The International Journal Of Engineering And Science (IJES)
|| Volume || 6 || Issue || 1 || Pages || PP 87-98|| 2017 ||
ISSN (e): 2319 – 1813 ISSN (p): 2319 – 1805

Software Architecture Design Recovery through Run-Time Source Code Collaboration Pattern Analysis

Lei Wu[1], Sankalp Vinayak[2], Hua Yan[3]
[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*
[3]*Institutional Research, University of Houston-Clear Lake, Houston, Texas U.S.A*

ABSTRACT
Interoperability between enterprise legacy systems and contemporary information systems requires more efficient ways to analyze the architecture and design of legacy systems. However, the original source code collaboration design information is dispersed at the implementation level. The analysis of system design architecture therefore becomes a difficult task. In this paper, the authors present a novel approach to efficiently recover and analyze legacy system architecture and design through run-time source code collaboration pattern and role analysis. The extraction of code artifact collaborations and their roles is therefore an important support aspect in legacy software comprehension and architecture design recovery. The authors' approach consists of two major parts, both of which are supported by their reverse engineering tools. The first part focuses on the dynamic analysis of target legacy systems and the automatic discovery of the legacy system's architecture. The second part is concentrated on the recovery and study of legacy system design through collaboration pattern and role analysis. The study demonstrates that this novel approach is promising.

Keywords: Design analysis, software architecture, dynamic analysis, system decomposition, collaboration pattern, role, design recovery, software visualization, reverse engineering

Date of Submission: 31 December 2016 Date of Accepted: 05 February 2017

I. INTRODUCTION
When facing a legacy system integration project, interoperability among enterprise legacy systems and contemporary information systems requires a more efficient method to analyze the legacy system's architecture and design [Bas03][Bri02]. Software engineers inevitably encounter the difficult task of understanding legacy software system's architecture and design [Ede03][Elo02][Jac00]. Hall's research work shows that understanding the documentation and logic of programs occupies about 50 to 60 percent of a maintenance programmer's time [Hal88]. In many cases, even the original developers find it difficult to comprehend their own code after a long period of time [Som00]. As a consequence, integration tasks tend to be difficult, expensive, and error prone [Mic03][Gan00][Gan01].

This research focuses on following two aspects of legacy system architecture and design analysis: (i) Legacy system source code collaboration pattern and role recovery and (ii) Legacy system decomposition. The code collaboration pattern and role recovery concentrate on the identification of legacy system architecture features and constructional structures. This work informs how legacy functionalities are implemented through modular units' collaboration, and the patterns/roles of various types of code cooperation forms. The system decomposition makes it possible to further apply a divide-and-conquer approach to implement enterprise legacy system interoperability re-architecturing.

II. SYSTEM ARCHITECTURE & DESIGN RECOVERY WITH COLLABORATION PATTERN ANALYSIS
Software functionality and behavior are accomplished by the cooperation of code artifacts. Consideration of this type of modular unit collaboration provides an important aid to system interoperability and legacy software evolution. However, the original collaboration design information is dispersed at the implementation level. The extraction of code collaborations and their roles is therefore an important method in legacy software design recovery. In this paper, the novel approach to recover and study legacy system architecture and design with code collaborations and roles analysis are presented.

2.1 APPROACH INTRODUCTION

Large enterprise legacy systems are normally organized in a structured form [Let99][Lak97], with code divided into separated source files based on different design criteria [Pau94][Let99]. For example, in COBOL, FORTRAN, C and Ada, the functions that relate to the same topic (such as "error") are usually grouped together into a single program file. Source files are further structured into different directories according to the functionalities to which they contribute [Let99]. This kind of program code organization reflects the original legacy design rational [Pau97]. Each source file and directory represents a certain design concept [Let99]. Each code cooperation instance contains a limited number of such code units. These construction units are viewed as source code modules, which interact with each other to realize the system functional behavior [Let99]. Moreover, each module plays a set of conceptual roles inside of the cooperation. The role relationship among code modules reflects the control characters of source code, and the code collaboration pattern reveals code organizational structure.

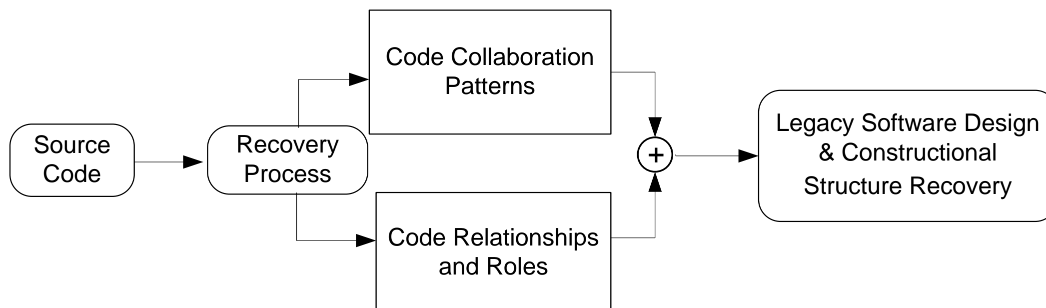


Figure 1: Legacy System Architecture & Design Recovery

Recovering such code collaboration patterns and conceptual roles from code artifacts is therefore an important aspect for better understanding and re-engineering legacy code [Ric02]. It further facilitates the recovery of legacy software design and constructional architecture, as illustrated in Figure 1. However, the large number of code modules and the complexity of dynamic relationships make discovering and analyzing module collaboration patterns and code roles a difficult task.

2.2 GENERAL CONCEPTS

This section introduces the underlying analysis concepts, terminologies, and formalisms used in the research work of legacy system analysis.

Source code module: the source code of a system is usually organized in a structured form [Let99][Lak97]. Code units related to the same concept or topic lay in a single source file and are further stored into different directories, which reflect the original design rational [Let99]. The source file or directory were viewed as source module, or simply module.

Interaction instance: an interaction instance is a dynamic information transaction between two modules. It triggers a message flow from sender module to receiver module.

Collaboration instance: a collaboration instance is the sequence of contiguous interaction instances, which together form a chain of events.

Collaboration (or cooperation) pattern: a collaboration pattern is a frequently repeated serial of several collaboration instances. During the whole process of interactions, modules show strong cooperative forms: certain modules always cooperate together to implement a particular type of task. This kind of phenomenon were viewed as a collaboration pattern. (See the detailed analysis terminology in Figure 2.)

