A METHOD FOR REVERSE ENGINEERING OF USE CASE REALIZATIONS IN UML

Dragan Bojic, Dusan Velasevic
University of Belgrade, Faculty of Electrical Engineering
Bulevar revolucije 73, 11000 Belgrade, Yugoslavia
{bojic,velasevic}@buef31.etf.bg.ac.yu

ABSTRACT

We propose a novel method for recovering certain elements of the UML model of a software system. These include relationships between use cases as well as class roles in collaborations that realize each use case, identifying common functionality and thus establishing a hierarchical view of the model. The method is based on dynamic analysis of the system for the selected test cases that cover relevant use cases. The theory of formal concept analysis is applied to obtain classification of model elements, obtained by a static analysis of code, in terms of use case realizations.

Keywords
Object-oriented systems, UML modeling, reverse engineering, use cases

INTRODUCTION

A support for reverse engineering a system model from its code has become a standard part of modem analysis and design tools to satisfy the needs for round-trip engineering, updating a model to the evolved code, and taking the legacy context into account. Our method is devised to supplement a model obtained by a typical static analyzer. Such models lack the support for reverse engineering of the use case architectural view. Therefore, we have decided to address this issue by recovering use case realizations, which serve as a basis for classifying model elements in a hierarchically nested package structure that is normally a part of a design model as prescribed by Jacobson (1999).
We have adopted the approach to conduct reverse engineering guided by use cases (i.e. scenarios of system use). A use case is a means of describing system functionality, and represents the collection of interactions between the system and its environment or between various parts of the system for some specific use of the system, Booch (1998). Jarke (1998, 1999) reports a number of advantages of a scenario driven development or reengineering process:

- Use cases are easily comprehensible for users as well as developers so they represent a good communication medium.
- Design efforts are focused on the most important functionality of the old/changed system before the detailed design.
- Design decomposition is task-oriented, enabling the exploration of design alternatives, iterative development and reusability of design knowledge.

While code reuse is certainly an attractive option to software developers, reuse of the most general specifications, i.e. use cases and interaction diagrams, will prove to be more effective. In particular, Blok (1998) states that reuse of requirement specifications can lead to a significant decrease in development time and cost, as it usually leads to the reuse of work-products further down-stream in the development process. Specifically, there exist techniques for storing and retrieving reusable UML use case specifications to and from a large collection of UML design components, and also, for automatically generating a framework design model from UML use case diagrams, class diagrams and sequence diagrams, Kim (1998). Clearly there is a need to have a use case view of the system model.
The presented method takes into account the following practical considerations:

- The cost of understanding and applying it must be much less that the cost of manually performing the recovery. User is required to have priory knowledge of the system functionality only, to define relevant use cases and corresponding test cases that execute those use cases.
- The method uses only those kinds of data about the existing system, its structure and behavior, which is readily available in most contemporary commercial development tools, thus the cost of implementing it on various platforms is not very high. In fact, to construct a prototype tool to support our method, we only had to adapt the output of Microsoft Visual C++ profiler to ConImp concept analysis tool, Burmeister (1998), and update the model extracted by Rational Rose design tool.

THE PROPOSED METHOD

There are several activities to perform when applying the proposed method:

1. Identify a set of use cases for a given system or a part of it.
2. For each use case, define one or more test cases that cover relevant use case.
3. Collect profiling information for each test case.
4. Construct the context relation between use cases and covered system code entities.
5. Construct the conceptual lattice (i.e. acyclic digraph) from a context relation by applying formal concept analysis. Informally, each concept is a pair of two components: a set of use cases and a set of system code entities that is a shared part of realization of these use cases. If an edge connects two concepts in the lattice, then the parent concept represents "broader" functionality that in some way "uses" the functionality of the child concept.
6. Estimate the quality of decomposition from the topology of the lattice, and if necessary, modify the test cases or apply some purification of the context relation, to eliminate the effects of interleaving functionality in code and other undesired effects, and repeat activities necessary to obtain the updated lattice.
7. Construct the basic UML model using a static analyzer and decompose the model using the concept lattice. Such analyzers have become an integral part of most general-purpose OOAD tools to support round trip engineering.

These activities are further elaborated in this chapter. In the next section we present the results obtained by applying the method to a small sample application.

Using a prototype tool on a small example

We have constructed a prototype tool URCA that supports the proposed method and uses four other tools. The C++ profiler from Microsoft Visual Studio environment is used for data gathering, Rational Rose for extracting initial UML model of the system and also for presentation of the resulting decomposition. ConImp concept analysis tool is used for constructing a concept lattice, and RistanCASE Development Assistant for C (DA-C) for viewing the lattice. The data flow graph of the system is represented in Figure 1. Data are exchanged through textual files, except for the communication between Visual Studio and other tools that is based on automation. Tools other than URCA dictate file formats. The components of URCA are Test Recorder application written in Visual Basic, and ca_input, rose_input and dac_input command line applications written in C++. Recently we replaced ConImp tool with our own implementation of concept analysis algorithm to overcome some ConImp's limitations. A batch process activates the latter three applications once test recording is done to prepare data for other tools.

We shall now present the use of the prototype tool to analyze a sample application – Scribble step 3 tutorial application from Microsoft Developer Network Library (Figure 2). It is an MDI application that supports elementary drawing, which can be saved and subsequently retrieved. It has a total of 34 classes and 401 member and non-member functions. MFC framework classes were excluded from the analysis. That leaves 11 user defined classes and 121 functions. We defined the following use cases that roughly corresponds to items from the main menu:

U1. Working with Documents (generalization of U2 and U3)
U2. Creating, Opening and Saving
U3. Printing
U4. Working with Multiple Windows
U5. Drawing
U6. Configuring
U7. Help

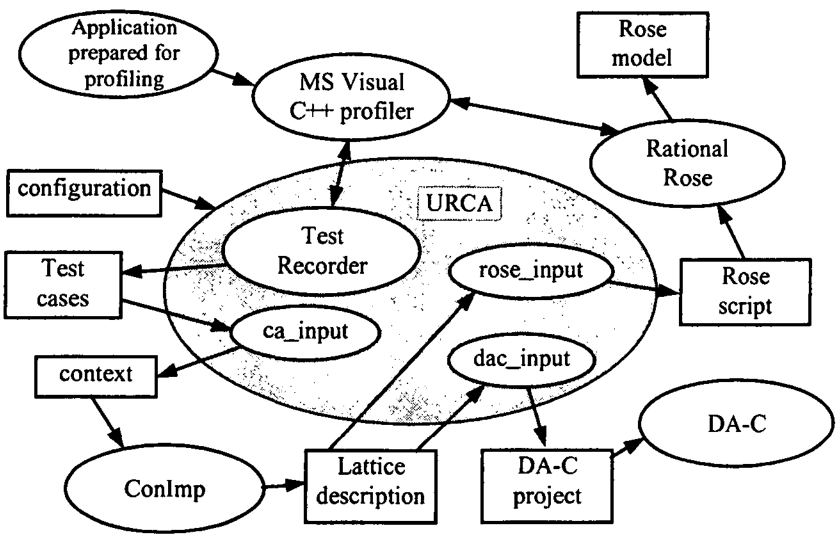


Figure 1. Structure of the prototype system

We defined twelve test cases to cover these use cases (we subtracted execution counts of the Application Start and Exit test case from the profiling data for other test cases).

