# A Use-Case Driven Method of Architecture Recovery for Program Understanding and Reuse Reengineering

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

*This work focuses on architectural recovery for program understanding and reuse reengineering of legacy object-oriented systems. The proposed 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 decompose the logical hierarchy of subsystems, so that parts of the system which implement similar functionality are grouped together.*

## 1. Introduction

The software architecture is the overall structure of a software system. It describes the organization of the software as a composition of components. It lays down the global control structures and the protocols for communication, synchronization and data access. It defines how functionality is assigned to design elements and how design elements are composed. It also addresses the physical distribution, the scaling and the performance, the dimensions of evolution and the selection among design alternatives [34].

Architectural views show how architectural components, usually large sets of classes, are composed to form a software system. In this paper the focus is on the functional view, where system entities that implement similar functionality are grouped together. UML [4] supports the 4+1 view model [21].

Existing systems often have never had their architecture documented, or have only documented one view of the architecture, or the system's documentation has become out of date with respect to the implementation. Among other benefits, architecture recovery technologies can facilitate system understanding and maintenance, planning for reengineering, help in identification of commonality within a family of related systems, and also bring about the possibility of reusing system structure and component in the development of new systems [25].

We propose a method to partially recover elements of UML architectural views. Collaborations (i.e. use case realizations) are recovered, and elements from the logical view are related to relevant use cases. This is a specific solution for the general problem of locating functionality in code.

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 [16]. A number of advantages are reported if a process of developing or reengineering a system is driven by scenarios (i.e. use cases) [14], [15]:

- 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, 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 [3]. 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 [22], thus stressing a need to have a use case view of the system model.

## 2. Related work

Numerous methods were proposed for recovering various views of system architecture. 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, In [24], suboptimal algorithms are proposed, based on search (hill climbing) and genetic algorithms. Wiggerts [36] presents an overview of clustering algorithms that could be used for software remodularization.

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 [23] use formal concept analysis on relations 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 [35] also use formal concept analysis in a similar context, but make use of negative information, e.g. function f does not use structure x. They claim more promising results than [23].

Plan recognition techniques [29], [30] 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 subplans 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 [20]. 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 [2] present a multi-stage reduction strategy using software metrics and structural properties to extract some structural design patterns from the code.

Aside from 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. The Rigi tool [26] allows a user to create multiple views, and also has some graph arranging algorithms to cluster related system entities. In [32], [31] 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 model techniques [27], 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 the 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 the source.

The Dali tool [17] 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.

De Hondt [13], 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 [1] 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 et. al [37] 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 [19] helps to understand large programs, by visualizing the control flow slices at the level of call graphs. In [7] a static slicing technique is used to identify user interface software layer.

## 3. The proposed method

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

1. Identify a set of use cases for a given system or a subset thereof.
2. For each use case, define one or more test cases which cover that particular use case.
3. Collect profiling information (functional coverage) 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 impacted by the execution of these use cases. If an edge connects two concepts in the lattice, then the parent concept represents a "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 sections 3.2. to 3.5. In the next section we present the results obtained by applying the technique to a small sample application.

### 3.1. 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. Microsoft Visual C++ profiler is used for data gathering, Rational Rose for extracting initial UML model of the system and also for presentation of the resulting decomposition. ConImp [6] 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 which 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++. A batch process activates the latter three applications once test recording is done to prepare data for other tools.

In the present configuration, URCA can analyze systems with at most 255 functions and 128 use cases, due to limitations of the ConImp tool. Other components do not impose such limitations, and we are currently working on replacing ConImp with our own implementation of a concept analysis algorithm, which will enable us to analyze real world systems.