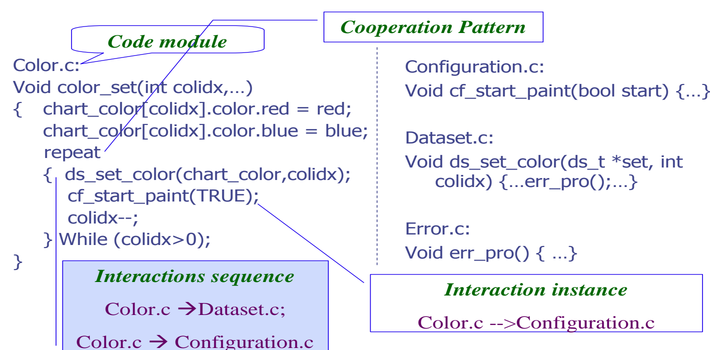


Figure 2: Analysis Terminology Illustration

Dynamic trace record: the dynamic trace record is the program trace information captured during the execution of target legacy system. A sample segment of the trace is given in Table 1.

Fun_id,	Level,	Module,	Routine,	Direction
3,	5,	indateentry.c,	in_dateentry_set_text,	In
3,	6,	intransinp.c,	on_input_data_changed,	In
3,	7,	transaction.c,	*trans_get_typelist,	In
3,	7,	transaction.c,	*trans_get_typelist,	Out
3,	6,	intransinp.c,	on_input_data_changed,	Out
3,	5,	indateentry.c,	in_dateentry_set_text,	Out

Table 1: Dynamic Tracing Data Format

An interaction instance includes six major components, namely system functionality ID, invocation (call) level, sender module, receiver module, invocated routine, and direction. In Table 1, the module represents the receiver module at that call level. The sender module is the module that locates at the nearest former interaction instance which has one call level less than the call level of the current interaction instance. The Fun_id stands for system functionality identification number, which correspondent to the specific system functionality that was performed. The level represents the invocation depth. Direction is the orientation of message flow. The module and routine represent the module name and the function/procedure that carries out the invocation event.
Conceptual role: a conceptual role is the predictable stereotype of an individual module. It represents the general characteristic of the module's utility in program.
Role definition: the definition of a specific role that a code module plays in a collaboration pattern reflects the relationship between two modules. A particular module may have multiple roles in different relationships with other modules, but normally it has a major dominant role. From the construction point of view, a simple job-dispatch relationship can be metaphorized into "manager-worker" roles for the sender and receiver modules. This role-pair relation explains that the module with high level "manager" role dispatches tasks to the modules with lower level "worker" role. Four pairs of conceptual role relationships can be defined based on the invocation contributions among the relationship.

III. COLLABORATION PATTERN AND ROLE RECOVERY APPROACH

This section presents the program analysis approach for the recovery of code collaboration patterns and conceptual roles from source code artifacts. The study applies dynamic program analysis and software visualization techniques to accomplish the goal. Two reverse engineering program analysis tools, namely Dynamic-Analyzer and Collaboration-Investigator, have been designed to carry out this approach. The tools are used to automate the process of detecting, recovering, and analyzing legacy source code collaboration patterns and roles.
Collaboration and conceptual roles are two design concepts that have been scattered throughout source code [Ric02]. Inside collaborations, participant modules interact with each other to carry out specific tasks. The cooperation is confined in an interaction structure form, which describes a set of allowed collaboration behaviors for each module. Such structure is implemented with two major design concepts and dispersed in code: the repetitive code cooperation pattern and conceptual role. Each participant plays a certain type of role in the collaborations. Recovering such information can largely facilitate the program comprehension process and promote program analysis into a deeper level. Since procedural languages do not provide explicit means to capture such design information, maintainers have to rely heavily on human efforts to investigate these design logics in legacy software.
To recover collaborations and roles from legacy source code, the study proposes a new approach that uses dynamic analysis, software visualization, and automatic/semi-automatic detection techniques to achieve the goal, as illustrated in Figure 3. The study first executes system functionalities separately and captures dynamic interaction information among modules during system execution period.

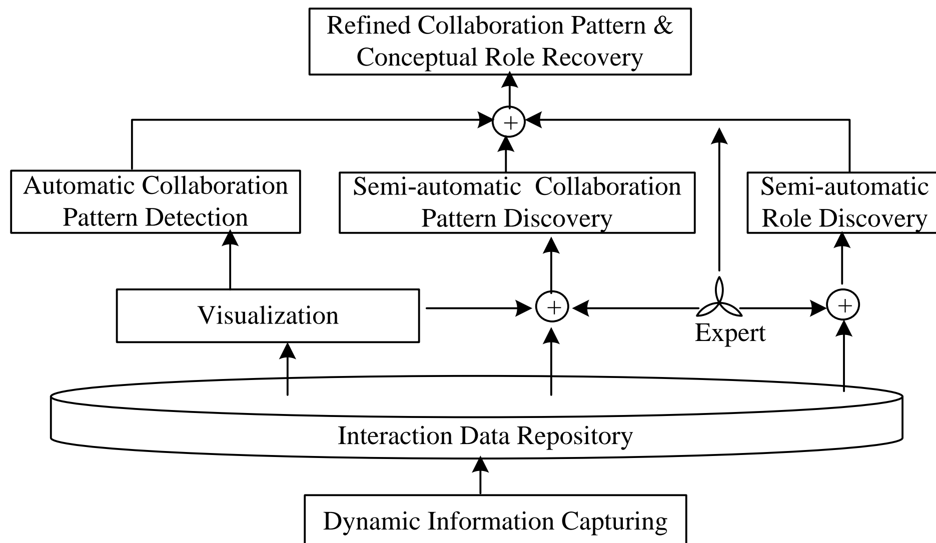


Figure 3: Recovery Approach Schema

It then applies software visualization technique to analyze and identify dynamic program features. Later on, automatic pattern detection process is performed to recover all significant repetitive collaboration instances. Meanwhile, with the intervention of maintainers, the semi-automatic process detects the collaboration pattern and participants' roles, and investigates their features. In addition, a crosscheck and refinement process is conducted to combine the two outcomes, and distill the final refined results. The following are the key issues addressed in the approach.

Dynamic information capturing. The study applies dynamic analysis technique [Bal99] to capture module interaction messages, data transformation routes and control flow information during program execution.