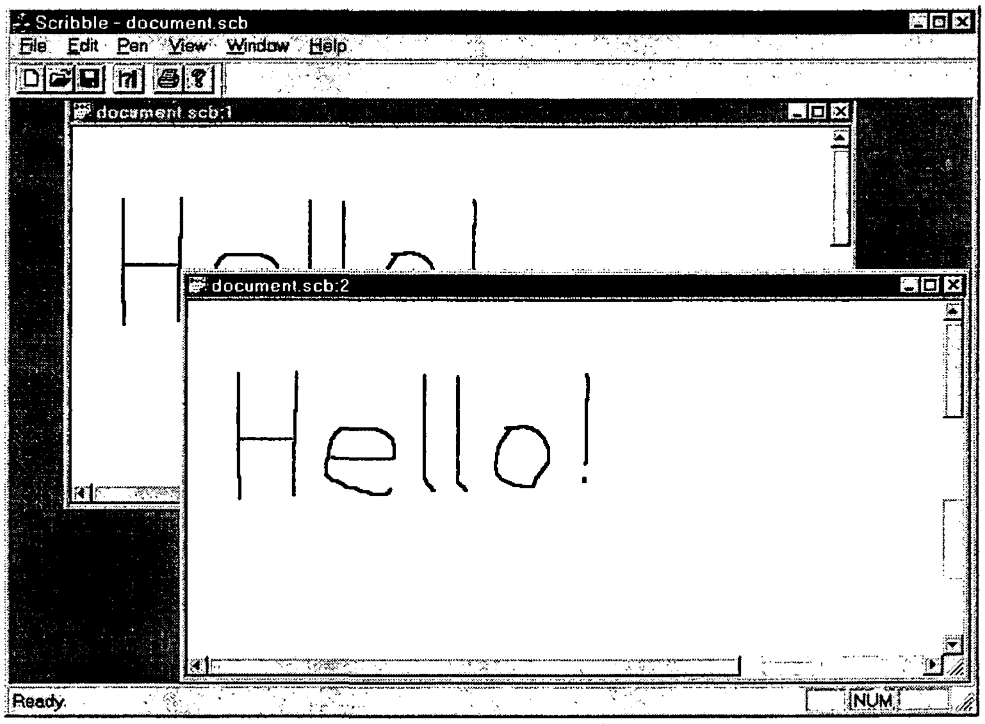
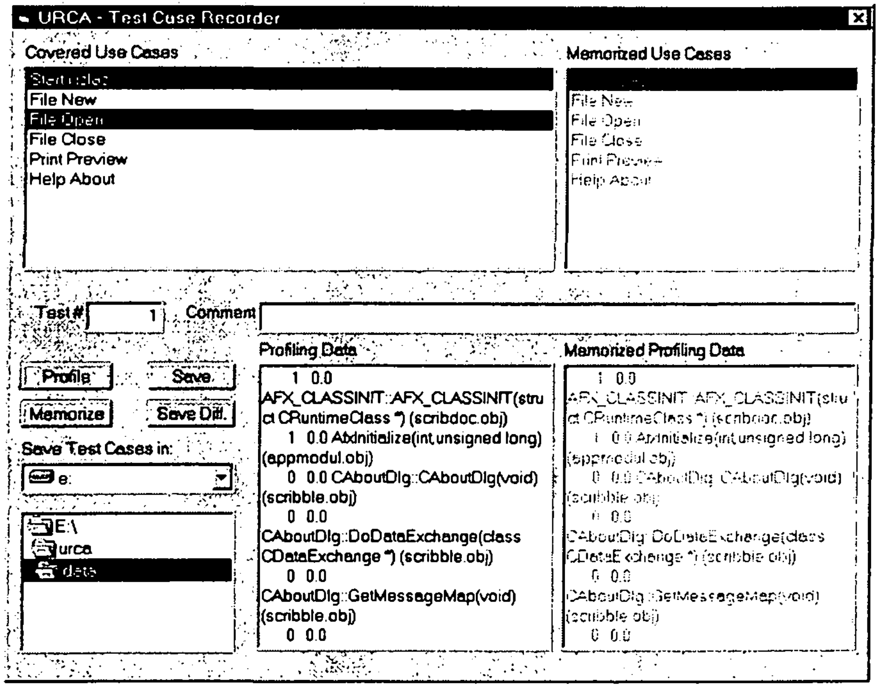


Figure 2. Scribble sample application screenshot

Figure 3 shows a Test Recorder application screen of our prototype recovery system. This application is used to activate a profiling session in the Visual Studio environment for each test case and to collect profiling data (the number of executions of each function) using the automation interface. The user then selects use cases covered from a list of all use cases. This information and profiling data can be either saved to a file or memorized internally for test differing. A difference between previously memorized data and current data can also be saved to a file as a separate test case.

For example, the first test case was to Start the application, select File New from the menu, and then exit. It covered U1 and U2. On the basis of coverage data, a so-called context matrix is computed by the ca_input application. This input is presented to the ConImp tool, which calculates the resulting description of the corresponding concept lattice. A concept lattice can be viewed using the Development Assistant for C tool (Figure 4). URCA dac_input application was used to create a DA-C project from the concept description file. The lattice is represented in the form of C source code, each concept is a C function that calls functions for the successor concepts. The concept lattice, i.e. call hierarchy graph, for our example is shown in Figure 5. The information content of each concept (its use cases and functions) is present in the form of comments and can also be examined in DA-C. Generalized use cases are marked Δ, externally visible use cases (which are directly covered with test cases) bear no special mark, and included use cases (a common functionality revealed by a concept analysis) are marked #. In Figure 5, included use cases were named after common functions that are contained in the corresponding concepts.



Covered Use Cases:

```
Start ...
File New
File Open
File Close
Print Preview
Help About
```

Memorized Use Cases:

```
File New
File Open
File Close
Print Preview
Help About
```

