Architecture Recovery of Dynamically Linked Applications: A Case Study

Igor Ivkovic and Michael W. Godfrey
Software Architecture Group
Department of Computer Science, University of Waterloo
{iivkovic, migod}@uwaterloo.ca

Abstract

Most previously published case studies in architecture recovery have been performed on statically linked software systems. Due to the increase in use of middleware technologies, such as CORBA, and OOP concepts, such as polymorphism, there is an opportunity and a need to analyze architectures of these dynamically linked systems. This paper presents the results of software architecture extraction of the Nautilus file manager, which employs CORBA in its implementation. A combination of existing static analysis and use-case modeling architecture recovery techniques was used with the expectation of complex but complete architecture extraction of a system such as Nautilus. We have found that this combined approach named Dynamo-1 presented in this paper provided successful focused architecture recovery and guidance for the future work in complete architecture recovery of dynamically linked applications.

Keywords: Nautilus, GNOME, program comprehension, architecture recovery, software architecture, dynamically linked applications, Dynamo, PBS, Focus.

1. Introduction

Software re-engineering is one of the most important activities in the software industry used to improve the understanding and extend the life expectancy of complex systems that lack proper documentation. The two steps in software re-engineering, reverse and forward engineering, attempt to extract and improve the concrete or as-implemented architecture of the system.

The re-engineering approach selected for particular software must conform to the type and the model of the system (e.g., distributed DBMS for Linux) and its domain of application (e.g., financial industry for contact management). Without the clear linkage between the selected approach and the application's characteristics, the usefulness of the re-engineering efforts to system's stakeholders will be significantly decreased [1, 2].

Previous case studies in software reverse engineering have been mostly focused on the statically linked software [3, 11, 12]. The applications that employ dynamic linking through middleware technologies (e.g., CORBA and COM) and OOP concepts (e.g., polymorphism and dynamic binding) are not emphasized, and the complete architecture recovery of these systems is considered to be an open problem [13].

An increase in use of the dynamic linking in both proprietary and open-source software indicates a wider range of opportunities for architectural analysis, and implies a need for a solution to architecture recovery of such systems to better support their program comprehension and software evolution needs.

This paper presents a case study in reverse engineering an application that employs significant dynamic linking in its implementation. The selected software system is the Nautilus file manager, which is an integral part of the GNOME desktop environment [4]. Nautilus' use of CORBA and its relative lack of design documents makes it an appropriate candidate for architecture recovery of dynamically linked software systems. The approach used in the analysis combines the static analysis methodology [3], based on the filtering and clustering reverse engineering framework, and the use-case modeling technique, based on the Focus reverse engineering and software evolution approach [8].

More accurately, our paper discusses the work used in or related to our case study in the section two, and talks about the importance of dynamically linked applications in the evolution of software in general in the section three. Nautilus specifics are described in the section four while the actual architecture extraction experiences and conclusions are presented in the last two sections.

2. Related work

Holt et al. have significant experience in performing architecture recovery of statically linked software systems [3, 7, 11, 12]. Their analysis tool named the Portable

Bookshelf (PBS), based on filtering and clustering reverse engineering framework, provides automated analysis and extraction of architectural artifacts that relies on the existence of code structure and relevant design documentation. In more detail, their approach starts by extracting a conceptual or as-designed architecture of the system from existing documentation and program structure. This model is then used to extract the concrete or as-implemented architecture based on the identification of static relations and a system clustering using the PBS and the human judgment.

