SOFTWARE DESIGN ANALYSIS WITH DYNAMIC SYSTEM RUN-TIME ARCHITECTURE DECOMPOSITION

Lei Wu[1] and Sankalp Vinayak[2]

[1]Software Engineering, University of Houston-Clear Lake, Houston, Texas U.S.A
[2]Computer Science, Clear Creek Independent School District, Houston, Texas U.S.A

ABSTRACT

Software re-engineering involves the studying of targeting system's design and architecture. However, enterprise legacy software systems tend to be large and complex, making the analysis of system design architecture a difficult task. To solve this problem, the study proposes an approach that dynamically decomposes software architecture using the run-time system information to reduce the complexity associated with analyzing large scale architecture artifacts. The study demonstrates that dynamic architecture decomposition is an efficient way to limit the complexity and risk associated with re-engineering activities of a large legacy system. This new approach divides the system into a collection of meaningful modular parts with low coupling, high cohesion, and a minimal interface. This division facilitates the design analysis and incremental software re-engineering process. This paper details two major techniques to decompose legacy system architecture. The approach is also supported by automated reverse engineering tools that were developed during the course of the study. The preliminary results indicate that this novel approach is very promising.

KEYWORDS

Design analysis, reverse engineering, software architecture, dynamic analysis, system decomposition

1. INTRODUCTION

Software re-engineering involves the study of the target system's architecture and design. However, enterprise legacy software systems tend to be large and complex [1][2][3][32]. System decomposition is important to limit the complexity and risk associated with the re-engineering activities of a large legacy system [3][4][5][6].

Architecture decomposition divides the system into a collection of meaningful modular parts with low coupling, high cohesion, and a minimal interface, thus facilitating the incremental approach to implement the progressive software re-engineering process [4][7][8][33][30][37].

To fulfill this goal, the study developed two major techniques to decompose legacy system architecture. The approach is supported by reverse engineering tools that developed during the course of the study. The preliminary results indicate that this novel approach is very promising.

DOI : 10.5121/ijsea.2017.8106

2. SYSTEM RUNTIME DYNAMIC ANALYSIS & VISUALIZATION

System usability is embodied in detailed atomic system functionalities, which represent the concrete utility of the system [9][24][21][45]. This visualization and dynamic analysis technique enhance the linkage between legacy source code construction and system functionality based on dynamic system run-time information, thereby facilitating the legacy system decomposition based on system execution information (see Figure 1).

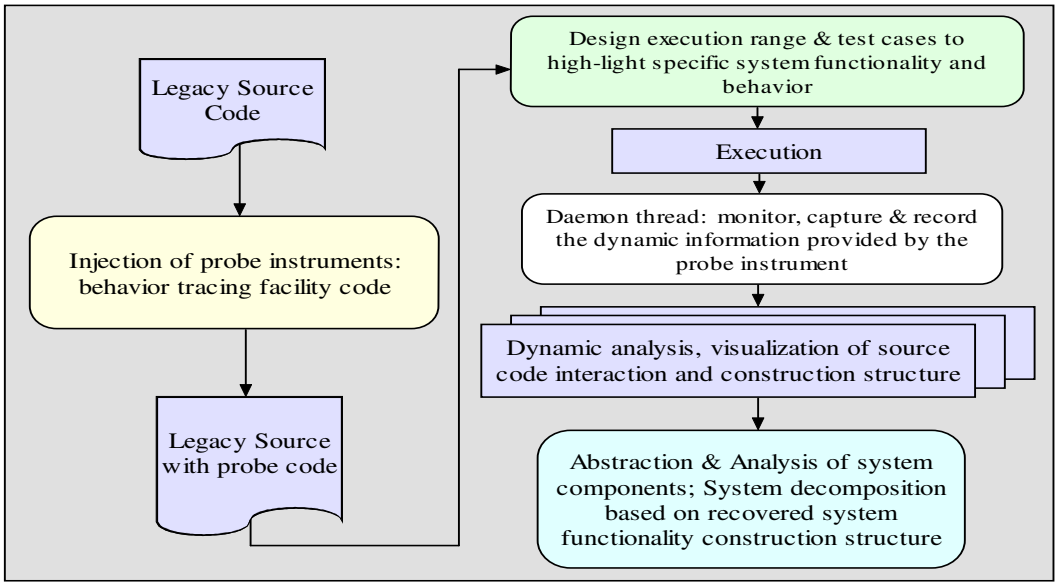


Figure 1: Dynamic Analysis & Visualization Process

A modified legacy source version is generated by injecting a probe code into original source code. During the execution of a legacy system with detection code, a range of test cases allows for retrieval of specific system functionality. Additionally, visual effects link system behavior with legacy source code modules. The visualization and animation present a meaningful method to investigate the interactions among architecture components. The method further generates the visual diagram to reflect the modular structure of the observed system functionality.

The system decomposition tasks were performed based on the recovered source code segments that had participated in the system functionality of interest. Visualization was implemented at several levels based on different granularity of abstraction.

2.1 Architecture Decomposition Partition

System utility is performed by those detailed atomic system functionalities, which embody the service of the system [10][11][12]. As a consequence of this limitation, static analysis does not present sufficient information to study the interactions of source modules [26][27][34]. Recording dynamic information of a program can provide us with sufficient knowledge about message exchanges and modular interactions during the program execution period [40][41][19]. However, this technique faces two major issues: (i) the overwhelming volume of tracing data [29][36] and (ii) incomplete coverage of the code [25][13]. In order to have a full coverage of the target code during each experimental session, this approach focuses on a specific set of observable system functionalities and behaviors. Therefore, the dynamic coverage contains only the relevant code

artifacts. In fact, this focus turns out to aid in the resolution of the first issue by reducing the volume of tracing data. Based on this approach, this study has developed a reverse engineering tool called the Dynamic-Analyzer to automate the dynamic capturing and visualization of source code modular interactions and to generate the architectural view of target systems.

One desirable feature of the Dynamic-Analyzer is that the observed system and analysis tool run in parallel. Analyzers are now able to observe visualization patterns of the system behavior and module interaction at the same time. Therefore, specific system behavior can be directly related to the visual effects of module interactions in a real-time manner, reducing the cognitive load required to remember and match these subjects.

