Obtaining Architectural Descriptions from Legacy Systems: The Architectural Synthesis Process (ASP)

A Thesis
Presented to
The Academic Faculty

by

Robert L. Waters

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
October 2004

Obtaining Architectural Descriptions from Legacy Systems: The Architectural Synthesis Process (ASP)

Approved by:

Professor Gregory Abowd, Advisor

Professor Michael McCracken

Professor Spencer Rugaber

Professor Rick Kazman
(Carnegie-Mellon University)

Professor Colin Potts

Date Approved: 21 October 2004

To my wife,

who followed me all over the world

during my military service,

and patiently waited while

I decided what I wanted to do

when I grew up.

PREFACE

In the early to mid twentieth century, many reputable engineers and scientists considered exceeding the speed of sound as being impossible. In fact, the popular literature referred to the sound "barrier" as something that could never be broken. Jet aircraft would approach that *barrier* and yet could not seem to break through. It was not until the principle of area ruling was discovered, that aircraft could slip into Mach flight with apparent ease.

In the same way today, many reputable computer scientists feel the concept assignment barrier will never be bridged in an automated manner. There are too many challenges to emulating those incredible matching and analytic abilities that are inherent in the human brain.

I set out in this research effort to show that it was in fact possible to break through the concept assignment barrier and do it in an automated manner. Unfortunately, I was not able to achieve that lofty goal, but I hope this work does move us one step closer to the day when our software does conceptual analysis with the same ease as the human brain does today.

ACKNOWLEDGEMENTS

As most students, I have to acknowledge the extensive help and guidance I received from my committee members. I also acknowledge their continued support and encouragement when I was ready to throw in the towel.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

A majority of software development today involves maintenance or evolution of legacy systems. Evolving these legacy systems, while maintaining good software design principles, is a significant challenge. Research has shown the benefits of using software architecture as an abstraction to analyze quality attributes of proposed designs. Unfortunately, for most legacy systems, a documented software architecture does not exist. Developing a good architectural description frequently requires extensive experience on the part of the developer trying to recover the legacy system's architecture.

This work first describes a four-phase process that provides a framework within which architectural recovery activities can be automated. These phases consist of: extraction (obtaining a subset of information about the legacy system from a single source), classification (partitioning the information based upon its viewpoint), union (combining all the information in a particular viewpoint into a candidate view), and fusion (cross-checking all candidate views for consistency.

The work then concentrates on the major problem facing automated architectural recovery—the concept assignment problem. To overcome this problem, a technique called semantic approximation is presented and validated via experimental results. Semantic approximation uses a combination of text data mining and a mathematical technique called concept analysis to build a lattice of similar concepts between higher-level domain information and low-level code concepts. The experimental data reveals that while semantic approximation does improve results over the more traditional lexical and topological approaches, it does not yet fully solve the concept assignment problem.

CHAPTER I

INTRODUCTION

Although the exact percentage varies, most researchers agree that somewhere between 40 and 80 percent of software development activities are focused on maintenance, enhancement or evolution of existing systems [15]. These existing systems are typically referred to as legacy systems. Historically, software engineers working on these legacy systems try to recover information from the source code or documentation to understand the system well enough to make required repairs or enhancements. Often these changes were made with minimal understanding of the impact they had on overall system quality over time. This led to the characterization of many legacy systems' structure as "spaghetti code." Recently the emphasis in reengineering of legacy systems has shifted to applying more of an organized forward engineering approach to their modification or enhancement [9].

The Software Engineering Institute has recently emphasized the importance of using architecture as a vehicle for guiding a reengineering effort. In describing the ten most important reasons that reengineering efforts fail, failure to take an architectural approach is identified as one of the most critical errors [8].

It is only natural then, that recovering the software architecture of a legacy system is an important task. Unfortunately, it is not an easy task. Alexander Ran, project architect of the ARES project summed up the state of the art when he said [17], "the major problem is managing information about software from various sources. There is a need for creation, management, and use of a software information base, for multiple views of software systems and most often a single, non-representative view is adopted. Consequently the view is not useful and is not used." Many projects may develop a software architecture, but few seem to develop anything actually useful.

1.1 Problem Statement

The overall problem of software architectural recovery is a two-edged sword. First it requires an analyst to have experience in both architecture and the application domain itself. Secondly, the low-level concepts embodied in the implementation must be mapped to the higher-level domain concepts. Figure Often, architectural recovery is as much an art as an engineering discipline. Two possible ways to attack these problems are to communicate the "art" of architectural recovery to software engineers through education and training, or to increase the amount of an architectural description that can be automatically recovered. This dissertation examines the latter choice.

The ultimate long-term goal of this research is to allow a software engineer to provide information about a system—its documentation, source code and design—and have a complete and useful architectural description be automatically generated. To achieve this overall goal, many sub-problems need to be addressed:

1. The architectural information is spread across a multitude of sources. These include textual information sources such as domain and user documentation, original design documents, the source code itself, and even partially-formed ideas in the heads of the original developers.

2. The software engineer recovering the architecture is frequently not a domain expert. Since the concepts embodied in the top level of the architecture use concepts from the domain, this can present a real problem. Architectures are ultimately recovered to aid in accomplishing the business objectives of an organization. The time necessary for developers to become acquainted with an application domain could better be spent achieving other business objectives.

3. Systems evolve over time. There is drift and erosion of the original architecture [57], and thus there are potential inconsistencies between the information obtained from different sources. At the early stages of the recovery effort, it is almost impossible to determine what the inconsistencies are, but as the effort progresses and more information is obtained, these inconsistencies can be discovered and corrected.

4. No one person has a complete picture of the architecture of any non-trivial system. The older and larger the legacy system is, the more this is true. An analyst frequently has to integrate information from several original developers, each of which is familiar with only a subset of the overall system.

5. The source code contains the ultimate truth of system functionality, but because of the concept assignment problem [10], it is not a trivial task to map the code to architecture. The architecture frequently uses the language of the domain and user, while the code uses the language of the implementation and developer. This boundary between architectural abstraction and implementation is referred to as the concept assignment boundary. (The phrase *concept assignment* comes from the problem of assigning abstract concepts to specific sections of code). Most architectural recovery research has focused on other areas than bridging this gap [24].

Crosscutting concerns and nonfunctional requirements embedded in the code are especially difficult to ferret out [5]. The term *crosscutting* is borrowed from aspect-oriented programming [20] and refers to features whose implementation is distributed across multiple architectural modules. For example, security might be a nonfunctional requirement, while activity logging would be a crosscutting concern supporting security. Detecting these concerns and recognizing them in the implementation becomes an important task in understanding how specific nonfunctional requirements and cross-cutting concerns might have been embodied in the code.

These problems lead to the following research questions:

1. How can we integrate architectural information to derive a more complete architectural description informed by multiple sources?

2. How much of this integration can we automate?

3. How can we overcome the concept assignment problem, that is, how can we assign implementation details to higher-level abstractions in an automated manner?

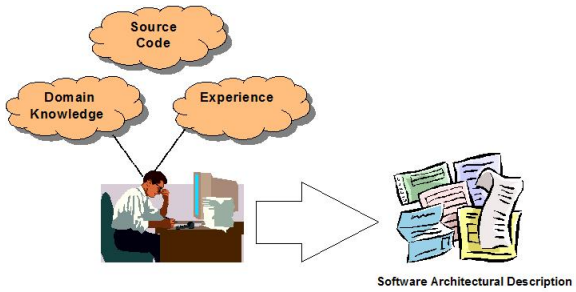

Figure 1: Current Recovery Methods

4. How can we identify crosscutting concerns (design concepts whose implementation spreads across multiple components) in an automated manner?

5. How can we identify and eliminate inconsistencies in the information obtained from multiple sources?

This dissertation research examines each of these questions in varying levels of detail. We focus primarily on questions 3 and 4. The goal of this research is to improve the automation of architectural recovery and thus to move one more step from an art of experts to a defined and repeatable process for use by developers.

1.2 Thesis Statement

It is possible to improve the amount and quality of architectural information that is automatically recovered and integrated from legacy systems by using semantic approximation to bridge the concept assignment gap.

To evaluate this thesis, the following hypotheses were developed:

It is possible to define a process that can be automated, allows synthesis of multiple sources of information, identifies inconsistency, and improves the results of the recovery over the use of code-based approaches alone. Figure 1 shows the current state of architectural recovery techniques. These techniques rely heavily on the software engineer's expertise and level of domain knowledge to process and understand the source code and turn it into a software architectural description. Figure 2 depicts the ultimate long-term goal of this research-that of fully automated recovery. Due to a number of current technological and research shortcomings, this goal cannot currently be met. Figure 3 depicts the scope of

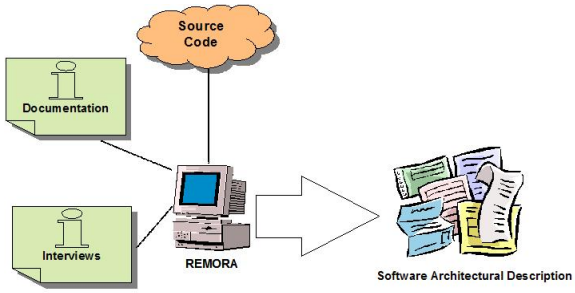

Figure 2: Ultimate Goal

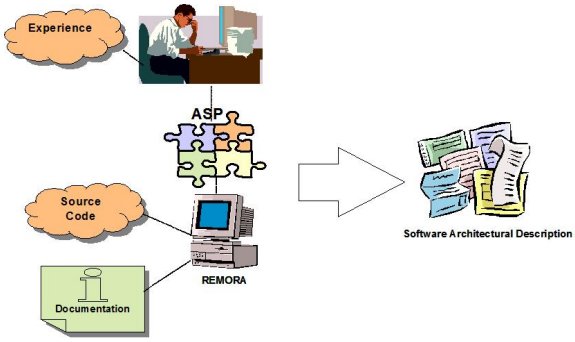

Figure 3: Goal of this Research

this dissertation. The puzzle labeled ASP represents the hypothesized process. This process uses information from a variety of sources, develops multiple views of the architecture, synthesizes the information and checks its consistency. The automated synthesis of information using various techniques is the focus of the remainder of this dissertation. The process includes a technique called *semantic approximation*, specifically developed during this research, that improves the quality of the information integration task.

Using semantic approximation provides better-automated synthesis of information than traditional lexical or topological approaches. Other researchers have focused on using techniques for combining or synthesizing architectural information using lexical or topological methods. These methods use superficial information to make this determination, disregarding particular semantic characteristics of the elements in the architecture. This dissertation discusses a technique that allows an approximation of an architectural element's semantics to be specified in a lightweight manner. This approximation is then used to aid lexical techniques to improve the accuracy of the combined information.

1.3 Contributions

The contributions of this research include:

1. An analysis and comparison of current architectural recovery techniques.

2. Definition of an architectural synthesis process that provides a framework for automating the recovery process.

3. Development of REMORA, a toolkit to support the architectural synthesis process and improve automated recovery of software architectures.

4. Definition of a technique called *semantic approximation* that improves the quality of the information-synthesis task, when the information comes from multiple sources.

5. Validation of the semantic approximation technique against traditional lexical and topological approaches.

1.4 Overview of Dissertation

Chapter II provides a common vocabulary and background in software architecture and then discusses and categorizes several typical architectural recovery approaches currently in use. Chapter III discusses in detail a defined process framework (ASP) within which software architectures can be recovered across a broad range of domains. This framework is the puzzle piece shown in Figure 5. Chapter III also discusses REMORA, a toolkit that supports the proposed recovery framework. Chapter IV of this dissertation provides an introduction to concept analysis followed by a detailed description of the Semantic Approximation technique for architectural information integration. Chapter V discusses the experimental validation of this technique for integrating functional information. Chapter V uses case studies of architectural recovery of the ISVis reverse engineering tool and the Linux operating system kernel as the basis for the experimental data. Chapter VI ends the dissertation with a discussion of the conclusions and directions for future work.

CHAPTER II

BACKGROUND AND ANALYSIS OF RECOVERY APPROACHES

In this chapter we provide three important prerequisites to understanding the remainder of this dissertation. First, we establish a common frame of reference and vocabulary concerning just what software architecture is. The software architecture literature and current discussions contain a wide variation in their definition of architecture and in their description of the contents of an architectural description. If we want to make any claims about recovering software architectural descriptions, we must have a common frame of reference. Since architectural information comes from multiple sources, we also introduce the idea of an information extraction space as a conceptual aid to thinking about where the architectural information that needs to be integrated comes from.

Secondly, we analyze current architectural recovery techniques. Recall our thesis statement is that we can improve the amount and quality of architectural information that is automatically recovered and integrated from legacy systems by using semantic approximation to bridge the concept assignment problem. To be able to judge whether this might be true, we must understand the current techniques in use and analyze their potential shortcomings.

Finally, for those readers interested in recovery as a practical matter, the analysis of approaches provides important information on types of views recovered, intermediate representations and visualization capabilities of each of the techniques. For those readers desiring a quick overview and reference, the analysis of approaches is summarized in Table 2.

2.1 The Concept Assignment Problem

We have already alluded to the concept assignment problem in the introductory chapter. According to Biggerstaff, the *concept assignment problem* consists of discovering human-oriented concepts and assigning them to their realizations within a specific program. Solving this problem forms the basis of most architectural recovery efforts. A central hypothesis of Biggerstaff's work is that a parsing-oriented (code-based) approach is necessary but not sufficient for solving the general concept-assignment problem.

Biggerstaff's approach to solving this problem used a toolsuite called DESIRE (Design Information Recovery Environment). DESIRE works as a human-in-the-loop system to help reverse engineer source code. As contrasted with our automated approach, DESIRE requires the the human to provide the domain level concepts, and to perform many of the mappings.

2.2 A Software Architecture Primer

2.2.1 Definition of Software Architecture

To understand both the benefits and the complexities of architectural recovery, one must first understand what constitutes a software architecture. Two seminal papers were published in the early nineties that attempted to define the phrase *software architecture*. Garlan and Shaw [27] define software architecture as comprising *components* (elements which provide computation services or passive data stores), *connectors* (elements which provide interactions between the components such as protocols) and *configuration* (the topology of the system). The authors also introduced the idea of *architectural styles*. Styles are commonly occurring configurations in which the components and connectors interact according to an agreed-upon set of constraints. Common styles include pipe and filter, implicit invocation and layered.