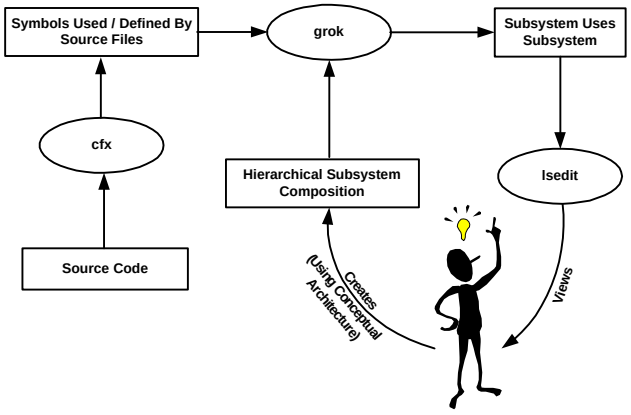

Figure 1. PBS architecture extraction process

Medvidovic et al. use an approach to architecture extraction based on use-case modeling called Focus [8]. Focus does not aim for complete analysis of the application nor does it try to significantly automate the analysis and architecture extraction. Instead, it is attentive to the key parts of the system that are most relevant for the system's continued evolution. In addition, instead of relying on existence of valid design documentation and code structure, it utilizes the use-case modeling to capture the rationale and the structure of the system being examined.

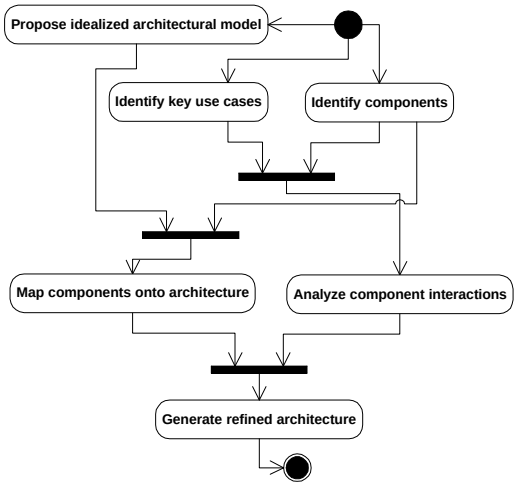

Figure 2. Focus architecture extraction process

Kazman et al. have also dealt with architecture recovery and suggested the use of other system structures such as build files, make files, and executables to enhance the source-code oriented analysis. Kazman's group also discussed the use of code profilers to deal with dynamically linked applications [13].

3. Dynamically Linked Applications

From the reverse engineering perspective, applications that employ late binding techniques such as middleware technologies and OOP concepts belong to dynamically linked applications. Such programs employ these techniques for several reasons:

- the ability to share commonly used code across many applications and platforms,
- the ability to use distributed services,
- the ability to hide the implementation of services from the application, and
- the flexibility to allow multiple implementations which are selectable at runtime [14].

Given the importance of cost and time efficiency in development and maintenance, and a need for distributed applications due to the popularization of the Internet, the use of dynamically linked applications is likely to increase significantly in the next several years.

4. Nautilus

Nautilus is an open-source file manager and graphical shell for UNIX-like operating systems that employs CORBA to achieve flexibility in implementation and selectability of its services. Its source code is available through the GNOME CVS repository.

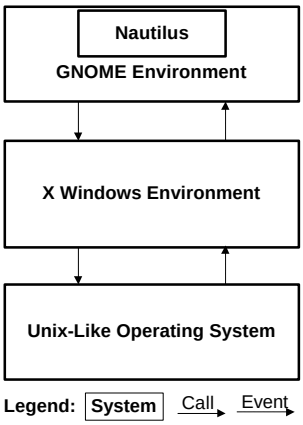

Figure 3. Nautilus in the environment

Nautilus provides an interface similar to Microsoft Windows Explorer or Linux Midnight Commander, and extends it through customizable and re-programmable views. File management is provided through the GNOME Virtual File System (VFS), while access to the remote data files is available through the various supported protocols (e.g., HTTP, FTP) [5].

Given that GNOME is built on top of the X Windows environment, Nautilus operating system calls have to go through both the GNOME and the X Windows layer before reaching the actual operating system services. The benefit of this approach is the easier implementation through the unified GNOME API with the slight overhead of the call proxying through two different architectural layers.