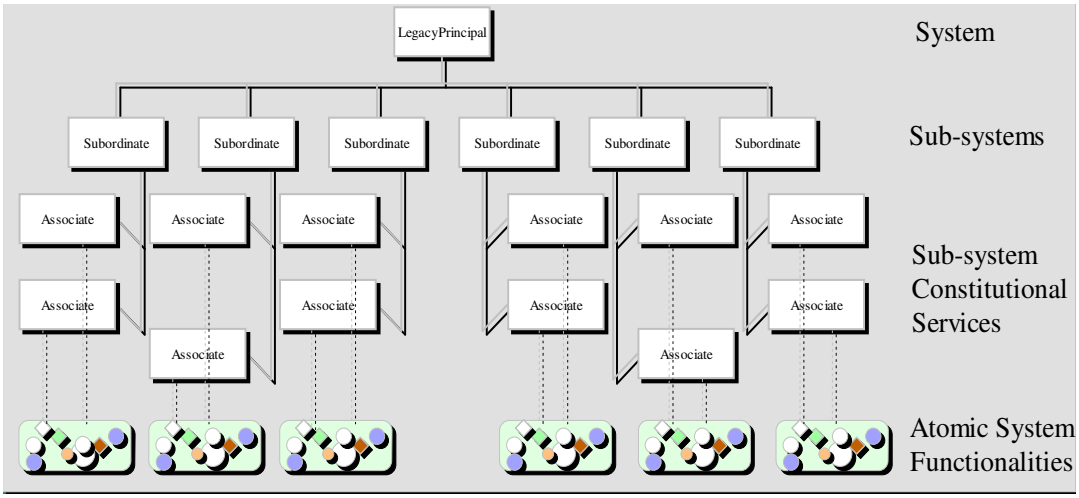


Figure 2: Hierarchical System Function Abstraction

The Dynamic-Analyzer toolset is able to produce various views to exhibit information at distinct granularity levels, and facilitate the smooth navigation among those levels. Two types of information can be visualized: the pure interactions and the statistical data information. The executable system hence can be viewed in a hierarchical manner as follows: whole system, subsystems, subsystem constitutional services, detailed atomic system functionalities for each service, etc. (see Figure 2).

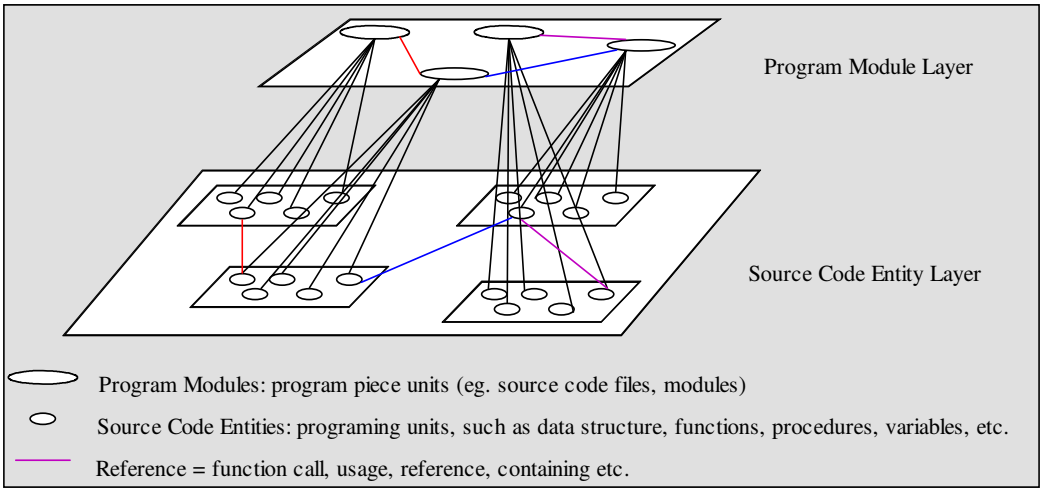


Figure 3: Source Code Hierarchical Abstraction

The physical program source code can also be viewed in a hierarchical way as program module layer, which includes source code files/modules, and source code entity layer which includes data structure components, database components, functional component (variables, functions, procedures, routines, data, etc.,) (see Figure 3). This study attempts to answer the following question: Which program artifacts realize the observed atomic system functionality provided by the system, and how? Mapping atomic system functionalities to the realizing code fragments and recovering the interaction relationships among source code artifacts would divide the whole source code into constructional parts, revealing the structure of legacy software [16] [13]. Based on this approach, the study has developed a dynamic and visualization analysis technique to analyze active system functionality behavior, and build up the linkage between the corresponding code artifacts, their interaction structures, and observed system functionalities. One of the most important features of this technique is the mapping ability for different abstraction layers. The dynamic run-time information can be automatically captured and analyzed though Dynamic-Analyzer, the reverse engineering software toolkit. Dynamic-Analyzer can generate a variety of system architectural analysis views:

Routine Interaction View: This view illustrates the inside of source code module and demonstrates which routines (program functions or procedures) interact with each other to contribute to the performance of a certain kind of atomic system functionality.

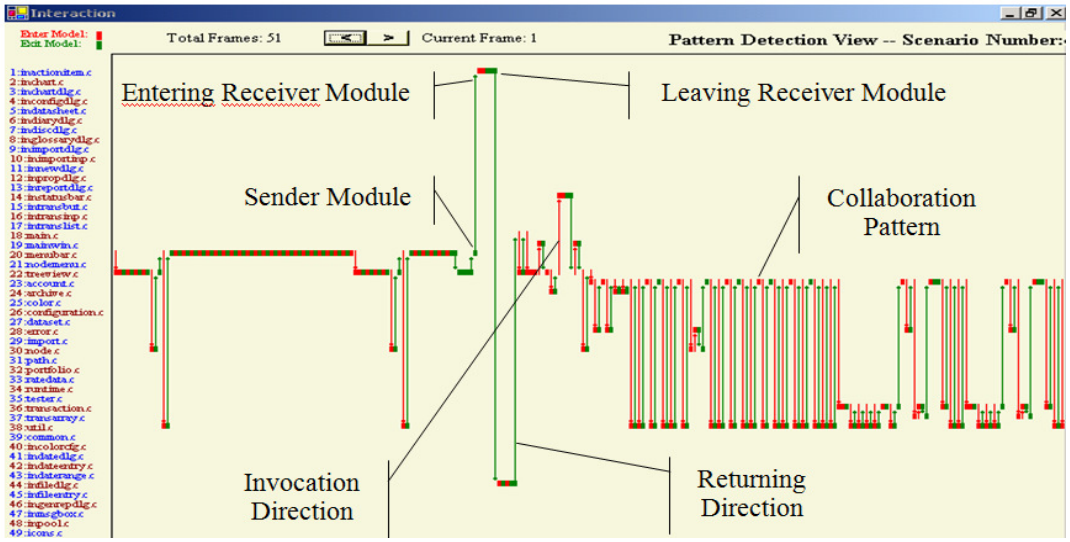


Figure 4: Module Interaction & Collaborative Pattern Detection Generated by Dynamic-Analyzer

