COMPUTATIONAL ANALYSIS OF CODE COLLABORATION PATTERN AND SEMANTIC ROLE

Lei Wu[1], Sharon White[1], Yi Feng [2], James Helm[1], Nathanial Wiggins[3]

[1]Software Engineering, University of Houston-Clear Lake, Houston, USA
wul,whites,helm@uhcl.edu
[2]Computer Science, Algoma University, Sault Ste. Marie, Canada
feng@algomau.ca
[3]Mathematics and Engineering, San Jacinto College, Houston, USA
Nathanial.Wiggins@sjcd.edu

ABSTRACT

Software functionalities and behavior are accomplished by the cooperation of code artifacts. The understanding of this type of source code collaboration provides an important aid to the maintenance and evolution of legacy systems. However, the original collaboration design information is dispersed at the implementation level. The extraction of code artifacts' collaborations and the roles is therefore an important support in legacy software comprehension and design recovery. In this paper, we present a novel approach to efficiently recover and analyze code collaborations and semantic roles based on dynamic program analysis technique. We also demonstrate the software tools that we have developed to support our approach and illustrate the viability of our approach in a case study.

KEYWORDS

Data mining, code collaboration pattern, semantic role, design recovery, dynamic program analysis, software visualization, reverse engineering

1. INTRODUCTION

During the last three decades, a considerable amount of software was developed using procedural languages. For example, Coyle et. al. estimated the size of legacy systems written in procedure and OO languages, such as Cobol, Fortran, ADA, etc., to be more than 100 billion LOC [17][43][45]. These types of systems have undergone several code revisions without a real concern of maintaining the documentation up-to-date [18][22][24][31]. As a consequence, the higher level of entropy combined with imprecise documentation about the design and architecture has made their maintenance more difficult, time consuming, and costly. On the other hand, these systems have important economic value and many of them are crucial for their owners [19][17][[18]. All these factors underline the importance of legacy system understanding.

Large legacy procedural systems are normally organized in a structured form [2][3][25]. Code is divided into separated source files based on different design criteria [1][2]21]. 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 they participate [2][26]29]. This kind of program code organization reflects the original legacy design rational [2][4]30]. Each source file and directory represents a certain design concept [2][18][37]. Each code cooperation instance contains a limited number of such code units. We view these construction units as source code modules which interact to realize the system functional behavior [2][41][34]. Meanwhile, each module plays specific conceptual semantic roles inside the cooperation[32][35][39]. Recovering collaborations and semantic roles from source code artifacts is an important factor for better understanding and evolving legacy code [16][33][18]. However, the large number of modules and the complexity of their relationships make discovering and analyzing interactions a hard task [18][17]. It is therefore very difficult to study system behavior by only using static information.

In this paper, we propose a novel program analysis approach for the efficient recovery of code collaborations and conceptual semantic roles from source code artifacts. We apply dynamic program analysis, software visualization and knowledge recovery techniques to facilitate legacy code understanding. We have developed two types of analysis tools set, namely *DynamicViewer* and *Collaboration-Investigator*, to validate the viability of our approach. Through examples, we illustrate how these two tools are used to detect, recover and analyze collaborations and roles in an automatic/semi-automatic way. The paper is structured as follows: in the next section, we introduce our approach. In section 3, we discuss our approach in detail and present the tools we have developed to support the recovery process. In section 4, we work through a case study of program comprehension using our approach. In section 5, we further discuss the issues and lessons learned from the experiment. In section 6, we review related work. In section 7, we give the conclusion of our study and future work

2. DYNAMIC RECOVERY APPROACH

Collaboration and conceptual semantic role are two design concepts that have been scattered throughout source code [10][42]. Inside of collaboration, participant modules interact with each other to carry out detailed tasks. The cooperation is confined in an interaction structure form, which describes a set of allowed collaboration behaviors for each module [36][40]. Such structure is implemented and dispersed in code with two major design concepts: the repetitive interaction pattern and role. Each participant plays a certain type of conceptual role in the collaborations [44][17]. Meanwhile, conceptual semantic role also enforces the information processing in a consistent and meaningful (understandable to maintainer) way [47][27]23]. Recovery such information can largely facilitate the program comprehension process and promote program analysis into a deeper level [28][46][38]. Since procedure computing languages do not provide explicit means to capture such design information, maintainers have to heavily rely on human efforts to investigate these design logics in legacy software.

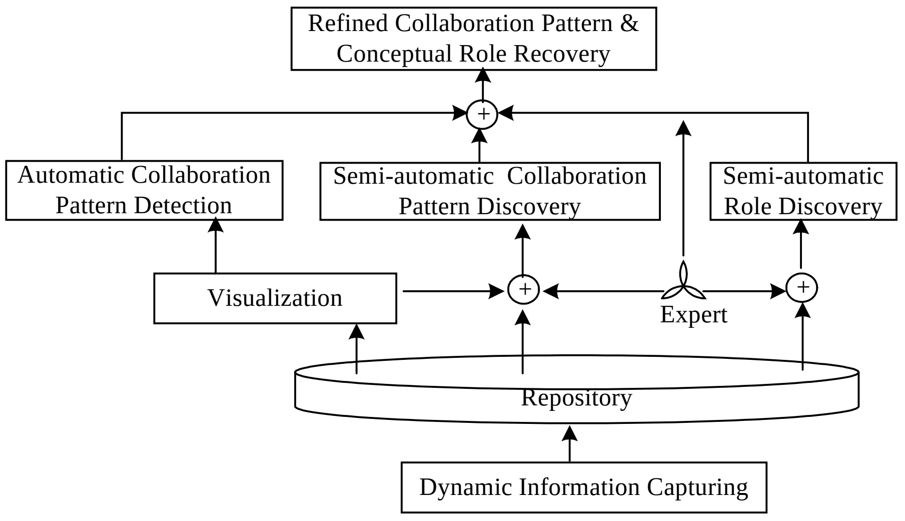

Refined Collaboration Pattern & Conceptual Role Recovery
Automatic Collaboration Pattern Detection
Semi-automatic Collaboration Pattern Discovery
Semi-automatic Role Discovery
Visualization
Expert
Repository
Dynamic Information Capturing

Figure 1. Schema of the dynamic analysis approach

To recover collaborations and roles from legacy source code, we propose an approach that can significantly increase maintainers' work efficiency in discovery such information. Our approach uses dynamic analysis, software visualization, automatic and semi-automatic detection techniques to fulfill the goal, see figure 1. We first capture dynamic interaction information among modules during the target system execution period. Then we analyze the features with dynamic visualization, such as the interaction composition, transaction sequence and recurrence frequency etc. Later, automatic pattern detection process will be performed to recover all the significant repetitive transaction serials. Meanwhile, with the intervention of maintainers, our tool may also be able to semi-automatically detect the collaboration pattern and participants' roles, and investigate their features. In addition, the crosscheck and refinement process will be conducted to combine the two results, and distill the final refined results. The following are the key issues that addressed in our approach.

Dynamic information capturing. We use dynamic analysis technique to capture module interaction message, data transformation route and control flow information during program execution period.

Visualization. We use computer graphic simulation to represent the captured information into a more understandable visual form. Meanwhile, we symbolize all those analysis results into graphical views. There are two kind of information we will visualize: one is the pure interaction information that represents what is going on inside the code; the other one is the statistical analysis results based on data mining. For the first one, we use both static visualization and animation to simulate the dynamic nature of message transactions. For the latter one, we use graphical diagrams and graphs to visualize the statistical analysis results.

