is a valid clone pattern represented in File 12 and File14 in Figure 6.

Mining all frequent itemsets returns many frequent itemsets that are subsets of bigger frequent itemsets. The correct solution in our case is to perform “Frequent Closed Itemset Mining” (FCIM), where only those itemsets are reported which are not subsets of any bigger frequent itemset. One of the parameters for FCIM is the support count, which means the number of files that contain the detected pattern of simple clones In our case, we have hard coded the support to be 2, so that it will report a clone pattern, even if it is present only in 2 files, as it could be still be significant for maintenance based on its size. The output from FCIM is in the format shown in Figure 7. Each row represents one frequent clone pattern along with its support count, indicating the number of files containing this clone



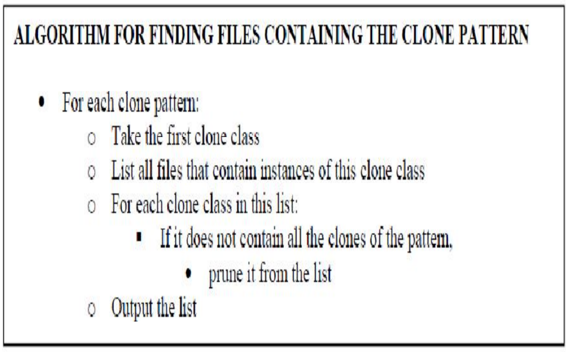
FREQUENT CLONE PATTERN	SUPPORT
9,9,9,15	2
60 44 40 42 3 49 59 63	4

Fig 7 - frequent clone patterns with support count

FCIM only deals with detecting frequent patterns. These clone patterns can also be considered as unrestricted gapped clones, where any number of gaps are allowed with arbitrary size and ordering. More work is required to isolate clone patterns where the gaps are small and the clones are more cohesive, to have more meaningful gapped clones. Finding gapped clones in this way also provide the flexibility to detect rearranged gapped clones, where the cloned parts can occur in arbitrary order and should not necessarily be arranged in the same way. In figure 8 algorithms for finding clone pattern.

3.2 CLUSTERING HIGHLY CLONED FILES

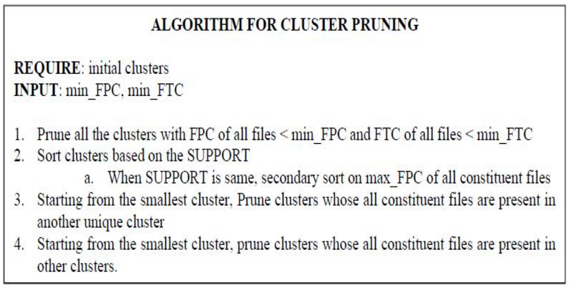
To measure file coverage by a clone pattern, we calculate two metrics, namely the File Percentage Coverage (FPC), which indicates the percentage of a file covered by a clone pattern, and the File Token Coverage (FTC), which tells the number of tokens in a file covered by the clone pattern. These metrics are calculated for each file containing the clone pattern. One complication here is that some clones may overlap in a file, as discussed earlier, so we cannot simply add up the size of all clones in a pattern to find the file coverage. The clustering based on these values and other parameters can also be made totally customizable to suit the needs of the different users. Currently, we let the user specify a minimum FPC and FTC value to indicate the significance of a cluster. The cluster will be considered significant even if one file has the FPC or FTC value greater than threshold values. The expected output is to find all the significant clusters that cover maximum number of files and no file is preferably repeated in two clusters.



```
ALGORITHM FOR FINDING FILES CONTAINING THE CLONE PATTERN

• For each clone pattern:
    o Take the first clone class
    o List all files that contain instances of this clone class
    o For each clone class in this list:
        ▪ If it does not contain all the clones of the pattern,
            • prune it from the list
    o Output the list
```

Fig 8 – Algorithm for finding Simple Clone pattern



```
ALGORITHM FOR CLUSTER PRUNING

REQUIRE: initial clusters
INPUT: min_FPC, min_FTC

1. Prune all the clusters with FPC of all files < min_FPC and FTC of all files < min_FTC
2. Sort clusters based on the SUPPORT
       a. When SUPPORT is same, secondary sort on max_FPC of all constituent files
3. Starting from the smallest cluster, Prune clusters whose all constituent files are present in
   another unique cluster
4. Starting from the smallest cluster, prune clusters whose all constituent files are present in
   other clusters.
```

Fig 9 - Algorithm for cluster pruning

Step 1 of the algorithm says that we remove from consideration all those clusters where no file passes the minimum criteria of FPC and FTC values. These are clusters where very small clones exist between files. In step 2, we sort clusters based on their support count. When the support count of two or more clusters is same, the clusters are sorted based on the maximum FPC value of the constituent files. Steps 3 and 4 prune clusters. These clusters of highly similar files give us the next level of structural clones that we call file clone classes or FCC.

IV. TOOL IMPLEMENTATION

CCFinder implements the structural clone detection techniques presented in this paper. CCFinder is written in C++, and it has its own token-based simple clone detector [6]. For frequent closed item sets mining (FCIM), we are using the algorithm from [21] .For manipulation of clones' data, CCFinder makes use of the STL containers from the standard C++ library. The output from CCFinder is generated in the form of text files so that any visualization tool developed in the future can easily interface with Clone Miner. For performance evaluation, we ran CCFinder on full J2SE 1.5 source code,3 consisting of 6,558 source files in 370 directories, 625,096 LOC (excluding comments and blank lines), and

70,285 methods, using different values of minimum clone size. For forming FCSets and MCSets, a value of 50 tokens is used for the clustering parameter minLen, where the Len is measured in terms of tokens. Likewise, for minCover, a value of 50 percent is used in all cases. The tests were run on a Pentium IV machine with 3.0 GHz processor and 1GB RAM. Each time it took around two to three minutes to run the whole process from finding simple clones to the analysis of files, methods, and directories for structural clones.