Perry and Wolf [57] take a slightly different approach to the definition of software architecture. Their definition couches software architecture as a three-tuple consisting of elements, form and rationale. Elements encompass the components and connectors of the Garlan and Shaw view. Form is similar to the idea of configuration, but also includes the

idea of constraints similar to those in Garlan and Shaw's idea of styles. Rationale is not explicitly accounted for in the Garlan and Shaw notion of architecture and embodies the design choices made in defining the architecture. For whatever historical reasons, the Garlan and Shaw notion of architecture has become the more generally accepted one. Many author's today use the Bass et al. [6] definition (which is a refinement of the Garlan and Shaw definition) in their work:

> The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

IEEE Standard P1471, Recommended Practice for Architectural Description (1998), defines architecture as the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution. This definition includes the Bass definition and explicitly captures the ideas of rationale from Perry. It also adds the important notion of understanding how the software system (via its architecture) interacts with its external environment. When we use the phrase software architecture in the remainder of this dissertation, it carries the meaning of the P1471 definition.

2.2.2 Views and Viewpoints

Just as a building architect needs multiple diagrams to describe the structure of a complex building, a software architect often needs multiple descriptions of a software system. *Architectural views* reflect a set of specific interests that concern a given group of stakeholders [42] applied to a specific representation of an architecture. Perry and Wolf also discuss the need to provide multiple views of an architecture. Typical views include:

1. Physical (Hardware): This view maps software onto hardware. Physical views are especially useful for depicting the context of the software architecture as part of the overall system's architecture. Components in this view are typically hardware devices such as processors, while connectors are communications paths.

2. Logical (Conceptual, Functional): This view depicts the software as a set of components cooperating to fulfill the functional requirements of the system. Components in this view usually provide either active computation services or passive data storage. Connectors are typically data or control flows between the components.

3. Module (Development, Code): This view depicts the actual implementation structure of the software system. Components in this view are usually source file directories and code modules. Connectors represent "uses" or "depends on" relationships.

4. Process (Coordination, Execution, Runtime): This view depicts the run-time behavior of the system. Components are processes or threads and the connectors are inter-process communication (IPC) mechanisms.

The P1471 standard also refines the idea of Krutchen's [42] view into a view and viewpoint. A *viewpoint* is similar to the idea of general views described above. A viewpoint is basically a model or set of rules for analyzing, building and interpreting views. A viewpoint can be instantiated through one or more views. A view in P1471 is the representation of a specific viewpoint for a specific software architecture. For example, a data viewpoint might be supported with an ER diagram as a view. We will use terminology consistent with the standard for the remainder of this dissertation.

2.2.3 The Information Extraction Space

Figure 4 depicts what we call the extraction information space. This extraction space is simply a qualitative visualization of the different sources of architectural information that might be used during a recovery effort. The labels on the graph are adapted from a paper by Egyed [18]. Sources of architectural information provide various amounts of coverage over the information space. For example, architectural meta-models provide abstract concepts such as the idea of components, connectors and viewpoints, and this type of information is also generic across domains. On the other hand, interviews with human experts about a system can span a variety of topics and thus might cover both generic and specific topics about the system at an abstract or concrete level. Examining the information space graph

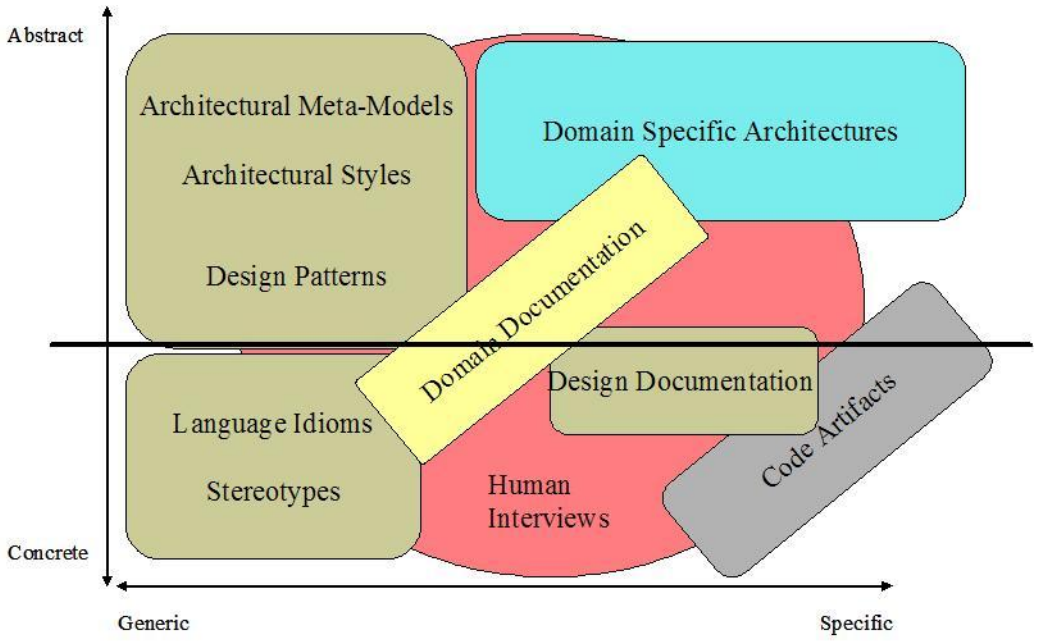

Figure 4: The Extraction Information Space

helps reinforce the idea that code analysis alone is not sufficient to provide the high level concepts associated with software architecture. Architectural information lives in the upper half of the graph, while code-based implementation information resides in the lower half. Spanning that gap is the concept-assignment problem.

2.3 Analysis of Architectural Recovery Approaches

We now analyze various approaches being taken to recover software architectures from legacy systems. Each of these approaches will be discussed in terms of their underlying support for assumptions about architectural recovery, architectural views developed (which viewpoints are supported), visualization support, sources of information, compliance with P1471, and ability to make claims about coverage of the architectural information space and consistency checking between recovered views. We classify these approaches into three broad categories: Pattern-based recovery, Visualization-based recovery and Process-based recovery.

2.3.1 Pattern-Based Recovery

One group of techniques is centered on variations in the idea of pattern detection—the premise that we can find the architecture of a system by finding standard styles or patterns that were used to originally develop the software. Techniques in this category include CANTO, ManSART, Dali and ARM. We now examine each of these techniques individually.

The Code and Architecture Analysis Tool (CANTO)[22] [23][4] detects clichés in source code. Clichés are recurring patterns in implementation which represent commonly used programming abstractions. For instance C programmers writing a server might create a socket, then call the `listen` function to wait for connections and finally use an `accept` function to handle client connection requests. If we could find these sequences of instructions in a code module, it would indicate a server cliché and we could infer a higher-level architectural structure. CANTO uses the Refine [34] code analysis tool to extract an abstract syntax tree (AST) that is then traversed and analyzed to find the clichés necessary to build up the architectural description. Not surprisingly, the intermediate representation for architectural analysis is an annotated AST. A CANTO representation is made up of two views: a module view depicting major code module relationships and a task view that gives a runtime or dynamic view of the architecture. These views are presented visually through use of the ATT dotty [25] graph layout program. Since dotty is primarily non-interactive, this technique of visualization limits user interaction with the graphically displayed architecture.

CANTO has several shortcomings that impact its ability to be a total solution for recovering P1471 compliant architectural descriptions. First it develops only two of the four principle architectural viewpoints. The physical and logical views are not supported. Secondly, it depends totally on the Refine toolkit for code analysis, and all representations are derived from this code analysis. This limits its completeness in that input from the original human designers or design documentation is not explicitly considered. Third, the use of an AST as an intermediate representation necessarily limits CANTO's analysis engine to operations on code-like structures. Finally, there is no direct support within the CANTO tool for attempting to maintain consistency between views.

The Mitre Software Architecture Recovery Tool (ManSART) [68] [14] [13] shares many

of the same characteristics as CANTO. ManSART uses recognizers to recover architectural features that then serve as abstractions within the architectural representations created. Like CANTO, ManSART uses the Refine toolkit to build an AST representing the source code for the system. The recognizers then traverse the AST looking for patterns of control and data flow that might indicate architectural-level information. ManSART supports several views of an architecture, although many (like the call-graph) are more for program understanding in-the-small rather than architectural-level understanding. The principle architectural views supported by ManSART include what they call the task-spawning (process) view, repository, service-invocation and abstract data type views. A combination of these last three would support the standard logical viewpoint of an architecture. After an initial automated analysis by ManSART the architecture is visualized as a graph. Like CANTO, the graph visualization is basically a static view. Activating user-defined functions (like containment operators) does manipulate the architecture on the displayed graph, albeit not interactively. To manipulate the visualization, a function is applied to the representation, the graph is recalculated and redisplayed. The intermediate representation for ManSART is called analysis module interfaces (AMI). AMI is an Architectural Description Language (ADL)-like language that allows for the hierarchical specification of analysis results. Like CANTO, ManSART's representations are derived solely from source code artifacts and ManSART's built-in recognition rules. Unlike CANTO however, ManSART does attempt to provide some consistency between its developed representations by allowing views to be merged.

Most of ManSART's shortcomings are identical to CANTO's, which is not surprising given the similarity of their design philosophies. ManSART provides no support for arbitrary or user-defined viewpoints of an architecture. It does provide a view that merges the module and logical viewpoints of the architecture. It is interesting to note that both CANTO and ManSART stress the importance of the process (or runtime) view of the architecture, yet both use only static code analysis techniques to derive their information.

Dali [37][36] (which does not stand for anything in particular) takes a different tack than either CANTO or ManSART. Dali recognizes the limitations of relying on a single tool for

extracting architectural information and thus uses the philosophy of an open workbench to integrate other tools into its functionality. Dali uses a database as the integration mechanism; thus its intermediate representation is the database tables that record the information from the tools it uses. Standard Dali extraction and analysis tools include the Lightweight Source Model Extraction (LSME) [55], gprof, Reflexion Model Tool (RMTool) [54], and IAPR [38]. The Rigi [67] environment provides Dali visualizations. Unlike CANTO and ManSART, Rigi allows an analyst to directly manipulate the architectural visualization and create new views via drag-and-drop interactive manipulations. Like ManSART, the analyst manipulates the architecture primarily by invoking functions to be performed on the current view. These functions are typically SQL and RCL (Rigi Control Language) scripts which describe how to retrieve and consolidate architectural information stored in the database. DALI provides logical and runtime views of the architecture, but the views developed are driven by the types of tools integrated into the workbench. Dali is sufficiently open that any given view could be presented within the workbench using a technique the author's call *view fusion*. By supporting fusion, Dali allows verificationof consistency between different views of the recovered architecture.

Dali attempts to overcome the shortcomings of depending on a single tool but still fails to account for all possible sources of architectural information. Like CANTO and ManSART, its explicit information source is the system's source code. It is important to note that in the example usages, the analysts have used domain information and user interactions to obtain the high-level architectural description. The effectiveness of Dali is somewhat limited to the experience of the analyst and his ability to create the scripts and queries that will reveal the needed architectural information.

The Architecture Recovery Method (ARM)[30] is an extension of Dali by an integrated methodology and by additional tools added to the open workbench. The workbench is also supplemented by prepared scripts and heuristics, which allow less experienced analysts to achieve adequate results with the tool. ARM encapsulates common design patterns into standard SQL and RCL queries allowing semi-automated recovery of architectural views. ARM, like Dali, is still source-code bound and thus does not take explicit advantage of all

available architectural information.

To summarize the pattern-based techniques, they extract information using a limited set of tools that emphasize various types of code analysis. The number and types of views that can be supported are driven by the tools' designs. Consistency checking is also implicit in the use of the tools. Architectural views produced are consistent primarily because they obtain information from only one source (the source code of the system) and use that information to develop only one or two views.

2.3.2 Visualization-Based Recovery

Rather than looking for common patterns semi-automatically as the first group of tools and methods did, this next group of tools uses information visualization as the key component of the recovery activity. These tool designers feel that if information is presented in the proper format, a human is the best pattern detector possible. FEPPS, ISVis, Shimba and the Software Bookshelf best typify this category of tools.

The Flexible and Extensible Program Comprehension and Support System (FEPPS)[43] provides a three-dimensional (3D) viewing and manipulation interface for architectural information. Using this system, an analyst recovers the architecture by viewing complex relationships among elements of architectural information. This information comes from ASTs, program slicing and control-flow graph(CFG) and data-flow graph(DFG) analyses, (Note that this technique requires compilable code). This information is stored in a multi-layer, multi-representation (MLMR) graph format. The fundamental elements of FEPPS data are the function elements and file elements within the source code. These are then presented visually for manipulation based upon analyst-specified commands entered in a navigation control. FEPPS supports both a module and logical viewpoint of the architecture. As we have seen in all the other tools so far, FEPPS suffers from a lack of completeness because it fails to explicitly use any information other than code-based information in the recovery process.

The Interaction Scenario Visualizer (ISVis)[35] is the first of the recovery tools that tries to incorporate dynamic code information with static information. An analyst first

instruments the system's source code and then executes the program to capture an event trace. A static analyzer such as ctags is used to extract static code information, which is then used to correlate the event traces. The event trace and static information are then imported into the ISVis visualization environment. The analyst uses the ISVis visualization to extract architectural components and connectors based upon the captured events from the system's execution. ISVis uses an object-oriented (OO) data model as its internal representation. Its visualization technique is one of the only ones that does not use a graph model. Instead the event traces are displayed in a custom 2D widget called an information mural. The analyst cannot directly manipulate the visualization, but rather manipulates the OO model, which then updates the mural. ISVis provides a single architectural view of the run-time organization of the application. Its overall effectiveness is highly dependent on an analyst's ability to interpret and effectively use the mural to interpret the event data.

The third technique is Shimba [62]. Although Shimba is restricted to the reverse engineering of Java applications, it is a good example of the use of dynamic program execution information in recovering high-level program structure. Shimba uses Rigi (as described above in DALI) for the visualization of static program information. It uses SCED [39] for the visualization of event trace and dynamic program information. The intermediate representation for recovered information is a combination of RSF (rigi standard format) and SCED' custom format for the dynamic information. The interesting part of Shimba is that it allows the analyst to define dynamic scenarios in more explicit manner than ISVis, and uses automated techniques to build state machines representing the dynamic execution of Java programs. Also, in contrast to ISVis, Shimba allows dynamic actions to be isolated to one component, which could prove useful for discovering connectors in a legacy system. This technique produces two views, a static structural view similar to a logical viewpoint and two dynamic views (state and scenario) which both map to the runtime viewpoint.