Automatic and semi-automatic collaboration pattern and role detection. With the results from former two processes, we will be able to study the features of dynamic transactions, such as interaction composition, direction, sequence and frequency etc. To discover the collaborations and conceptual roles that dispersed over the huge amount of transactions, we apply automatic and semi-automatic approaches to accomplish our recovery goals. The difficulty lies in how to efficiently identify those significant repetitive interactions, which together form a meaningful collaboration pattern in the large transaction space. We apply our automatic detection technique to directly use dynamic visualization result to produce fine-grained collaboration pattern outcomes. The recovery does not need expert's intervention. The advantage is that it is capable to detect a wide range of collaboration patterns, while the disadvantage is maintainer may lost the control of expressing her focus in discovering patterns. To overcome this shortage, we also adopt a guided semi-automated recovery of collaborations and roles with the interventions of maintainers. According to maintainer's different emphasis, she may interactively give the recovery criteria. The collaboration pattern discovery results will reflect the focused interests of maintainer, thus they only provide the most interested features of various collaborations and roles. Finally, these two types of results will be combined and refined to produce the final recovery result.

3. CODE COLLABORATION PATTERNS AND SEMANTIC ROLES RECOVERY

In this section, we present our approach in detail, and introduce the tools which were developed to automate the recovery process. We first explain the underlying terminology and concepts; then, we present our automatic and semi-automatic mechanisms which are supported by these tools.

3.1. Terminology and analysis formalism

We first introduce the dynamic trace record - the program trace information that we captured during the execution of target legacy system. A sample segment of the trace is given below.

Sid,	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

Sid stands in the above segment for scenario identification number, which correspondent to the specific system functionality that is performed at that moment. The level represents the invocation depth. Direction is the orientation of message flow. This trace segment sample includes three routine invocation events. Each event record has five elements. We use the second record to illustrate them: the scenario id (3), invocation level (5); sender module (indateentry.c); receiver module (inransinp.c); receiver invocated routine (on_input_data_changed) and direction (in).

Source module: The source code of a system is normally organized in a structured form [2,3]. Code unit related to same concept or topic are usually grouped in a single source file and further stored into different directories, which reflect the original design rational [2]. We view source file or directory as source module, or simply say module.

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

Collaboration instance: A collaboration instance is the sequence of successive interaction instances, which together form a chain of continuous events that generate a message propagation tree.

Collaboration 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 appear and cooperate together to implement a certain type of task. We view this kind of phenomenon as a repetitive collaboration pattern.

Conceptual semantic role: A conceptual semantic role is the characteristic and predictable behavioral stereotype of an individual module based on its computational feature. It represents the general conclusions of the module’s utility in program. From the construction point of view, the role can be semantically transformed into director-manager-worker relationships. Module with high level role dispatches tasks to modules with lower level roles. From information process point of view, consumer and supplier are the most common roles.

3.2. Pattern discovery criteria

To efficiently recover collaboration patterns from execution trace information, we have to give our tools a certain type of criteria to emphasize what aspect is more important in deciding which sequences of collaboration instances are related, and may thus be further grouped together to form a concrete pattern. The criteria can be chosen among the following three categories:

Interaction instance component. An interaction instance includes six major components, namely scenario id, invocation level, sender module, receiver module, invocated routine and direction. Based on different bias, 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 message propagation tree. For this reason, we may select to view only the interaction instances involving different modules. Meanwhile, to limit the observation scope, we may also define the consideration boundary that confines the recovery process within a certain depth range.

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, we may let our tool omit some considerations of the ordering sequence when comparing two collaboration patterns.

3.3. Automatic discovery with *DynamicViewer*

As we discussed in the introduction, static analysis does not present sufficient information to study the interactions of source modules. Recording dynamic information of program can provide us with sufficient knowledge about message exchanges during program execution period. However, this technique faces to two major issues: the overwhelming volume of tracing data and incomplete coverage of the code. In our approach, since the focus is on a limited set of system exercising functionalities and behaviors, the dynamic coverage only contains the relevant code artifacts that concerned with specific system functionality. In fact, this turns to benefit the resolution of the first issue of dynamic analysis technique [5]. Meanwhile, to significantly reduce the large volume of tracing data, we use automatic pattern discovery technique implemented in our toolkit to accelerate the recovery process. We have developed a tool, the *DynamicViewer*, to automate the dynamic capturing and visualization of message process flows among source modules, see following figure 2.

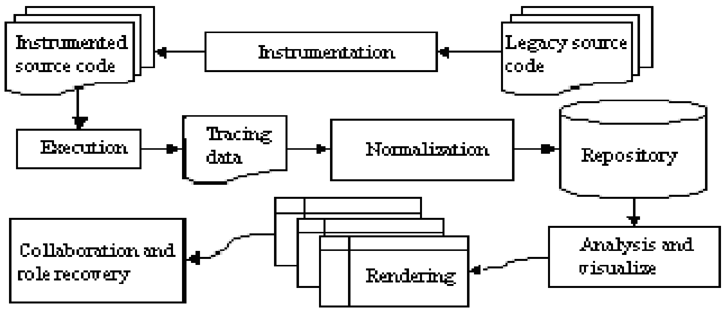
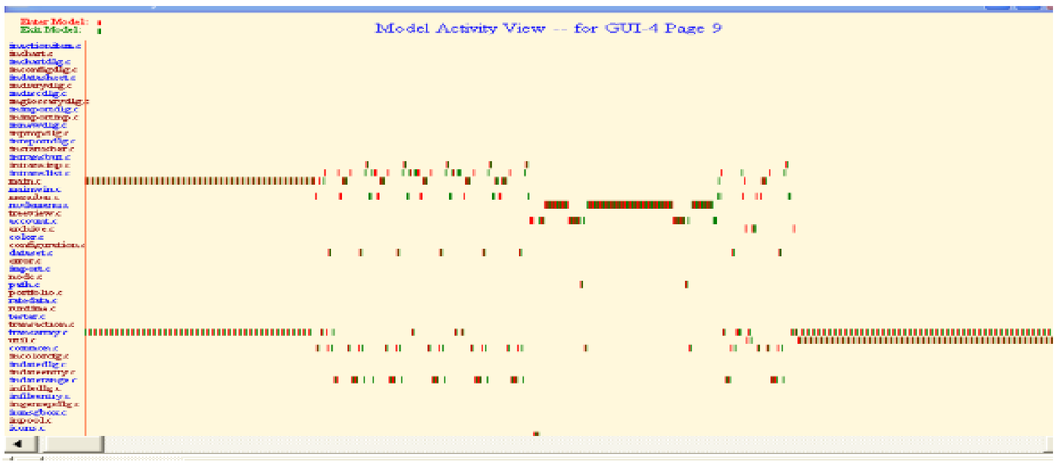