Test# 1

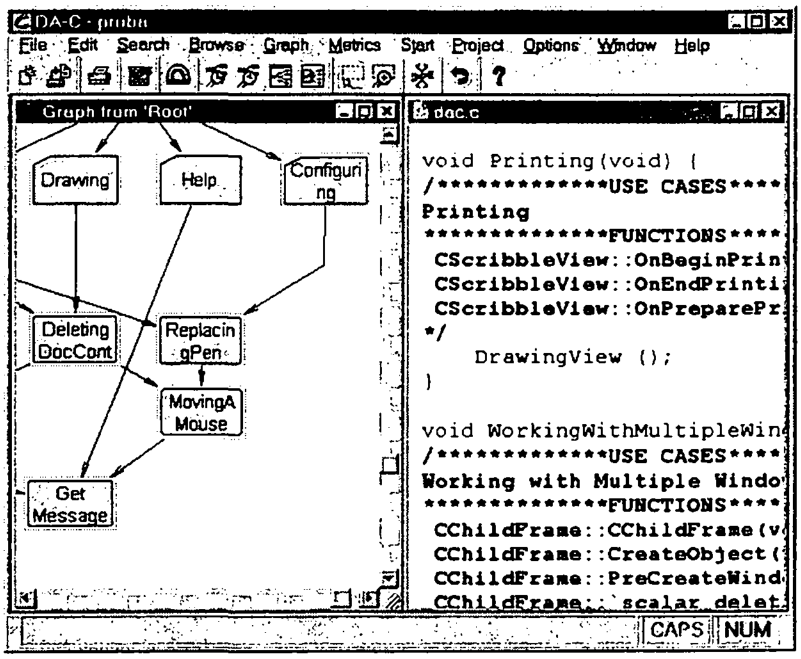
Profiling Data:

```
    1  0.0
AFX_CLASSINIT::AFX_CLASSINIT(stru
ct CRuntimeClass *) (scribdoc.obj)
    1  0.0 AfxInitialize(int,unsigned long)
(appmodul.obj)
    0  0.0 CAboutDlg::CAboutDlg(void)
(scribble.obj)
    0  0.0
CAboutDlg::DoDataExchange(class
CDataExchange *) (scribble.obj)
    0  0.0
CAboutDlg::GetMessageMap(void)
(scribble.obj)
    0  0.0
```

Memorized Profiling Data:

```
    1  0.0
AFX_CLASSINIT::AFX_CLASSINIT(stru
ct CRuntimeClass *) (scribdoc.obj)
    1  0.0 AfxInitialize(int,unsigned long)
(appmodul.obj)
    0  0.0 CAboutDlg::CAboutDlg(void)
(scribble.obj)
    0  0.0
CAboutDlg::DoDataExchange(class
CDataExchange *) (scribble.obj)
    0  0.0
CAboutDlg::GetMessageMap(void)
(scribble.obj)
    0  0.0
```

Figure 3: Test Case Recorder Application



```
void Printing(void) {
/**************USE CASES****
Printing
***************FUNCTIONS****
 CScribbleView::OnBeginPrin
 CScribbleView::OnEndPrinti
 CScribbleView::OnPreparePr
*/
    DrawingView ();
}

void WorkingWithMultipleWin
/**************USE CASES****
Working with Multiple Windo
***************FUNCTIONS****
 CChildFrame::CChildFrame(v
 CChildFrame::CreateObject(
 CChildFrame::PreCreateWind
 CChildFrame::`scalar delet
```

Figure 4: DA-C Application Screenshot

Using the Rational Rose 98i Visual C++ add-in operating with Visual C++ fuzzy parser, we have created a reverse engineered model of the Scribble application. The URCA rose_input application uses a concept lattice description to create a Rose script file (Figure 6) for updating the reverse engineered UML model of the Scribble application. The updated model contains a package hierarchy as a logical view corresponding to a concept lattice.
Figures 7–10 present the partial logical structure of the model. For each concept, there is one logical package. A

lattice determines the containment relationship between packages. The package that corresponds to some concept U contains or references only those parts of classes relevant to the realization of the use case U, which are not already present in contained packages. Those parts are shown in the pictures. A static analyzer that is a part of Rose induced relationships between classes.
Now we shall discuss the decomposition process in detail.

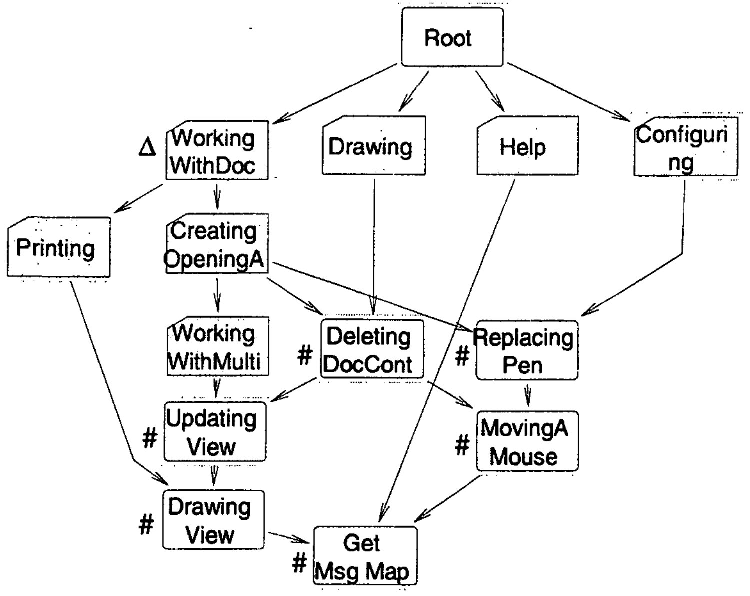


Figure 5: The concept lattice for Scribble

```
Sub Main
Dim C0 As Category
Dim C1 As Category
Dim C2 As Category
. . .
Dim Dummy As ClassDiagram
Dim aClass As Class
Set C0 = RoseApp.CurrentModel.RootCategory
Set C1 = C0.AddCategory("Help ")
Set Dummy = C1.AddClassDiagram("Main")
d = C0.ClassDiagrams.GetFirst("Main").AddCategory(C1)
Set C2 = C0.AddCategory("Drawing ")
Set Dummy = C2.AddClassDiagram("Main")
d = C0.ClassDiagrams.GetFirst("Main").AddCategory(C2)
Set C3 = C0.AddCategory("Working with Documents ")
Set Dummy = C3.AddClassDiagram("Main")
```

Figure 6: Part of Rose script generated by URCA

Identifying Use Cases

Each use case represents one functional requirement. For the reasons of efficiency for large systems, we are interested in those use cases that are "architecturally significant" as defined by Jacobson (1999), so that separate use cases should not be used for too elementary system functions. Relationships between use cases, as defined by Booch (1998), are not introduced in this step. They will be determined automatically in the next phases.

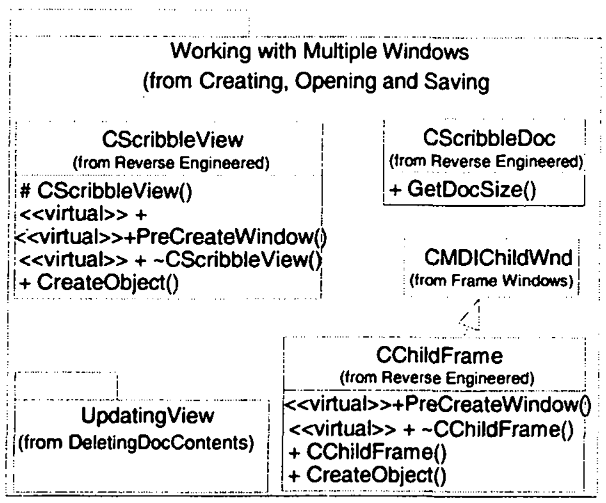

Working with Multiple Windows
(from Creating, Opening and Saving
CScribbleView
(from Reverse Engineered)
# CScribbleView()
<<virtual>> +
<<virtual>>+PreCreateWindow()
<<virtual>> + ~CScribbleView()
+ CreateObject()
CScribbleDoc
(from Reverse Engineered)
+ GetDocSize()
CMDIChildWnd
(from Frame Windows)
CChildFrame
(from Reverse Engineered)
<<virtual>>+PreCreateWindow()
<<virtual>> + ~CChildFrame()
+ CChildFrame()
+ CreateObject()
UpdatingView
(from DeletingDocContents)