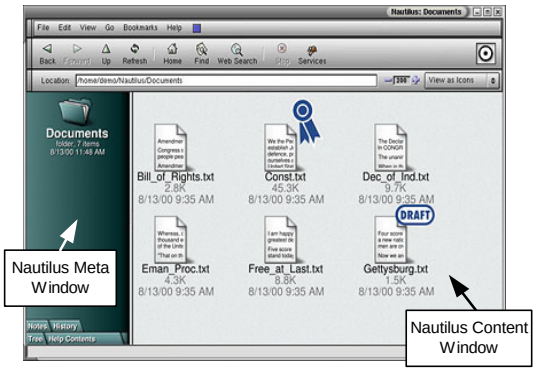

Figure 4. Nautilus file management

Nautilus is also a fairly large-scale project with almost 250 KLOC [6] that consists of the container application called Nautilus that serves as the main controller, and a number of view components that are loaded on demand. Inside of the container application, there is a main content window that shows various data views (e.g., images are shown as thumbnails that can be enlarged or shrunk using the zoom command) and meta-windows on the left that provide specific functionality (e.g., information about or functionality related to the data shown in the main content window).

When navigation to a particular URI is requested by a view, Nautilus determines what content view and meta-views should be displayed. Content and meta-views are determined based on the requested URI, the referring URI, the content type of the requested URI, and the application configuration.

4.1. Nautilus and GNOME

Being the integral part of GNOME, Nautilus directly utilizes other GNOME components such as CORBA-compliant ORB and GNOME VFS [4].

CORBA is an open standard, distributed-object, computing infrastructure being standardized by the Object Management Group (OMG). CORBA provides automation of many network-programming tasks such as object registration and activation, and operation dispatching. Look up service is provided by the Object Request Broker to get a reference to the function (service).

GNOME VFS is defined as a user-level file system abstraction that provides UNIX-like functionality over multiple protocols and extends the file system concept. Currently, it supports Native File System, HTTP, FTP and User-Level NFS.

5. Architecture Recovery of Nautilus

To extract the architecture of a dynamically linked system such as Nautilus, parts of the static analysis approach using the PBS tool and the use case modeling approaching based on the Focus method were combined. Each of the steps in this approach is selected based on its compatibility with the properties of the analyzed system. The resulting approach, named Dynamo-1 for its specialization in dynamically linked systems, is presented as Figure 5 and used with Nautilus**.**

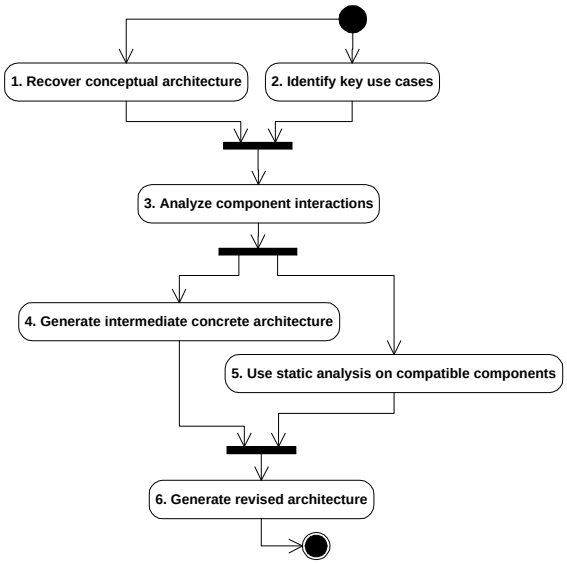

Figure 5. Dynamo-1 architecture recovery process

5.1. Recovering conceptual architecture

Using Holt et al. PBS-based approach, the system's conceptual architecture is first extracted from Nautilus' original documentation, the source code structure and

comments, file and directory naming conventions, and if available existing reference model for this type of application [3].

Based on the examination of the Nautilus' documentation as well as at the corresponding structural elements of Nautilus, it is recognized that each instance of the Nautilus Application has a NautilusWindow object that encapsulates all of the main window controls and uses NautilusView objects to establish communication with Nautilus view components. This communication is primarily one-way, which suggests a layered architectural style where the NautilusWindow object calls methods on the NautilusView objects and responds to returning messages. This structure is depicted as Figure 6.