We used the prototype tool to analyze a sample application – Scribble step 3 tutorial application from Microsoft Developer Network Library. It is MDI application that supports elementary drawing, which can

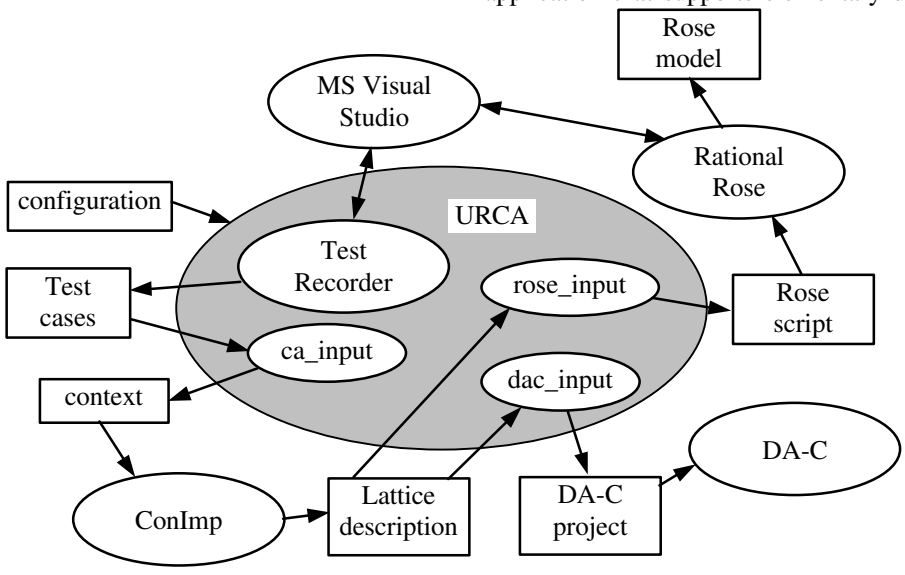



**Figure 1. Structure of the prototype system**

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. It does not violate the current limitation of ConImp in the number of objects. We defined the following use cases that roughly corresponds to items from the main menu:

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

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 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 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 Asistant for C tool. 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 3. 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.