Figure 7: Working With Multiple Windows package

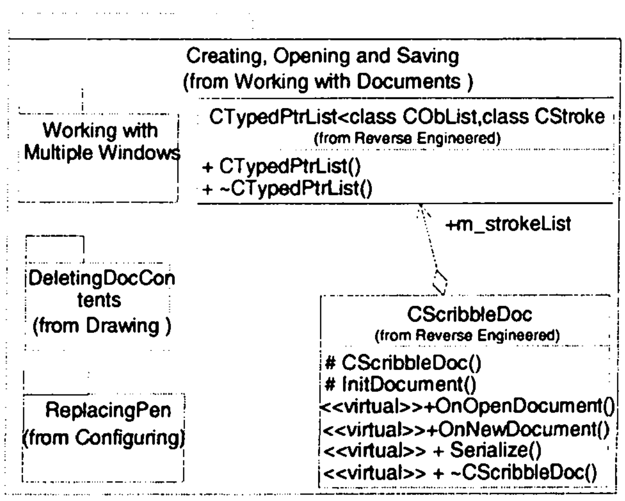

Creating, Opening and Saving
(from Working with Documents)
Working with Multiple Windows
CTypedPtrList<class CObList,class CStroke
(from Reverse Engineered)
+ CTypedPtrList()
+ ~CTypedPtrList()
+m_strokeList
DeletingDocContents
(from Drawing)
CScribbleDoc
(from Reverse Engineered)
# CScribbleDoc()
# InitDocument()
<<virtual>>+OnOpenDocument()
<<virtual>>+OnNewDocument()
<<virtual>> + Serialize()
<<virtual>> + ~CScribbleDoc()
ReplacingPen
(from Configuring)

Figure 8: Creating, Opening and Saving package

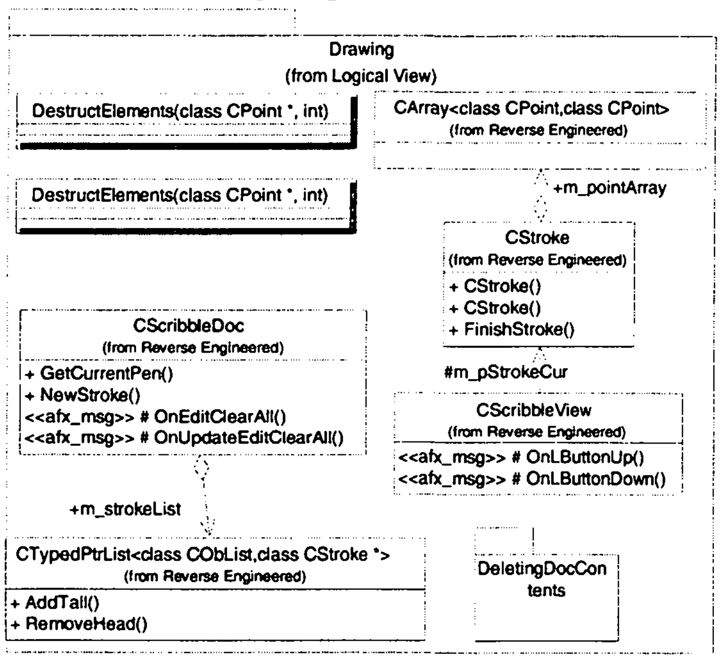

Drawing
(from Logical View)
DestructElements(class CPoint *, int)
CArray<class CPoint,class CPoint>
(from Reverse Engineered)
DestructElements(class CPoint *, int)
+m_pointArray
CStroke
(from Reverse Engineered)
+ CStroke()
+ CStroke()
+ FinishStroke()
CScribbleDoc
(from Reverse Engineered)
+ GetCurrentPen()
+ NewStroke()
<<afx_msg>> # OnEditClearAll()
<<afx_msg>> # OnUpdateEditClearAll()
#m_pStrokeCur
CScribbleView
(from Reverse Engineered)
<<afx_msg>> # OnLButtonUp()
<<afx_msg>> # OnLButtonDown()
+m_strokeList
CTypedPtrList<class CObList,class CStroke *>
(from Reverse Engineered)
+ AddTail()
+ RemoveHead()
DeletingDocContents

Figure 9: Drawing package

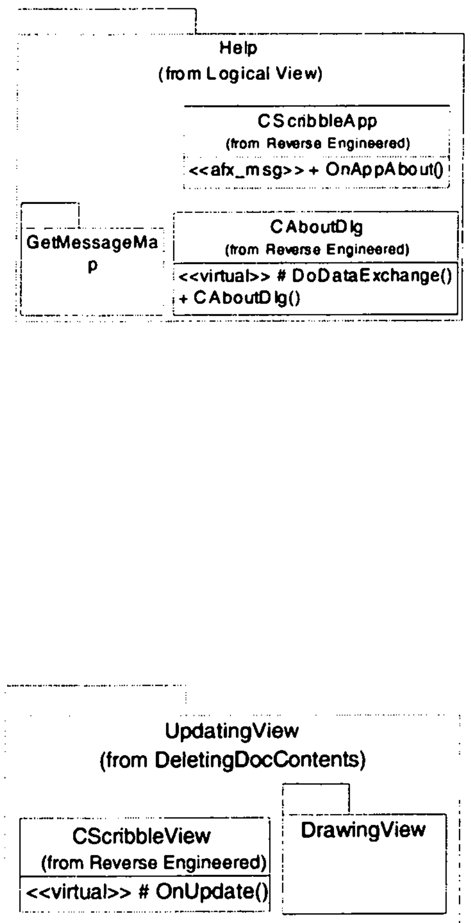
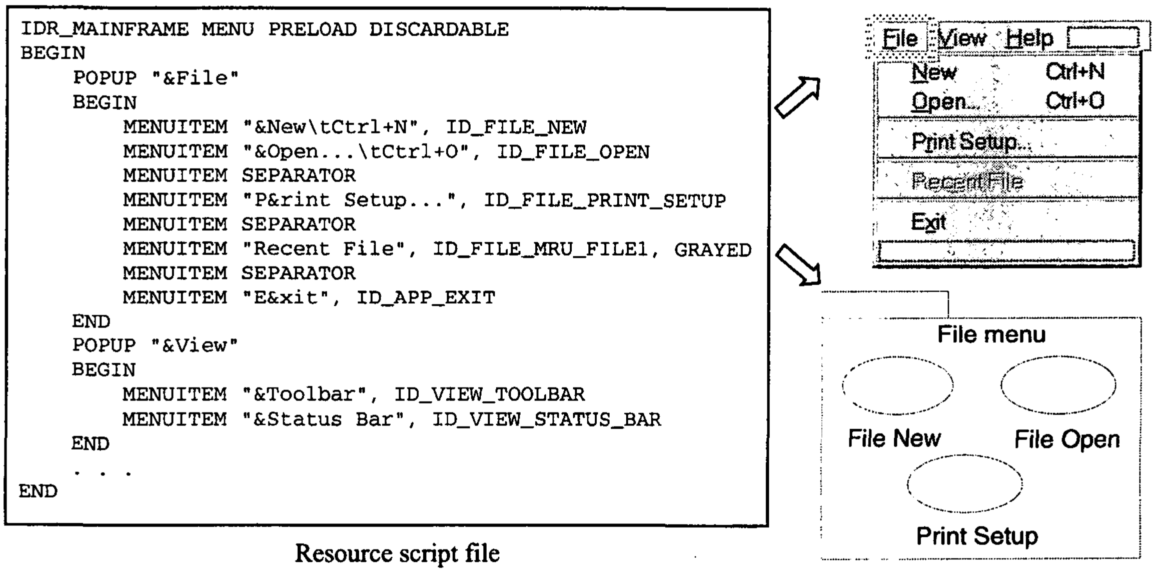