Software visualization. The study applies graphic simulation to represent the captured information into a more understandable visual form as a set of comprehensive graphical diagram views. Two kinds of information are visualized: first is the pure interaction information that represents what is going on inside the code; the second is the statistical data information. For the first, both static visualization and animation were used to simulate the dynamic nature of code artifacts cooperation. For the latter, graphical diagrams and graphs were used to visualize the statistical analysis results.

Automatic and semi-automatic collaboration pattern and role detection mechanism. With the results from former two processes, the features of dynamic code interaction instances—such as the components of the code cooperation, their directions, the serials of code collaboration sequence and their frequencies—can be studied. To discover the collaborations and conceptual roles dispersed over the huge amount of code transactions, it is crucial to limit the searching space. The difficulty lies in the efficient identification of those significant repetitive interactions, which jointly form a meaningful collaboration pattern and role relationships in the large transaction space. By applying automated detection technique on the visualization results, the study was able to extract fine-grained collaboration patterns. The advantage of such an approach is that it is capable to detect a wide range of collaboration patterns, while the disadvantage is that a maintainer may lose the control of expressing the emphasis on discovering patterns. As a remedy to this shortage, the study adopted a semi-automated recovery of collaboration patterns and roles with human intervention.
Based on an investigator's emphasis, the recovery criteria can be interactively selected, which gives the preferable weight on different aspects of automated pattern recovery. The collaboration pattern and role relationship discovery results reflect the emphasized interests of maintainer. As a result, the system can only recover those collaboration patterns and roles, which exhibit the most interesting features defined by human. Finally, these two types of outcomes are compared and refined to produce the final recovery result.
In order to efficiently recover collaboration patterns from execution trace information, certain types of criteria have to be designed to emphasize aspects that are more important in detecting sequences of collaboration where instances are related, thereby creating opportunities to further combine instances together to form a concrete collaboration pattern. The criteria are divided into the following three categories:

Interaction instance component. An interaction instance includes six major components, namely system functionality id, invocation level, sender module, receiver module, invocated routine, and direction. Based on different emphasis, a maintainer may use any combination of these components to define the recovery criteria.

Collaboration instance selection. The main purpose of finding collaboration instance is to recover the invocation path. For this reason, only the interaction instances involving specific modules can be selected to view. Meanwhile, to limit the observation scope, a consideration boundary that confines the recovery process to a certain depth range may also be defined.

Collaboration pattern matching. When several frequently repeated collaboration instances form one collaboration pattern, the shape of that pattern may not be unique. Different interaction sequence may lead to various visual outlines, while the semantics of these patterns are identical. Therefore, the study defines the criteria to let the tools compare two collaboration patterns.

IV. AUTOMATION WITH DYNAMIC-ANALYZER

As discussed in the introduction, static analysis does not present sufficient information to study the interactions of source modules. Recording dynamic information of a program can provide one with sufficient knowledge about message exchanges during program execution period. However, this technique faces two major issues: the overwhelming volume of tracing data and incomplete coverage of the code. In the current approach, since the focus is on a limited set of system functionalities and behaviors, the dynamic coverage contains only the relevant code artifacts. In fact, this focus is beneficial because it reduces the volume of tracing data. Based on the previously presented, the Dynamic-Analyzer, a reverse engineering tool has been developed to automate the dynamic capturing and visualization of dynamic source code message process information among source modules (See Figure 4). First, legacy source code is instrumented to record execution information. Then, the target system functionalities are executed to observe system behaviors; meanwhile, program dynamic information is retrieved, processed (normalized) and fed into data repository.

Subsequently, the visualization and animation program would present the information through visual effects to create a meaningful way to investigate the interactions. Finally, the automatic collaboration pattern detection process will be performed to distil all the patterns. The Dynamic-Analyzer tool can automatically discover all fine-grained collaboration patterns. Another reverse engineering tool has been developed to incorporate human intervention in the discovery process and combine those with the outcomes from Dynamic-Analyzer to improve the result. This semi-automatic approach is detailed in following section. One advantageous feature of the Dynamic-Analyzer is that both the observed system and analysis tool run in parallel. Maintainers are now able to observe system behavior and the visualization of source module interactions/patterns simultaneously. Thus, any specific system behavior can be directly related to the visual effects of module interactions in a real-time manner, reducing the cognitive load to remember and match these two subjects.

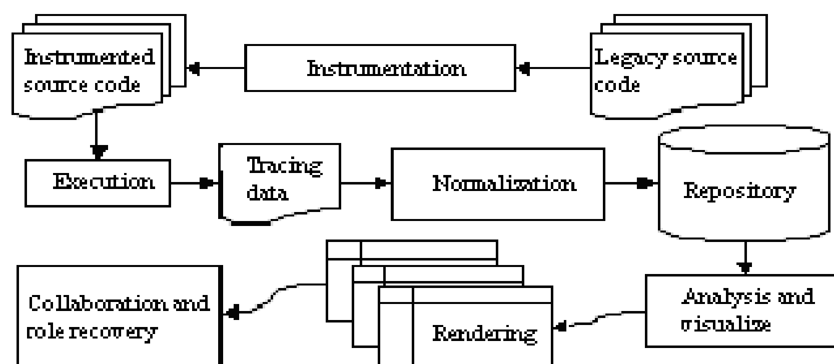
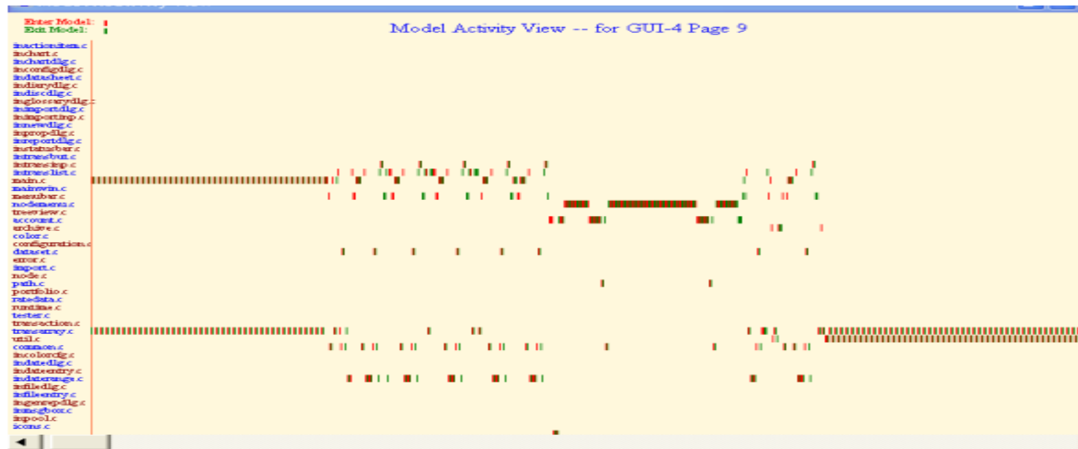