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

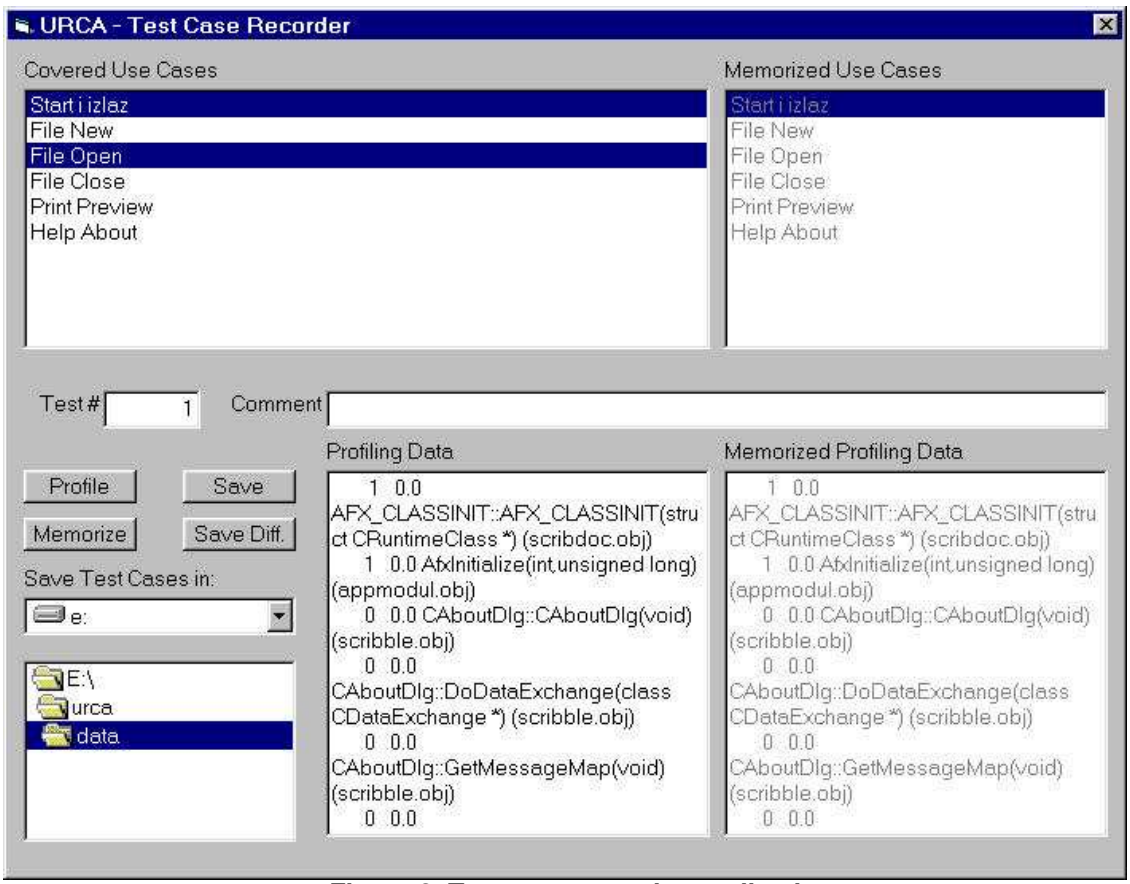



**Figure 2. Test case recorder application**

lattice description to create a Rose script file 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 4–8 present the logical structure of the model. For each concept, except the root and the leaf, 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. The only exception is the "Working With Multiple Windows" package, in which member functions are not shown at all to conserve space. A static analyzer that is a part of Rose induced relationships between classes.