Help
(from Logical View)
CScribbleApp
(from Reverse Engineered)
<<afx_msg>> + OnAppAbout()
GetMessageMap
CAboutDlg
(from Reverse Engineered)
<<virtual>> # DoDataExchange()
+ CAboutDlg()
UpdatingView
(from DeletingDocContents)
CScribbleView
(from Reverse Engineered)
<<virtual>> # OnUpdate()
DrawingView

Figure 10: Help and Updating View packages

Use case identification can be done manually, in a systematic manner similar to one described in Noffsinger (1998). For example, for Graphical User Interface applications, each menu item, toolbar button and other items that initiate some function can be assigned one use case. This procedure can even be automated by analyzing resource file that contains definitions of such items (Figure 11). Resource descriptions can be extracted even from executable file for Windows applications.

Selecting and Executing Test Cases

A test case specifies one way of testing the system, including what to test with which input or result, and under which conditions to test. Each test case can be derived from, and is traceable to a use case or a use case realization, to verify the result of the interaction between the actors and the system or between various components of the system – Jacobson (1999).
In our case, the purpose of executing tests is not to verify system behavior, but to collect profiling data. For each use case defined in the previous step, we devise one or more scenarios of system use. A particular scenario can possibly exercise more than one use case, if they are indirectly related, but the exact kind of relationship need not be specified. Each test case specifies how to test some specific scenario from the set of devised scenarios. When we obtain the initial architectural model, we can add more test cases that concentrate on some specific use case realization, to refine the model.
The above methodology is related to high-level functional testing as described in Kit (1995).
Each test case can be related to more than one use case. For example, a test case can cover general editing functionality and specific text replacement functionality; or a test can cover opening test file and then previewing that file for printing.
The profiling information is then collected for each test case. We have chosen to collect executions counts at the function (class members) level (the code lines level seemed too detailed). Using execution counts, and not just coverage information enables what we called "test case differing": suppose we conduct a test case in which a file is opened in editor, and another in which a file is opened and then previewed for printing. By subtracting execution counts of the first test case from those of the second, we obtain profiling information for just the print preview use case, which can now be treated as a separate test case.



```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New\tCtrl+N", ID_FILE_NEW
        MENUITEM "&Open...\tCtrl+O", ID_FILE_OPEN
        MENUITEM SEPARATOR
        MENUITEM "P&rint Setup...", ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "Recent File", ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "E&xit", ID_APP_EXIT
    END
    POPUP "&View"
    BEGIN
        MENUITEM "&Toolbar", ID_VIEW_TOOLBAR
        MENUITEM "&Status Bar", ID_VIEW_STATUS_BAR
    END
    . . .
END
```

Figure 11: Identifying Use Cases from menus

Introduction to Formal Concept Analysis

Concept analysis provides a way to identify sensible groupings of objects that have common attributes – Burmeister (1998). A context relation indicate which object has which attributes. Figure 12 present a sample context relation for four objects $O_1 - O_4$ and eight attributes, A_1–A_8.

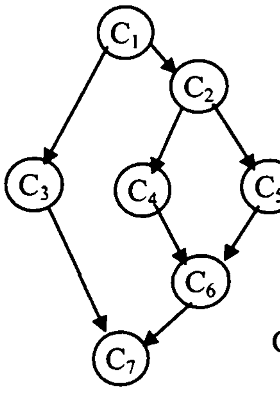
	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
O_1	X	X						
O_2			X	X	X			
O_3			X	X		X	X	X
O_4			X	X	X	X	X	X

Figure 12. A sample context relation

For any set of objects O, let ca(O) represent the set of attributes which every object in O posses. Similarly, for a set of attributes A, let co(A) represent the set of objects which posses all attributes in A. A pair (O,A) where A = ca(O) and O = co(A) is called a formal concept. Such formal concepts correspond to maximal rectangles in the context relation matrix, where of course permutations of rows or columns do not matter. For a concept c = (O, A), the first component O is called the extent and the second component A is called the intent of c.

A concept (X_1, Y_1) is a subconcept of concept (X_2, Y_2) if, and only if, X_1 is a subset of X_2 (or, equivalently, Y_2 is a subset of Y_1). The subconcept relation forms a complete partial order (the concept lattice) over the set of concepts. An algorithm to compute a concept lattice can be found in Siff (1997). The concept lattice for a sample context relation from Figure 12 is presented in Figure 13.

Let us now define the notions of object and attribute concepts. The *object concept* (E,I) that corresponds to an object O is the smallest (closest to the leaf) concept for which $O \in E$. The *attribute concept* (E,I) that corresponds to an attribute A is the largest (closest to the root) concept for which $A \in I$. Figure 14 depicts the lattice from Figure 13 annotated with this information. For example, C_5 concept is the object concept for O_3 and the attribute concept for A_6, A_7 and A_8.



$$C_1 = (\{O_1,O_2,O_3,O_4\}, \{\})$$
$$C_2 = (\{O_2,O_3,O_4\}, \{A_3,A_4\})$$
$$C_3 = (\{O_1\}, \{A_1, A_2\})$$
$$C_4 = (\{O_2,O_4\}, \{A_3,A_4,A_5\})$$
$$C_5 = (\{O_3,O_4\}, \{A_3,A_4,A_6,A_7,A_8\})$$
$$C_6 = (\{O_4\}, \{A_3,A_4,A_5,A_6,A_7,A_8\})$$
$$C_7 = (\{\}, \{A_1,A_2, A_3,A_4,A_5,A_6,A_7,A_8\})$$

Figure 13. A sample concept lattice

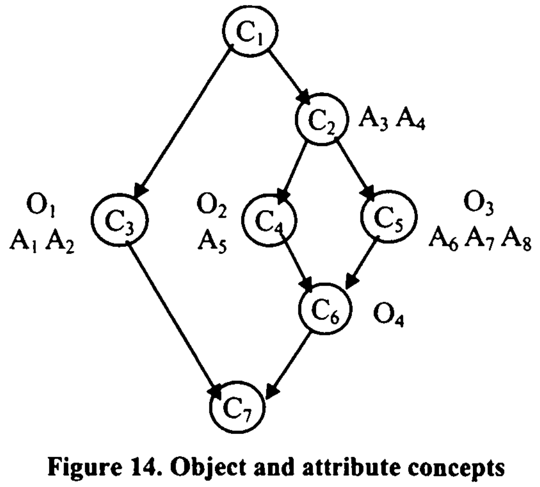


Figure 14. Object and attribute concepts

Applying Formal Concept Analysis

For our purposes, the set of objects O includes all functions, class member as well as non-member, in our application. The set of attributes A includes all use cases as defined by the user. Context relation between functions and use cases is defined with a meaning "F is related with U if, and only if, F is a part of code that implements U". Relation *implements* is established on the basis of profiling data obtained by executing test cases, and is defined as follows:

F *implements* U, if, and only if, there exist at least one test case that executes F (at least once) and covers U.