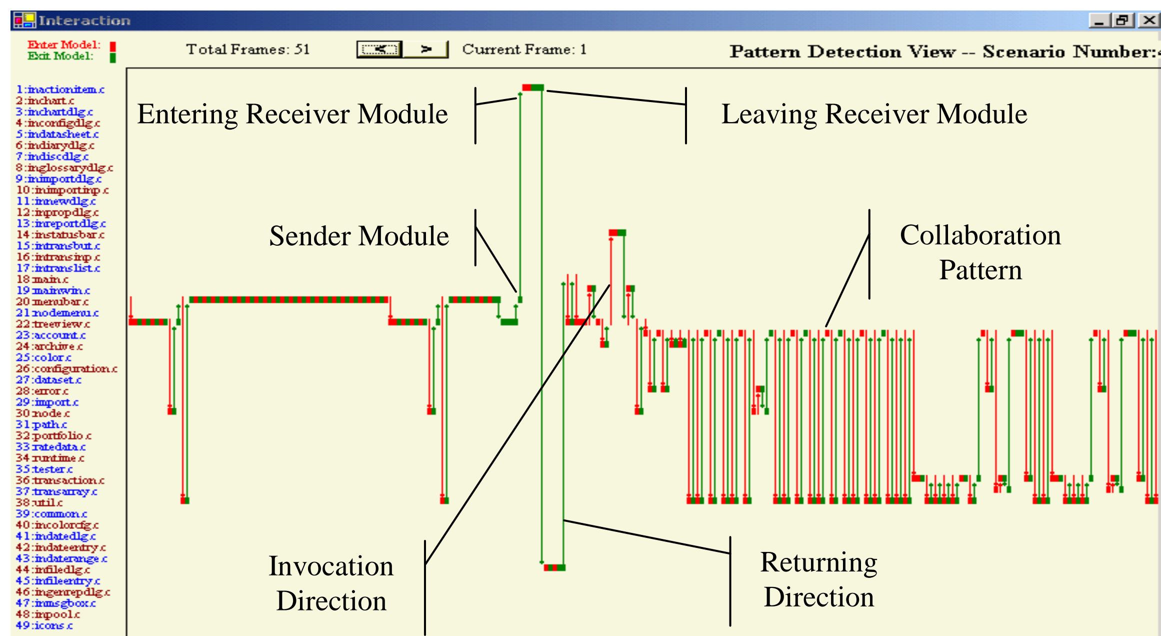
Figure 4: Workflow of Dynamic-Analyzer

The study used the Dynamic-Analyzer tool to define different types of views, to exhibit information at different granularity levels, and to facilitate the smooth navigation among those levels. Two types of information were visualized: the pure interactions and the statistic data information. For the first, static and animate visual effects were applied to enhance the recovery process. Figure 5 illustrates the fine-grained dynamic module interaction footprint view and animated pattern detection views produced by Dynamic-Analyzer.



(i) Dynamic Module Interaction Visualization: Footprint View

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 sender module towards receiver module; the gray (green) line with direction point represents the returning of the interaction message from receiver module back to sender module.
The Dynamic-Analyzer automatically detected all the repetitive collaboration instances, and distilled them as candidate collaboration patterns. To further reveal the construction structure of particular system functionality, a more effective means to discover the dynamic module interaction space is needed.
The study proposes a new approach to visualize the dynamic information in a form that illustrates the relationships between difference source code modules which are involved in the activities that generate the specific system functionality.



(ii) Automated Collaboration Pattern Detection

Figure 5: Footprint (i) and pattern detection (ii) views from Dynamic-Analyzer

As demonstrated in Figure 6, the visual representation of the comprehensive module interaction relationships reveals system constructional structure that implements observed system functionality.
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 & module), there's 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 the module has among all the levels. Suppose the "Full_Size" square has 1cmX1cm height and width. The mathematic formula is expressed as following:

Size(Module_a,Level_b)=FullSize*Invocations(Module_a, Level_b)/Invocations(Module_a, All Levels)

Here, the Full_Size represents the maximum square space between two nodes. The size of each rectangle shows the relative impact of invocations that a module has at certain level, which indicates the activity intensity degree at each invocation depth for one particular module.

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 total invocations of all modules. Color scale schema is applied to reflect the weight. The notation of color is shown in Figure 7.
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, the study applies 10 color scale schema to symbolize the weight variance gradient from "High" to "Low." Figure 7 illustrates the color representation scheme of the weight variance gradient.