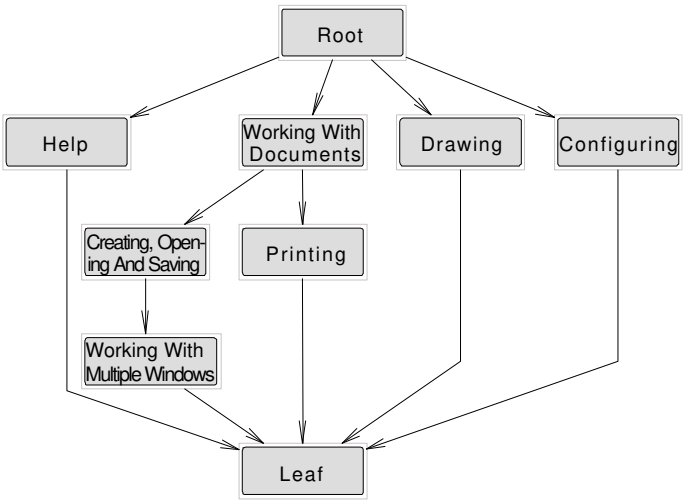



**Figure 3. The concept lattice for Scribble**

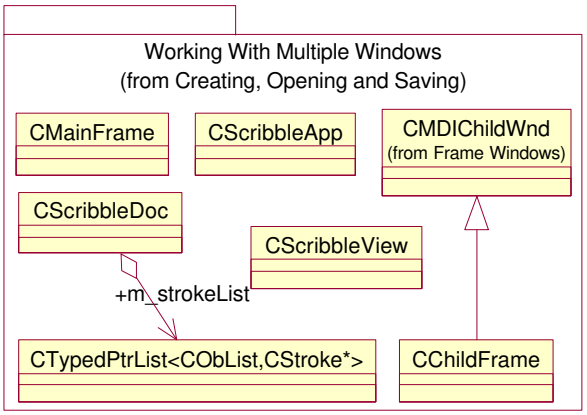



**Figure 4. Working with multiple windows package**

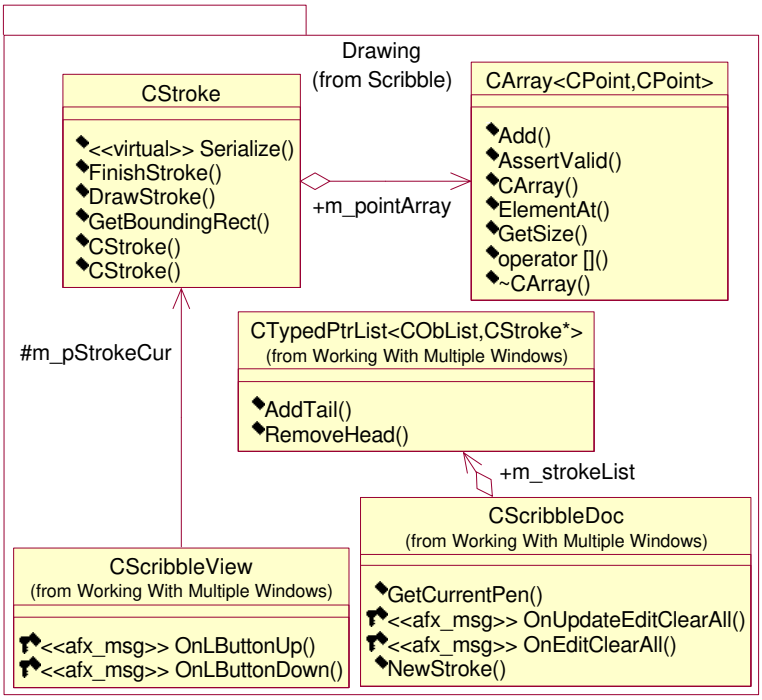



**Figure 5. Drawing package**

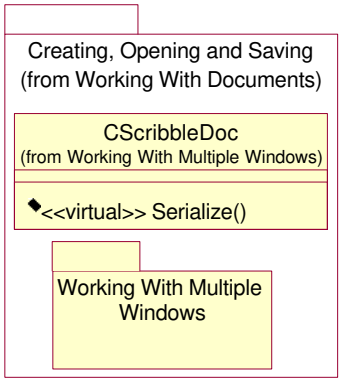



**Figure 6. Creating, opening and saving package**

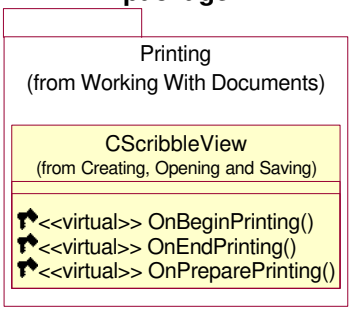



**Figure 7. Printing package**

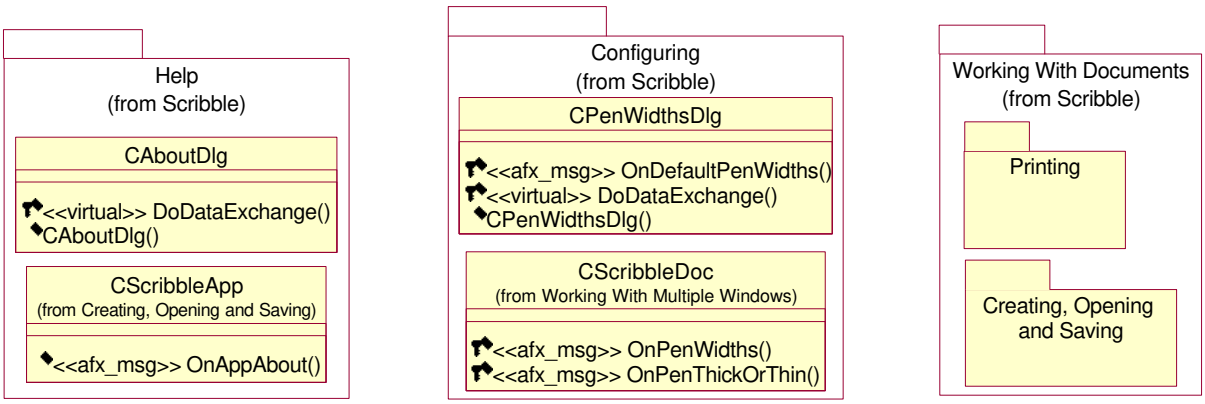



**Figure 8. Help, Configuring and Working with documents packages**

### 3.2. Identifying use cases

According to Jacobson et. al [16], most classes, with operations, interfaces, and attributes that are private to subsystems or service subsystems (hidden in the rest of the system), are not architecturally significant. Subsystems that are variants of other subsystems are not important from an architecture perspective. Experience indicates that less than 10% of the classes are relevant to the architecture. The remaining 90% are not significant because they are not visible to the rest of the system. A change to one of them does not affect anything substantial outside the service subsystem. Nor are most use-case realizations architecturally significant since they do not impose any additional constraints on the system. Most use-case realizations represent simple add-on behavior that is easy to implement even though they constitute most of the functions that the system offers.

Architecturally significant use cases are the ones that help us mitigate the most serious risks, those that are the most important to users of the system, and those that help us cover all important functionality so that nothing is left in the shadows.

### 3.3. Selecting 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 [16].

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 [18].

### 3.4. Applying formal concept analysis

Concept analysis provides a way to identify sensible groupings of objects that have common attributes [6]. A context relation indicate which object has which attributes. Figure 9 present a sample context relation for four objects $O_1$ – $O_4$ and eight attributes, $A_1$–$A_8$.

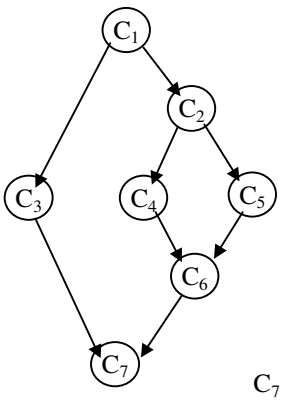
| | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ | $A_8$ |
|---|---|---|---|---|---|---|---|---|
| $O_1$ | X | X | | | | | | |