The fourth visualization-intensive technique is the Software Bookshelf or Portable Bookshelf (SBS/PBS) [61]. SBS uses a web-browsing paradigm to present architectural information. An analyst can then select any of the elements displayed, which then takes him through a hypertext link to the appropriate place in the architecture. SBS is populated

using static code analysis tools. The visualization technique is a combination of a graph representing the architectural structure and textual information supporting the nodes and edges of the graph. The internal representation is kept in relational tables, which are accessed via GCL (a query language for information retrieval). SBS can export and import information using a storage scheme known as TA (for tuple algebra). SBS supports a single architectural viewpoint corresponding most closely to the module view. Since SBS develops a single viewpoint consistency is implicit in the technique. Recently, SBS/PBS has been renamed Swagkit [33], but is essentially the same technique.

2.3.3 Process-Based Recovery

All of the method and tool strategies mentioned so far have had the same weakness-dependence on the source code of an application as the primary source of information for architectural recovery. Process-based approaches try to incorporate explicitly information from other sources in the information extraction space. The principle examples in this category are Krikhaar's Reverse Architecting Approach, SAAM, ARF and Hybrid.

Krikhaar's Reverse Architecting Approach (RAA) [40], consists of three phases, extraction, abstraction and presentation (these steps are common to Dali and SBS also). After obtaining architectural information from source code, documentation and human experts, the information is grouped and filtered to obtain a relevant subset of the information. Finally the information is presented to the analyst in prototype visualization environment called Teddy.

Information extraction from the source-code uses a lightweight method to find source code relations such as imports, part-of and uses. This information is then stored and manipulated in the relation partition algebra. This algebra allows the analyst to use specific operators to abstract (or lift) the information to obtain a view of the architecture that seems to correspond closest to the module view.

The Software Architecture Analysis Method (SAAM) [3] while not primarily an architectural recovery strategy, does produce architectural descriptions in a human-centric fashion. A facilitator leads key stakeholders in the development of an architectural representation.

Extraction is primarily performed though shared consensus building. The representation is usually a diagram drawn on a white board or easel. Usually a single viewpoint is developed which corresponds to the logical viewpoint. The degree of consistency of the architectural information is dependent on the quality of the knowledge of the stakeholders participating in the SAAM session.

The architectural recovery framework (ARF)[7] [19] attempts to build a single viewpoint representation of an architecture by integration of information from multiple sources. It represents information internally using the DARWIN [47] ADL. The viewpoint recovered seems to support predominately the logical viewpoint of the architecture. This technique is essentially a manual one, so the visualization is constructed by displaying the DARWIN model using a specialized tool. There is no interactive capability yet, so the analyst must modify the model off-line and then use the DARWIN display tool to redraw the view. ARF has a clear methodology and therefore should be usable by practitioners in the field without extensive training.

ARF builds a single viewpoint of the architecture normally expressed in terms of a quality attribute such as safety or security. It tries to build as complete a model as possible of this one area. There is no attempt to make any statement about the overall completeness of the Architectural Description (AD). While ARF looks at multiple sources of information, it builds only one viewpoint, which limits its ability to check for inconsistency in the AD.

The Hybrid process [64] combines source code derived information with developer interviews to produce a representation of the recovered software architectural description. They define an iterative process, supported by SBS, that consists of the following steps:

1. Choose a domain model. This step in the language of P1471, determines which viewpoint you are interested in recovering.

2. Extract facts from source code. This corresponds to the extraction phase of the other techniques discussed.

3. Cluster into subsystems. In the language of P1471, this step builds the components and connections for the view that will support the viewpoint chosen in step 1.

4. Refine clustering using information derived from developer interviews.

5. Refine the layout to accommodate new information.

Since the tool support is based on SBS, the internal representation is also the Tuple Algebra (TA). Consistency in the Hybrid approach is obtained manually by discussing the emerging architectural description with the human experts on the system.

2.3.4 Summary of Recovery Techniques

Table 2 and Table 1 summarize each of the recovery strategies based upon the elements of our conceptual framework. All have a related information extraction phase using either custom-designed or open toolkits. None have an overt classification phase since each strategy is oriented towards developing a specific set of views rather than being a general-purpose technique. In this case, the designer of the recovery strategy has performed the classification task implicitly by designing the extraction task to get information for a particular set of viewpoints. The union phase is also fairly constrained in most of the approaches, since information is extracted from a single source to support a single view. Finally, with the exception of Dali, fusion of views for consistency checking is either implicitly accomplished by the tool by constraining the types of viewpoints and information sources, or it is accomplished by discussing the derived viewpoints with human experts. Like the strategy Dali takes, any new architectural recovery technique or process needs to leverage work already done well by other tools and techniques. This fact makes this particular analysis useful for identifying intermediate representations produced by other tools. These representations can then be imported and analyzed in support of any new techniques or frameworks that might be proposed.

As evidenced by the wide range of approaches, and differing areas of interest in each approach, architectural recovery strategies are highly influenced by the preconceived ideas of the original developer. When a particular strategy constrains the information sources and the developed viewpoints, the ability of the strategy to produce compliant architectural descriptions becomes constrained. Only one (Dali) of the architectural recovery strategies reviewed support explicit techniques for consistency checking or development of arbitrary

views/viewpoints. Likewise, few attempt explicit coverage of the extraction information space. This explains why some experts liken architectural recovery to an art form. Those analysts who are good at creating architectural descriptions usually have knowledge and experience that cover the other three quadrants of the information space that do not involve specific code-based knowledge.

From this analysis, we believe there is a need to define a generalized framework, into which we can place the output of any or all of the techniques analyzed in this section, so that we can leverage off their strengths and mitigate their weaknesses. We define just such a process framework in the next chapter.

Table 1: Summary of Architectural Recovery Strategies, Pattern and Visualization Based

Recovery Strategy/ References	Explicit Information Sources	Internal Representation	Viewpoints Views Supported	Visualization Support	InterView Consistency Mechanism
CANTO /[22] [23][4]	Source Code	Annotated AST	Module Task	Dotty	Implicit
ManSart/ [68] [14] [13]	Source Code	Analysis Module Interfaces(AMI)	Task-Spawning Repository Service-Invocation ADT	Static Graph	Implicit
DALI/ ARM/ [37][36]	Source Code	Database	Logical Runtime Module	Rigi	Partial
FEPPS/ [43]	Source Code	Multi-Layer Multi-Graph (MLMG)	Module Logical	Custom 3D Display	Implicit
ISVis/ [35]	Source Code Execution Traces	Internal OO	Logical Runtime	Information Mural MSC	Implicit
Shimba/ [62]	Java Source Code and Dynamic call Information	RSF and SCED	Logical Runtime	Rigi and SCED	Partial

Table 2: Summary of Architectural Recovery Strategies, Process Based

Recovery Strategy/ References	Explicit Information Sources	Internal Representation	Viewpoints Views Supported	Visualization Support	InterView Consistency Mechanism
RAA/ [40]	Source Code Human Experts	Relational Partition Algebra	Module	Teddy Graph	Implicit
SAAM/ [3]	Human Experts	Paper/ White-Board	Any based on Participants	Paper Graph	Implicit
ARF/ [7]	Source Code Documentation Human Experts	DARWIN ADL	Centered around specific NFR	Darwin renderer	Implicit
Hybrid/ [64]	Source Code Human Experts	See SBS	Module	See SBS	Developer Interviews

CHAPTER III

DESCRIPTION OF THE ARCHITECTURAL SYNTHESIS PROCESS

Recall our thesis statement.

> *It is possible to improve the amount and quality of architectural information that is automatically recovered and integrated from legacy systems by using semantic approximation to bridge the concept assignment gap.*

In order to reach the ultimate goal of someday having fully automated architectural recovery, we must first define an overarching framework for software architectural recovery. By defining a high-level process that encompasses all facets of recovery, we have a framework which provides a basis to reach that ultimate goal. The remainder of the dissertation will look at validating the thesis statement. This chapter defines the overarching framework within which the subsequent work described in chapter VI is situated.

3.1 Process Overview

Architectural information exists in many locations and can be derived from many sources, as we discussed in Chapter II in the section on the information extraction space. Unfortunately, many current reverse engineering and architectural recovery techniques explicitly rely on code-based recovery strategies. While these approaches use the most authoritative source (the code itself), they do not lend themselves to identifying the original high-level abstractions the designers intended the system to have or to mapping the low-level source code information to these higher level concepts. The latter mapping problem is commonly referred to as the concept-assignment problem which was discussed in the previous chapter.

A comprehensive strategy for architectural recovery should be realized through a defined, tailorable process for obtaining architectural descriptions. This process should meet several

general goals. The target process should be:

- *Fully-Informed*: We would like the architectural descriptions produced by the process to be complete descriptions. Since completeness has a precise mathematical connotation in computer science, we avoid that term and its semantic baggage and use fully-informed instead. A process is said to be fully-informed if it explicitly uses information from a variety of sources covering all quadrants within the extraction information space.

- *Consistent*: We want the description produced by the process to be consistent. That is we want to identify conflicting information between different viewpoints in the description, and ultimately provide a mechanism for resolving that conflict. We also want to identify incorrect information and inconsistencies caused by drift and erosion.

- *Compliant*: We want the description to have all the information that is considered important by the software architecture community—thus we want the information to be compliant with existing standards. IEEE Standard P1471 is an example of an existing standard governing content of architectural descriptions.

- *Useful*: Architectural descriptions are derived for a purpose. We want any architectural description produced by the process to support that purpose and provide the necessary information for an analyst to achieve their goals in performing the architectural recovery. Tilley's [63][65] survey of reengineering efforts has determined that evolution of existing systems is the common purpose.

IEEE Standard P1471 states that any compliant architectural description must contain one or more views. Recall that a view is defined as a representation of a whole system from the perspective of a related set of concerns. The rules for what a view means are called a view template (or in P1471 a viewpoint). The process developed must therefore accommodate and use the idea of viewpoints and their corresponding views. This again reinforces the need for multiple sources of information as described above, since it is unlikely that any single recovery technique would adequately address multiple sets of concerns.

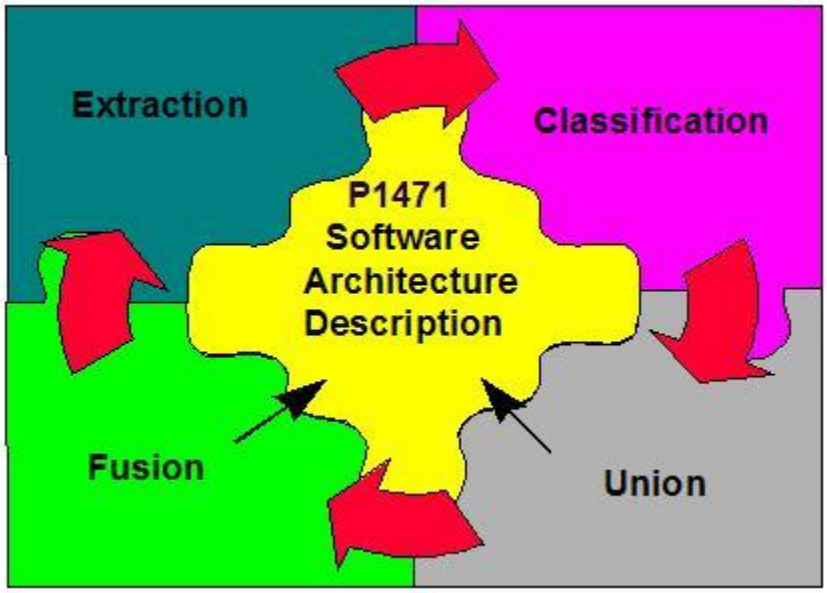

Figure 5: Architectural Synthesis Process

Finally, if we have multiple views of an architecture, all derived from multiple sources, it is unlikely that they would all be initially consistent. Section 5.5 of P1471 requires a compliant architectural description (AD) to contain an analysis of the consistency across all views provided. This places a requirement on any process description that it address some type of consistency checking for the recovered description. It's important to note that P1471 is similar to the Capability Maturity Model (CMM) [56] in that it states what has to be in a compliant AD, but not how to derive or even present the AD.

In designing a process for architectural recovery, there are two fundamental options. The first is to adopt a strategy of identifying the weaknesses of existing extraction tools and then building a new "ultimate" extraction tool to address them. This approach is similar to one proposed by Mendonca and Kramer [50]. The other approach recognizes that there will always be new approaches developed with their own strengths and weaknesses. A general process that can integrate the different recovery frameworks would be the most extensible and useful.

Given these high-level requirements, the Architectural Synthesis Process (ASP) was developed. It consists of four phases that are shown graphically in Figure 5. The extraction

phase allows for the use of multiple tools and techniques to obtain raw architectural information. In the classification phase, this raw information is categorized based upon the viewpoint (stakeholder concerns) to which it refers. In the union phase, all information related to each viewpoint is combined into a complete representation (or view). At the end of the union phase, the architectural description will have one or more views (depending on the amount and diversity of information obtained). Finally, the fusion phase allows for consistency checking between views. There is no requirement that a compliant architectural description resolve inconsistencies that are detected, only that they are detected and identified in the description. We represent inconsistency resolution, by showing the process as a cycle, where we reenter the extraction phase after fusion if there are identified inconsistencies remaining.

The central part of the puzzle is the Software Architectural Description. This represents the architectural description that is being produced by the synthesis process. We would like the AD being developed to be compliant with P1471.

We now have defined the overarching conceptual model and the general process phases forming the methodology to be followed. The next section will provide implementation-level details for ASP.

3.2 ASP Implementation

This section presents the details of the proposed solution to the problem of obtaining a fully-informed and consistent architectural description for a legacy system using a tailorable process. This overall process is called architectural synthesis.

This section is divided into two parts: a more detailed discussion of ASP, introduced initially in the overview and a description of the tool that will provide automated support. Figure 6 presents an overall context diagram for the ASP. Recall that context diagrams show how a system interacts in its environment. The ASP is the system (the center circle). External systems and data repositories are shown as rectangles. Finally input and output data flows are shown as labeled arrows between the external systems and ASP.