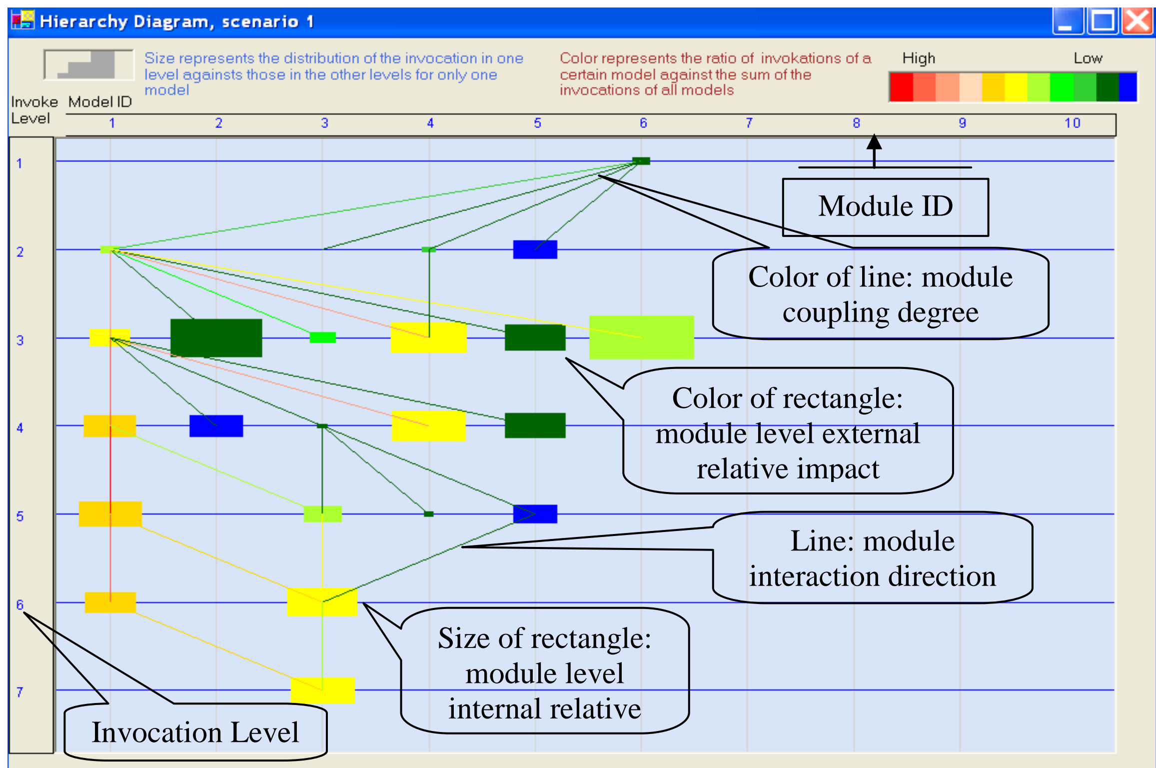


Figure 6: Construction Structure View of System Functionality

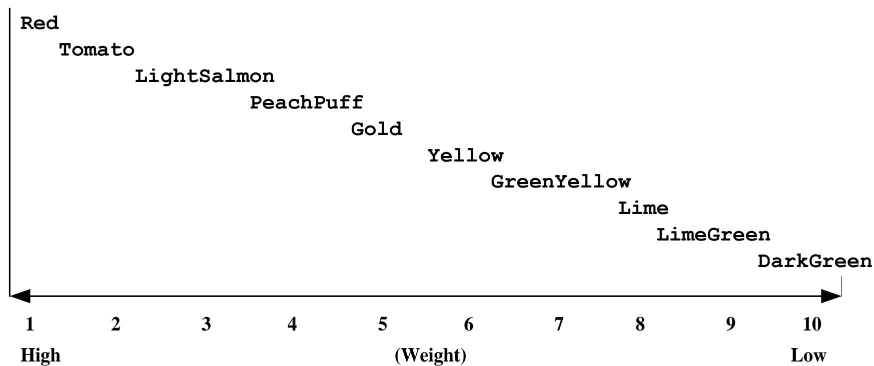


Figure 7: Color Representation Scheme of Weight Variance Gradient

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

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

V. RECORDING SYSTEM FUNCTIONALITY SCENARIO

While analyzing system functionalities and their source code construction structure, it is desirable to record system behavior for the comparison and analysis. As such, Figure 8 illustrates a scenario recording function for Dynamic-Analyzer to capture the screen snapshots of system functionality and behavior.

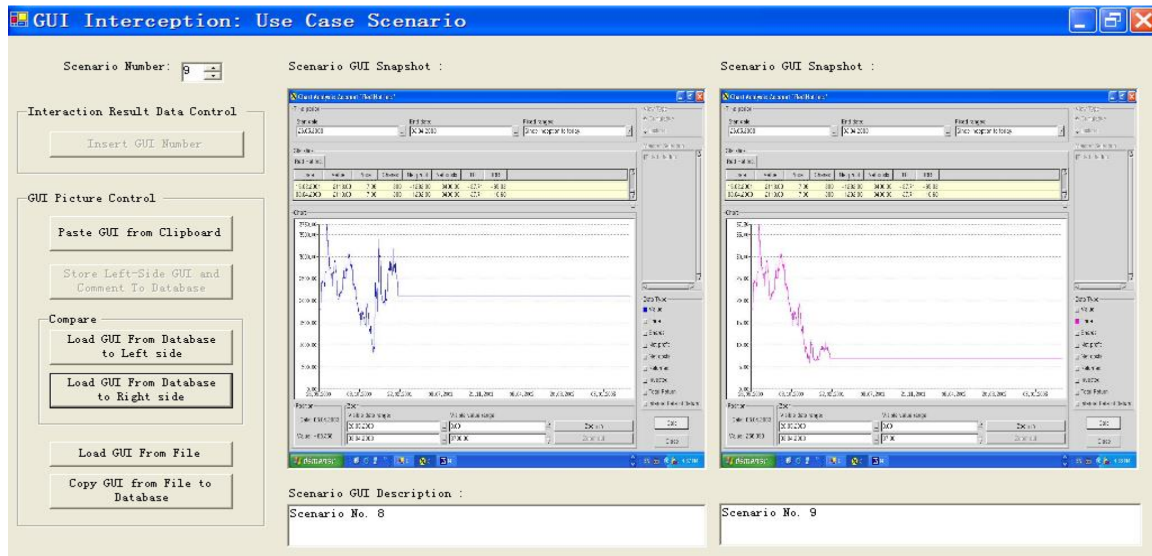


Figure 8: System Functionality Scenario Recorder

The recorded scenario screen snapshots were labeled and stored into repository database for analysis usage. Later, when performing the dynamic visual analysis, the user is able to retrieve the scenario pictures to reflect the observed system functionality, thereby linking the system behavior and the visual analysis results (views and diagrams). In this way, the target system does not need to be executed every time there is a need to study it.

VI. ANALYSIS WITH COLLABORATION-INVESTIGATOR

Implementation of observed system functionality: As discussed previously, a collaboration pattern is a frequently repeated series of collaboration instances, which involves different modules cooperating together to perform a certain type of work that contributes to the system functionality. During the whole process of interactions, those modules involved in the pattern show strong collaboration manners: cooperatingto implement a particular task.

To study the coloration patterns, and further analyze the roles of participant modules, *Collaboration-Investigator* was designed as another reverse engineering tool (see Figure 9) to carry out the approach explained in the previous section.