| $O_2$ | | | X | X | X | | | |
| $O_3$ | | | X | X | | X | X | X |
| $O_4$ | | | X | X | X | X | X | X |

**Figure 9. 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

[35]. The concept lattice for a sample context relation from Figure 9 is presented in Figure 10.




$C_1 = (\{O_1,O_2,O_3,O_4\}, \{\})$

$C_2 = (\{O_2,O_3,O_4\}, \{A_3,A_4\})$

$C_3 = (\{O_1\}, \{A_1, A_2\})$

$C_4 = (\{O_2,O_4\}, \{A_3,A_4,A_5\})$

$C_5 = (\{O_3,O_4\}, \{A_3,A_4,A_6,A_7,A_8\})$

$C_6 = (\{O_4\}, \{A_3,A_4,A_5,A_6,A_7,A_8\})$

$C_7 = (\{\}, \{A_1,A_2, A_3,A_4,A_5,A_6,A_7,A_8\})$

**Figure 10. A sample concept lattice**

For our purposes, the set of objects O includes all functions, class members as well as non-members, in our application. The set of attributes A includes all use cases as defined by the user. A 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 the 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, the following conditions hold:

1. There exist at least one test case that executes F (at least once) and covers U, and
2. Every test case that executes F (at least once) also covers U.

Note that it is not required that every test case that covers U must also execute F, because there can exist different execution paths for 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. This operation can be performed automatically by a tool, 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 a 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

The quality of decomposition can be estimated from the topology of the lattice, and could serve as a basis for a decision to reformulate test cases or to modify the context matrix. Figure 11 shows an example of the preferred topology. When we remove the root concept (representing all functions and no use cases) and the only leaf concept (if such exists, it represents all use cases), we should obtain a forest (the set of proper trees).

### 3.5. Updating UML model from the lattice

From the concept lattice, we can determine various elements of a UML model. To specify model elements, let us first define the notions of object and attribute concepts. The *function concept* (E,I) that corresponds to a function f is the smallest (closest to the leaf) concept for which $f \in E$. The *use case concept* (E,I) that corresponds to a use case u is the largest (closest to the root) concept for which $u \in I$.