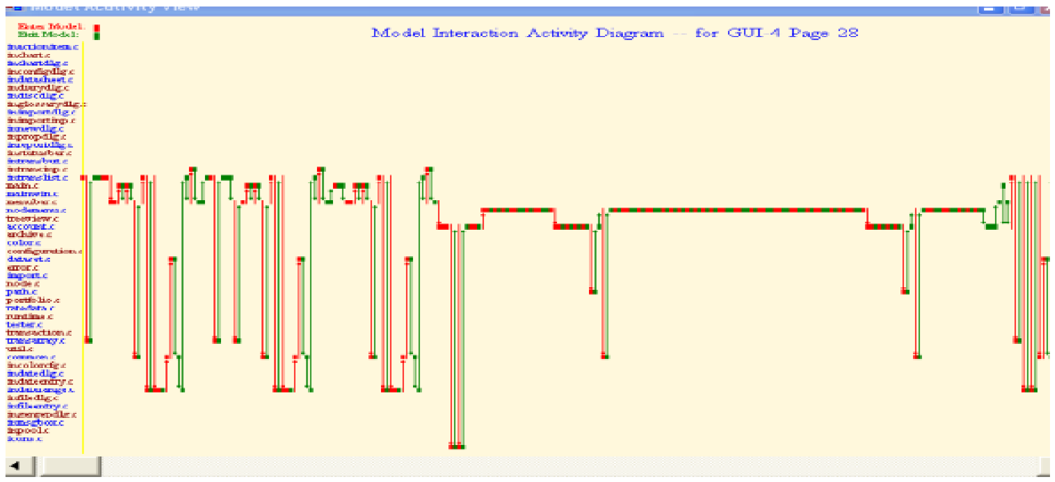
Figure 2. Workflow of *DynamicViewer*

First, legacy source code is instrumented to record execution information. Then, the interested system functionalities are executed to observe system behaviors; meanwhile, program dynamic information is retrieved and processed into repository. Later, the visualization and animation program will present the information through visual effects to provide a meaningful way to investigate the interactions. Finally, the automatic collaboration pattern detection process will be performed to distil all the patterns. Currently, *DynamicViewer* can automatically discover all the fine-grained collaboration patterns. We've also developed another tool to incorporate human intervention in the discovery process, and combine the outcomes from *DynamicViewer* to generate the best result. That semi-automation approach will be detailed in section 3.4.

One desired feature of *DynamicViewer* is both legacy system and analysis tool will run in parallel. Maintainers now are able to observe system behavior and the visualization of source module interactions/patterns at the same time. In this way, they can directly relate any specific system behavior with the visual effects of module actions in real-time, thus largely reduce the memory work to match these two concepts.



(i)



(ii)

Figure 3. Footprint (i) and pattern detection (ii) views from *DynamicViewer*

DynamicViewer provides maintainers with an efficient way to automatically discover the collaboration patterns, and look inside of the program execution space. It defines different types of views to cover information at different granularity levels. Furthermore, it also facilitates the smooth navigation among various granularity levels. It visualizes two types of information: the pure interactions, and the knowledge mining result. For the first one, it supports static and animate visual effects. The above figure 3 illustrates the fine-grained footprint and animated pattern detection views produced by *DynamicViewer*. The left-most vertical part shows the name of modules; the horizontal direction represents the time sequence; the red box indicates an invocation interaction instance from the send module; the green box shows the return of interaction instance from the receiver module; the red line with direction point shows an outgoing message from sender module towards receiver module; and green line with direction point represents the returning of the interaction message from receiver module back to sender module. *DynamicViewer* can automatically detect all the repetitive serial of collaboration instances, and distil them as candidate collaboration patterns.

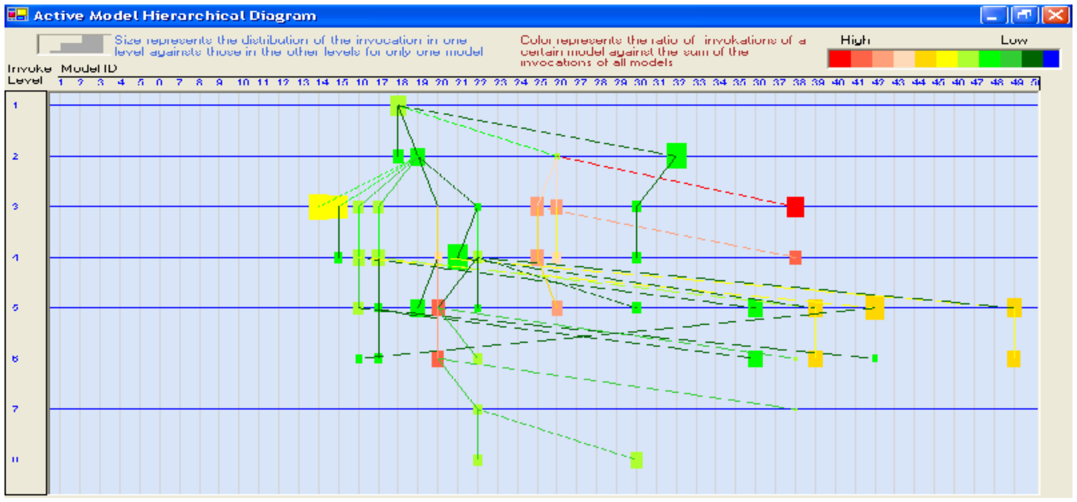

Active Model Hierarchical Diagram
Size represents the distribution of the invocation in one level againsts those in the other levels for only one model
Color represents the ratio of invokations of a certain model against the sum of the invocations of all models
High
Low
Invoke Level
Model ID
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
1
2
3
4
5
6
7
8

Figure 4. Module dispersal & scale diagram from *DynamicViewer*

For the second type of information, *DynamicViewer* represents the knowledge mining results with graphical diagrams. The conclusive visual report of statistic data gives maintainers an efficient way to analyze the source code interaction behaviors. The dynamic module dispersal and scale graph shows the overall performance of each module in different transaction levels (see figure 4). The statistical data is rendered in the forms of size and color. The row represents the invocation depth and the column indicates each module. When sender module invokes a message to a receiver module, the receiver will lie on one depth below the sender. The rectangle color box represents the quantity of interaction instances at different depths. Its color shows the overall invocation scale of the whole interaction space, and the box's size exhibits the scale compared with all of its own interactions. The line displays the invocation direction from higher depth to lower depth, and its color shows the intensity. *DynamicViewer* also provides a scenario recording function to capture the system functionality and behavior, see following figure 5. The recorded scenario screen snapshots will be labeled and stored into repository database for analysis usage.