V.STRUCTURAL CLONES IN SOFTWARE MAINTENANCE AND REUSE

To improve the design of the legacy systems, various re-structuring or refactoring techniques can be applied [Opd92] [Fow99]. Analysis of structural clones is helpful in locating places where high-level duplication is present, which can be restructured or refactored. For redesigning the system to enhance maintainability of the legacy code, we propose some Structural clone based techniques. A good starting point is to analyze the file clone classes (i.e., groups of cloned files). After choosing file clone classes for refactoring, simple clones or method clones within those groups of files can be more easily refactored because of the context information. It may also be possible to apply several small refactoring simultaneously, for example moving together several cloned methods to the parent class, or simply changing the inheritance structure to remove duplicates. Having only the knowledge of simple clones, possibility of making such bigger changes is not very apparent, and one has to go step by step, with the risk of missing the bigger picture altogether. Analysis can also be done at code fragments, methods, or directories level, depending on how intense the cloning is and how major reengineering is practical. The other analysis features built inside Clone Analyzer, such as the structural clone configurability and the Diff feature can aid the user in the finer details of refactoring. This proposed method is somewhat general due to the varying objectives; it gives a basic framework for the analysis process.

VI. RELATED WORKS

Clone detection tools produce an overwhelming volume of simple clones' data that is difficult to analyze in order to find useful clones. This problem prompted different solutions that are related to our idea of detecting structural clones. Clone detection techniques using Program Dependence Graphs (PDG). In addition to the simple clones, these tools can also detect noncontiguous clones, where the segments of a clone are connected by control and data dependency information links. Such clones also fall under the premise of structural clones. While our technique detects structural clones with segments related to each other based only on their colocation, with or without information links, the PDG-based techniques relate them using the information links only. Moreover, the clustering mechanism in Clone Miner, to identify to identify groups of highly similar methods, files, or directories based on their contained clones, is missing from these techniques.

PR-Miner is another tool that discovers implicit programming rules using the frequent item set technique. Compared to structural clones found by Clone Miner, these programming rules are much smaller entities, usually confined to a couple of function calls within a function. The work by Ammon's et al. is also similar, finding the frequent interaction patterns of a piece of code with an API or an ADT, and representing it in the form of a state machine. These frequent interaction patterns may appear as a special type of structural clone, in which the dynamic relationship of cloned entities is considered. Similar to Clone Miner, this tool also helps in avoiding update anomalies, though only in the context of anomalies to the frequent interaction patterns.

VII. CONCLUSIONS

We emphasized the need to study code cloning at a higher level. We introduced the concept of structural clone as a repeating configuration of lower-level clones. We presented a technique for detecting structural clones. The process starts by finding simple clones (that is, similar code fragments). Increasingly higher-level similarities are then found incrementally using data mining technique of finding frequent closed item sets, and clustering. We implemented the structural clone detection technique in a tool called Clone Miner. While Clone Miner can also detect simple clones, its underlying structural clone detection technique can work with the output from any simple clone detector. Structural clone information leads to better program understanding, maintenance, reengineering and reuse.

REFERENCES

[1] H.A. Basit and S. Jarzabek, “Detecting Higher-Level Similarity Patterns in Programs,” Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng., pp. 156-165, Sept. 2005

[2] J.R. Cordy, “Comprehending Reality: Practical Barriers to Industrial Adoption of Software Maintenance Automation, “Proc. 11th IEEE Int’l Workshop Program Comprehension, (keynote paper), pp. 196-206, 2003.

[3] A.De Lucia, G. Scanniello, and G. Tortora, “Identifying Clones in Dynamic Web Sites Using Similarity Thresholds,” Proc. Int’l Conf. Enterprise Information Systems, pp. 391-396, 2004.

[4] J.Y. Gil and I. Maman, “Micro Patterns in Java Code,” Proc. 20th Object Oriented Programming Systems Languages and Applications, pp. 97-116, 2005.

[5] G.Grahne and J. Zhu, “Efficiently Using Prefix-Trees in Mining Frequent Itemsets,” Proc. First IEEE ICDM Workshop Frequent Itemset Mining Implementations, Nov. 2003.

[6] J. Han and M. Kamber, “Data Mining: Concepts and Techniques”. “Morgan Kaufman Publishers”, 2001.

[7] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, “ARIES:Refactoring Support Environment Based on Code Clone Analysis,”Proc. Eighth IASTED Int’l Conf. Software Eng. and Applications, pp. 222-229, Nov. 2004.

[8] S. Jarzabek, “Effective Software Maintenance and Evolution: Reused- Based Approach”. “CRC Press, Taylor and Francis”, 2007.

[9] S. Jarzabek and S. Li, “Unifying Clones with a Generative Programming Technique: A Case Study,” J. Software Maintenance and Evolution: Research and Practice, vol. 18, no. 4, pp. 267-292, July 2006.

[10] C. Kapser and M.W. Godfrey, “Toward a Taxonomy of Clones in Source Code: A Case Study,” Proc. Int’l Workshop Evolution of Large Scale Industrial Software Architectures, pp. 67-78, 2003.

[11] C. Rich and L.M. Wills, “Recognizing a Program’s Design: Ac Graph-Parsing Approach,” IEEE Software, vol. 7, no. 1, pp. 82-89, Jan. 1990.