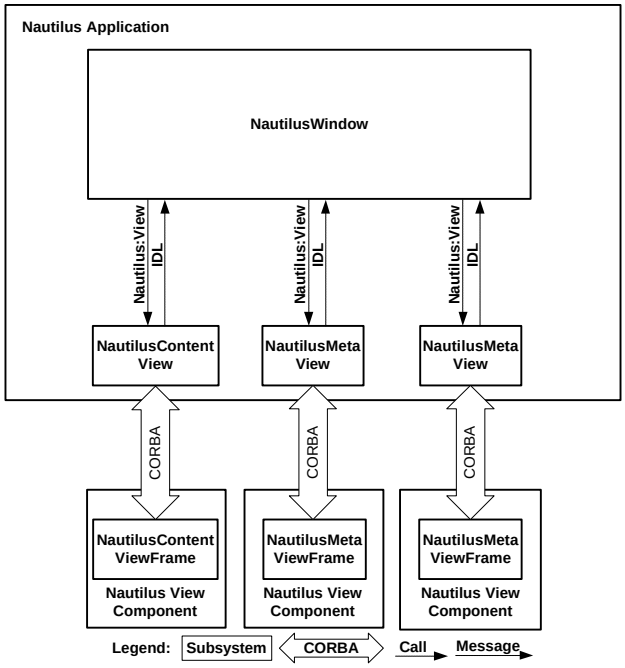

Figure 6. Nautilus conceptual architecture

NautilusView objects proxy all of the communication from the NautilusWindow to view components which use CORBA for communication. Nautilus:View IDL method calls arrive from NautilusWindow and are translated to the direct CORBA calls for the view components. The NautilusView objects also convert the returning CORBA calls into messages that correspond to the Nautilus:View IDL that are proxied back to the NautilusWindow layer.

NautilusViewFrame objects that are part of the view components accept the CORBA calls and convert them into the corresponding Nautilus:View IDL method calls. These objects also perform the backwards conversion whereas view component calls to the Nautilus:View IDL are converted into CORBA calls to the Nautilus application that are then accepted by the NautilusView object as returning messages.

5.2. Identifying key use cases

The Focus methodology identifies the key use cases by analyzing the user-level behavior of the application or the desired evolution requirements. For this process, Focus relies on the user-level documentation, which is not easily available for Nautilus. Instead, by examining the behavior of Nautilus through its common uses and related work on the file manager functionality [15], the key use cases were identified as follows:

- changing the current URI for the current window,
- changing the current URI into a new window,
- updating all visible views,
- showing status messages, and
- stopping the URI loading process.

These are by no means the only system-level use cases for Nautilus; rather they represent the focus of our architecture recovery efforts on the key functionality that is currently most relevant for the software evolution and program comprehension of the file managers [15].

5.3. Analyzing component interactions

The Focus method uses the identified use cases and analyses their interactions to discover more details on the architecture of Nautilus. Two steps in this process are

- the identification of the control flow among functions, and
- abstraction of this flow to the component level.

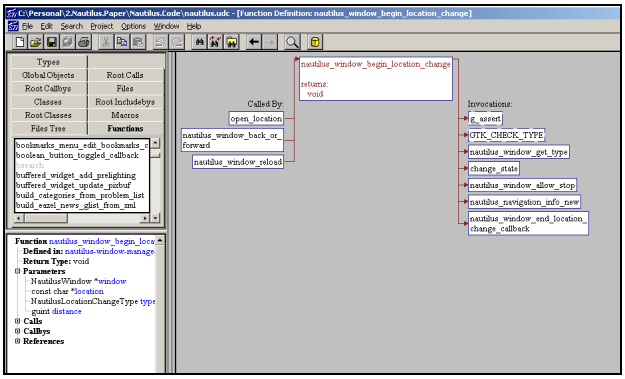

Figure 7. Code navigation in Understand for C++

Since Nautilus is a large application with over 250 KLOC, the "Understand for C++" tool [6], which provides efficient code navigation and analysis of the C/C++ code bases, is used to help with the analysis. Using this tool, typical scenarios for the identified key use

cases are traced and the traces of these scenarios are abstracted to the component level and presented through the UML sequence diagrams for the verification purposes. An example sequence diagram is shown as Figure 8.