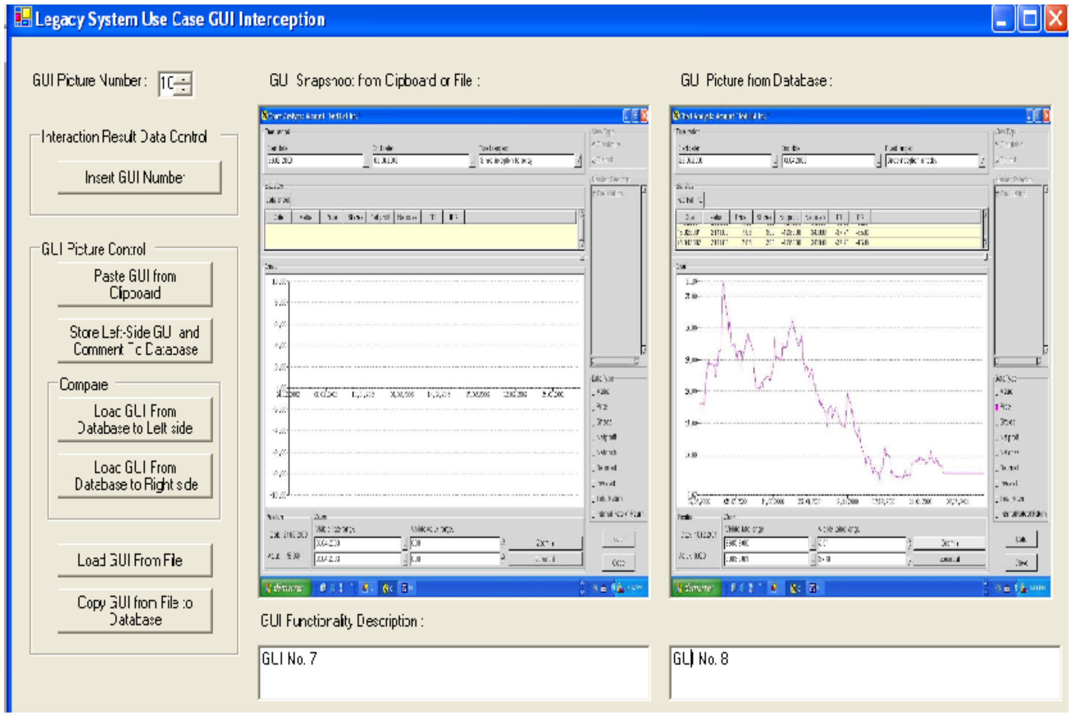

Legacy System Use Case GUI Interception
GUI Picture Number:
GUI Snapshot from Clipboard or File :
GUI Picture from Database :
Interaction Result Data Control
Insert GUI Number
GUI Picture Control
Paste GUI from Clipboard
Store Left-Side GUI and Comment To Database
Compare
Load GUI From Database to Left side
Load GUI From Database to Right side
Load GUI From File
Copy GUI from File to Database
GUI Functionality Description :
GUI No. 7
GUI No. 8

Figure 5. Scenario recorder

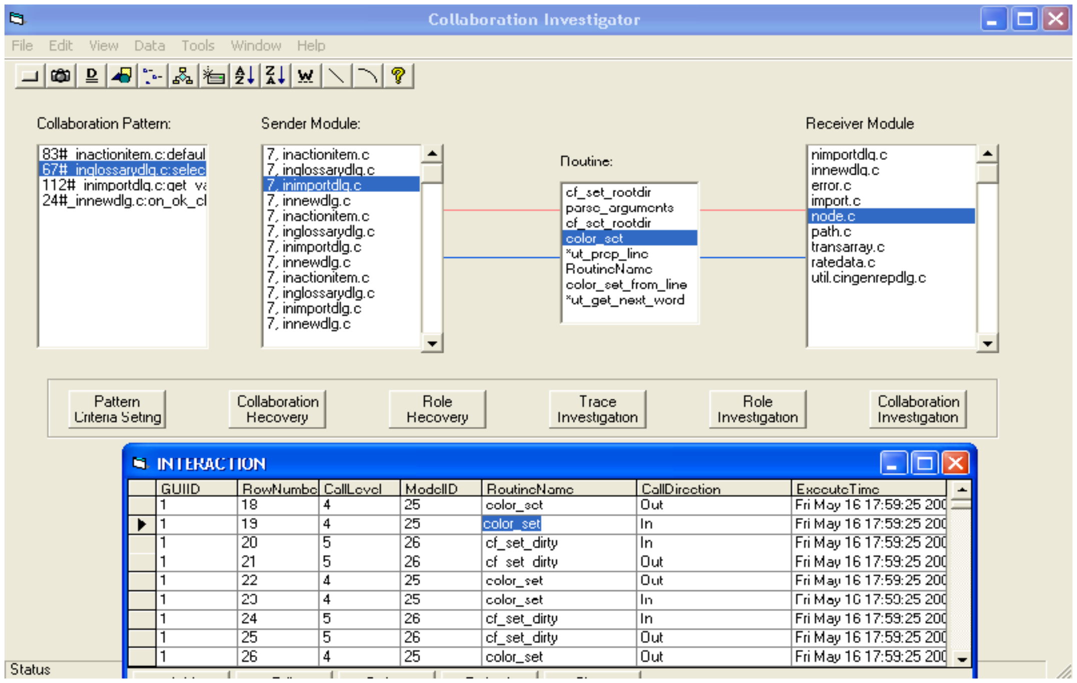
Later, when maintainer wishes to explore the dynamic interaction space, she may also be able to retrieve the scenario pictures to link the system behavior and the visualization results. In this way, she may not need to execute the target system every time when she wants to investigate.

3.4. Semi-automatic recovery with Collaboration-Investigator

The main functionality of *Collaboration-Investigator* is to help maintainers efficiently discover the featured/screened collaboration patterns and involved participation roles, as well as assisting them thoroughly to investigate the details. It provides five major operations for the study of collaborations and conceptual roles, namely pattern criteria setting, collaboration recovery, role recovery and their investigations. It also retrieves other useful information related to module collaboration and roles. Maintainers may retrieve, abstract and compare different system behavior, collaboration patterns and roles in an operational manner.

The snapshot (see figure 6) of the demonstration sample is from the analysis of “Interest” legacy system, which will be presented in detail as case study in the following section. Here, we will introduce the functionalities of *Collaboration-Investigator*.

Collaboration pattern criteria establishment. The “Pattern Setting” function will prompt the criteria building form for maintainer. She has to check three groups of pattern recovery criteria, namely interaction component, collaboration instance selection and pattern reshaping. These three categories are detailed in previous section. The choice of optional items in each category reflects maintainer’s observation emphasis of pattern selection aspects.



Collaboration Pattern:

83# inactionitem.c:defaul
67# inglossarydlg.c:selec
112# inimportdlg.c:get va
24#_innewdlg.c:on_ok_cl

Sender Module:

7, inactionitem.c
7, inglossarydlg.c
7, inimportdlg.c
7, innewdlg.c
7, inactionitem.c
7, inglossarydlg.c
7, inimportdlg.c
7, innewdlg.c
7, inactionitem.c
7, inglossarydlg.c
7, inimportdlg.c
7, innewdlg.c

Routine:

cf_set_rootdir
parse_arguments
cf_set_rootdir
color_set
*ut_prep_line
RoutineName
color_set_from_line
*ut_get_next_word

Receiver Module

nimportdlg.c
innewdlg.c
error.c
import.c
node.c
path.c
transarray.c
ratedata.c
util.cingenrepdlg.c

GUIID	RowNumbe	CallLevel	ModelID	RoutineName	CallDirection	ExecuteTime
1	18	4	25	color_set	Out	Fri May 16 17:59:25 200
1	19	4	25	color_set	In	Fri May 16 17:59:25 200
1	20	5	26	cf_set_dirty	In	Fri May 16 17:59:25 200
1	21	5	26	cf_set_dirty	Out	Fri May 16 17:59:25 200
1	22	4	25	color_set	Out	Fri May 16 17:59:25 200
1	23	4	25	color_set	In	Fri May 16 17:59:25 200
1	24	5	26	cf_set_dirty	In	Fri May 16 17:59:25 200
1	25	5	26	cf_set_dirty	Out	Fri May 16 17:59:25 200
1	26	4	25	color_set	Out	Fri May 16 17:59:25 200

Figure 6. *Collaboration-Investigator*: the semi-automatic pattern and role recovery toolset

Collaboration and role recovery: This function will recover the most significant collaboration patterns based on previously set criteria. The result will be shown in the “collaboration pattern” frame. The naming convention of distilled collaboration is the unique sequential id number plus the first sender module’s name and the first invocation routine name. “Role Recovery” function is used to generate the participant role table. Currently, it provides “director-manager-worker” role stereotype and simplified “supper-consumer” role stereotype.