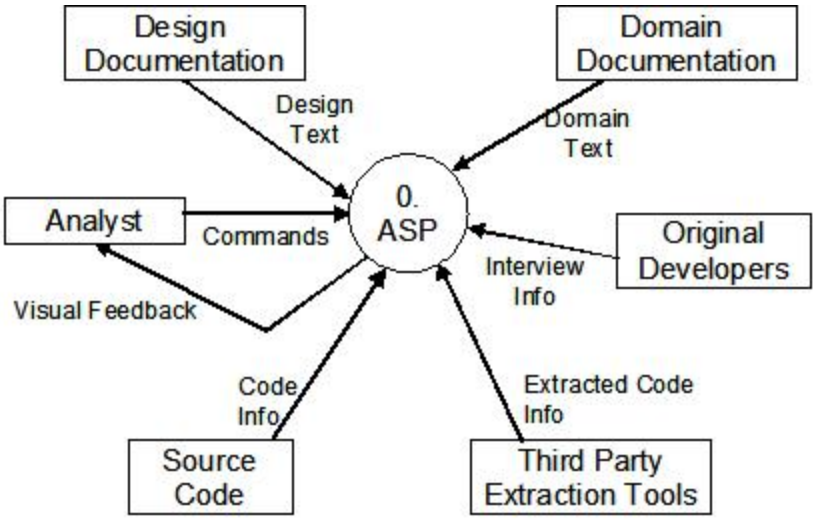

Figure 6: ASP Context Diagram

3.2.1 Process Overview

ASP is fundamentally an information-processing task. Information is obtained about the architecture using a set of tools and techniques. Within ASP, we refer to a the information obtained from a single source or technique as a *perspective*. It may be convenient to think of perspectives as "raw" information. This information is then processed to filter out incorrect or inconsistent information and produce a subset of the total information that is consistent. This of course raises the question how much information needs to be obtained to ensure we have enough to make this a consistent subset? A standard formulaic answer to this question is probably not possible. Every legacy system varies as to its size, complexity and architecture. Most legacy systems have been modified so often that the existing documentation is inaccurate or the system is so old that documentation may be missing entirely. Even interviewing designers or implementers may not provide adequate information because their individual understanding is usually limited to the portion of the system they were involved with. These conflicts drive the need to explicitly obtain information from all quadrants of the extraction information space. It was stated earlier that a general goal of ASP is to produce fully-informed and consistent representations. How then do we make a claim about any description being fully-informed? Clearly we can never know with 100% confidence that we have uncovered every scrap of information because there is no "oracle" against which

we can compare our information. The key to understanding the notion of fully-informed is to think about another goal of the process, usefulness. An architectural description is fully-informed if it contains sufficient information for the analyst to accomplish his goal. This leads to the situation where a description considered fully-informed for a Rapide [46] simulation may not be fully-informed for a SAAM evaluation and vice versa.

This notion of fully-informed completeness is consistent with current thought in the software architecture community. It is a generally accepted premise that there is no one "true" software architecture. Rather we prepare an architectural description that supports some goal of the preparer. The architectural description will contain at least one view that we can then use as an accepted representation of the system. This matches our intuition since if I were going to discuss the performance of the system; I would sketch a view that emphasized the architectural elements that deal with performance. This might be very different from a data view or security view of the system, but for the purpose we need, it is the software architecture of the system.

Consistency is also an important concept. There are two aspects to consistency that we seek to ensure during the synthesis of architectural information. First we want to be able to eliminate erroneous information from consideration by the analyst. By erroneous information, we mean information primarily from sources like documentation and human subjects which is incorrect, either because the system has changed from its documented baseline, or because of a human's misunderstanding of how the system is structured. Secondly, we want to identify information that is in conflict with other information that has been collected. A conflict can have two causes: we may be missing information necessary to remove the conflict or a piece of the conflicting information may be in error. This is similar to the idea of database consistency [21].

A frequent problem in legacy information systems is that the same information is entered multiple times and resides in multiple places in the database. This information often gets out of synchronization and gives inconsistent results. Architectural information suffers from the same problem. Information comes from many sources and some of those sources are outdated and have minor errors while other sources may be grossly in error.

From the preceding discussion then, we can infer several requirements that any synthesis process should possess:

- It should provide a method for combining information obtained from a variety of sources.
- It should provide a mechanism for finding inconsistencies in architectural information.
- It should support iteration so that inconsistent or missing information can be resolved.

Zave and Jackson studied the problem of combining formal specifications from multiple sources to produce a single coherent specification. In their well-know paper[71] on composing these specifications, the authors present three primary goals that a solution must meet. These goals are adapted here for ASP:

- ASP should accommodate a wide variety of architectural recovery paradigms and techniques.
- It should be possible for ASP to combine partial architectural representations regardless of overlaps or gaps in coverage; regardless of which paradigms they represent, and regardless of where boundaries between techniques and representations are drawn.
- Intuitive expectations of a combined architectural representation should be met. ASP should not define as inconsistent sets of partial representations that are intuitively consistent and meaningful. It should not map intuitively interdependent properties or elements in a representation onto spuriously independent ones.

Figure 7 presents the top-level process diagram for ASP. While this diagram may look a little daunting at first, much of the convoluted information flow is due to traceability data and the iterative nature of obtaining and then refining the architectural information.

As the process is discussed, we will use a continuing small example to illustrate the different concepts of the ASP. We will use the Interaction Scenario Visualizer (ISVis). For details on ISVis, refer to chapter 5 and Appendix C.

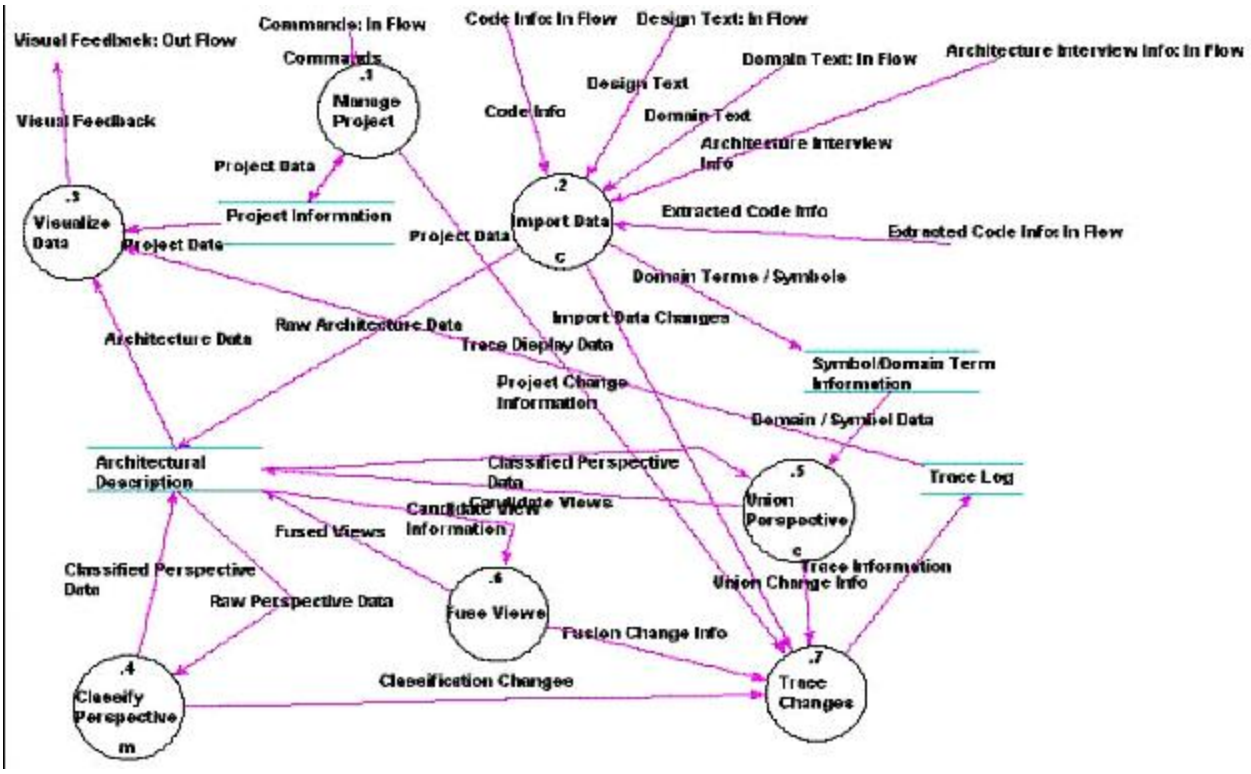

Figure 7: Top-Level ASP Process Diagram

3.2.1.1 Extraction

The first step in performing the synthesis process is to obtain the perspectives to be synthesized. These perspectives may come from existing documentation, source code analysis, domain analysis, interviews with human experts, or any other source that may provide an idea of what the legacy system's architecture might be.

Following the philosophy of Dali [36], there is no prescribed set of extraction mechanisms that must be used to develop a set of perspectives. It is unclear at this time whether there might exist a canonical set of extraction methods which could be prescribed that would give an adequate set of perspectives from which to derive a set of representations which are complete, consistent, useful, and have the desired content. It is unlikely that such a canonical set would exist given the wide range of domains which legacy system's support. This increases the motivation to accommodate as many extraction methods and information sources as possible. In general, it is desirable to get some type of coverage across the extraction information space to obtain the widest range of data sources for the

Table 3: Some Typical Sources of Potential Perspectives by Viewpoint Viewpoint Source

Physical (Hardware)	System Inventory, Design Documents, Interviews
Process (Runtime, Execution)	Design Documentation, Dynamic Execution Traces, ManSART, Interviews, Makefile analysis
Module (Code, Source)	MakeDepend, Makefile analysis, File System Examination,PBS, Rigi,Project Documentation (Work breakdowns)
Conceptual (Logical, Functional)	Rigi, DALI, Interviews, ManSart , Call-Graph Generator, DSSA, Domain Models

architecture.

The principle requirement of ASP extraction is that the output of any extraction technique must be expressible as an attributed graph. This is not really a limitation since even if the output pertains to a single component, the graph would be one node with all relevant information shown as attributes. Components become nodes and connectors (the relationships between components) become edges. Any other architectural information extracted is attached to the nodes, edges or to the graph as a whole as attributes. We do not anticipate this to be a major limitation of the technique. Architectures are typically described as a graph where the nodes are components, the edges connectors and the configuration the topology of the graph.

We also must determine the relevant viewpoints that must be addressed to meet the purpose to which the architectural description will be put. We can generally limit our selection to the four major viewpoints (physical, logical, process and module). In larger systems, or in specialized domains, the number of viewpoints may need to be expanded to include other concerns. For instance, the logical viewpoint might be separated into a data and control view if this is necessary to better understand a complex system. In certain domains, security or performance views might also need to be constructed. The ultimate decision about which viewpoints are significant and what views need to be prepared is a function of the domain and the purpose to which the architectural description will be put.

Finally, after the viewpoints are chosen, any mappings between the viewpoints must be

Table 4: Summary of Mappings

Source Viewpoint	Destination Viewpoint	Component Relationship	Connector Relationship
Process	Physical	runs-on resides-on deployed-on	transported by
Logical	Module	implemented-by defined-by	implemented-by
Process	Logical	defined-by	defined-by

identified. These mappings are important because they indicate what information we must extract about elements so that we can perform fusion in a later stage of the process. For instance, for the standard four viewpoints we have the following mappings (summarized in Table 4: A component in the process view maps to a component in the physical view via the runs-on or resides-on relation. (We must determine which processors a given process or thread runs on, or which processor a given data store resides on). A connector in the process view maps to either a processor or communications path via the runs-on relation. A component in the module view maps to a component or connector in the logical view via the implements relation. (We must determine which logical elements are implemented by which code modules).

If the analyst selects other viewpoints, similar mappings must be selected so that the proper information is extracted for use later in the fusion phase. We define the mappings during extraction rather than later so that they can help determine information that the analyst extracts from the legacy system.

EXTRACTION PHASE PROCESS SUMMARY:

Preconditions Recognize the need to develop an architectural description of a legacy system.

Process Steps

- Determine the purpose of the architectural description that ASP is being used to develop. By *purpose* we mean to understand the underlying motivation for creating the architectural description. The purpose will drive many of the subsequent decisions that must be made. For instance, if the purpose if to enhance the security features of

the system, then this will focus the extraction efforts on security aspects of the system. The purpose should be as specific as possible to allow the process to be focused as tightly as possible. Write a short statement of the purpose to focus and guide the effort.

In our example, we might imagine that we want to port ISVis from its current C++/Motif implementation to a Java/Swing implementation. The purpose is then an evolution task to transfer functionality, but not add additional functionality to the new platform.

- Determine the viewpoints that must be represented (understanding the purpose to which the AD will be put) in order to produce a complete AD for the task. Usually, it is best to recover all four of the basic viewpoints (physical, process, module and conceptual) in order to have a basis for checking the consistency of the views and insuring that they are complete.

  In our example, we recognize that ISVis is a single threaded, single process application, therefore the viewpoints of interest will be the logical and module viewpoints. For the remainder of the example we will focus on the logical viewpoint.

- Determine the mappings (relationships) between the viewpoints that have been selected in the previous step.

  The principle mapping in our example is the "implemented-by" relation. A component or connector in the logical viewpoint must be implemented by a component in the module viewpoint. Likewise, each component in the module viewpoint should implement an element (component or connector) in the logical viewpoint.

- Gather Legacy System Information: Stable Code Base, Design and Domain Information and a list of original designers and current maintainers for potential interviews.

  For our example, we have available the original design documentation, the source code, research papers describing the system and the results of an interview with the original designer/developer.

- Determine an initial set of tools to use for the extraction effort. These tools should give you a range of information-from low-level code constructs to high-level abstractions. Table 3 presents a summary of some possible sources of perspectives based upon the viewpoints chosen.

  For our example, we select a dynamic architectural extraction tool (ISVis itself). We also extract the design information from the design documents available.

- Follow the instructions for each tool that was chosen in the previous step to develop an initial perspective of the architecture. (Some of this data may be in electronic format, while other information may be in paper form).

  For our example we end up with the two perspectives presented in Chapter V, Figures 25 and 21.

- Convert the information in the previous step into an attributed graph, with components being nodes, connectors being edges and associated information being attributes.