Seven major tasks are carried out for the recovery and analysis of collaboration patterns and conceptual roles.

1. Pattern criteria establishment
2. Interaction investigation
3. Collaboration pattern recovery
4. Visualization of collaboration pattern
5. Role recovery
6. Collaboration pattern investigation
7. Role investigation

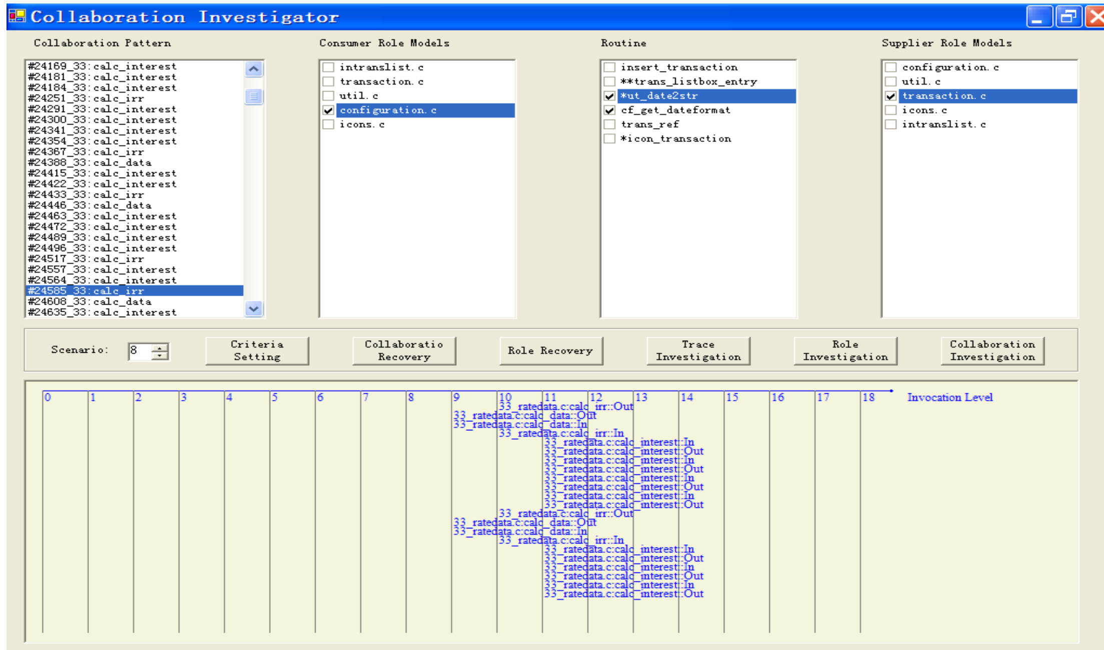


Figure 9: Collaboration-Investigator: A Reverse Engineering Tool for Collaboration Pattern and Role Recovery and Analysis

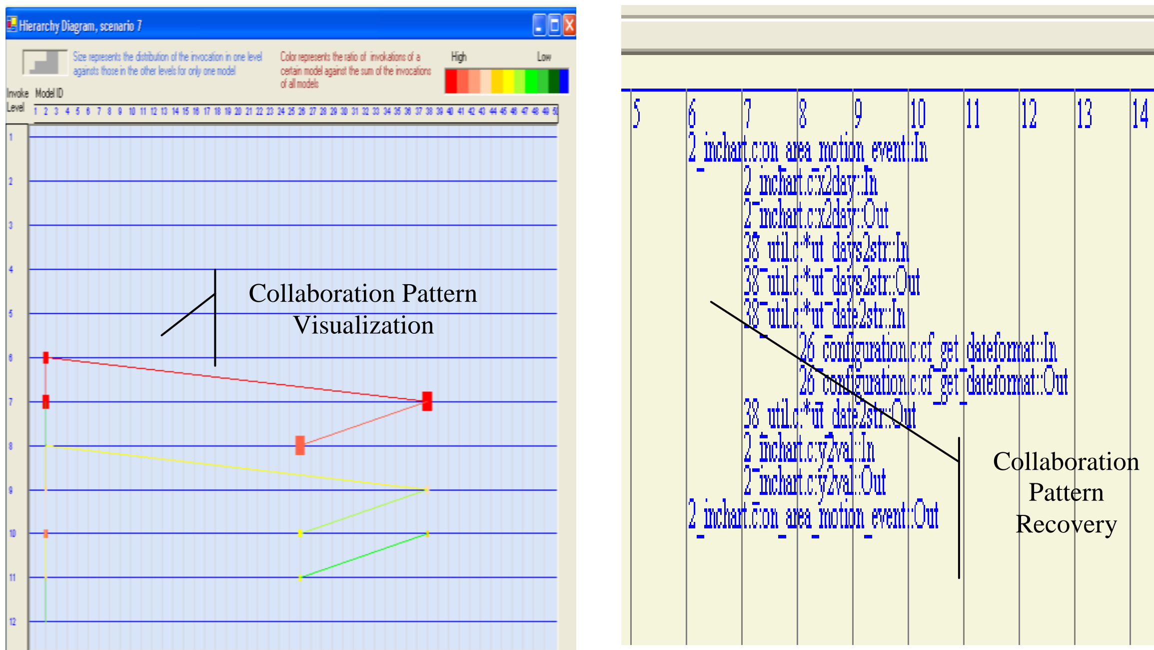


Figure 10: Collaboration Pattern Recovery and Visualization

Collaboration pattern criteria establishment: Three categories of pattern recovery criteria were set up, namely interaction component selection, collaboration instance selection, and pattern matching. The choice of optional items in each category reflected the maintainer's observation emphasis of pattern recovery aspects. The naming convention of distilled collaboration pattern is the unique sequential id number plus the first sender module's name and the first invocation routine name.