Module Interaction View: This view demonstrates which source code modules work together to carry out a certain kind of system functionality (see Figure 4). This information will be used to facilitate the system decomposition process. The left-most vertical part shows the name of modules; the horizontal direction represents the time sequence; the dark (red) box indicates an invocation interaction instance from the sender module; the gray (green) box shows the return of interaction instance from the receiver module; the dark (red) line with direction point shows an outgoing message from the sender module towards the receiver module; the gray (green) line with direction point represents the returning of the interaction message from the receiver module back to the sender module. The Dynamic-Analyzer automatically detects all the repetitive serial of collaboration instances, and distills them as candidate collaboration patterns.

Construction Structure View: To further reveal the construction structure of particular system functionality, more effective means to discover the dynamic module interaction space are needed. The approach is to visualize the dynamic information in a form that illustrates the relationships between different source code modules involved in the activities that generate the specific system functionality.

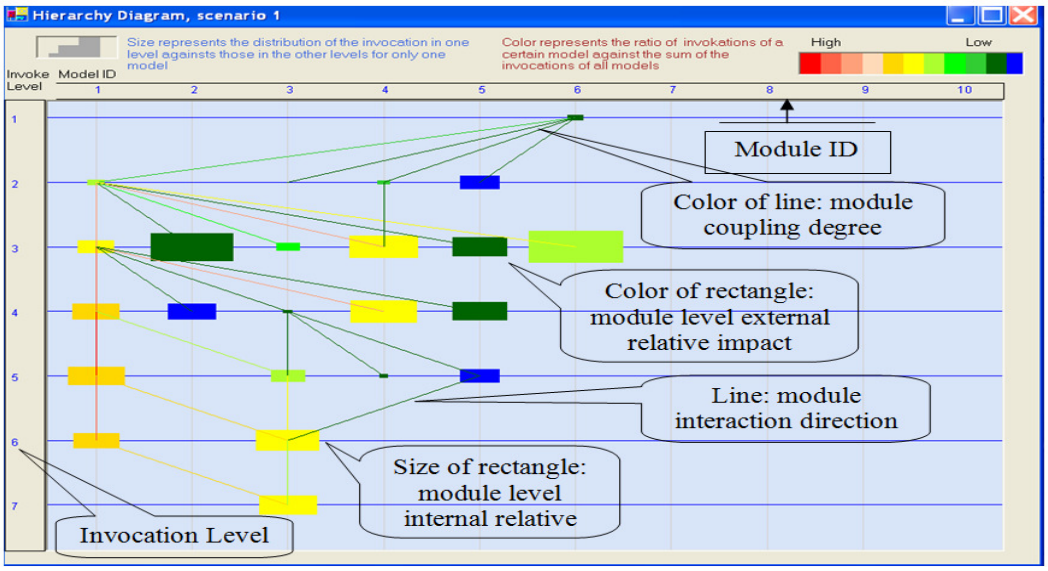


Figure 5 Architectural View of System Functionality Generated by Dynamic-Analyzer

As demonstrated in Figure 4 and Figure 5, the visual representation of the comprehensive module interaction relationships reveals a system constructional structure that implements the observed system functionality.

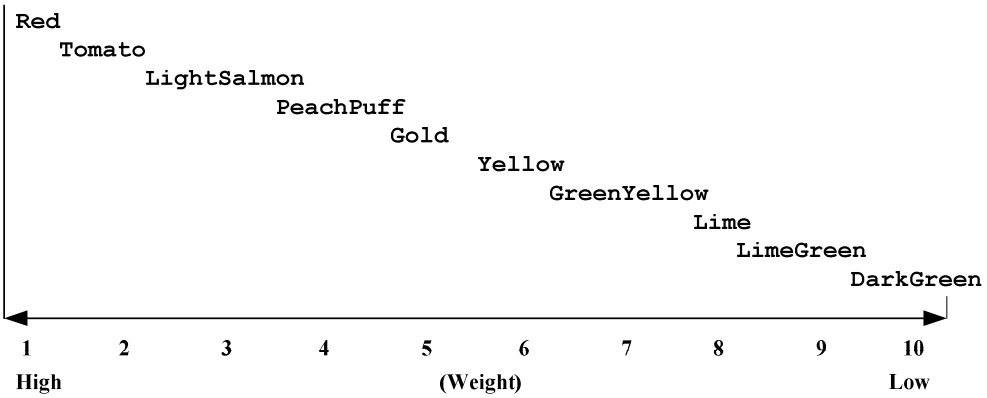


Figure 6: Color Representation Scheme of Weight Variance Gradient

- **The invocation level:** corresponds to the call depth from sender module to receiver module.
- **The link between modules:** represents the invocation instance from higher level module to lower level module.

- **The location of rectangle:** shows at the specific location (invocation level and module) a certain amount of module invocation activities.

- **The size of rectangle:** stands for the percentage of invocations the module has at the particular level, compared with the total number of invocations that module has among all the levels. Suppose the Full_Size rectangle has 1cmX1cm height and width. The mathematic formula is expressed as Figure 6:

The color of both link and rectangle uses "weight" to represent the percentage of invocations it occupies compared with the total number of invocations that all modules have. The mathematical formulas are expressed as following:

Weight_Link (a,b)= 10 * Invocations (Module_a →Module_b) / Invocations (All Modules)
Weight_Rectangle (Module_a, Level_b) = 10 * Invocations (Level_b) / Invocations(All Modules)