  This step would give us a GML file for Figure 21 like:

```
graph [
version 2
directed 1
node_style [

node [
id 9
label "View"
]
node [
id 12
label "ViewManager"
]
node [
id 15
label "ProgramModel"
]
node [
id 18
label "TraceAnalyzer"
]
node [
```

```
id 21
label "EventStream"
]
node [
id 24
label "SessionFile"
]
node [
id 27
label "EventTrace"
]
node [
id 30
label "InstrumentedSourceCode"
]
node [
id 33
label "TraceInfoFile"
]
node [
id 36
label "SourceCode"
]
node [
id 39
label "SolarisDBDatabase"
]
node [
id 42
label "StaticAnalyzer"
]
node [
id 45
label "Instrumentor"
]
node [
id 48
label "StaticInfoFile"
]
edge [
source 48
target 15
label "Data8"
]
edge [
source 48
target 45
label "Data6"
```

```
]
edge [
source 36
target 39
label "Data1"
]
edge [
source 45
target 30
label "Data5"
]
edge [
source 30
target 27
label "Data4"
]
edge [
source 27
target 21
label "Data3"
]
edge [
source 18
target 24
label "Data2"
]
edge [
source 12
target 18
label "Control5"
]
]
```

- Use text data mining to extract attribute information (for example word frequency and ngrams) about the system being recovered for use in later steps. *NGrams* are used in text processing to improve the analysis of textual content over single-word analysis. In this dissertation we use ngram to mean using more than one word as if it were a single word. For instance a 1-gram might be "file" while a 2-gram would be "information file" and a 3-gram "static information file." This technique is detailed further in the next chapter.

  For our example, we would derive the word sets in Appendix F, Table 25.

- Assign a subset of this text information to the elements in the perspective graph as an attribute. This step is elaborated upon in the next chapter.

  For our example, we would obtain the formal context shown in Appendix F, Table 26.

Postconditions

- A short statement of purpose of architectural recovery effort has been prepared.
- A list of viewpoints that are relevant for the recovery effort has been prepared.
- A list of mapping relationships between the viewpoints has been prepared.
- A set of perspectives (attributed graphs) representing the architectural data that has been extracted has been prepared.

3.2.1.2 Classification

The next step is for the analyst to group perspectives into their respective viewpoints (selected during the previous phase of ASP). This helps an analyst to focus initially on synthesis of perspectives that are intended to describe the same aspects (or concerns) of the system undergoing recovery. Note that this is not necessarily a one-to-one function, as parts of an individual perspective may map to two or more viewpoints.

To determine how to classify the information in a perspective, the analyst must use the type of viewpoint, the source of the perspective and the attributes of the elements in the perspective to determine how to classify the perspective. Some general rules for classification are the following:

1. If the perspective came from an object-model diagram (such as an OMT diagram) it is typically a logical view.
2. If the perspective came from a call-graph representation then it is typically a logical view.
3. If the perspective contains hardware elements then it is typically a physical view.

4. If the perspective contains component names that can be matched to static source code entities, it is typically a logical view.

5. If the perspective contains names that can be matched to targets in a build or `make` file, it is typically a process view.

6. If the perspective came from a legacy "box-and-arrow" diagram, it is typically a logical view.

7. If the perspective contains information derived from the source code directory structure or depends sections of the make file it is typically a module view.

CLASSIFICATION PHASE PROCESS SUMMARY:

Preconditions Identification of the Viewpoints that are of interest to the recovery. A set of perspectives (represented as attributed graphs).

Process Steps For each perspective: Examine the nodes and edges of the graph. Classify either the whole graph or the appropriate subset of nodes and edges to a particular viewpoint using the attributes of each element and the perspective source to make the decision.

For our example, the perspectives obtained would both be part of the logical viewpoint.

Postconditions Set of perspectives classified according to the viewpoints they represent.

3.2.1.3 Union

The union phase analyzes and combines all perspectives representing a specific viewpoint to build a view. During this step we manipulate a perspective as a graph where nodes are components (boxes) and edges are connectors (lines). We refer to components and connectors collectively as *elements* for convenience.

Each element within a perspective has some set of attribute values that describe properties of that element. These attributes include the name of the component (or its domain synonym), topological characteristics such as port count, general attributes, and textual documentation that has been mined from the available system artifacts.

General attributes consist of system characteristics expresses as attribute-value pairs. Some common examples of these general attributes are:

attribute = *Abstraction Level* possible values = *composite* or *atomic*

attribute = *Behavior Type* possible values = *passive* or *active*

attribute = *Component Type* possible values = *function* or *procedure*

attribute = *Connector Type* possible values = *shared-memory, socket* or *file*

The union phase derives its name from its similarity to the set union operation. During union we combine perspectives by matching elements or by recognizing new elements in the perspectives until we have combined all the perspectives into a single view. We repeat the union process for each viewpoint into which we classified perspectives during the classification phase.

Specifically, we combine perspectives in a similar manner to that used in database schema integration. First a perspective is selected as the base representation for the viewpoint being unioned. Then every other perspective (or partial perspective for those which were classified into more than one view) in the viewpoint is combined with the base representation, one perspective at a time. After each combination is completed, the result becomes the new base representation. When we have combined all perspectives, we have a view that encompasses all available architectural information pertaining to the viewpoint of the architecture being considered.

Although selection of a base representation is fairly flexible, in practice it is best to use a perspective that contains elements with a high level of abstraction. We do this because it appears that it is easier conceptually to work top-down in building a view than to work bottom-up.

The union process is an elaboration of the architectural element-matching problem-that is, given two perspectives, how can we determine when an element in one perspective matches some element in a second perspective, or an element is a new element to be added. There are three major techniques that might be used to solve this problem: lexical, topological, and semantic approximation. Chapter V discusses the theory behind each of these

techniques in some detail. Solving this matching problem is a major step towards automation of the ASP process. Our inconsistency handling during union uses the delay strategy. We expect some inconsistencies to resolve themselves during union of later perspectives. Any remaining inconsistency is removed during the Fusion phase.

UNION PHASE PROCESS SUMMARY:

Preconditions A set of perspectives classified according to the viewpoints they represent.

Process Steps For each viewpoint in the recovery: Select a perspective to be the base representation. For each remaining perspective in this viewpoint union the elements in the new perspective to the base representation.

For our example, we would select Figure 19 as the base representation and union Figure 20 to it. To get a sense of the union process activities, we examine the subset of these figures as shown below in Figure 8. We then use lexical information to perform the first match giving us the result in Figure 9. We continue to match components to give us the relationships shown in Figure 10. The connectors can then be inferred from the component relationships. This is especially helpful because the design document perspective had no information on the connectors other than data and control. By unioning this with the dynamically-derived perspective, we can obtain more information about the relationships between the original design components.

Postconditions A set of views-One supporting each viewpoint that has been selected by the analyst.

3.2.1.4 Fusion

We adopt this term from DALI, but use it to represent analysis across multiple viewpoints to check for commonality and create compositions of views. Fusion serves two purposes: first it provides a check on the consistency of the views and secondly, it provides additional information about the architecture through the composition of related views.

Our approach to view fusion is through the use of mappings. We define a mapping

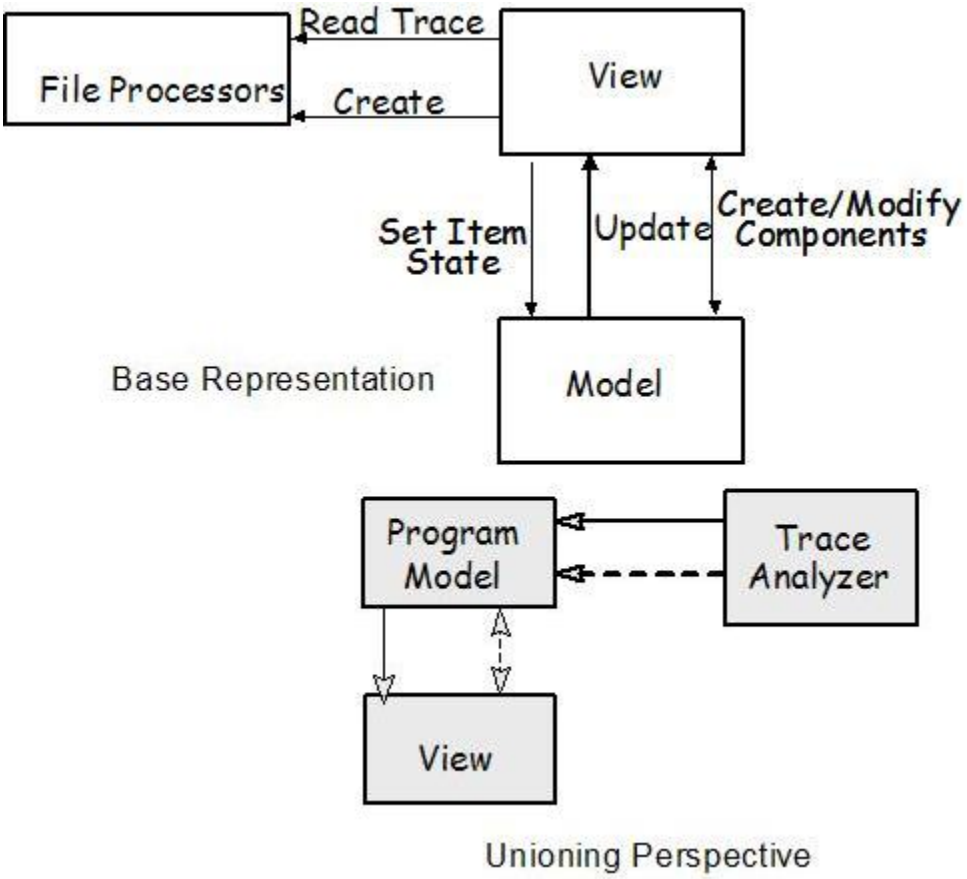

Figure 8: Base and Unioning Perspectives

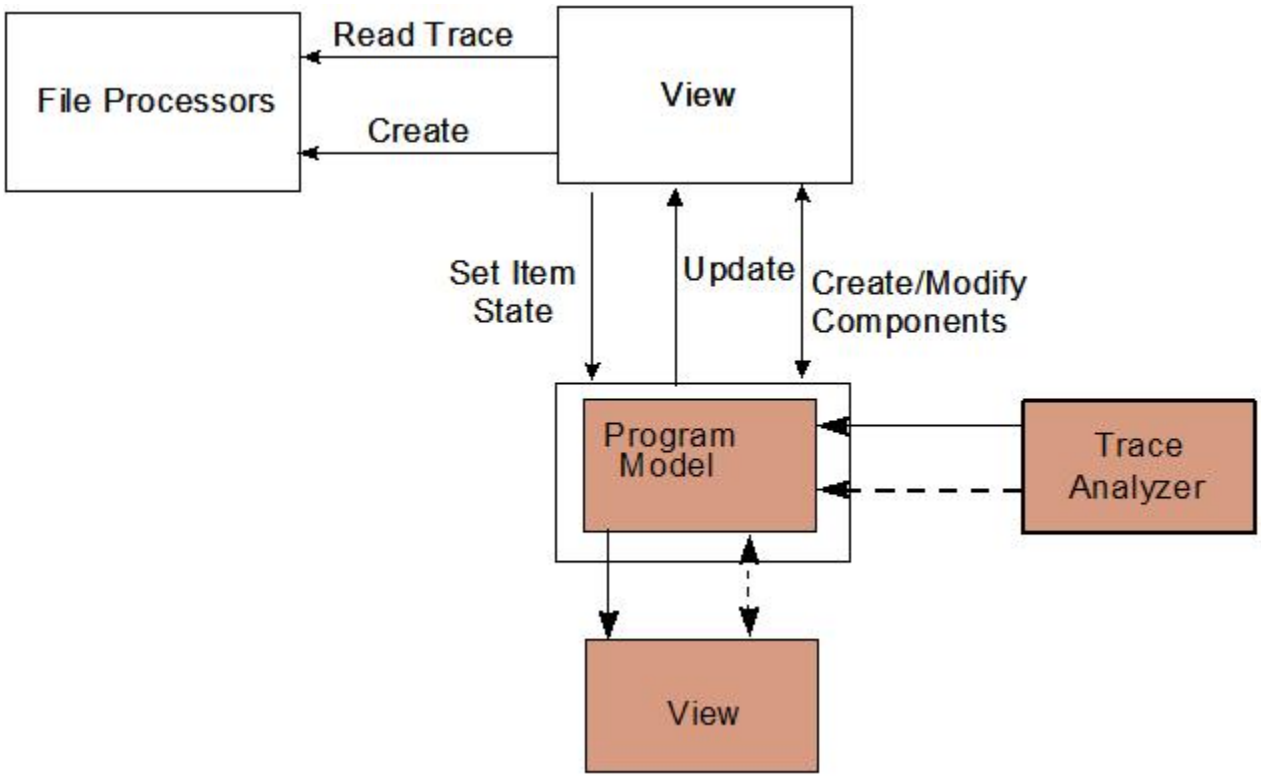

Figure 9: Results of Union After One Match

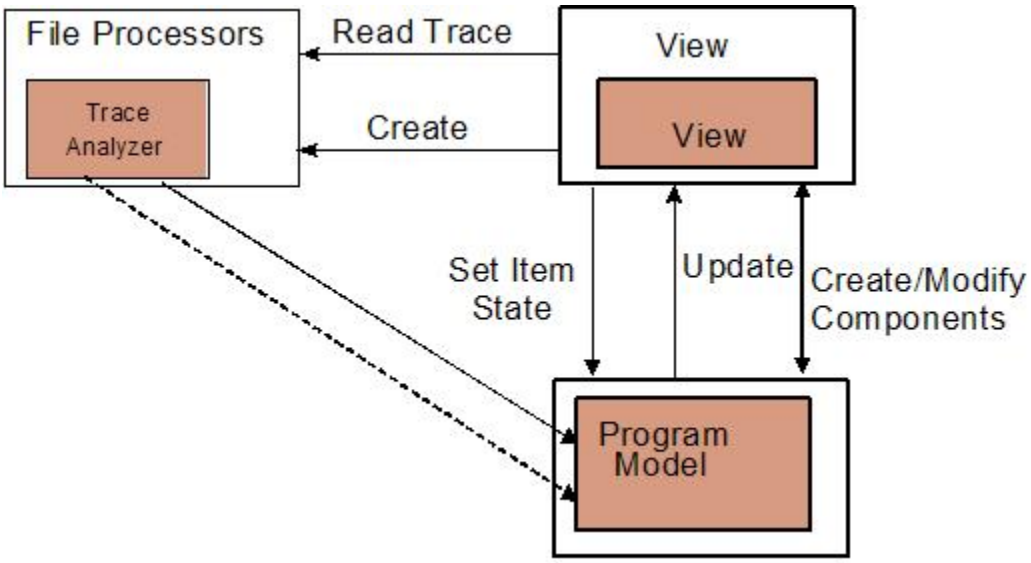

Figure 10: Final Union Results on Two Perspectives

as simply a relation between an element in one viewpoint and an element in another. The nature of this relationship varies depending upon the semantics of the architectural elements in each viewpoint. For some viewpoints there is a close correspondence between the views that represent them. For other combinations of viewpoints, the relationship is primarily transitive. For example, there is no direct mapping between the physical and module (or code) viewpoints. To get a mapping between them, we can transitively use the process view as an intermediary. First we map the process view to the physical view, then the module view to the process view thus arriving transitively at a module to physical mapping.