With this definition of context relation, the extent E of each concept C in a resulting concept lattice is a set of functions that are the shared part of implementation code of the set of use cases that forms the intent I of C.

In a real world situation, two problems could possibly arise with this interpretation. First, if all test cases that cover a certain use case U_1 also cover some unrelated use case U_2, than all functions that implement U_1 will be mistakenly related to U_2 also. This problem can be eliminated by carefully choosing test cases.

The second problem is code interleaving, a fact that the same function F can actually participate in implementing two unrelated use cases U_1 and U_2. The above definition works fine for this situation, relating F to both U_1 and U_2, but a resulting concept lattice mistakenly relates together U_1 and U_2. Splitting the row F of the context matrix in two or more rows, one for each of the implemented use cases, and recalculating the lattice solves this problem. A tool can perform this operation automatically, without the need for the user to examine the context relation manually, although URCA prototype does not support it yet.

The second modification of context relation is the elimination of all functions that belong to standard library or foundation classes because they are used in many different contexts. This does not apply to classes that are inherited from these standard classes to perform some specific functionality.

In this interpretation of concepts we use the term *function concept* instead of object concept, and *use case concept* instead of attribute concept.

We have chosen to modify the resulting concept lattice by eliminating all such concepts that are neither function concepts nor use case concepts for at least one function or use case. These concepts have no contribution to the model decomposition, but only clutter the lattice. Pairs of concepts in the original lattice that were connected via these eliminated concepts are connected directly to each other in the resulting lattice. Usually, if test cases covered all functions and use cases, the root and the leaf concepts of the lattice would be eliminated. The remaining concepts can be classified in the following categories (differently marked in Figure 5):

1. Use case concepts for generalized use cases (marked Δ in Figure 5). Generalized use cases are identified by the fact that they never appear alone in test cases, and that their corresponding concept is larger than the concepts of those other related use cases.
2. Use case concepts for "ordinary" use cases (without marking in Figure 5) are all other use case concepts.
3. Function concepts that are not at the same time use case concepts (marked # in Figure 5) are usually lower in the lattice than use case concepts and represent some shared functionality between different use cases.

Updating Model from the Lattice

From the concept lattice, we can determine various elements of a UML model.

A particular concept can be use case concept for zero or more use cases. In the former case, a concept represents some common functionality that is not externally visible. In the latter case, if there is more than one use case corresponding to a particular concept, further test cases are needed to differentiate between those use cases. In case of one to one correspondence, each use case concept corresponds to the collaboration (use case realization) of the corresponding use case. Extents of these concepts define analysis classes (parts of real, implementation classes that take role in the realization of relevant use cases).

For every concept in a lattice, calculated by the rules described in the previous section, we introduce one package in a logical view (more precisely, in a distinct sub-view of this view). The relation between concepts in the lattice corresponds to the containment relation between packages. If there is a concept in the lattice with more than one parent concept, a corresponding package will be owned by one parent package and referenced from others. Packages that correspond to use case concepts are named after the relevant use cases.

Each non-member function belongs to the package that corresponds to its function concept. In case of classes, there could be more than one function concept for its member functions. In that case, we assign the class to the package that corresponds to the concept that is function concept to most of its functions, and reference the class from other packages.

It is well known that the use case and the logical view of a UML model are related with each other by interaction diagrams describing particular use cases, showing interaction (function member calls) of instances of classes from the logical view that are implementing these use cases.

Having only the information obtained from static analysis and execution counts, one cannot determine exact interaction diagrams but only model elements relevant for the implementation of each use case. They all belong to the package that corresponds to use case concept for a particular use case.

The ordering relation on the set of concepts can be used to determine relationships among particular use cases. If the use case concept of some use case U is greater the use case concept of some use case U' then there exists a unidirectional relationship between them. If U corresponds to generalized use case, then U "generalizes" U'. In other cases, the relation is either "includes" or "extends" but a deeper analysis is needed to establish which kind. The preceding rules are summarized in Figure 15.

Actors (external entities) can also be introduced in the model by assigning them to some functions in code. Good candidates are event handling functions and I/O library functions. If function F is assigned to actor A, and the object concept for F corresponds to the realization of use case U then we associate A and U in the model.

How do modules, subsystems and components fit in this model?

This hierarchy is independent to functional decomposition (it also includes other criteria: physical distribution, technology concerns, etc.

Figure 16 depict possible relationships between physical subsystems and use case realizations. If we wish to introduce this additional classification in our model we can extend the context relation to include containment relation between functions and subsystems.