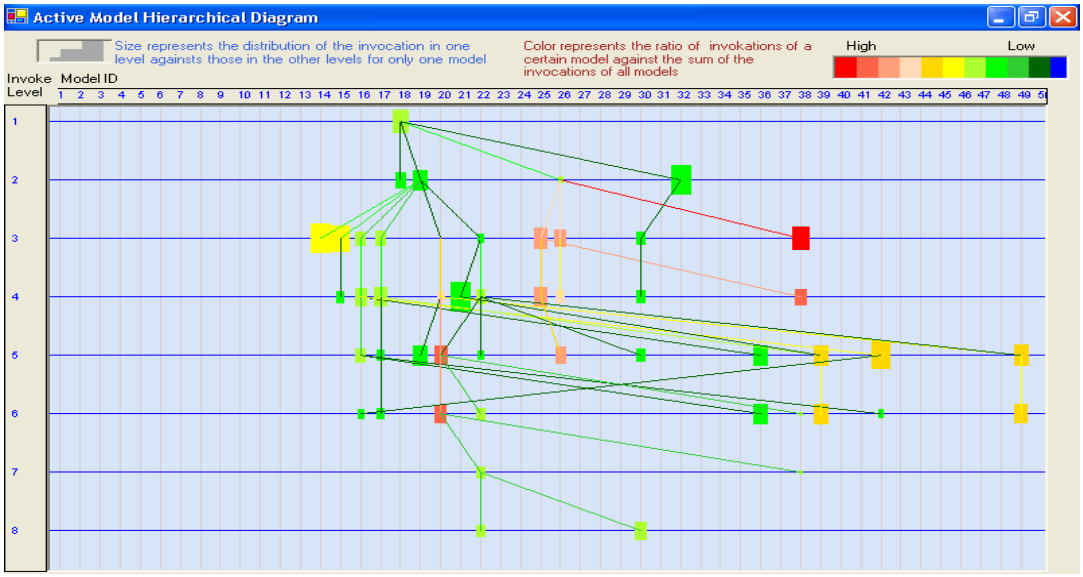


Figure 7: Run-time Partial System Architecture Visualization Generated by Dynamic-Analyzer

- **The color of rectangle:** reveals the module at certain level external relative impact, which specifies the activity intensity degree ("weight") at each invocation level for one particular module in comparison to the total invocations of all modules. Color scale schema are applied to reflect the weight (see Figure 6).

- **The color of link:** illustrates the coupling degree of these two linked modules, which reflects the weight of that link. For the color of link and color of rectangle, 10 color scale schema to symbolize the weight variance gradient from High to Low were applied. Figure 6 illustrates the color representation scheme of the weight variance gradient.

With the help of visual expression provided by the Dynamic-Analyzer, comprehensive system functionality construction structures to reveal partial system architecture are able to be generated.

Figure 7 illustrates the system modular architectural construction structure that implements one sub-system's functionality in a case study. It exhibits the inter-relationships among all the modules that contribute to the implementation of one specific system functionality. The integrated diagram can be further decomposed into a set of visual representations of collaboration patterns (see Figure 4), with assignments of major role for each module.

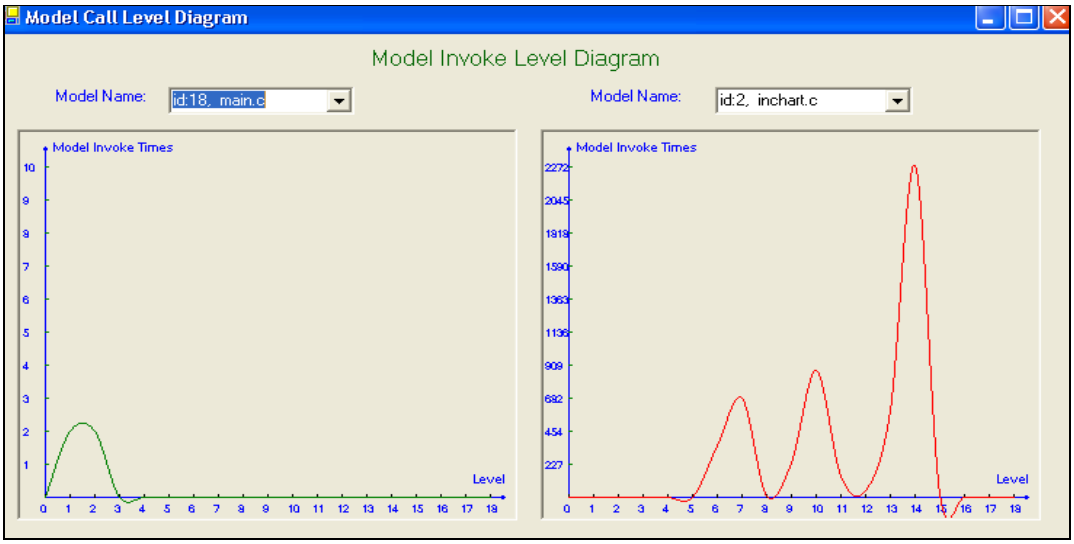


Figure 8: Module Contribution Comparison

Module Contribution Comparison View: This view of two modules provides the visual comparison of any two source code modules automatically generated by Dynamic-Analyzer (see Figure 8). It is used to analyze the detailed efforts of modules at each invocation level. The vertical coordinates represent the number of invocation times, whereas the horizontal direction shows the invocation depths. Cardinal splines are used to diminish the sharp angles of the curve. This method creates a drawing under the horizontal coordinate to reduce the radical changes from high value to zero. Any drawing under the horizontal line is ignored.

Module Participation View: This view is provided by Dynamic-Analyzer to analyze the overall participation percentage of each source code module for the observed system functionality (see Figure 9). The vertical coordinate indicates the name of each source code module, whereas the horizontal axis shows the scale of invocations. The histogram demonstrates the overall efforts of each module that contributes to the implementation of the observed system functionality. The view shows the difference of contribution for each source module during the execution of specific system functionality.

System Functionality Construction View: Normally, a mapping between fine-grained code artifacts and system behavior will cause the maintainer to be lost in the middle of the huge code interaction space, limiting the usability of the technique and reverse engineering tools. By mapping particular system functionalities with different abstract layers of program artifacts, a more efficient high level view of system structure will be revealed. Consequently, the relationships among different system functionalities and corresponding program artifacts at various abstraction levels were discoverable. The result is later used to facilitate the legacy system decomposition task.