Interaction investigation: The “Trace Investigation” function is used to explore all the components of an interaction instance. In figure 6, the pattern name “67#_inglossarydlg.c:select_event”, the sender module “imimportdlg.c” and receiver module “rode.c” are selected. After push “Trace Investigation” button, all the routines (functions and procedures) that were invoked from sender module to receiver module within the selected collaboration pattern will be listed in the *Routine Module* panel. If we select routine “color_set”, then the full detail interaction information will be presented in the Interaction Table.

Collaboration investigation: This function is to generate the query results for related collaboration patterns. For example, in figure 6, when maintainer selects a set of routines and presses “Collaboration-Investigation” button, all the patterns that involve any one of these routines will be listed in “Collaboration Pattern” panel. The selecting of a sender module or receiver module in order to find the other related parts has similar effects. We also can use any combination of these four elements, (pattern, sender module, receive module and routine), to generate the list of the remaining elements.

Role investigation: This function will help maintainer to explore the role of each module in a selected collaboration pattern. We assign to the sender module a “consumer” conceptual role, and to the receiver module a “supplier” one. If three modules have sequential consumer-supplier role relationships, we consider that they realize a “director-manager-worker” role stereotype. This kind of role information will help maintainer recover the structure of the program. When user selects one module and presses *Role Investigation* button, all the modules that have a role relationship with that targeted module will be listed in a prompted *Role Module* list window.

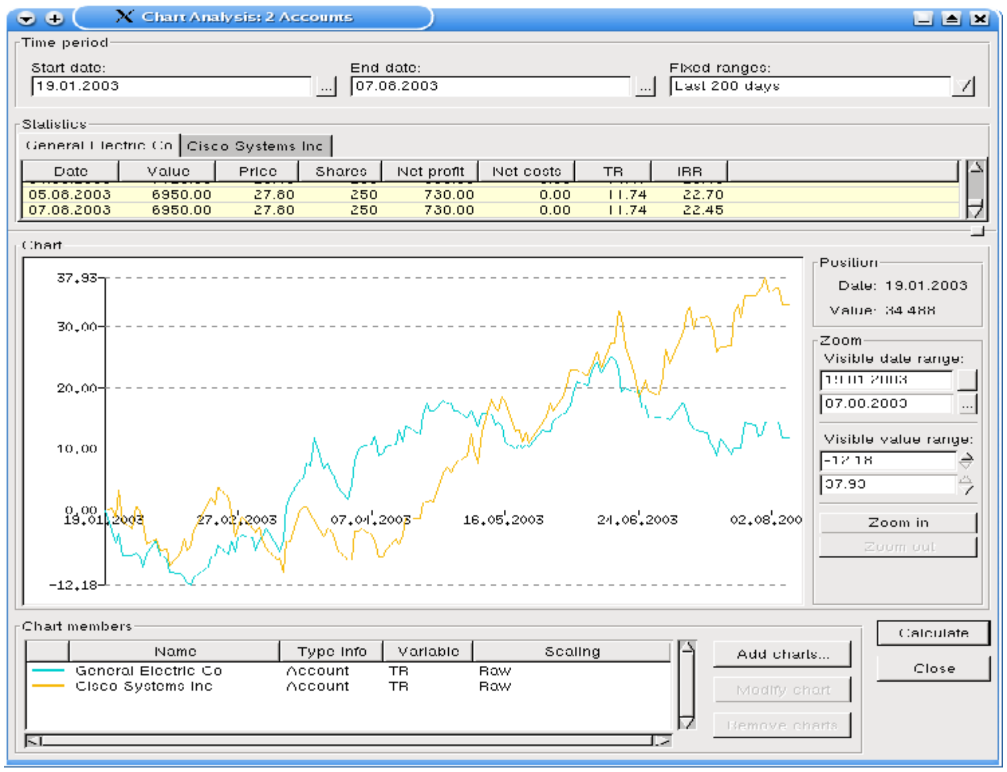
Visualize collaboration pattern: The *DynamicViewer* will be used to generate the recovered collaboration pattern, and it can zoom in/out to facilitate the study. For example, in figure 6, when the pattern called "87#_inglossarydlg.c:init" is selected, then *DynamicViewer* will generate the visual representations of its correspondent collaboration instances in another window.

4. CASE STUDY: UNDERSTANDING "INTEREST"

In this section, we demonstrate how our approach supports the understanding and recovery of code collaborations and semantic roles in a legacy system. The example software "*Interest*" is a financial management system for personal investments. It is written in C with 94 source files of approximately 28KLOC. It is designed to analyze individual stock market investment performance (see figure 7). It is free software, distributed under the GNU General Public License, and can be obtained from web site http://sourceforge.net/projects/interest

4.1. The analysis question and hypothesis

The legacy system provides a bunch of tools to help users analyze their stock investment performance. To better understand how these tools are implemented, we put forward several questions and hypotheses to start our study. We notice that the code is divided into three major directories. Source root "src/" contains two subdirectories, namely "src/base/" and "src/widgets/" respectively. Meanwhile, the source file names under each directory have different characteristics. For example, the files under "/base" subdirectory have names like "color.c", "error.c", "transaction.c" etc. We thus hypothesize that the modules under different directory deal with different functional issues.



Start date: 19.01.2003
End date: 07.08.2003
Fixed ranges: Last 200 days

Statistics

General Electric Co | Cisco Systems Inc

Date	Value	Price	Shares	Net profit	Net costs	TR	IRR
05.08.2003	6950.00	27.80	250	730.00	0.00	11.74	22.70
07.08.2003	6950.00	27.80	250	730.00	0.00	11.74	22.45

Position
Date: 19.01.2003
Value: 34.488

Zoom
Visible date range:
19.01.2003
07.08.2003
Visible value range:
-12.18
37.93

Chart members

	Name	Type Info	Variable	Scaling
🟦	General Electric Co	Account	TR	Raw
🟧	Cisco Systems Inc	Account	TR	Raw

Figure 7. "Interest"- a financial management system for personal investments

Questions:

- What's the relationship between these source files? What are the roles and general functions they represent?
- Which modules work together to make the most significant contributions?
- How do they cooperate with each other?

Hypotheses:

- We suspect that the modules from root directory provide major system functions, while modules from the other two subdirectories yields support for those modules.
- Modules follow a certain pattern in cooperating with each other in order to implement the functionalities.
- Besides, we also guess that each module represents a certain system constructional concept, which reflects a distinct role in the whole software.

4.2. Code collaboration patterns and semantic roles dynamic recovery

The Rather than try to understand the whole legacy software in one step, we study system behavior and its implementation based on individual system functionality. In this case, we select the most interesting part of our target tool, the functionality of graphical analysis of stock investment. We would like to find out whether these modules follow a pattern in their cooperation. Furthermore, we also want to know if these patterns recur in other system functionalities as well. Moreover, we wish to investigate which modules participate in the patterns, how they interact, what's the relationship among them and what's the role of each module. The recovery of collaborations and roles will substantially help maintainers gain better understanding of target legacy system.

Recording dynamic information. We execute the target system with instrumented code. The scenario is focused on stock performance analysis tool. We observe system behavior and functionalities within this tool. *DynamicViewer* collects the interaction information into repository, and visualizes the dynamic effects of module interactions. The scenario creates 68,123 function invocation events.

Setting pattern discovery criteria. We define the following four criteria. (1) select four out of six interaction components to observe, namely sender module, receiver module, invocated routine and direction; (2) set the deepest invocation level as 18. (3) do not ignore self-interaction. (4) omit considerations of the ordering sequence when comparing two collaboration patterns.