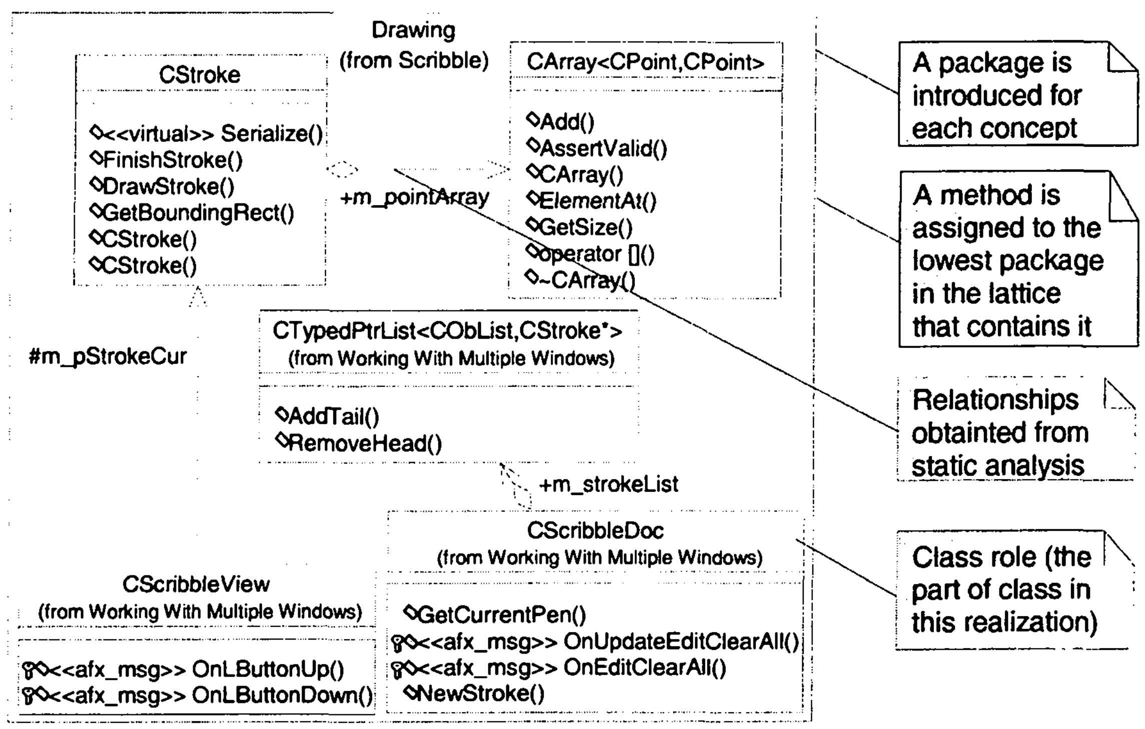

Drawing
(from Scribble)
CStroke
<<virtual>> Serialize()
FinishStroke()
DrawStroke()
GetBoundingRect()
CStroke()
~CStroke()
+m_pointArray
CArray<CPoint,CPoint>
Add()
AssertValid()
CArray()
ElementAt()
GetSize()
operator []()
~CArray()
#m_pStrokeCur
CTypedPtrList<CObList,CStroke*>
(from Working With Multiple Windows)
AddTail()
RemoveHead()
+m_strokeList
CScribbleDoc
(from Working With Multiple Windows)
GetCurrentPen()
<<afx_msg>> OnUpdateEditClearAll()
<<afx_msg>> OnEditClearAll()
NewStroke()
CScribbleView
(from Working With Multiple Windows)
<<afx_msg>> OnLButtonUp()
<<afx_msg>> OnLButtonDown()
A package is introduced for each concept
A method is assigned to the lowest package in the lattice that contains it
Relationships obtainted from static analysis
Class role (the part of class in this realization)

Figure 15. Summary of rules for updating UML model from the lattice

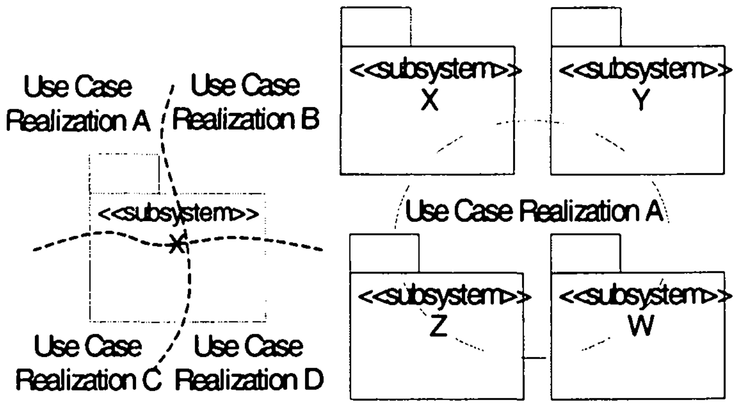


Figure 16. Relationships between physical subsystems and use case realizations

RELATED WORK

Numerous methods were proposed structuring models of software systems. Structural clustering techniques define strictly structural criteria for decomposing a system in a hierarchy of subsystems – minimal coupling between subsystems and maximal coupling between entities in the same subsystem. For example, Mancoridis (1998) proposes sub-optimal algorithms based on search (hill climbing) and a genetic algorithm. Wiggerts (1997) presents an overview of clustering algorithms that could be used for software re-modularization.

Clustering techniques that focus on data abstraction, consider the use of data of the same type as the grouping criterion. These techniques find their use in discovering candidates for object and classes in non-object-oriented software. Lindig and Snelting (1997) use a formal concept analysis on relation between procedures and global variables. However, experiments suggest a limited usability for discovering system structure (partly because of noise in input data). Siff and Reps (1997) also use formal concept analysis in a similar context, but also make use of negative information, e.g. function f does not use structure x. They claim more promising results than Lindig and Snelting (1997).

Plan recognition techniques, Qulici (1995, 1998), need a library of predefined programming plans, which are matched against parts of code to build a hierarchy of concepts (abstractions such as complex data structures or a specific functionality). Each plan consists of two parts: a pattern, used for matching and a concept that is recognized with the pattern. A pattern is a combination of components (the language items or sub-plans that must be recognized) and constraints (the relationships that must hold between these components to have the plan recognized). Plan matching can be performed by a variety of techniques including deductive inference, constraint satisfaction or pattern matching, and could be combined with a top-down search of plan libraries for a plan that matches a particular concept. Plan recognition could be applied to maintenance related program transformations, Kozaczynski (1992). These techniques typically require the existence of a large plan database and at the present guarantee only partial model recovery. There is also a problem of varying code patterns that implement the same plan and a scaling problem when considering large systems.

Recently, a work has been done on identifying standard design patterns in code. For example, Antoniol et. al (1998) present a multi-stage reduction strategy using software metrics and structural properties to extract some structural design patterns from code.

Besides automatic decomposition techniques, with a possibility for a user to guide a process in some phases, there exist a number of manual and semi-automatic techniques. These techniques allow collecting, filtering and presenting in a suitable form, data obtained by means of either static or dynamic analysis of the system. The user is, however, responsible for identifying and grouping system entities into subsystems, or for connecting implementation entities with a functional model of the system. Rigi tool, described in Müller (1993), allows a user to create multiple views, and also has some graph arranging algorithms to cluster related system entities. Dali tool from Kazman (1997) uses a two-phase process – in the view extraction phase, various elementary views are extracted from the system using static and dynamic analysis, and are stored in a view repository. In the view fusion phase, these views are joined under user manual control into more complex views.

In Richner (1998, 1999), static and dynamic information is modeled in terms of logic facts in Prolog, which allows creating views with high-level abstractions using declarative queries.

In software reflexion modeling technique, Murphy (1995), users define a high-level model (a graph of logical subsystems) and specify how the model maps to the source model (a call graph) – by assigning functions to logical subsystems. A tool then computes software reflexion model, a graphical representation that shows how well the connections between logical subsystems defined in high-level model agree with those actually found in source model.

De Hondt (1998) proposes an architectural recovery model based on software classifications. In his model, classifications are containers of software artifacts, and artifacts can be classified in multiple ways. Among other methods, he proposes virtual classifications. There is an explicit (e.g. declarative) description which elements are intended to belong to a classification.

Finally, there has been some work on locating functionality in code using static or dynamic analysis. The Software Reconnaissance technique implemented in χSuds toolset, Agrawal (1998) locates plans i.e. code fragments associated with a particular feature i.e. functionality by comparing traces of test cases that exhibit or do not exhibit a feature. Every feature must be considered separately from others, and the focus is on small-scale features. Wilde (1996) proposed a technique to partition a program code in equivalence classes such that all code in the same class has the same execution count in each of multiple test cases. The results of the case study were mixed. On the one hand several meaningful and non-trivial program plans were identified. On the other, a significant fraction of the equivalence classes did not represent meaningful program plans.

Slicing techniques make use of call and data dependency graphs to identify those parts of code that contribute the computation of the selected function, for all possible inputs, in case of static slicing, or for a given input, in case of dynamic slicing. A dynamic slicing technique presented in Korel (1998) helps to understand large programs, by visualizing the control flow slices at the level of call graphs. In Canfora (1998), a static slicing technique is used to identify user interface software layer.

CONCLUSIONS

The proposed method is conceived to automate a part of task that a reengineer has to do manually after creating basic UML model with a static analyzer.