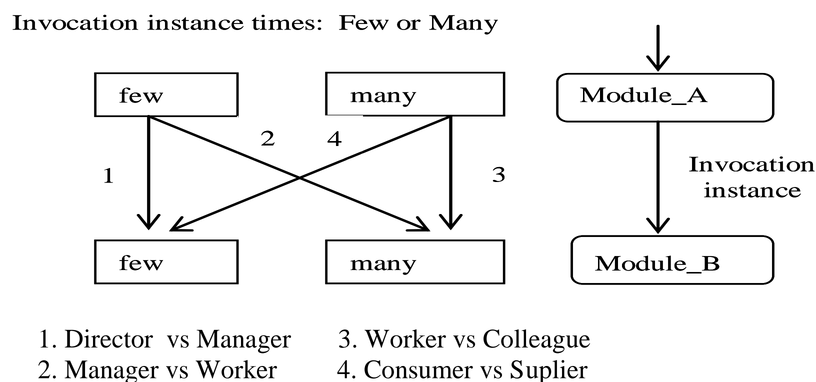
Interaction Investigation. This component is designed to explore all the components of any interaction instance. When selecting the pattern name, the sender module and receiver module, the user can generate all the routines (functions and procedures) that are invoked from sender module to receiver module within the selected collaboration pattern. By selecting a routine name, the tool will also generate the full interaction information in

the collaboration pattern which contains that routine. Through this component, the module interaction space can be fully explored.

Collaboration Pattern Recovery: This component recovers the collaboration patterns based on the criteria that have been previously created. It identifies those significant repetitive interactions, which jointly form a collaboration pattern and role relationships in the large transaction space. The result will be shown in the "collaboration pattern" frame; see Figure 10.

Visualization of Collaboration Pattern. The Dynamic-Analyzer is used to generate the visual effects of recovered collaboration pattern. It also generates the visual representations of the corresponding collaboration instances for a pattern. Figure 10 illustrates a sample recovered collaboration pattern.

Role Recovery: Based on the recovered collaboration patterns, the module relationships and the module roles will be defined according to the role pairs classification, illustrated in Figure 11. For each pattern, a module role table will be generated for the participant. The Collaboration-Investigator would recover four role relationship pairs based on module invocation frequency: "director-manager" (few, few); "manager-worker" (few, many)"; "consumer-supplier" (many, few); and "worker-colleague" (many, many).



1. Director vs Manager
2. Manager vs Worker
3. Worker vs Colleague
4. Consumer vs Suplier

Figure 11: Conceptual Role Pairs

The invocation times indirectly reflect a particular module's level of involvement while contributing to system functionality. The study uses fuzzy concepts "few"" and "many" to symbolize the invocation frequency. It reveals the amount of computation work contributed by this module to the specific system functionality. "Few" indicates that this module generally dispatches the work load to partner modules, and it is in charge of management work whereas "many" reveals that this module has taken a considerable work load portion of the system functionality and is in charge of the implementation part of system functionality.

Four role-relation pairs and eight roles to reveal the relationships among code modules have been designed: (i) Director vs Manager (Few-Few); (ii) Manager vs Worker (Few-Many); (iii) Retailer vs Wholesaler (Many-Few); (iv) Worker vs Colleague (Many-Many).

Collaboration Investigation. This component will generate the query results for related collaboration patterns. The collaboration pattern list can be produced which contains a set of selected routines, and can discover the features by exploring the collaboration pattern components. The selection of a sender module or receiver module in order to find out the other related parts had similar effects. Any combination of these four elements (pattern, sender module, receive module and routine) can be used to generate the list of the remaining elements.

Role Investigation. This component explores the role space of each module in a selected collaboration pattern. When the sender module assigns a certain conceptual role, the modules that have the corresponding role in a given collaboration pattern can be retrieved. Further, roles for a single module can be compared in different patterns and produce the dominant role for a particular module based on the multiple role information that it has carried out in different collaboration patterns.

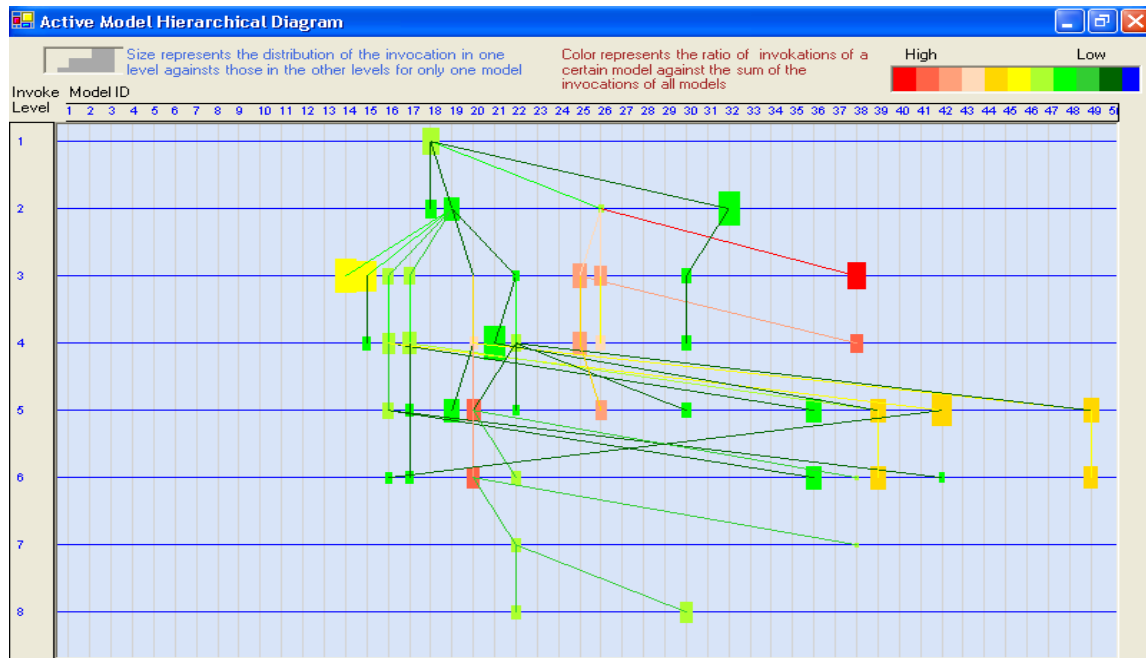


Figure 12: Partial System Constructional Structure Visualization

With the help of visual expression provided by the Dyanmic-Anaylzer and the analysis components provided by the Collaboration-Investigator, a comprehensive system functionality construction structure can be generated to reveal partial system architecture.

Figure 12 illustrates the system module constructional structure that implements one sub-system's functionality in a sample 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, and the assignments of the major role for each module.

VII. CONCLUSION

Interoperability among an enterprise legacy system and a contemporary information system requires a more efficient way to analyze the legacy system's architecture and design. This paper presents a novel approach to efficiently recover and analyze legacy system architecture and design through collaboration pattern and role analysis.