Automatic pattern detection with *DynamicViewer*. We use *DynamicViewer* to generate the initial fine-grained result of recovered patterns from the whole interaction serials. As shown in following figure 8, the total interaction sequence lasts for 85 visual screen frames.

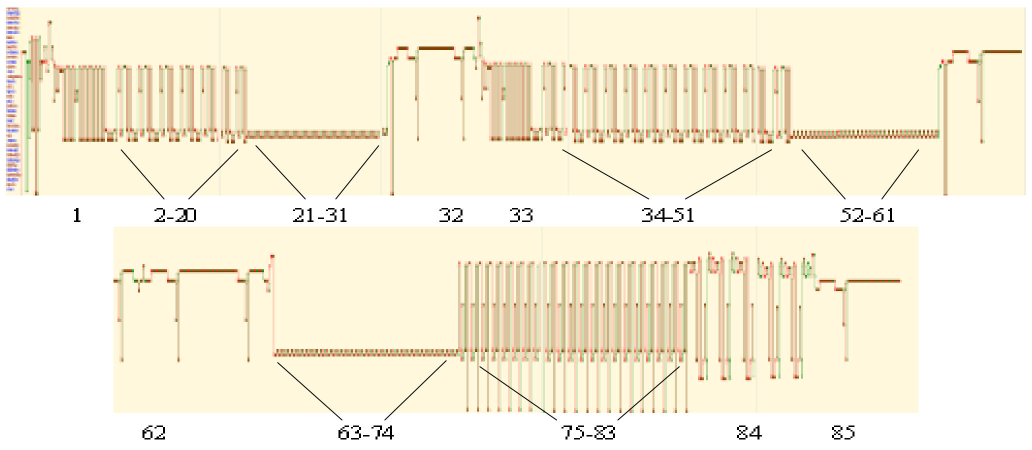


Figure 8. Automatic discovery of code collaboration patterns. (The number indicates frame id)

Early result indicates only three important patterns contribute almost the whole of this system functionality. Based on frame number, we can see that one lies on "from 2 to 20" plus "from 34 to 51" frames; another one lies on "from 21 to 31" plus "from 52 to 61" plus "from 63 to 74" frames and the last one lie on "from 75 to 83" frames respectively. These three major collaboration patterns account for over 90% of the interaction frames. More significantly, the participants are limited to less than fifteen modules. This result shows that, although the "Stock analysis tool" has very complex system functionalities and various dynamic behaviors, with the help of collaboration pattern analysis, maintainers can quickly focus on studying several important patterns to understand the whole implementation.

Deep study with *Collaboration-Investigator*. With the help of *Collaboration-Investigator*, further analysis will help maintainers to study the details of collaboration patterns and the semantic relations (roles) among the participants inside of each single pattern. We use *collaboration-investigator* to recover all the patterns that satisfy discovery criteria which have been settled previously. A total of nine patterns have been recovered. They are listed in table 1. The name of each collaboration pattern is composed of its identity number, the first module's name and the first routine name that invoked by that module. We notice that some patterns consist of several smaller patterns, and some patterns' appearance may not be contiguous. These kinds of patterns are not easy to find by only using *DynamicViewer*.

Recovered collaboration patterns
#01_account.c : ac_init_attributes
#02_import.c : parse_header_data
#03_transaction.c : trans_compare_by_date
#04_nodemenu.c : nodemenu_new
#05_menubar.c : analyze_flags
#06_configuration.c : clear_autoload
#07_intranslist.c : in_trans_list_set_account
#08_dataset.c : ds_show
#09_intransbut.c : in_trans_but_init

Table 1. List of recovered patterns

To answer the questions and verify the hypotheses in the previous section, we now select one pattern to study in detail. The No.7 pattern has one of the most significant repetitive characteristics among all the recovered patterns. It has 31 out of 85 visual screen frames with a nested pattern No.3. The following interaction fraction shows the retrieved components and collaboration instances from the *Collaboration-Investigator* tool. From the composition relationships illustrated in the segment, we can find that when module Intranslist.c sets transaction accounts, it in fact fills in the transaction list. To accomplish this job, it first asks module Translist.c to reset transaction list. Then it delegates the whole task to module Transaction.c. The latter one deals with the details of comparing and compiling work with the support from module Transarray.c.

```
          |  Intranslist.c : in_trans_list_set_account : in
          |    Intranslist.c : fill_translist : in
          |       Translist.c : reset_translist : in
          |       Ttranslist.c : reset_translist : out
          |       Transaction.c : tarray_compile : in
          |         |  Transaction.c : compare_by_date : in
Pattern 7 |         |     Transarray.c : trans_compare_by_date : in
          |  Pattern 3    Transarray.c : trans_compare_by_date : out
          |         |  Transaction.c : compare_by_date : out
          |            ... ..   {repeat for many frames}    ... ...
          |        Transaction.c : tarray_compile : out
          |    Intranslist.c : fill_translist : out
          |  Intranslist.c : in_trans_list_set_account : out
```

Figure 9. Details of automatically recovered code collaboration patterns

Now we further wish to know what is the specific semantic role of each module inside of pattern 7 and pattern 3. This time, we use *Collaboration-Investigator* tool to inspect the role of each module. As described in previous section, there are two role stereotypes defined in the tool, namely “Director-manager-worker” role type and “Consumer-supplier” role type respectively. Based on the nesting information, each module in the pattern will be assigned at least one of these two role stereotypes. Maintainer is prompted to add more detailed observation roles to specific module that she is interested in.

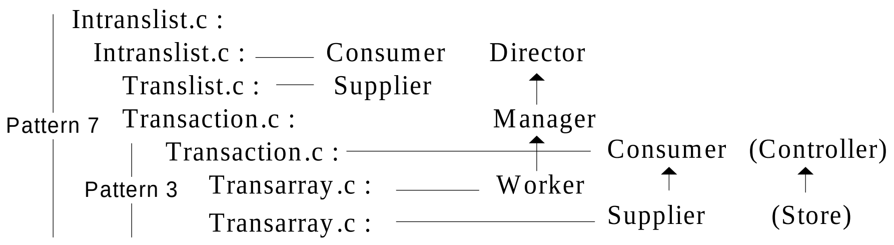


Figure 10. Module conceptual semantic role recovery

In this case, there are three groups of semantic roles assigned to modules, as shown in figure 9 and 10. For collaboration pattern 3, maintainer may find that this consumer-supplier role relationship can also be described as “Controller-store (data)”. Further investigation shows that module Intranslist.c belongs to root directory “/src/”, whereas both modules “Transaction.c” and “Transarray.c” belong to subdirectory “/src/base/”. From the role relationship between these three modules, maintainers can also figure out the same relationships between the underlying directories. Therefore, this confirms our first hypothesis that modules from subdirectory are service providers for the modules from the root directory.

To better understand a single module's function in the scenario, maintainer can choose a single module, then query all the roles it plays inside of any collaboration pattern, therefore to get a broad understanding of what kind of role that module has in general. In our case, module Transarray.c works as "worker" and "supplier". These give maintainers a strong suggestion that this module implements a certain system constructional concept, which means "labor", who does the real job, and contributes to those who dispatches tasks to others.

Within a single collaboration pattern, we can also use *DynamicViewer* to analysis two modules which have strong cooperation, especially when these two modules have a certain type of role relationships inside of that pattern. See figure 10, the vertical axis represents the number of invocation times; the horizontal direction represents the invocation depths.