It is also desirable to map between smaller views, for instance a security view and a performance view of a legacy system's architecture. We accomplish these types of subview mappings by first abstracting them up to a top-level viewpoint and then doing the fusion. For instance the security view is usually a subset of the conceptual or logical viewpoint. We would then use the mapping rules for the conceptual view to check the security view for consistency.

Meta-information about potential mappings and the semantics of a given set of viewpoints might come from multiple sources of information. Information about what constitutes a viewpoint and general semantics is derivable from existing architectural metamodels such as those described in the IEEE P1471 specification. More specific information about the meaning of components and connectors in a specific domain comes from information contained in domain-specific architectures and so on. The following paragraphs discuss mappings between the four basic viewpoints.

System→Physical Mapping While the system view is not one part of the basic four viewpoints, it is a commonly available view that provides valuable information about the environment in which the legacy system operates. A system's diagram (or view) is normally found in design documentation or product brochures and typically consists of a top-level representation of all hardware components in the legacy system, whether they have software associated with them or not. The systems view in legacy documentation may often be referred to as the systems architecture. Recall from Chapter II, that the P1471 definition

of a software architecture includes its relation to the external environment. This mapping helps express that relationship.

The Physical view can frequently be checked for consistency against the overall System's architecture. Since the system's architecture by definition should contain all the hardware system components and the physical view contains only the hardware components that host software, the physical view should be fully contained in the system view.

The actual process of mapping the system to the physical view is straight-forward. There should be a correspondence between the elements in each view. Additionally, since these elements are of the same type (i.e. actual hardware components), we should be able to match them using basic lexical analysis and attribute comparisons.

An inconsistency exists if we have any elements in the physical view that do not exist in the system view or if there are elements in the system view which we know to host software, yet these elements are missing from the physical view. Complexity is introduced even into this simple process by aliases and multiple connection types. For example the larger system diagram may call a component the business-rules server while the physical view used by developers may refer to it as the application server.

Process→Physical Mapping the processes to the physical view is much more difficult. Components in the process view represent runtime processes or threads while the connectors represent some type of inter-process or inter-thread communication. Information about the physical devices that processes exist on at run time must be obtained from design documentation, human interviews or dynamic trace information obtained during system execution. For this mapping, trying to match names or topology is not effective. We would not expect the names of processes to have much in common with the names of processors or communications paths. Topologically, many of the connectors in the process view do not map onto connectors in the physical view, but rather onto processors (components) in the physical. This phenomenon is caused by the ability of multiple processes to run on a single processor; therefore the inter-process communication (IPC) connectors would also be mapped to that processor. It is the case however that any IPC between processes on

different processors must map onto a physical communications connector. This illustrates why a simple graph-matching algorithm is inappropriate. Nodes (components) in one graph do not necessarily map to nodes in another, they might map to edges (connectors)!

Since elements in the physical view are not included unless they host software, we would expect that all processors have some process running on them. An inconsistency exists if there are processors that have no processes or processes that have no processor to run on.

Conceptual→Module Mapping of a conceptual viewpoint to a module viewpoint requires information about which code modules implement the various components and connectors in the conceptual view. Often the connectors in views supporting these two viewpoints provide information that is not related to consistency issues. For instance, the connectors in a module view usually represent a "uses" or "contains" relationship between the components denoting source code modules. While providing important information within the module viewpoint, they do not bear any relation to information in views supporting the conceptual viewpoint.

Likewise in the conceptual viewpoint, the connectors often represent aggregations of function calls, message passing or other semantics that are implicitly implemented in software rather than explicitly implemented by the code base. Another complication is caused by the use of COTS (commercial off-the-shelf) components. Some of these may be used by the software system in a third-party binary format, and thus they have no corresponding implementation in the module view. For this reason, during extraction we must identify components and connectors in the logical views as internal or external based upon whether the element is COTS.

An inconsistency is detected when there are internal components and connectors that have no code modules that implement them. Likewise, we have an inconsistency when there are code modules that do not implement any elements in the logical view.

Other Mappings We did not consider all possible combinations of views because the other combinations have transitive mappings that allow them to be derived from the ones presented. For instance, what consistency considerations are there for the physical to code

module view? None, except transitively through the fact that processes run on processors, and processes are made up of portions of logical components that are themselves implemented by code modules.

Besides these transitive considerations, we also have the situation where the "basic four" views are not sufficient or are not used in the domain. For instance in some information systems environments, it might be necessary to create views supporting the Zachman Framework[69] which contains views supporting up to 30 different viewpoints. Within the Department of Defense, views might be generated conforming to the C4ISR architecture that would put them into three major categories: Operational, Technical or Systems. While there are mappings which can be derived between these views, unfortunately, each mapping is unique to the semantics of the given viewpoint/view combination. If we can use architectural description languages (ADL's) to express the meaning of these different views, then we have the possibility of further automating these mappings.

Summary It is useful at this point to address a couple of key questions about view fusion and inconsistency detection. First, we might ask what kinds of inconsistencies can be detected? Clearly using the approach of mapping, most of the inconsistencies we find are related to a mismatch between viewpoints. This mismatch is most commonly caused by an architectural element in one view failing to have a corresponding match in another view. For example, in the code module view, we might have a source file which implements functionality that is not represented by anything in the conceptual view. Conversely, we might have a logical component in the conceptual architecture which has no code module associated with it (and the logical component is not an external or COTS piece of the system).

This leads us to the second question, what kinds of inconsistency cannot be found. Clearly, if we have missing information from multiple views, then inconsistencies that are actually there might not be found. In ASP, we attempt to overcome the problem of missing information by deriving information from the entire spectrum of sources shown in the information extraction space, rather that just source code alone. While missing information

is generally an issue of completeness rather than consistency, it does impact on our method of consistency checking. In an ideal world, we might wish for an oracle that had complete knowledge of every viewpoint for a legacy system, but sadly this oracle does not exist. For this reason we strive for a relative consistency between the views/viewpoints rather than some utopian absolute consistency measure.

Another question that might be asked is how inconsistent are views in practice? The degree of inconsistency is usually directly related to the diversity of information sources used for extraction. If we recover an architecture based solely on source code analysis, we will probably have a small set of views that are very consistent (although their completeness and accuracy might be less). As we diversify the extraction sources to cover the information space, we begin to uncover information that reflects differences between the design documents, interviews and code that do introduce inconsistencies in the different views.

Finally, we might ask how are inconsistencies handled? Is any of this detection even important? Again, as P1471 states, there is no requirement that all inconsistencies be resolved in an AD, only that they are identified. This acknowledges the fact that for any given situation, the resolution of an inconsistency is left to the discretion of the analyst conducting the architectural recovery. If we are conducting a SAAM session to evaluate the impact of changes on the code structure, an inconsistency in a physical view might be unimportant and would not need to be addressed. On the other hand, if we were preparing an architectural description to evaluate extensive new functionality to be implemented in a distributed fashion, then it would be critical to resolve any inconsistencies between the physical view and the processes specifically running on the physical devices in the view.

FUSION PHASE PROCESS SUMMARY:

Preconditions Set of Views representing viewpoints of interest for the recovery.

Process Steps For each pair of views: If there is a mapping between the views, then check the views for consistency by using the defined mappings. If any inconsistencies are detected, decide on the appropriate course of action (resolve, ignore, delay). If resolution of the inconsistency is necessary, determine the type of information which will resolve the

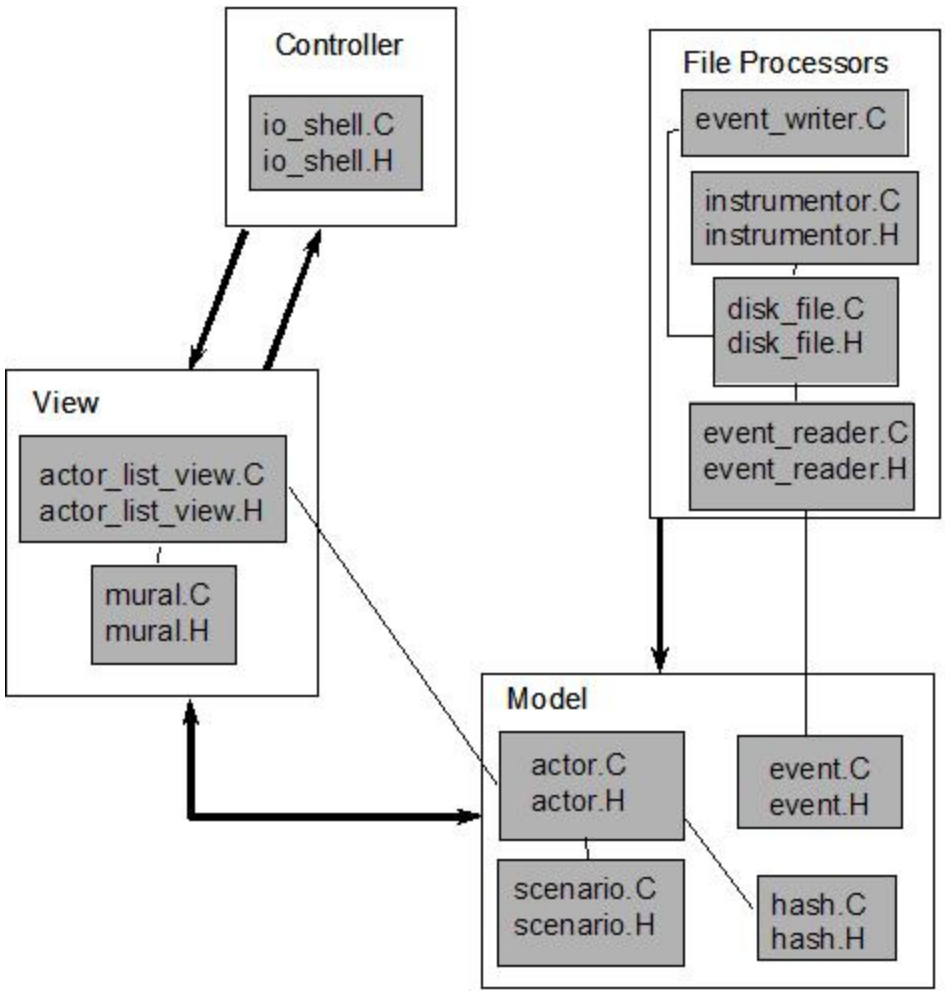

Figure 11: Partial Fusion Results for ISVis

inconsistency and return to the extraction phase.

For our running example, we would perform fusion between the module and logical viewpoint. Figure 11 depicts the results of this fusion. Note that we have code modules implementing all our logical components, but we have been unable to identify connector information. If the purpose of our recovery effort required us to understand these connectors, then this indicates the need to go back to the extraction phase and obtain additional information to resolve this missing information.

Also since our original purpose was to port this tool, the modules implementing the View component would be of particular interest since they encompass the graphical user interface, which is the hardest part of the application to port.

Postconditions List of inconsistencies between views List of inconsistencies which must be resolved

3.3 REMORA Toolkit

Two things enhance developing a repeatable process that will move architectural recovery from art to engineering discipline. First we can define a standard process as in the previous section. Second, we can develop automated support that performs routine actions in a standardized way. This section details the design for REMORA (Resolution of MORALE[59] Architectures). REMORA has a conceptual architecture as shown in Figure 12.

The conceptual architecture of REMORA represents a component-based approach to the development of an architectural synthesis toolkit. Major functional elements of the toolkit are implemented as independent COM components. This allows them to be reused at the binary level in a variety of different languages and platforms. Also, by using the remote execution feature, workload-intensive components can run on different machines to accomplish their major tasks. In Figure 12, the shaded components are building block components that are not normally directly used by an application.

REMORA currently uses either the VisEd (Graphlet) package or the LEDA library for graph display and layout services. Christian Lindig's [44] concepts package computes concept lattices from formal contexts. The text analyzer component performs text manipulation and analysis on the documents imported into the current project. The Bow library and the Text Analyst SOM analyzer are used as COTS products. Nauty is used as the graph manipulation library. This component also contains the WordNet software package, which performs significant computation services for synonym and semantic similarity detection.