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, except the root and the leaf ones, we introduce one package in a logical view. The ordering relation between concepts 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 the function concept to most of its functions, and reference that class from other packages.

It is well known that the use case and the logical view of a UML model are related via interaction diagrams attached to particular use cases, showing the 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

exactly the interaction diagrams but only model elements relevant for the implementation of each use case. They all belong to the package that corresponds to the use case concept for a particular use case.

Interface classes of some package P (denoted by stereotype <<interface>>) are those which are also referenced in some other package that is not contained in P. Finally, the ordering relation on the set of concepts can be used to determine relationships among particular use cases, however here additional work is required. Information contained in the lattice is inadequate for this purpose.

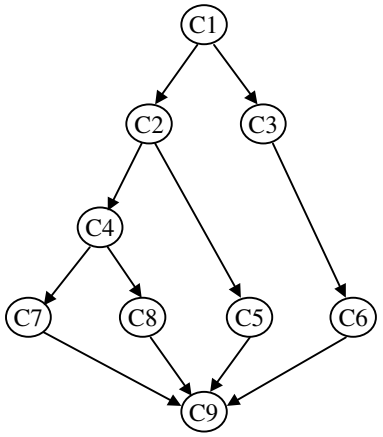



**Figure 11. A preferred topology of the concept lattice**

## 4. Conclusions

We plan to demonstrate the usability of our technique on some more realistic examples. 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.

There is an experimental evidence of applying formal concept analysis, although in a different manner, for recovering the structure of large systems [23].

Further work is needed to extend the technique to include the generation of interaction diagrams, on the basis of more detailed dynamic analysis, possibly using animation. A second possible research direction is to incorporate other criteria (possible through adding attributes in the context relation) such as layering, which is, as [13] points out, the decomposition criterion orthogonal to functional decomposition. The presented technique 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.

## 5. References


[1] H. Agrawal, J. L. Alberi, J. R. Horgan, J. Jenny Li, Saul London, W. Eric Wong, Sudipto Ghosh, Norman Wilde, Mining System Tests to Aid Software Maintenance, IEEE Computer, July 1998, pp. 64-73

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

[3] M. C. Blok, J. L. Cybulski, Reusing UML Specifications in a Constrained Application Domain, Asia Pacific Software Engineering Conference, Taipei, Taiwan, Dec. 1998, pp 196-202

[4] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1998.

[5] Elizabeth Burd, Malcolm Munro, A method for the identification of reusable units through the reengineering of legacy code, The Journal of Systems and Software (44), 1998, pp. 121-134

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

[7] G. Canfora, A. Cimitile, A. De Lucia, G. A. Di Lucca, 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, 1998, pp. 136-144

[8] Gerardo Canfora, Andrea De Lucia, Malcolm Munro, An integrated environment for reuse reengineering C code, The Journal of Systems and Software 42, 1998, pp. 153-164

[9] G. Canfora, A. Cimitile, M. Munro, A reverse engineering method for identifying reusable abstract data types, Proceedings of Working Conference on Reverse Engineering, Baltimore, 1993, pp. 73-82.

[10] A. Cimitile, G. Visaggio, Software salvaging and the dominance tree, The Journal of Systems and Software Vol. 28, No. 2, 1995, pp. 117-127.

[11] Paul C. Clements, Nelson Weiderman, Report on the Second International Workshop on Development and Evolution of Software Architectures for Product Families, SPECIAL REPORT, CMU/SEI-98-SR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213

[12] H. Gall, M. Jazayeri, R. Klosch, W. Lugmayr, and G. Trausmuth. Architecture Recovery in ARES. In Proc. 2nd ACM SIGSOFT Int. Software Architecture Workshop, pages 111–115, San Francisco, USA, Oct. 1996.