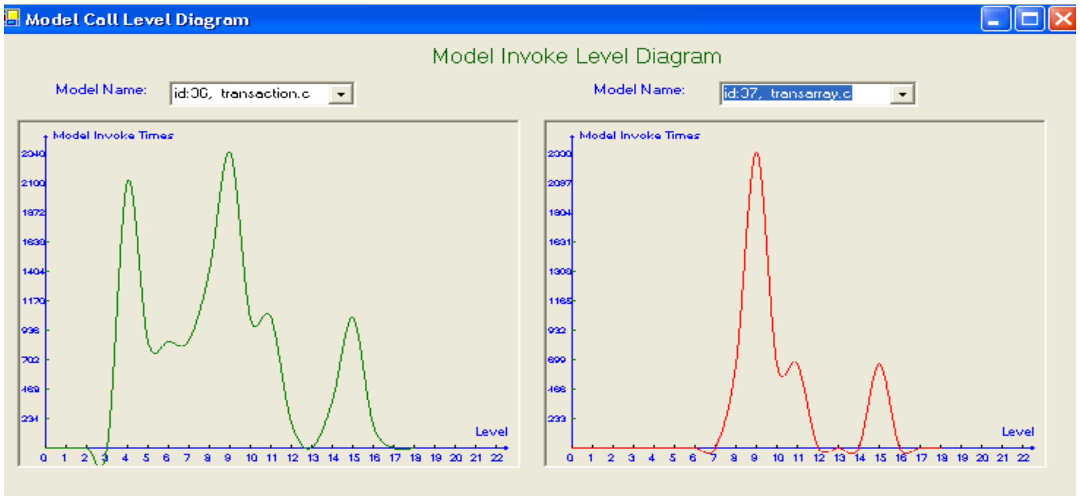


Figure 10. Analyzing two modules in collaboration: Transaction (left) and Transarray (right)

Two modules will be compared with the density of their interaction instances, the depth level, the activity frequency and time period. Figure shows the comparison of module Transaction.c and Transarray.c. We can find that starting from depth 7, they have same fluctuation pattern of invocation frequency and calling instances. This is confirmed by our previous collaboration pattern analysis, since these two modules cooperate together to form collaboration pattern 3, and this pattern occupies a relatively large segment in the whole interaction serials (31 out of 85 observing frames). This fact suggests that both modules have tight coupling with each other after depth level 7. But from depth 1 to 7, module Transarray has no interaction instances, while module Transaction is still active from depth 4 to 6. This indicates that, the module Transaction not only cooperates with Transarray, but also participates in other activities, maintainer can further use our tools to trace it; while from depth level 7, the rest part of module Transaction only faithfully cooperates with module Transarray. From this analysis result, maintainers can further build up a more clear comprehension model of target analyzing code artifacts.

5. Discussion

In this case study, we've demonstrated how to apply our approach, with the help of tools support, to automate the recovery process of collaboration patterns from stock analysis software. We also use these tools to discover the semantic roles of modules inside of patterns, therefore to get a deeper comprehension of the source code. Each recovered collaboration pattern represents a concrete implementation block of the observed system functionality. By characterizing such kind of program construction, we gain a better understanding of how the system behavior is carried out through the interactions. This will largely facilitate maintainers' cognition process in understanding target software system.

It is also very useful for us to apply the discovered collaboration patterns to a further decomposition of the whole system into a role-based hierarchical representation. It will help maintainer rapidly acquire the desired general comprehension of target system. Maintainer can use this information to study each module within various collaboration patterns, thus to regain more detailed source code modularization information. Within a collaboration pattern, its composition modules intensively cooperate together to perform a concrete task inside of the system functionality. This type of cooperation represents a highly cohesive source code unit. Many object identification research works agree that highly cohesive program parts normally are primary candidates for object structures [6][7][18]. Cohesive measurement is widely used to perform legacy re-construction [8][9][17]. Therefore, collaboration pattern recovery can be further used in the re-modularization of legacy system.

6. RELATED WORK

Our work on recovering collaboration and role from legacy software is a part of research work for legacy migration [11][12]. The recovery of collaborations provides us with a decomposition view of legacy software. Most work on understanding interactions has focused on visualization techniques, where the challenge is to develop efficient way to visualize the large amount of dynamic information [13][14]. The work of DePauw et al. [15], now integrated with Jinsight, allows engineer to visually recognize patterns in the interactions of classes and objects. ISVis [16] displays interaction diagrams using a mural technique and also provides pattern matching. Our work in the visualization part is similar to these two approaches. Contrary to the merely focus on visualization, our approach emphasizes more on the recovery of collaboration and the understanding of roles. Tamar et al. also propose an approach to analysis role within collaborations [10], but they purely use the invocation methods as representative of roles. This is not sufficient in our research to analyze the general function of a module inside of recovered collaboration pattern. We use predefined conceptual role stereotypes for the recovery of role based on the invocation relations with other modules. This gives maintainers a better understanding of how the modules cooperate within a collaboration pattern.

7. CONCLUSIONS

In this paper, we have presented an approach to recover collaboration patterns and semantic roles from legacy system for the purpose of legacy software understanding. It consists of two major parts, both of which are supported by software toolsets that we have developed. The first part focuses on the dynamic analysis of target legacy systems and the automatic discovery of collaboration patterns. The second part is concentrated on the recovery and analysis of collaboration patterns and semantic roles with human intervention. We have illustrated our approach through the analysis of a sample legacy system in a case study, in which we use our software tools to automate the discovery of collaborations patterns and the semantic roles of the participants. The initial experience shows the result is very promising. Our approach has demonstrated the feasibility and utility of using dynamic information to recover and analyze collaborations and semantic roles from code, hence to support legacy software understanding. We are continuing this work by improving the analysis functions of our tools.

REFERENCES

[1] Anquetil, N.,Lethbridge, T.C. "Recovering software architecture from the names of source files", Journal of Software Maintenance: Research and Practice, 11, 1999, pp. 201-221.

[2] Arun Lakhotia. "A unified framework for expressing software subsystem classification techniques". Journal of Systems and Software, 1997, mar, 36, pp.211—231.

[3] S. Palthepu, J. Greer, and G. McCalla. "Cliche recognition in legacy software: A scalable, knowledge-based approach". IEEE Working Conference on Reverse Engineering, IEEE Comp. Soc. Press, Oct 1997. pp. 94—103

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

[5] Houari A. Sahraoui, Hakim Lounis, Walcelio Melo, Hafedh Mili, "A concept formation based approach to object identification in procedural code", Automated Software Engineering Journal, 1999.

[6] 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, 2000. vol. 54, pp. 99-110

[7] A. van Deursen, L. Moonen. "Exploring legacy systems using types". Proceedings 7th Working Conference on Reverse Engineering, IEEE Comp. Soc, 2000, pp 32-41.

[8] Tobias Kuipers and Leon Moonen. "Types and concept analysis for legacy systems". Proceedings of the International Workshop on Programming Comprehension (IWPC 2000). IEEE Computer Society, 2000. June.

[9] Tamar Richner and Stephane Ducasse. "Using dynamic information for the iterative recovery of collaborations and roles" Proceedings of International Conference of Software Maintenance, IEEE Computer Society, 2002, pp 34-43.

[10] L.Wu, H. Sahraoui, P. Valtchev, "Legacy design recovery with dynamic visualization". Proceedings of the 16th International Conference Software & Systems Engineering and their Applications, 2003

[11] L.Wu, H. Sahraoui, P. Valtchev, "Migrating legacy software towards new technologies", Proceedings of the Migration and Evolvability of Long-life Software Systems Workshop, NetObjectDays, 2003.