3.3.1 User-Interface Module

The user-interface (UI) component is always active and acts as mediator for the various features of REMORA. A screen shot of the UI is provided in Figure 3-1. The view is divided into several areas each displaying different types of information to the analyst. On the left is a tabbed tree view that lists all the information for a specific project. The tabs provide organized access to diagrams, documents, and scenarios. Currently the tool

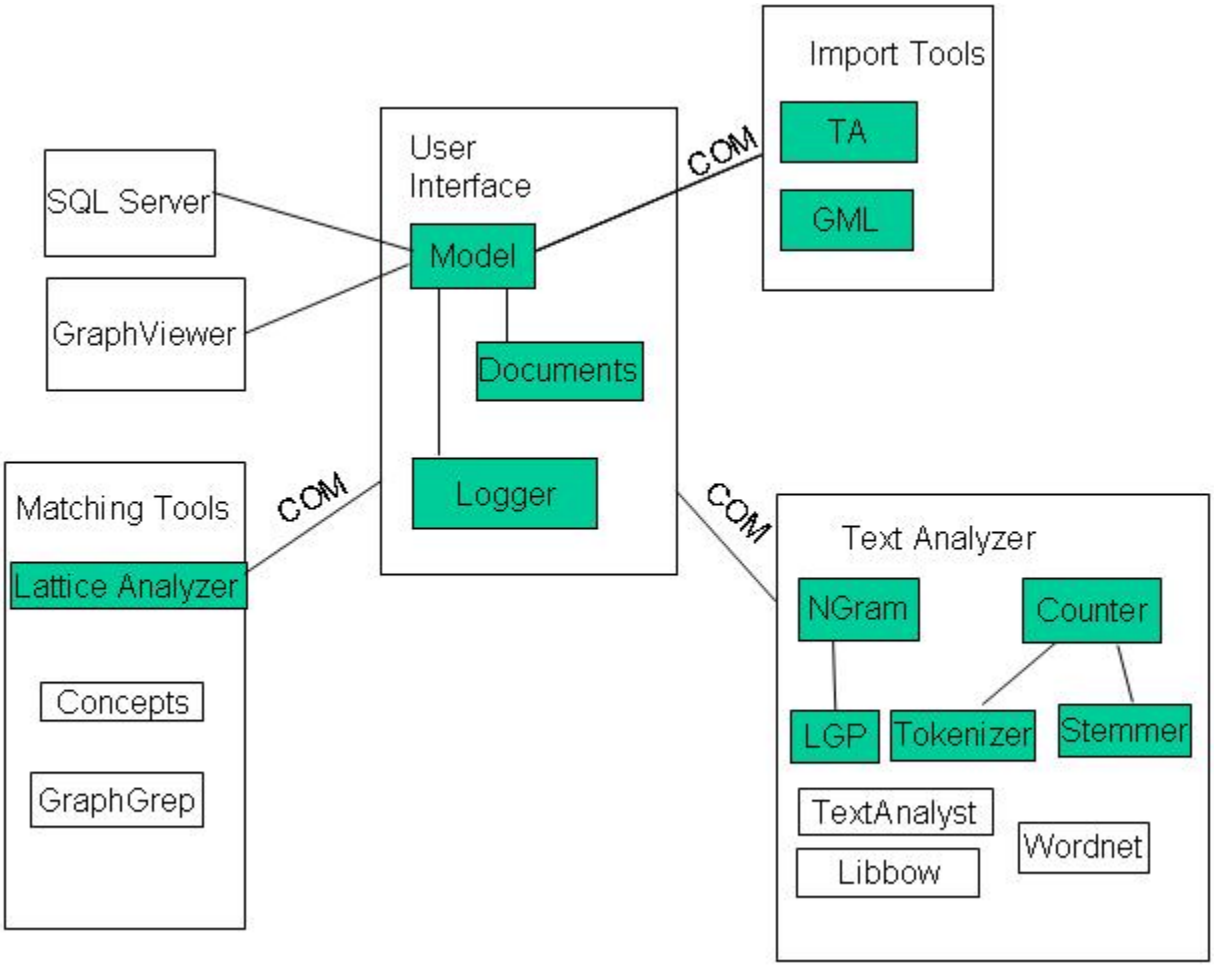

Figure 12: REMORA Conceptual Architecture

does not provide scenario support, but does provide a placeholder for future expansion. An analyst creates a project that identifies the set of perspectives, documents and their associated information that will be used for the remainder of the process. The extraction phase is supported through importation of architectural information via various filters. After importing the perspective, other attributes can be assigned to the representation if desired. The different representations created can be grouped using the UI module.

The UI Module is implemented using the Microsoft Foundation Classes (MFC) and the standard event-driven windows paradigm. The model (derived from the standard MFC CDocument class) of the project controls access to the supporting database using OLE DB COM calls to obtain information from the database. This allows easy swapping of database components since the base program only uses standard COM calls. Views of the data in the project are displayed either internally, using the MFC HTMLView class, or externally using a graph viewer. By using the MFC View class, REMORA can display a variety of formats including ASCII Text, native hypertext (html), extended markup (xml) or MS Word documents. The view has an integral button bar which allows access to the supporting COM-based Text Analysis modules.

The UI module also supports log files to provide both feedback and undo functions. The error log view pane displays a chronological report of abnormal conditions that occurred in the project. Examples of errors include failure to successfully import a Rigi Standard Format (RSF) file due to format problems or an inability to launch a required COM component.

The trace log on the other hand provides a basic record of how the analyst got to the point in the synthesis process that the state of the project depicts. Any actions affecting the final product are recorded and time stamped in the trace log. The analyst can use this log to determine where certain views came from by tracing through the perspectives that were combined. The log can also serve to undo certain actions and restore a previous project state.

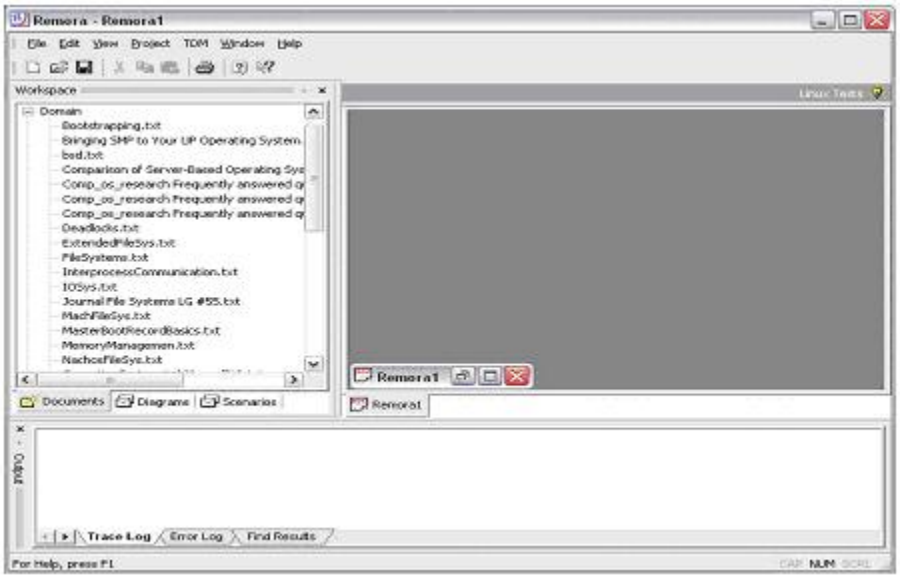

Figure 13: REMORA Main UI

3.3.2 Import Tools Module

The Import Tools module allows data from other extraction tools to be brought into REMORA. Currently there are two primary importation tools. The first uses Graph Markup Language (GML). This allows the analyst to quickly draw an architecture in either VisEd or LEDA, export it as GML and bring it into the project. The second tool is a Tuple Attribute (TA) and Rigi Standard Format (RSF) importer. This allows data from several research extraction tools to be brought into REMORA. These tools include PBS, DALI and Rigi.

3.3.3 Matching Tools Module

To combine perspectives, we must be able to match elements between perspectives. This matching problem can be stated as: Given two perspectives P1 and P2, each comprised of a set of architectural elements, how can we combine the elements to create a new perspective P3 which is a combination (union) of P1 and P2? As mentioned previously, we want to use both syntactic and semantic information to do this matching. The Matching Tools Module provides access to both these techniques.

The lattice analyzer, a COM component that uses Christian Lindig's concepts program

to compute a concept lattice, provides the implementation of our custom algorithm for semantic approximation. The analyzer traverses the concept lattice using the algorithm described earlier in this chapter to find the appropriate matching relations. These are then displayed through the UI component to the analyst. Syntactic matching is provided by connecting to the Mapper component (in the Text Analysis module), which provides synonyms and word relations to help match similar components. This information is also provided through the UI component to the analyst.

3.3.4 Graph Viewer Module

While documents can be displayed correctly within the UI, diagrams must be viewed externally using the Graph Viewer Module. Currently, REMORA supports one of two viewers, Graphlet (VisEd) or LEDA. These run as threads within a COM component and support graphical manipulation and display of information.

3.3.5 SQL Server DB Module

Project information is persistently stored in a relational database. Currently, REMORA uses SQL Server 7.0[2], however this is not a hard requirement. Only the UI and Import component even knows there is a database, and all communication is via the OLE DB protocol. Thus, any OLE DB[32] provider can be used to provide the back end, even a custom one written by the analyst. Changing the backend does not require any rewrite of the other components. There are two models, a high-level and low-level model. These models are currently correlated and integrated by knowledge built into the REMORA application code rather than by the database model itself.

3.3.6 Text Analysis Module

The text analysis module is the heart of the document processing capability in REMORA. It provides several tools inspired by Dowser[16] to analyze the project architectural documents. The shaded components in Figure 3-3 represent smaller building-block COM components that are not directly used by REMORA. These are the Stemmer, Tokenizer, Link Grammar Parser (LGP)[28] and Wordnet[52] components. The tokenizer provides a simple way to

tokenize and return the words in a file. The stemmer accepts a word and returns its stem using either the Porter[58] or Morph (included in wordnet distribution) algorithm. Wordnet is an extensive language analysis program computing complex word relationships like synonyms, antonyms and hypernyms. LGP is the link grammar parser that parses sentences and returns the links between the words based upon the parts of speech.

Sitting on top of these utility components are the main REMORA tools: Ngram, Counter and Mapper. Counter is simply a single word counter which takes a document, filters out stop words if desired, stems the words if desired and counts the number of occurrences of the word in the document. Ngram works slightly more intelligently by recognizing that key ideas are often a combination of nouns and adjectives. Ngram submits the sentences in the document to the LGP and then processes the links to obtain ngrams composed of the nouns and their descriptive modifiers. We feel this Ngram version is an improvement over the original Dowser ngram tool that inspired it. First, unlike the Dowser tool, there is no need to specify the size of "n." Ngram allows "n" to range from 1 to any number. Consider for example the phrase "centralized operating system." Depending on whether the user requested 1, 2, or 3 grams, Dowser would return *system*, *operating system* or *centralized operating system*, while Ngram would automatically return the entire phrase *centralized operating system*. It does this by recognizing that centralized and operating are both modifiers of the noun system. Mapper takes a word and returns a set of related words based upon the user's request. It primarily uses Wordnet, but also uses a user provided custom dictionary of domain synonyms and abbreviation expansions.

The code data miner parses a source code file, and extracts keywords, associating them with files or function names in the program. Each function name is processed to extract the maximum meaning. For instance the function `mem_set` would get expanded to *memory set* and `procSched` would become *process scheduler* (based upon use of an operating systems domain dictionary).

CHAPTER IV

SEMANTIC APPROXIMATION

This chapter gets at the heart of our thesis statement:

> *It is possible to improve the amount and quality of architectural information that is automatically recovered and integrated from legacy systems by using semantic approximation to bridge the concept assignment gap.*

Previous chapters have provided a background in current automated support for architectural recovery and our general framework and process for recovering architectures, around which automated tools can be built. This chapter presents a technique within that framework which provides the mechanism for achieving our thesis statement. Specifically this technique focuses on improving integration of information during the union phase of ASP. The union process is discussed in detail in paragraph 3.2.1.3. The next chapter discusses validation of semantic approximation through an experimental comparison.

This chapter has two major parts. The first is a theoretical overview of concept analysis to provide background in the field. The second shows our application of this mathematical technique to architectural recovery.

4.1 Concept Analysis Primer

Concept analysis is based on lattice theory [66]. Concept analysis allows groupings to be formed based upon maximal sets of common attributes. Other software engineering researchers have used concept analysis for module identification [60] and configuration management [41]. Ganter and Willie provide a complete mathematical treatment of concept analysis in [26]. This section presents a brief introduction so the reader can understand the techniques presented in subsequent chapters of this dissertation.

Lattices are based on ordered sets. An ordered set is a pair (S, $*$) where S is a set and $*$ is an order relation on the elements of S. We can draw a Hasse diagram or line diagram

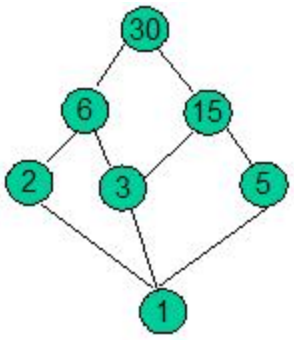

Figure 14: Sample Haase Diagram

based upon the ordering of the set. For example, let S be the set {1, 2, 3, 5, 6, 15, 30} and the order relation be / (that is the ordered set (S, /) where / means divisible by). A lattice then us an ordered set in which every subset containing exactly two elements has a greatest lower bound (also called the infimum or meet) and a least upper bound (also called the supremum or join). The corresponding Hasse diagram for our example set is shown in Figure 14. Refering to this figure we can see the join of 2 and 3 is 6, while the meet of 2 and 3 is 1. In our figure the elements 30 and 1 have a special designation known as top and bottom.

Concept analysis is an application of lattice theory that deals with objects and their attributes. Lattices for concept analysis are created from a *formal context*. A formal context is a triple {O, A, R } where O is a set of objects, A is a set of attributes and R is a relation between these objects and attributes. Specifically, R is a subset of O×A that relates objects to the attributes they possess such that (o,a) ∈R if object o has attribute a. In concept analysis, when we discuss attributes we are discussing actual values. We might say in other domains that an attribute of a creature is its skin covering which could be for example scales or fur. In concept analysis, we call scales and fur attributes (rather than a value of the attribute skin covering).

We then define two important functions:

Let $X \subseteq O$ and $Y \subseteq A$ then:

$$\sigma(X) = \{a \in A | \forall o \in X : (o, a) \in R\} \tag{1}$$

Table 5: A Formal Context

	Hair	Scales	Walks erect	Has thumb	Crawls	Breathes air	Breathes water
Dog	X					X	
Cat	X					X	
Person	X		X	X		X	
Monkey	X		X	X		X	
Snake		X			X	X	
Fish		X					X

$$\tau(Y) = \{o \in O | \forall a \in Y : (o, a) \in R\} \tag{2}$$

Thus equation 1 refers to the set of common attributes for the objects in X, while equation 2 refers to the set of objects which each have all the attributes in Y. A concept then is a pair of sets consisting of objects(X) and attributes(Y) that satisfy the constraint $Y = \sigma(X)$ and $X = \tau(Y)$. A concept can also be thought of as a maximal collection of objects who share the same attributes.

These two functions lead to the fundamental theorem of concept lattices shown in equation 3. Simply stated, this theorem states that a concept can be computed by intersecting their attributes and finding the common objects of the resulting intersection.

$$\bigcup_{i \in I}(X_i, Y_i) = (\tau(\bigcap_{i \in I} Y_i), \bigcap_{i \in I} Y_i) \tag{3}$$
$$where\ I\ is\ subset\ of\ integers\ ranging\ from\ 0\ to\ |Y|$$

For example, consider the following set of objects and attributes. O = {dog, cat, person, monkey, snake, fish}; A = {hair, scales, walks erect, has thumb, crawls, breathes air, breathes water}; R= formal context in Table 5.