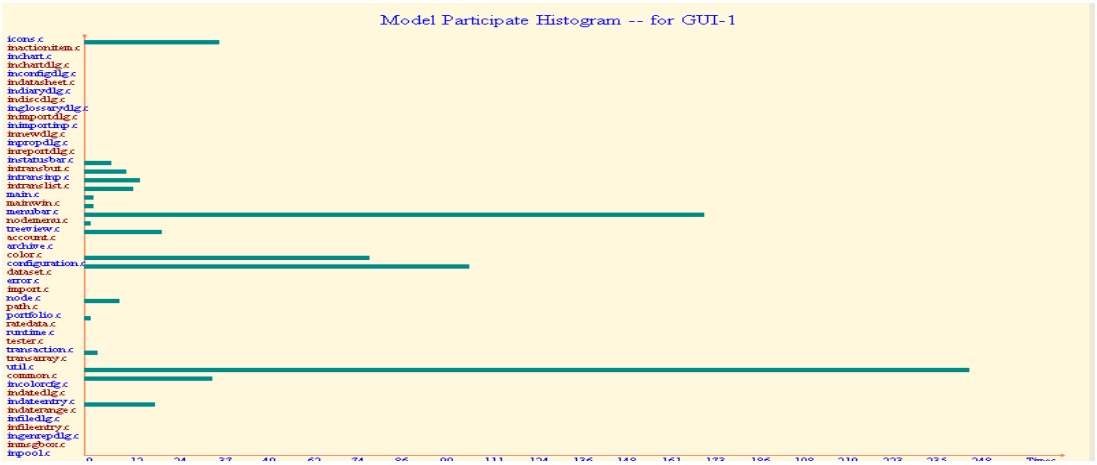


Figure 9: Statistical Module Participation View Provided by Dynamic-Analyzer

Statistic Information Analysis View: For a certain kind of system atomic functionality, different sets of code fragments at different abstract layers were discovered by executing test cases (see Figure 9). Dynamic-Analyzer cane be used to retrieve the following statistical information: which parts of each different abstract layer have persistently participated; which parts are conditionally involved; which parts undertake the heaviest computing part; and which parts mostly contribute to the task dispatching and management job, etc.

2.2. Architecture Decomposition Strategy

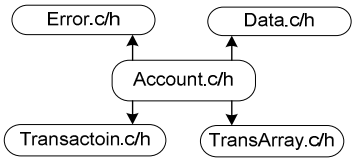
Recovering the architecture of legacy systems requires more than just one reverse engineering toolset to generate descriptive diagrams of systems. Software architecture is commonly defined in terms of components and connectors [14][15], which is lacking in legacy systems based on procedural methodology. Therefore, it is necessary to abstract the high-level architecture by identifying component-like parts and express it in the decomposed way. By dynamically analyzing atomic system functionalities with software visualization, the decomposition strategy separates source code artifacts based on their involvement in a certain type of system functionalities. Code modules that have participated in more than one system functionality are separated to form a new partition serving other functionalities. With the help of the visual dynamic analysis technique and automated reverse engineering toolset, whole program artifacts can be decomposed into cohesive elements revealing the system organization.

3. STATIC MODULE DEPENDENCY ANALYSIS & VISUALIZATION

For most legacy procedural languages, the whole system can be divided into program pieces (such as source code files) [1][22]. Such individual program units are seen as a single source code module. Normally, original developers had a certain kind of principle to organize their program artifacts. The dependency of user-defined data types (UDT) reflects the module relations between each other [18][22][9]. It can be further viewed as an indicator of the associations between the host modules and the rest of the system [17][23][28]. By analyzing the dependency of user defined data type, useful design information of legacy system architecture construction can be elicited. To validate this approach, a case study was undertaken using a code segment from a legacy finance software system. For the source code module, called Account.c/h, there is a User Defined Data Type:

```
typedef struct _ac_t {
node_t        node;             /* has to be the first member (inherit)*/
                                                              ---Error.c/h
char          *symbol;              /* symbol or id */
char          *altsymbol;     /* alternative symbol (for import) */
prec_t        precision;            /* precisions for in/output */        ---Data.c/h
double        fraction;             /* fraction of an account (factor) */
GPtrArray     *transarr;            /* GArray<trans_t> transaction list*/
                                                          ---TransArray.c/h
int    flags;       /* frozen == readonly */
period_t      lifetime;             /* time period: 1st to last transaction*/
                                                           ---Transaction.c/h
GArray        *rtarr;        /* GArray<rt_t>: "prepared" trans data*/
                                                           ---TransArray.c/h
} ac_t;
```

(i)



(ii)