[12] T.Systa,K.Koskimies and H.Muller. "Shimba – an environment for reverse engineering java software systems." Software –Practice and Experience, 1(1), January 2001.

[13] 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.

[14] 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

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

[16] F.P. Coyle "Legacy Integration Changing Perspectives", IEEE Software, IEEE Computer Soc. Vol. 17 No. 2, March/April Press, pp. 37-41. 2000

[17] 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

[18] K.H.Bennett, "Legacy Systems: Coping With Success", IEEE Software, January, Vol 12, No.11 pp 19-23. 1995

[19] Okun, V.; Gaucher, R.; & Black, P. E., eds. Static Analysis Tool Exposition (SATE) 2008 (NIST Special Publication 500-279). NIST, 2009.

[20] Dannenberg, Roger B.; Dormann, Will; Keaton, David; Seacord, Robert C.; Svoboda, David; Volkovitsky, Alex; Wilson, Timothy; & Plum, Thomas. "As-If Infinitely Ranged Integer Model," 91-100. Proceedings 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE 2010), San Jose, CA, Nov. 2010. IEEE 2010.

[21] Heffley, J. & Meunier, P. "Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?" Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9. Island of Hawaii, January 2004. IEEE Computer Society, 2004.

[22] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, "Recovering Traceability Links between Code and Documentation," IEEE Transactions on Software Engineering, 28(10):970–983, 2002.

[23] G. Antoniol, Y.-G. Guéhéneuc, "Feature Identification: A Novel Approach and a Case Study," Intl. Conf. on Software Maintenance (ICSM), 2006.

[24] T. J. Biggerstaff, B. G. Mitbander, D. Webster, "The concept assignment problem in program understanding," Intl. Conf. on Software Engineering (ICSE), 1993.

[25] B. Boehm, "Software engineering," IEEE Transactions on Computers, C-25(12):1226–1241, 1976. [6] M. Bruntink, A. v. Deursen, R. v. Engelen, T. Tourwe, "An evaluation of clone detection techniques for identifying cross-cutting concerns," Intl. Conf. on Software Maintenance (ICSM), 2004.

[26] G. Canfora, L. Cerulo, M. D. Penta, "On the Use of Line Co-change for Identifying Crosscutting Concern Code," Intl. Conf. on Software Maintenance (ICSM), 2006.

[27] K. Chen, V. Rajlich, "Case study of feature location using dependence graph," Intl. Wkshp. on Program Comprehension (IWPC), 2000.

[28] M. Eaddy, A. Aho, G. C. Murphy, "Identifying, Assigning, and Quantifying Crosscutting Concerns," Wkshp. on Assess. of Contemp. Modularization Techniques (ACoM), 2007.

[29] T. Eisenbarth, R. Koschke, D. Simon, "Locating features in source code," IEEE Trans. on Soft. Eng., 29:210–224, 2003.

[30] A. D. Eisenberg, K. De Volder, "Dynamic Feature Traces: Finding Features in Unfamiliar Code," Intl. Conf. on Software Maintenance (ICSM), 2005..

[31] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, "Advanced Candidate Link Generation for Requirements Tracing: The Study of Methods," IEEE Transactions on Software Engineering, 32(1):4–19, 2006.

[32] E. Hill, L. Pollock, K. Vijay-Shanker, "Exploring the Neighborhood with Dora to Expedite Software Maintenance," Automated Software Eng. (ASE), 2007.

[33] A. J. Ko, R. DeLine, G. Venolia, "Information Needs in Collocated Software Development Teams," Intl. Conf. on Software Engineering (ICSE), 2007.

[34] D. Liu, A. Marcus, D. Poshyvanyk, V. Rajlich, "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," Automated Software Eng.(ASE), 2007.

[35] A. Marcus, D. Poshyvanyk, R. Ferenc, "Using the Conceptual Cohesion of Classes for Fault Prediction in Object Oriented Systems," IEEE Transactions on Software Engineering, 2008.

[36] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," IEEE Transactions on Software Engineering, 33(6):420–432, 2007.

[37] W. Zhao, L. Zhang, Y. Liu, J. Sun, F. Yang, "SNIAFL: Towards a Static Noninteractive Approach to Feature Location," ACM Transactions on Soft. Eng. and Methodology, 15(2):195–226, 2006.

[38] Holger Schmidt, Jan-Patrick Elsholz, Vladimir Nikolov, Franz J. Hauck, and Rudiger Kapitza OSGi4C: Enabling OSGi for the Cloud. In Fourth International ICST Conference on OMmunication System softWAre and middlewaRE (COMSWARE '09), COMSWARE '09, Dublin, Ireland, June 2009. ACM.

[39] Marcin Solarski. Dynamic Upgrade of Distributed Software components. PhD thesis, Fakultat IV (Elektrotechnik und Informatik), Technische Universitat Berlin, 2004. ¨

[40] Marcin Solarski and Hein Meling. Towards Upgrading Actively Replicated Servers on-the-fly. In 26th Annual International Computer Software and Applications Conference (COMPSAC 2002), pages 1038–1043, 2002.

[41] N. Sridhar, S.M. Pike, and B.W. Weide. Dynamic Module Replacement in Distributed Protocols. In 23rd International Conference on Distributed Computing Systems, pages 620–627, May 2003.

[42] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. ACM Transactions on Programming Languages and Systems (TOPLAS), 29(4), August 2007.

[43] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In 15th European Conference on Object-Oriented Programming (ECOOP '01), ECOOP '01, pages 236–255. Springer-Verlag, 2001.

[44] Andre L. C. Tavares and Marco Tulio Valente. A Gentle Introduction to OSGi. SIGSOFT Software Engineering Notes, 33(5), September 2008.

[45] L.A. Tewksbury, L.E. Moser, and P.M. Melliar-Smith. Live Upgrades of CORBA Applications Using Object Replication. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), ICSM '01, pages 488–497. IEEE Computer Society, 2001.

[46] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. IEEE Transactions on Software Engineering, 33(12):856–868, December 2007.

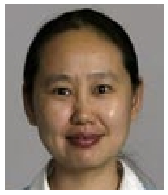
[47] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: towards a Cloud Definition. SIGCOMM Computer Communication Review, 39(1):50–55, January 2009.

Authors

Lei Wu, Assistant Professor of software engineering at University of Houston-Clear Lake, Houston, U.S.A. His major research interests include software engineering with artificial intelligence, secure service-oriented architectures, software for robotics and embedded system intelligence, game software development, and pervasive computing. He can be reached at wul@uhcl.edu

Sharon White, Associate Professor of software engineering at University of Houston-Clear Lake, Houston, U.S.A. Her research interests include domain specification languages, architecture design languages, and software architecture. She can be reached at whites@uhcl.edu

Yi Feng, Associate Professor of computer science, Algoma University, Sault Ste. Marie, Canada. Her major research interests include formal verification, software engineering, signal processing, and system description languages. She can be reached at feng@algomau.ca



James Helm, Associate Professor of software engineering at University of Houston-Clear Lake, Houston, U.S.A. His research interests include systems and software engineering, operations research, computer science, mathematics, physics, simulation, and modeling. He can be reached at helm@uhcl.edu

Nathanial Wiggins, Professor of Mathematics and Engineering, San Jacinto College, Houston, U.S.A. His major research interests include scientific computing, numerical analysis, dynamic system modeling, geometry topology, control theory and optimization, software engineering, formal verification. He can be reached at Nathanial.Wiggins@sjcd.edu