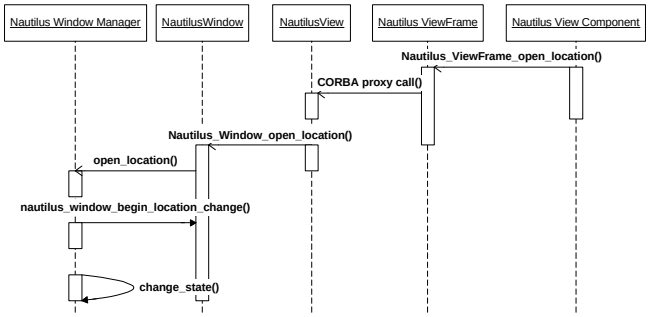

Figure 8. Changing the current URI

Concretely, in addition to the components found through the conceptual architecture recovery, this step indicated the existence of the Nautilus Libraries and Nautilus Window Manager components and their important interactions with other components. In particular, Nautilus Window Manager is recognized as a controller for the NautilusWindow and related NautilusView objects that deals with all of the changes in status on these Nautilus objects.

5.4. Generating intermediate architecture

Instead of separately developing an idealized architecture before the actual system structure is identified and consequently mapping the identified components onto this idealized architecture, our combined approach performs these two steps at once. The Chiron-2 architectural style for GUI software [9, 10] is used as a basis for the model that satisfies the recovered higher-level structures in Nautilus. This intermediate model is shown as Figure 10. This model does not cover the lower level structures in Nautilus that were not emphasized through the key use cases, so they are analyzed whether they contain any dynamic linking and if not are analyzed in the next step of our combined approach.

5.5. Using static analysis where applicable

For the statically linked lower-level components of the overall software architecture, the PBS tool is now used to extract the internal structure. The strength of this tool is the capability of total code coverage, and the detailed analysis of the underlying relations with an automated analysis and architecture extraction scripting support. This advantage will be used to deal with the size of the application's code base that would make manual use-case modeling approach unfeasible.

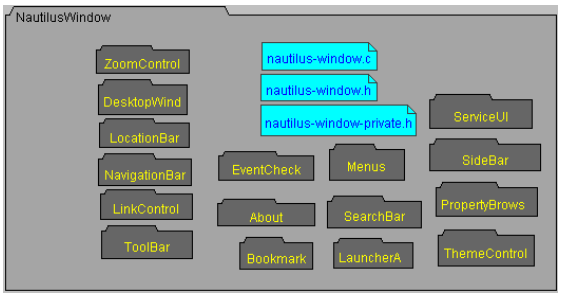

Figure 9. NautilusWindow architecture

Concretely, the NautilusWindow component was easily clustered as a part of the Nautilus Application and it contains various window elements and controls (e.g., side bar, location bar, menus). Figure 9 represents the internal structure of the NautilusWindow. Other elements such as the NautilusView objects are not as easily identifiable since their code is split among the libraries subsystem – an implementation specific structure – and the cluster identified as their actual code.