Figure 10. Source Code Module Dependency Analysis

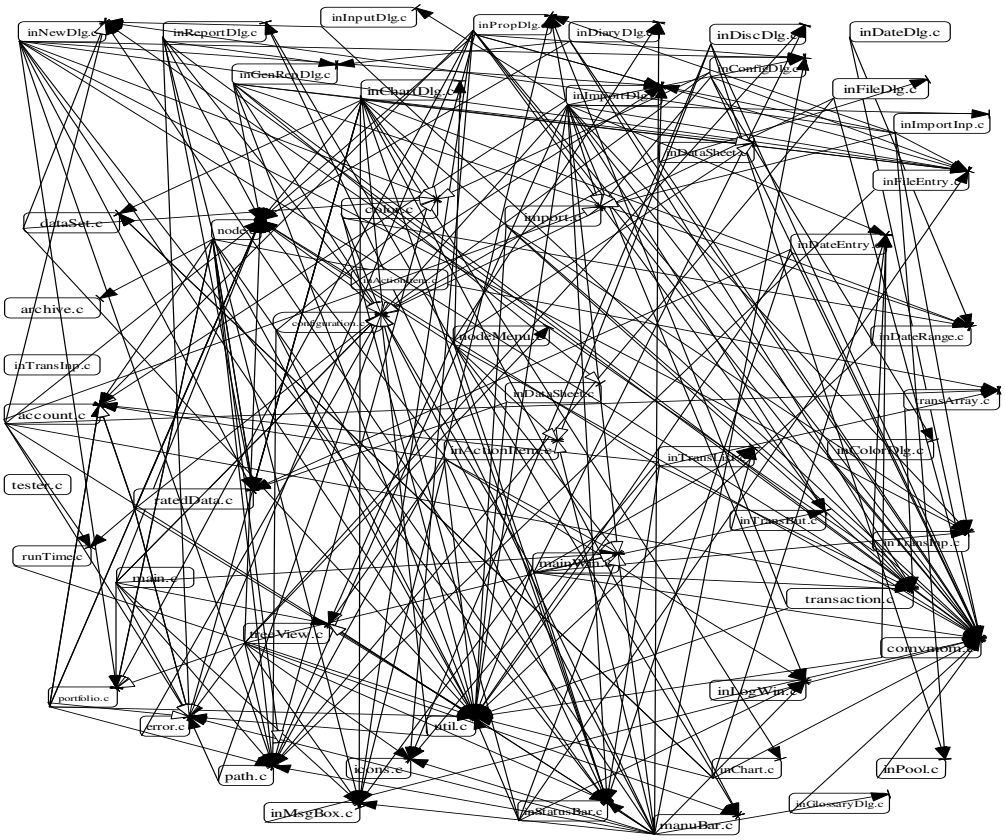


Figure 11. System Module Call Graph with Static Analysis

The user data type dependency (one UDT uses other UDTs) could reflect the reliance relationships between its host module and its dependent modules. To explain this, the study used the example illustrated in Figure 10. The model dependency diagram (ii)) is generated from the source code of a user-defined data type (Figure. 10 (i)). Source code module Account defines one UDT called ac_t, which uses four other UDTs defined in other four modules, namely Error, Data, Transaction, and TransArray. The relationships reflect the dependency between Account module and the other four modules. Those modules involved in this relationship were granted higher coupling value than others which do not have such relations. This process to assign coupling degrees among each module pair can be iterative. Therefore, the module dependency relationship can further be viewed as an indicator for dividing the whole system source code into organically integrated parts.

Fine-grain Detail Code Entity Exploring: By using traditional reverse engineering tools, useful information deep inside of legacy source code can be obtained, thus to ease the system decomposition task. The source code information includes the parsing result of AST (abstract syntax tree); the data-flow diagram; the routine, variable and data structure reference graphs; and more. In the current study, a reverse engineering tool called Source-Navigator [12][13] was used to parse legacy source code and generate various intermediate result information.

The borders of construction parts are the borders of the module dependency relationships. Consequently, according to the module dependency relations, the whole system can be decomposed into parts. The study has devised a second system decomposition approach based on source code module dependency analysis. Figures 11 and 12 illustrate an example of decomposing legacy finance system based on this approach. The first diagram of a call graph (Figure 11) demonstrates the relationships among all the source modules. It is difficult to get a meaningful insight of how system architecture i and how the source modules are organized as components to constitute system functions.

As a result, eventually, this type of call graph information becomes too overwhelming to comprehend: it is hard to distinguish which modules have higher coupling features and which do not, thus making it difficult to decompose the whole system based on coupling and cohesion analysis. The second diagram (Figure 12) is constructed based on module dependency analysis. It further divides the whole system according to module dependency relationships, represented in the diagram by the lines. The result created is more understandable and meaningful for further system decomposition of its architecture with same static information provided by source code.

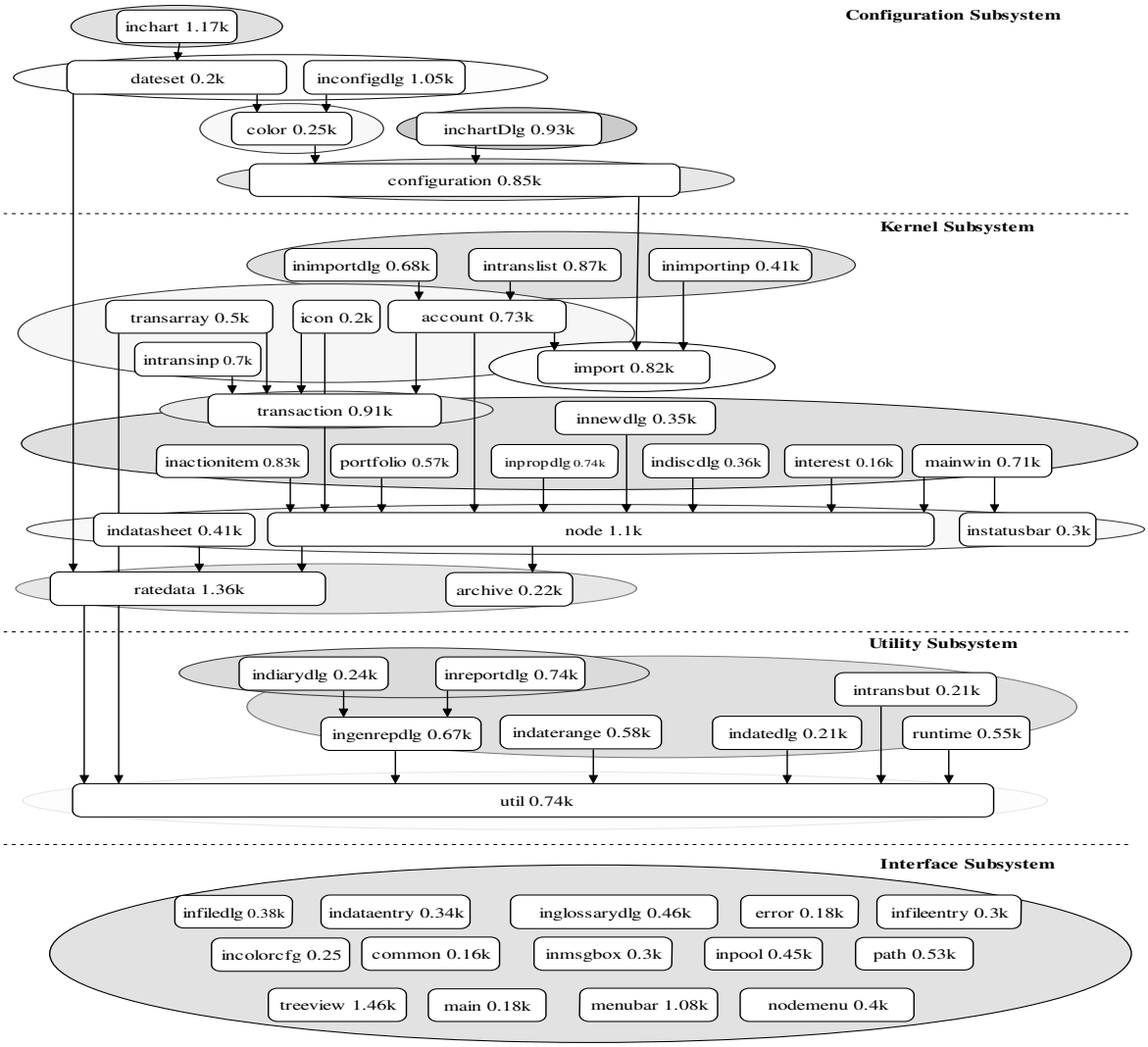


Figure 12: Architecture Decomposition with Module Dependency Analysis

4. DISCUSSION AND RELATED WORK

One of the key research issues of system design analysis and architecture decomposition is to identify program modular features from source code. This identification can be accomplished by viewing existing systems in a component manner [40][17], forming an active research direction of legacy system reverse engineering [43]. In the literature, many modeling techniques are proposed to discover architecture modular structure inside of legacy source code [42][44][38]. Some of these techniques can be partially automated [39], therefore greatly reducing the time to construct a decomposition model from legacy code.