[13] K.De Hondt, A Novel Approach to Architectural Recovery of Object-Oriented Systems, PhD Theses, Vrije Universiteit Brussel, 1998.

[14] M. Jarke, Scenarios for Modeling, Communications of the ACM, January 1999, Vol. 42, No. 1, pp. 47-48

[15] M. Jarke, R. Kurki-Suonio, Scenario Management – Introduction to the Special Issue, IEEE Transactions on Software Engineering, Vol. 24, No. 12, December 1998, pp. 1033-1035

[16] I. Jacobson, G. Booch, J. Rumbaugh, The Unified Software Development Process, Addison-Wesley, 1999.

[17] Rick Kazman, S. Jeromy Carrière, View Extraction and View Fusion in Architectural Understanding, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213

[18] E. Kit, Software Testing in the Real World, Addison-Wesley, 1995.

[19] B. Korel, J. Rilling, Program Slicing in Understanding of Large Programs, Proceedings of the 6th International IEEE Workshop on Program Comprehension IWPC 98, Ischia, Italy, 1998, pp. 145-152

[20] W. Kozaczynski, J. Ning, A. Engberts, Program Concept Recognition and Transformation, IEEE Transactions on Software Engineering, Vol. 18, No. 12, December 1992, pp. 1065-1075

[21] P. Kruchten, The 4+1 View of Architecture, IEEE Software, Vol. 12, No. 6, November 1995, pp. 45-50

[22] Dong Kwan Kim, Hyo Taeg Jung, Chae Kyu Kim, Techniques for Systematically Generating Framework Diagram Based on UML, Asia Pacific Software Engineering Conference, Taipei, Taiwan, Dec. 1998, pp. 203-210

[23] C. Lindig, G. Snelting, Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis, ICSE 97, Boston, USA, 1997, pp. 349-359

[24] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, E. R. Gansner, 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, 1998, pp. 45-52

[25] N. C. Mendonca, J. Kramer, Developing an Approach for the Recovery of Distributed Software Architectures, Proceedings of the 6th International IEEE Workshop on Program Comprehension IWPC 98, Ischia, Italy, 1998, pp. 28-36

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

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

[28] W.B. Noffsinger, R. Niedbalski, M. Blanks, N. Emmart, Legacy Object Modeling Speeds Software Integration, Communications of the ACM, Vol. 41, No. 12, December 1998, pp. 80-89

[29] A. Quilici, Q. Yang, S. Woods, Applying Plan Recognition Algorithms To Program Understanding, Journal of Automated Software Engineering , 5(3):347-372, 1998.

[30] A. Quilici, D. Chin, DECODE: A Cooperative Environment for Reverse-Engineering Legacy Software, Proceedings of the Second IEEE Working Conference on Reverse Engineering, pp. 156-165, July 1995.

[31] T. Richner, S. Ducasse, Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information, to appear in IEEE Proceedings ICSM'99

[32] T. Richner, S. Ducasse, R. Wuyts, 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 1998.

[33] M. Sefika, A. Sane, R. H. Campbell, Monitoring compliance of a software system with its high-level design models. In Proceedings ICSE-18, March 1996, pp 387–396

[34] Mary Shaw and David Garlan. Software Architecture. Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[35] M. Siff, T. Reps, Identifying Modules Via Concept Analysis, IEEE International Conference on Software Maintenance, Bary, Italy, Sept. 1997

[36] T. A. Wiggerts, Using Clustering Algorithms in Legacy System Remodularization, Working Conference of Reverse Engineering WCRE '97, pp. 33-43

[37] N. Wilde, M. Cotten, S. London, Using Execution Counts to Identify Delocalized Program Plans, Tech. Rpt. SERC-TR-81F, Software Engineering Research Center, OSE-301, University of Florida, FL 32611, 1996

[38] Steven G. Woods, Evolving Legacy Systems: New Technologies Supporting Architectural Views of Software, Tutorial presented at Dagstuhl 98, Dagstuhl, Germany, March 1998.