5.6. Generating revised architecture

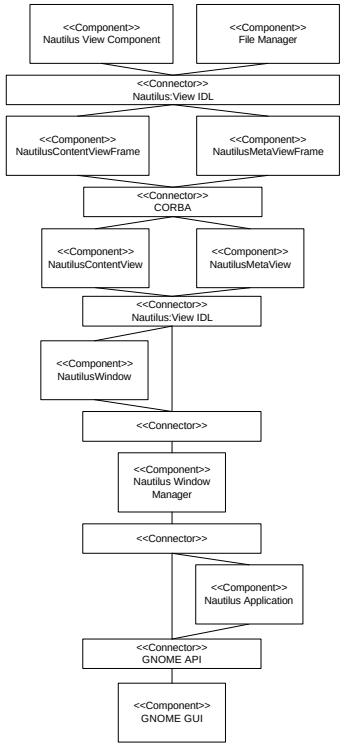

Figure 10. Refined Nautilus architecture

The resulting combination of the higher level as shown in Figure 10 and the lower level models as contained in the PBS tool represents the revised architecture of Nautilus. This architecture can be exported into a unified format such as UML since most the extracted architecture artifacts are already in UML and since the PBS supports exporting into recognized exchange formats such as TA or GXL [16].

6. Conclusions

This paper presents a case study in architecture recovery of a dynamically linked software system. Due to the inability of the existing frameworks to deal with this class of systems, a hybrid architecture recovery methodology called Dynamo-1 was proposed, and the analysis of Nautilus represents its initial case study.

The results of the architectural analysis of Nautilus showed that our proposed method when applied to a complex system such as Nautilus could offer successful focused architecture recovery. However, complete architecture recovery, which is analog to the results produced by the PBS tool, could be unfeasible due to the lack of automation in the use-case identification and call tracing, which is a crucial part of our method.

Future work on Dynamo-1 includes more case studies to confirm its validity, and development of a supporting technology that would allow more automated use-case extraction and that would make our method capable of successful complete architecture recovery.

7. References

[1] J. Bergey, D. Smith, N. Weiderman, and S. Woods, "Options Analysis for Reengineering (OAR): Issues and Conceptual Approach", *SEI Technical Report CMU/SEI-99-TN-014*, Software Engineering Institute, Carnegie Mellon University, Sep 1999.

[2] M. Shaw, and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[3] I.T. Bowman, R.C. Holt, and N.V. Brewster, "Linux as a Case Study: Its Extracted Software Architecture", *In Proceedings of the ICSE 99*, Los Angeles, May 99.

[4] "GNOME Resources", Online, www.gnome.org, 2002.

[5] "Nautilus Development", Online, nautilus.eazel.com, 2002.

[6] "Understand for C++" Analysis Tool, Scientific Toolworks Inc, Online, www.scitools.com/ucpp.html.

[7] R. C. Holt, "PBS: Portable Bookshelf Tools", Online, http://www.swag.uwaterloo.ca/pbs/, 2002.

[8] L. Ding, and N. Medvidovic, "Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution", *In Proceedings of the 2001 Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, the Netherlands, August 27-29, 2001.

[9] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. "A Component- and Message-Based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pages 390-406, June 1996.

[10] N. Medvidovic, "Formal Definition of the Chiron-2 Software Architectural Style", *Technical Report UCI-ICS-95-24*, Department of Information and Computer Science, University of California, Irvine, August 1995.

[11] A. E. Hassan, and R. C. Holt, "A Reference Architecture for Web Servers", *In Proceedings of the Working Conference on Reverse Engineering 2000*, Brisbane, Australia, November 2000.

[12] V. Tzerpos, and R. C. Holt, "A Hybrid Process for Recovering Software Architecture", *In Proceedings of the CASCON 1996*, Toronto, Canada, November 1996.

[13] R. Kazman, L. O'Brien, and C. Verhoef, "Architecture Reconstruction Guidelines", *SEI Technical Report CMU/SEI-2001-TR-026*, Software Engineering Institute, Carnegie Mellon University, August 2001.

[14] "The Dynamic Linking Extension", Online, *A White Paper from the X/Open Base Working Group*, The Open Group, March 1997.

[15] N. Berzukov, "The Orthodox File Manager (OFM) Paradigm", Online, www.softpanorama.org/OFM/Ofm_00.shtml.

[16] "Graph Exchange Language (GXL)", Online, www.gupro.de/GXL/