Similarity Clustering: the similarity clustering approach compares pairs of entities by their direct relationships in order to determine whether they belong to the same atomic component [16][10]. This technique groups base entities (subprograms, user-defined types, and global variables) according to the proportion of common features (entities they access, their name, the file where they are defined, etc.).

The intuition is that if these features reflect the correct direct and indirect relationships between these entities, then entities having the most similar relationships should belong to the same atomic component [16][11][6]. The key issue for applying this technique is determining the similarity of the program entities for a certain aspect. Subprograms are clustered into modules based on similarity metrics. Since the two most similar groups are combined per iteration, the order of combinations can be represented by a binary tree, in which the leaves are the initial groups and the inner nodes are combinations of groups. The farther a combination is away from the root of the tree, the higher is its degree of similarity. This procedure is called hierarchical clustering; see the following algorithm:

```
Place each entity in a group by itself;
Repeat
        Identify the most similar groups Si and Sj;
        Combine Si and Sj;
        Add a sub-tree with children Si and Sj to the clustering tree;
Until the existing groups are satisfactory or only one group is left;
```

In each iteration, the most similar groups are combined using a similarity metric. Many aspects can be considered to compare the similarity of two program entities, such as using the same user-defined data type, accessing the same files outside of the program [18][31][35], etc.

Dominance Analysis: Cimitile et al. proposed a dominance analysis to call graphs to identify candidates for system architecture modules [6][8]. A node N is said to dominate another node M in a directed graph G if each path from the root of G to M contains N. If N is a dominator of M and every other dominator N of M is also a dominator of N, then N is called an immediate or direct dominator of M. The dominance relationship can be represented as a dominance tree where a node's parent is its immediate dominator.

In their approach, cycles (i.e., strongly connected components) are collapsed before applying dominance analysis. It is used to detect additional entities. The algorithm involves the following basic steps:

1. All members of an atomic component are collapsed to a single node (this step is denoted by Collapse);
2. Dominance analysis is applied to the collapsed graph;
3. In the dominance tree, each component C absorbs its (transitively) dominated subprograms that are not dominated by any other component dominated by C.

Concept Analysis: The use of concept analysis has been applied as an automated technique for analyzing the modular structure of legacy software [39][15]. Concept analysis is a mathematical technique that provides a way to identify groupings of items that have common features. The main application is to derive the architecture structure of legacy software. It starts with a context: a binary table (relations) indicating the features of a given set of items. From that table, the analysis builds up so-called concepts which are maximal sets of items sharing certain features. The relations between all possible concepts in a binary relation can be given using a concise lattice representation. The component identification is done by deriving a concept lattice from the code based on data usages in source code procedures. The structure of this lattice reveals the modularization that is implicated in the code.

Concept analysis has a sound mathematical background and the insights into the relationships among system components. It is an interesting technique for atomic component detection. On the

other hand, it is also a time-consuming process when applying concept analysis to larger systems, creating a major barrier to applying concept analysis technique in object modeling tasks.

5. CONCLUSION AND FUTURE WORK

Software reengineering involves the activities of studying a target system's architecture [4][7][21]. However, enterprise legacy software systems tend to be large and complex. The analysis of system architecture therefore becomes a difficult task [10][15][40]. To solve this problem, the current study proposes an approach that decomposes software architecture to reduce the complexity associated with analyzing large scale architecture artifacts.

The study has demonstrated that architecture decomposition is an efficient way to limit the complexity and risk associated with the re-engineering activities of a large legacy system. It divides the system into a collection of meaningful modular parts with low coupling, high cohesion and a minimal interface, thus facilitating the incremental approach to implement the progressive software re-engineering process [7][5][9][12]. The system architecture decomposition focuses on how to decompose legacy system into parts, thus facilitating the next stage of applying a divide-and-conquer approach to implement the legacy system re-architecting and incremental re-engineering tasks [15][30][33]. The decomposition strategies are constructed based on different emphases of system analysis aspects. This paper presented two techniques developed to conduct legacy system architecture decomposition work: visualization & dynamic analysis of system architecture and module dependency analysis respectively by utilizing run-time dynamic and static information. These two techniques are constructed based on different emphasis on system analysis aspects. Our approach is supported by our automated reverse engineering tools called Dynamic-Analyzer. The preliminary experiment demonstrates that this approach yields promising results.

REFERENCES

[1] F. Balmas and Harald Wertz, "Identifying information needs for program understanding: An Iterative Approach," Proceedings of 7th International Conference on Reverse Engineering for Information Systems, Lyon (France) July 2001.

[2] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice (2nd edition), Addison-Wesley 2003.

[3] J. Bergey, D. Smith, N. Weiderman, and S. Wood. "Options analysis for reengineering (OAR): Issues and conceptual approach," Technical Note, Carnegie Mellon Software Engineering Institute CMU/SEI, TN, 2000.

[4] A.Bianchi, D.Caivano, G.Visaggio, "Method and process for iterative reengineering data in legacy system," Proc. IEEE Working Conference on Reverse Engineering, November 2000

[5] Liam O'Brien Christoph Stoermer, Chris Verhoef, "Software architecture reconstruction: practice needs and current approaches," Carnegie Mellon Software Engineering Institute, Technical Report, CMU/SEI-TR-024, 2002

[6] G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, "decomposing legacy systems into objects: An eclectic approach," Information and Software Technology, vol. 43, no. 6, 2001, pp. 401-412.

[7] A. Eden, R. Kazman, "Architecture, design, and implementation," Proceedings of the 25th International Conference on Software Engineering (ICSE 25), (Portland, OR), pp. 149-159, May 2003.