The approach consists of two major parts, both of which are supported by the reverse engineering tools used in this study. The first part focuses on the dynamic analysis of target legacy systems and the automatic discovery of legacy system's architecture. The second part concentrates on the recovery and analysis of legacy system design through collaboration pattern and role analysis.

Each recovered collaboration pattern represents a concrete implementation block of the observed system functionality. By characterizing program construction, the insight can be gained about the way legacy system behavior is carried out through the collaborations of its basic design unit. It is also useful to apply the discovered collaboration patterns to further decompose the whole system into a role-based hierarchical representation. Inspectors can use this information to study each module within various collaboration patterns, regaining more detailed architecture and design information. Cohesive measurement is also used to perform legacy re-construction [Van00][Kui00][Cha02]. Within a collaboration pattern, composition modules intensively cooperate together to perform a concrete system function. Therefore, both Dyanmic-Anaylzer and Collaboration-Investigator can be further used in the re-modularization of enterprise legacy system, providing the interoperability with other contemperary software systems.

The architecture and design recovery process provides a decomposition view of legacy software [Kos02]. Most research on understanding interactions has focused on visualization techniques[Lan03][Lud02][Boh11][Wat15], where the challenge is to develop efficient way to visualize the large amount of dynamic information [Wal98][Sys01]. The work of DePauw et al. [Pau98], which is currently integrated with Jinsight, allows engineers to visually recognize patterns in the interactions of classes and objects. ISVis displays interaction diagrams using a mural [Jer97] technique. The current work in visualization is the combination of these two approaches. Instead of the mere focus on visualization, the current approach places greater emphasis on the recovery of legacy sytem architecture and the understanding of legacy system design. Tamar et al. also propose

an approach to analysis of roles within collaborations [Ric02], using the invocation methods as representative of roles. Their approach is not sufficient to analyze the general function of a module inside of recovered collaboration pattern. The study proposes a significantly improved model that uses predefined conceptual role stereotypes for the recovery of roles based on the invocation relations with other modules. Test results using case studies presented in this paper confirm that the novel approach proposed in this paper offers a more robust, accurate and efficient solution for architecture recovery of legacy systems.

REFERENCES

[1]. [Bal99] T.Ball, "The concept of dynamic analysis." Proceedings of ESEC/FSE, LNCS, 1999, pp. 216-234

[2]. [Bas03] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice" (2nd edition), Addison-Wesley 2003:

[3]. [Bri02]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

[4]. [Cha02] S. M. Charters, C. Knight, N. Thomas, M. Munro: "Visualisation for informed decision making; from code to components." *Proceedings of the 14th international conference on software engineering and knowledge engineering,* July 15-19, 2002, Ischia, Italy. ACM, p765-772 2002

[5]. [Can00] G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, "Decomposing Legacy Programs: A First Step Towards Migrating to Client-Server Platforms," *The Journal of Systems and Software,* vol. 54, 2000, pp. 99-110.

[6]. [Can01]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]. [Ede03] 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]. [Elo02] J. Eloff, "*Software Restructuring: Implementing a Code Abstraction Transformation,*" *ACM International Conference Proceedings of SAICSIT* 2002.

[9]. [Hal88] HALL, R.P. "Seven Ways to Cut Software Maintenance Costs (Digest)," in PARIKH, G.: *Techniques of Program & System Maintenance* (QED Information Sciences Inc., 1988 2nd edition.

[10]. [Jac00]Daniel Jackson & Martin Rinard. "Software Analysis: a Road Map," "The Future of Software Engineering," Anthony Finkelstein (Ed.), ACM Press 2000.

[11]. [Jer97] D.Jerding, S.Rugaber, "Using visualization for architectural localization and extraction," *Proceedings* WCRE, IEEE Computer Society, 1997, pp. 56-65.

[12]. [Kos02] Rainer Koschke. "Atomic Architectural Component Recovery for Program Understanding and Evolution," *In Proceedings of the International Conference on Software Maintenance,* Montréal, Canada, October 2002.

[13]. [Lan03] Michele Lanza, Stephane Ducasse. "Polymetric Views - A Lightweight Visual Approach to Reverse Engineering." *IEEE Transactions on Software Engineering (TSE),* Vol. 29, No. 9, pp. 782-795, September 2003.

[14]. [Lak97] A. Lakhotia. "A Unified Framework for Expressing Software Subsystem Classification Techniques." *Journal of Systems and Software*, pp. 211–231, March 1997.

[15]. [Lud02]Martin Ludger, Anke Giesl, Johannes Martin. "Dynamic Component Program Visualisation." *In Proceedings of the 9th Working Conference for Reverse Engineering,* Richmond, Virginia, October 2002.

[16]. [Let99] A.N. Lethbridge, "Recovering Software Architecture from the Names of Source Files," *Journal of Software Maintenance: Research and Practice*, November, 1999, pp. 201-221.

[17]. [Mic03] Isabel Michiels, Dirk Deridder, Herman Tromp and Andy Zaidman, "Identifying Problems in Legacy Software," Elisa ICSM workshop 2003

[18]. [Pau98] W.D. Pauw, D. Lorenz, J.Vlissides, and M. Wgman, "Execution Patterns in Object-Oriented Visualization," *Proceedings Conference on Object-Oriented Technologies and Systems* (COOTS'98), USENIX, 1998, pp. 219-234

[19]. [Ric97]R. Richardson, D. Lawless, J. Bisbal, B. Wu, J. Grimson, and V. Wade "A Survey of Research into Legacy System Migration," Technical Report TCD-CS-1997-01, Computer Science Department, Trinity College Dublin. January 1997.

[20]. [Som00] Ian Sommerville "Software Engineering," 6th edition. Addison-Wesley 2000

[21]. [Sys01] T. Systa, K. Koskimies, H. Muller. "Shimba – An Environment for Reverse Engineering Java Software Systems." Software –Practice and Experience, 1(1), January 2001.

[22]. [Wal98]R.J. Walker, G.C. Murphy, B.F. Benson, D. Wright, D. Swanson and J. Issaak. "Visualizing Dynamic Software System Information Through High-Level Models," Proceeding OOPSLA'98, 1998, pp.271-283.

[23]. [Boh 11] 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.

[24]. [Wat 15] 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: 20-37.