We certainly do not imply that a presented decomposition strategy is "the best" strategy, in the sense that the original designer would create the same decomposition, or that it works equally well for all kinds of software systems. But it is our opinion that a functional decomposition is useful for the system-understanding task that faces a reengineer unfamiliar with system implementation.

We applied the method to 10K line Wordpad standard Windows accessory application. We focused on essential functionality and devised about 25 use cases and roughly the same number of test cases that covered about 70%

of application functions. Resulting lattice contains 35 concepts, 5 "generalized", 15 "ordinary" and 15 "included" ones. The decomposition quality was subjectively assessed as good, without making any transformations on the context relation. In the future, we also plan to demonstrate the usability of our method on even more complex applications. That might include a project the first author has been involved with for about four years. The application is an integrated development environment with reverse engineering capabilities that runs on Windows platforms. Its size is nearly 400.000 lines of C++ code, and it has many attributes of a legacy application: the original team left the firm, the application was initially developed without any explicit modeling or architectural considerations etc.

A work is now being done to extend the method to include creation of interaction diagrams, on the basis of more detailed dynamic data e.g. a complete activation tree for the given test execution, produced by some tools. The presented method can be applied even to non object-oriented systems, in combination with techniques that identify candidate objects in the code, to obtain an object-oriented model of such systems.

REFERENCES

Agrawal, H & Alberi, J. L. & Horgan, J. R. & Li J. & London, S. & Wong, W. E. & Ghosh, S. & Wilde, N. (1998) "Mining System Tests to Aid Software Maintenance", **IEEE Computer**, July, pp. 64-73.

Antoniol, G. & Fiutem, R. & Cristoforetti, L. (1998) "Design Pattern Recovery in Object-Oriented Software", **Proceedings of the 6th International IEEE Workshop on Program Comprehension IWPC 98**, Ischia, Italy, pp. 153-160.

Blok, M. C. & Cybulski, J. L. (1998) "Reusing UML Specifications in a Constrained Application Domain", **Asia Pacific Software Engineering Conference**, Taipei, Taiwan, Dec, pp 196-202

Booch, G. & Rumbaugh, J. & Jacobson, I. (1998) **The Unified Modeling Language User Guide**, Addison-Wesley.

Burmeister, P. (1998) **Formal Concept Analysis with ConImp: Introduction to the Basic Features,** Arbeitsgruppe Allgemeine Algebra und Diskrete Mathematik, Technische Hochschule Darmstadt, Schloßgartenstr. 7, 64289 Darmstadt, Germany.

Canfora, G. & Cimitile, A. & De Lucia, A. & Di Lucca, G. A. (1998) "Decomposing Legacy Programs, a First Step Towards Migrating to Client-Server Platforms", **Proceedings of the 6th International IEEE Workshop on Program Comprehension IWPC 98**, Ischia, Italy, pp. 136-144.

De Hondt, K. (1998) **A Novel Approach to Architectural Recovery of Object-Oriented Systems**, PhD Theses, Vrije Universiteit Brussel.

Jarke, M. (1999) "Scenarios for Modeling", **Communications of the ACM**, January, Vol. 42, No. 1, pp. 47-48.

Jarke, M. & Kurki-Suonio, R. (1998) "Scenario Management - Introduction to the Special Issue", **IEEE Transactions on Software Engineering**, Vol. 24, No. 12, December, pp. 1033-1035.

Jacobson, I. & Booch, G. & Rumbaugh, J. (1999) **The Unified Software Development Process**, Addison-Wesley.

Kazman, R. & Carrière, S. J. (1997) **View Extraction and View Fusion in Architectural Understanding,** Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213

Kit, E. (1995) **Software Testing in the Real World**, Addison-Wesley.

Korel, B. & Rilling, J. (1998) "Program Slicing in Understanding of Large Programs", **Proceedings of the 6th International IEEE Workshop on Program Comprehension IWPC 98**, Ischia, Italy, pp. 145-152.

Kozaczynski, W. & Ning, J. & Engberts, A. (1992) "Program Concept Recognition and Transformation", **IEEE Transactions on Software Engineering**, Vol. 18, No. 12, December, pp. 1065-1075.

Kim, D. K. & Jung, H. T. & Kim, C. K. (1998) "Techniques for Systematically Generating Framework Diagram Based on UML", **Asia Pacific Software Engineering Conference**, Taipei, Taiwan, Dec, pp. 203-210.

Lindig, C. & Snelting, G. (1997) "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis", **International Conference on Software Engineering**, Boston, USA, pp. 349-359.

Mancoridis, S. & Mitchell, B. S. & Rorres, C. & Chen, Y. & Gansner, E. R. (1998) "Using Automatic Clustering to Produce High-level System Organizations of Source Code", **IEEE Proceedings of 6th International Workshop on Program Understanding IWPC'98**, Ischia, Italy, June, pp. 45-52.

Müller, H. & Orgun, M. & Tilley, S. R. & Uhl, J. S. (1993) "A Reverse Engineering Approach To Subsystem Structure Identification", **Software Maintenance: Research and Practice**, 5(4), December, pp. 181-204.

Murphy, G. & Notkin, D. & Sullivan, K. (1995) "Software Reflexion Models: Bridging the Gap between Source and High-Level Models", **Proceedings of the ACM SIGSOFT '95**, Washington, D.C., pp. 18-28.

Noffsinger, W. B. & Niedbalski, R. & Blanks, M. & Emmart, N. (1998) "Legacy Object Modeling Speeds Software Integration", **Communications of the ACM**, Vol. 41, No. 12, December, pp. 80-89.

Quilici, A. & Yang, Q. & Woods, S. (1998) "Applying Plan Recognition Algorithms To Program Understanding", **Journal of Automated Software Engineering** , 5(3), pp. 347-372.

Quilici, A. & Chin, D. (1995) "DECODE: A Cooperative Environment for Reverse-Engineering Legacy Software", **Proceedings of the Second IEEE Working Conference on Reverse Engineering**, July , pp. 156-165.

Richner, T. & Ducasse, S. (1999) "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information", **Proceedings of the IEEE International Conference on Software Maintenance CSMR'99**

Richner, T. & Ducasse, S. & Wuyts, R. (1998) "Understanding Object-Oriented Programs with Declarative Event Analysis", **12th European Conference on Object-Oriented Programming**, Workshop on Experiences in Object-Oriented Re-Engineering, Brussels, Belgium, July.

Sefika, M. & Sane, A. & Campbell, R. H. (1996) "Monitoring compliance of a software system with its high-level design models", **Proceedings of the 18th International Conference on Software Engineering,** March, pp 387–396.

Siff, M. & Reps, T. (1997) "Identifying Modules Via Concept Analysis", **IEEE International Conference on Software Maintenance**, Bary, Italy, September.

Wiggerts, T. A. (1997) "Using Clustering Algorithms in Legacy System Remodularization", **Working Conference of Reverse Engineering WCRE '97**, pp. 33-43Wilde, N. & Cotten, M. & London, S. (1996) "Using Execution Counts to Identify Delocalized Program Plans", Tech. Rpt. SERC-TR-81F, Software Engineering Research Center, OSE-301, University of Florida, FL 32611.