[8] J. Eloff, "Software restructuring: implementing a code abstraction transformation," ACM International Conference Proceedings of SAICSIT 2002,

[9] D. Jackson and M. Rinard. "Software analysis: A roadmap," in The Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press 2000.

[10] R. Koschke, "Atomic architectural component recovery for program understanding and evolution," In Proceedings of the International Conference on Software Maintenance, Montréal, Canada, October 2002.

[11] T. Kuipers and L. Moonen. “Types and concept analysis for legacy systems.” In Proceedings of the International Workshop on Programming Comprehension (IWPC 2000). IEEE Computer Society, June 2000.
[12] A. De Lucia, G.A. Di Lucca, G. Canfora, A. Cimitile, “Decomposing legacy programs: A first step towards migrating to client-server platforms,” The Journal of Systems and Software, vol. 54, 2000, pp. 99-110.
[13] M. Ludger, A. Giesl, J. Martin. “Dynamic component program visualisation.” In Proceedings of the 9th Working Conference for Reverse Engineering, Richmond, Virginia, October 2002.
[14] I. Michiels, D. Deridder, H. Tromp and A. Zaidman, “Identifying problems in legacy software,” Elisa ICSM workshop, 2003
[15] D. Notkin, “Longitudinal program analysis,” ACM SIGSOFT Software Engineering Notes , Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools And Engineering, Volume 28 Issue 1, November 2002.
[16] K. Rainer, “Atomic architectural component recovery for program understanding and evolution,” Ph.D Thesis, Institut für Informatik, Universität Stuttgart 2000
[17] C. Stoermer, L. O'Brien, C. Verhoef, “moving towards quality attribute driven software architecture reconstruction,” Working Conference on Reverse Engineering, Victoria, BC, Canada, November 13th - 16th, 2003
[18] A. van Deursen and L. Moonen. “Exploring legacy systems using types”. In Proceedings 7th Working Conference on Reverse Engineering, (WCRE'2000), pages 32-41. IEEE Computer Society, 2000.
[19] M. Fowler, Refactoring - Improving the Design of Existing Code. Addison-Wesley Professional, 2000.
[20] E. J. Chikofsky and J. H. Cross II, “Reverse engineering and design recovery: A taxonomy,” IEEE Software, vol. 7, no. 1, pp. 13-17, 1990.
[21] T. M. Pigoski and A. April, Software engineering body of knowledge. IEEE Computer Society, 2004, pp. 6.1-6.16.
[22] M. Cohn, Succeeding with agile: Software development using scrum, K. Gettman, Ed. Addison-Wesley, 2009.
[23] E. Murphy-Hill and A. P. Black, “Refactoring tools: Fitness for purpose,” IEEE Software, vol. 25, Sep. 2008, pp. 38-44.
[24] G. C. Murphy, M. Kersten, and L. Findlater, “How are java software developers using the eclipse ide?” IEEE Software, vol. 23, Jul. 2006, pp. 76-83.
[25] P. Weissgerber and S. Diehl, “Are refactorings less error-prone than other changes?” in Proceedings of the International Workshop on Mining Software Repositories, 2006, pp. 112-118.
[26] T. Eisenbarth, R. Koschke, and D. Simon, “Locating features in source code,” IEEE Transactions on Software Engineering, vol. 29, no. 3, 2003, pp.210-224.
[27] D. L. Parnas, “Software aging,” in International Conference on Software Engineering, 1994, pp. 279-287.
[28] A. Telea and L. Voinea, “Case study: Visual analytics in software product assessments,” in Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis, 2009, pp. 65-72.
[29] F. Bourquin and R. K. Keller, “High-impact refactoring based on architecture violations,” in Proceedings of the European Conference on Software Maintenance and Reengineering, 2007, pp. 149-158.
[30] S. Demeyer, S. Ducasse, and O. Nierstrasz, Object oriented reengineering patterns. Morgan Kaufmann Publishers Inc., 2002.
[31] F. Simon, F. Steinbruckner, and C. Lewerentz, “Metrics based refactoring,” in Proceedings of the European Conference on Software Maintenance and Reengineering, 2001, pp. 30-38.
[32] S. Roock and M. Lippert, Refactoring in large software projects: Performing complex restructurings successfully. John Wiley & Sons, Inc., 2006.
[33] M. Lanza and S. Ducasse, “Polymetric views—a lightweight visual approach to reverse engineering,” IEEE Transactions on Software Engineering, vol. 29, no. 9, 2003, pp. 782-795.
[34] R. Wettel and M. Lanza, “Visually localizing design problems with disharmony maps,” in Proceedings of the Symposium on Software Visualization, 2008, pp. 155-164.
[35] D. Holten, “Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data,” IEEE Transactions on Visualization and Computer Graphics, vol. 12, 2006, pp. 741-748.
[36] J. Bohnet and J. Dollner, “Monitoring code quality and development activity by software maps,” in Proceedings of the International Workshop on Managing Technical Debt, 2011.

[37] C. Lewerentz, F. Simon, and F. Steinbruckner, "Crococosmos," in Graph Drawing, ser. Lecture Notes in Computer Science. Springer, 2002, vol.2265, pp. 72-76.
[38] D. Ratiu, S. Ducasse, T. Gˆırba, and R. Marinescu, "Using history information to improve design flaws detection," in Proceedings of the Euromicro Working Conference on Software Maintenance and Reengineering, 2004, pp. 223-232.
[39] R. W. Schwanke and S. J. Hanson, "Using neural networks to modularize software," Machine Learning, vol. 15, pp. 137-168, May 1994.
[40] R. N. M. Watson, J Woodruff, P. G. Neumann, et al. "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," 2015 IEEE Symposium on Security and Privacy. IEEE, 2015, pp 20-37.
[41] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in Proceedings of the International Symposium on Software Visualization, 2010, pp. 5-14.
[42] R. Chitchyan, A. Rashid. P. Sawyer, A. Garcia, MP. Alarcon, J. Bakker and A. Jackson. Survey of aspect-oriented analysis and design approaches. 2015.
[43] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," Journal of Systems and Software, vol. 83, pp. 391-404, Mar. 2010.
[44] S. Hayashi, M. Saeki, and M. Kurihara, "Supporting refactoring activities using histories of program modification," IEICE Transactions on Information and Systems, vol. E89-D, Apr. 2006, pp. 1403-1412.
[45] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice. Addison-Wesley, 2007.