$\sigma(Dog, Person) = \{Hair, BreathesAir\}$

$\tau(Hair, Walkserect) = \{Person, Monkey\}$

Figure 15 depicts the computed lattice for the example. For purposes of this illustration, note that objects are shown in bold type and attributes in regular type. The top of the lattice represents the concept shared by all objects (the universal concept), and is normally (as in this example) empty. Each node in the lattice represents a concept composed of one

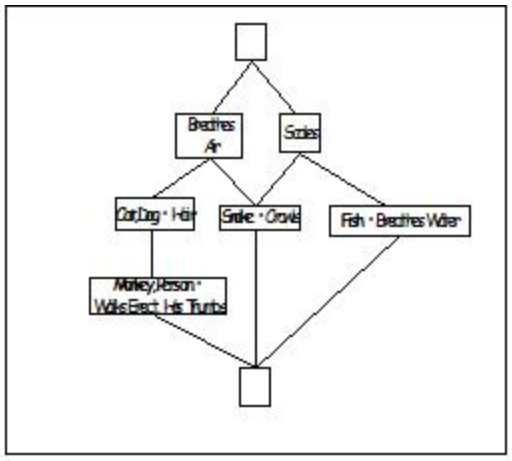

Figure 15: Concept Lattice Example

or more attributes and zero or more objects. For example, just below top are two concepts: {Breathes Air} and {Scales}. Concepts that are below a concept are called *subconcepts* while concepts above a concept are *superconcepts*. Subconcepts have all the attributes of their superconcepts in addition to any new attributes belonging to that concept. Take for example the concept corresponding to the object snake. Snake has the attributes from its superconcepts (Breathes Air and Scales) in addition to its new attribute Crawls. The bottom of the concept lattice represents the concept that has all the attributes in the set A. This concept is referred to as the empty concept. Typically (as in this example) each concept shows only the lowest point in the lattice where an object can be placed (that is where its maximal set of attributes occurs). For shorthand, only the new attributes at each concept are shown.

4.2 Automated Integration of Architectural Information

It useful to digress for a moment from a description of semantic approximation to consider the other two primary approaches being used for automated integration of architectural information: lexical and topological. We begin with a short description of each of these, followed by a more detailed description of our proposed technique which will allow more and higher quality information integration to take place.

4.2.1 Lexical Matching Techniques

The theoretical basis for this technique is founded on the idea that the names chosen for functions, code modules, source directories, etc. have a relation to their functionality. This has been explored in other approaches to reverse engineering such as library analysis [51] and function name analysis [12]. The basic technique is fairly simple. If the name of the architectural element in one perspective is the same as the name in the second perspective, they are considered to be a match—that is, they refer to the same element. While in an actual recovery effort we may have other information about an element, we are considering here using only the lexical information available.

To make this technique more robust, we can use domain synonyms and substring comparisons, rather than limit ourselves to exact lexical matches. Thus we can match "Model" to "Program Model", "Data Model", and other variations in addition to simply "Model." We also can expand common domain abbreviations using a domain dictionary. For instance, "mem" would expand to "memory" and "sys" to "system" within the domain of operating systems software.

4.2.2 Topological Matching Techniques

There are two possible approaches to using topological analysis for solving the element-matching problem. The first uses graph isomorphism. In this technique, the perspectives are matched using a graph-theoretic technique to match elements based upon matching in and out degree of the nodes to determine parts of the graph that are isomorphic. Unfortunately there are two difficulties. The first is that the general algorithm for determining isomorphic graphs is NP-Hard. The second is even more serious. Since each perspective is potentially only partially complete, there may be nodes and edges missing in the two graphs. Attempting to match only partially complete graphs is a fruitless exercise.

The second use of topological analysis is as an informal support to guide matching activities in conjunction with the lexical matching previously accomplished. Once a component in the new perspective has been matched with one in the base perspective, the edges can be used to select the next node to match. In this way, node matching also helps resolve the

edge matching.

4.2.3 Semantic Techniques

The shortcoming of lexical and topological based matching is that they both use superficial syntactic information (names and graph characteristics) to inform decisions about integrating information. Ideally, if we could match information about architectural elements based upon what we knew about their actual functionality, our matching accuracy would improve dramatically.

Previous research in this area has focused on heavyweight methods like specification matching [70]. The problem with these approaches is that although highly accurate, they require an analyst to write a specification for each architectural element, which is clearly not feasible for large scale systems. Our goal for semantic approximation was to provide a lightweight, automated method for approximating the semantics of an architectural component.

4.2.4 Matching Definitions

Before we discuss specifics of information integration, we need to establish a common vocabulary to discuss integrating information between two perspectives. We refer to the process of combining information about specific elements in two different perspectives as matching. Let P_1 and P_2 be two perspectives whose elements we wish to combine. For any element $e_1 \in P_1$ and $e_2 \in P_2$, we can have one of the following possible match results:

1. EXACT(el_1 , el_2). This relation is true if e_1 is an exact match for e_2—that is all attribute values of e_1 are also attribute values of e_2 and vice versa. Using our previous concept analysis notation:

$$EXACT(e_1, e_2) \doteq \sigma(e_1) = \sigma(e_2)$$

   Note this relation is symmetric, i.e., EXACT(e_1, e_2) = EXACT(e_2, e_1). Informally, one can reason that an exact match means each piece of information we know about element e_1 is also true for e_2 and vice versa-thus we can infer that the elements are the same.

2. SUBSUME(e_1, e_2). This relation is true if e_1 subsumes the description of e_2. This occurs when the set of attribute values of e_2 is a proper subset of e_1. More formally, $SUBSUME(e_1, e_2) \doteq \sigma(e2) \subset \sigma(e1)$. Note this relation is asymmetric, that is $SUBSUME(e_1, e_2) \neq SUBSUME(e_2, e_1)$. Informally, we can say that everything we know about e_2 is true about e_1, and that there is additional information about e_1 that is beyond anything we know about e_2. From this we can infer that most likely these are the same elements, but we have increased the amount of information we know about them using the whatever technique that created the perspective.

3. CONTAIN(e_1, e_2). Any component or connector within a specific representation may be decomposed into another representation made up of another set of elements. We refer to this set of elements as a subcomponent of the component or connector that was decomposed. The CONTAIN relation is true if e_2 is part of the subcomponent of e_1. This occurs when we can match e_2 using EXACT, SUBSUME or OVERLAP to an element in the subcomponent of e_1. This relation is also asymmetric such that $CONTAIN(e_1, e_2) \neq CONTAIN(e_2, e_1)$. Informally, we can infer e_2 as a subelement of the element represented by e_1.

4. OVERLAP(e_1, e_2). This relation is true when e_1 overlaps the description of e_2. This occurs when e1 has attribute values in common with e_2, but has other attribute values that are different. $OVERLAP(e_1, e_2) \doteq (\sigma(e_1) \cap \sigma(e_2) \neq \oslash) \wedge (\sigma(e_1) - \sigma(e_2) \neq \oslash) \wedge (\sigma(e_2) - \sigma(e_1) \neq \oslash)$. OVERLAP is symmetric thus OVERLAP(e_1, e_2) = OVERLAP(e_2, e_1). Informally, e_1 and e_2 have some information in common, but they also have additional information that is unique. This situation makes it difficult to infer the proper relationship between the two elements. The more attributes that the elements have in common, the more likely it is that they are related to each other.

5. NOREL(e_1,e_i). This relation is true if e_1 and e_i have no apparent commonality—that is they are not related by the EXACT, SUBSUME, OVERLAP or CONTAIN relations. $NOREL(e_1, e_i) \doteq \sigma(e_1) \cap \sigma(e_i) = \oslash$. NOREL is symmetric, that is NOREL(e_1 , e_i) == NOREL(e_i, e_1). Informally, if an element e_1 is NOREL with every element

in P_2, we have a newly discovered element in the architectural description.

4.2.5 Semantic Approximation

Our goal was to obtain the advantages of semantic matching, but in a fashion that would lend itself to automation. We have developed a lightweight approach to matching which we call *semantic approximation*. (By lightweight we imply both minimal effort and minimal training required to use the technique.)

The fundamental idea behind the technique is a variation on text summarization. If we were to write a paragraph that described an architectural element's semantics, and then eliminated all the noise words, we would be left with a list of terms that approximated the semantics of that element. If we did this for all elements in the system being recovered, we would have a set of elements, each of which had its own set of words that summarized its functionality.

Recall from section 4.1, we discussed concept analysis and stated that it was made up of the triple (O, A, R). For semantic approximation, we let O be the set of elements (components), A be the set of all the words extracted, and R be the relation that is true if word a is in the description of element o. From this triple, we can build formal contexts that allow concept lattices to be constructed. How then might we use lattice information to determine matching in some automated manner? We now provide a simple example that will demonstrate the technique in action.

For this introductory example we will discuss only components as elements and ignore connectors. This is done only for brevity in the description and not because connectors are deemed unimportant. Figure 16 depicts two simple perspectives of an architecture. In order to eliminate ambiguity caused by similar names in the two perspectives, we prefix the element names by their system names. After mining the text for information, we build a formal context (Table 6 and compute the concept lattice using the concepts software package [44] to produce Figure 17.

For this simple example, we can examine the lattice and detect the relations manually as follows:

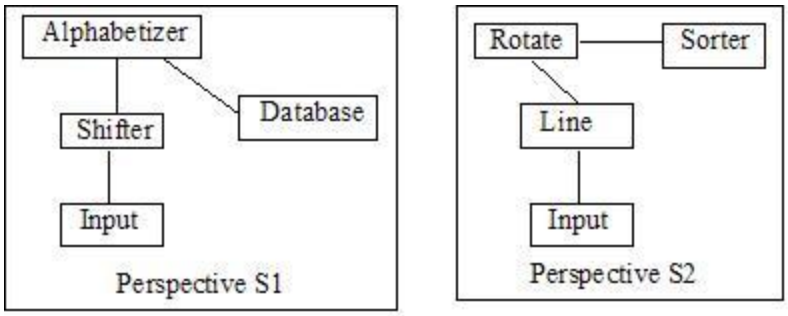

Figure 16: Two Perspectives for KWIC Architecture

The EXACT relation is the easiest to detect graphically. The two architectural elements form part of the same concept, meaning they are members of a maximal set of elements sharing the same domain attributes. This is shown in Figure 17 where Alphabetizer in system S1 and Sorter in system S2 map to the same concept in the lattice.

The NOREL relation is the next easiest to discern. Elements with a NOREL relation have no common attributes therefore their least upper bound (join) is the universal concept (or top) and their least lower bound (meet) is the empty concept (or bottom). This is shown by the concepts Line and Database in the lattice.

The SUBSUME relation means that one concept is a superconcept of another. The S1.Input and S2.Input nodes in Figure 17 show this relation.

Finally, the OVERLAP function can be found when two concepts have a least upper bound (join) that is not the universal concept. This is shown by nodes Shifter and Rotate, which have the concept Lexeme as their least upper bound.

To accomplish automated traversal of the lattice, we use the following algorithm that is implemented using the graph template library (GTL) (Raitner, Forster et al. 1999). This algorithm computes the EXACT, SUBSUME, OVERLAP, CONTAIN and NOREL relations from a concept lattice.

1. Set the start node to be the universal concept.

2. Conduct a depth-first search (dfs).

3. Iterate through the node list in dfs order

Table 6: Formal Context of Systems S1 and S2

Components	Change	IO	Data	Persistent	Lexeme	Reorder	Order	File	Lexical	Access
S1.Alphabetizer							X		X	
S1.Shifter	X				X					
S1.Input		X						X		X
S1.Database				X						
S2.Rotate					X	X				
S2.Line			X							
S2.Sorter							X		X	
S2.Input		X								

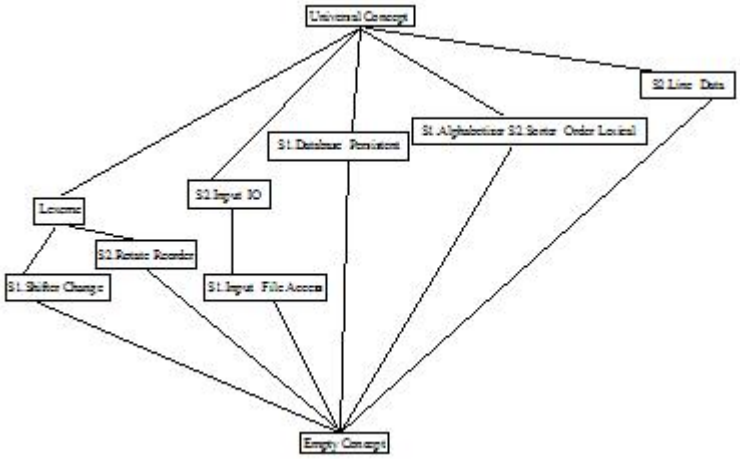

Figure 17: Computed Concept Lattice for KWIC

4. For each node we check first whether the node has multiple element names in the node label. If so, it is marked as an EXACT match and the all elements at that node are marked as used.

5. If the node is not an EXACT, we check to see if the indegree of the node ==1 and its parent is not the start node. If these conditions are true, we look at the parent to find if its label contains an element name. If so, we have a SUBSUME relation. If not we recursively check these conditions back up the node list until we either find an element name in a label or find the start node. In the latter case the SUBSUME check fails.

6. If the node is not EXACT or SUBSUME, we look for the weakest condition, OVERLAP. In this case we backtrack up the node list until we find a parent with an out-degree¿=2. In this case we follow the new portion of the tree to find the overlap condition.

7. After all nodes in the dfs visit list have been checked, then any that are not EXACT, SUBSUME, or OVERLAP are designated NOREL.

8. The CONTAIN relation is inferred from the EXACT, SUBSUME, or OVERLAP relation by examining the node labels (using lexical analysis) to recognize which are subsystem names and which are top-level system names.

9. We also compute a simple metric while checking for the OVERLAP relation. This metric is a measure of the degree of commonality of the elements in the relation. It is derived by taking the number of common attributes and dividing by the total number of attributes. For our example, the overlap measure for elements Shifter and Rotate would be .33. This measure provides a way for the analyst to determine whether the OVERLAP relation is significant enough to warrant a match.

We have now specified a matching technique based upon lightweight semantic information rather than more superficial syntactic information. Recall that we want to improve the quantity and quality of information that is automatically integrated during the architectural recovery process. We must now show that this technique will in fact improve the integration task and reduce human involvement in the recovery process. The next chapter will show that this is true by comparing the results of semantic approximation with the results obtained via typical lexical and topologic integration techniques.

4.2.6 Semantic Approximation Tool Support

Recall from Chapter 3, we discussed details of the REMORA toolkit. In this section we present typical tool outputs to allow the reader to better understand how the algorithm works.

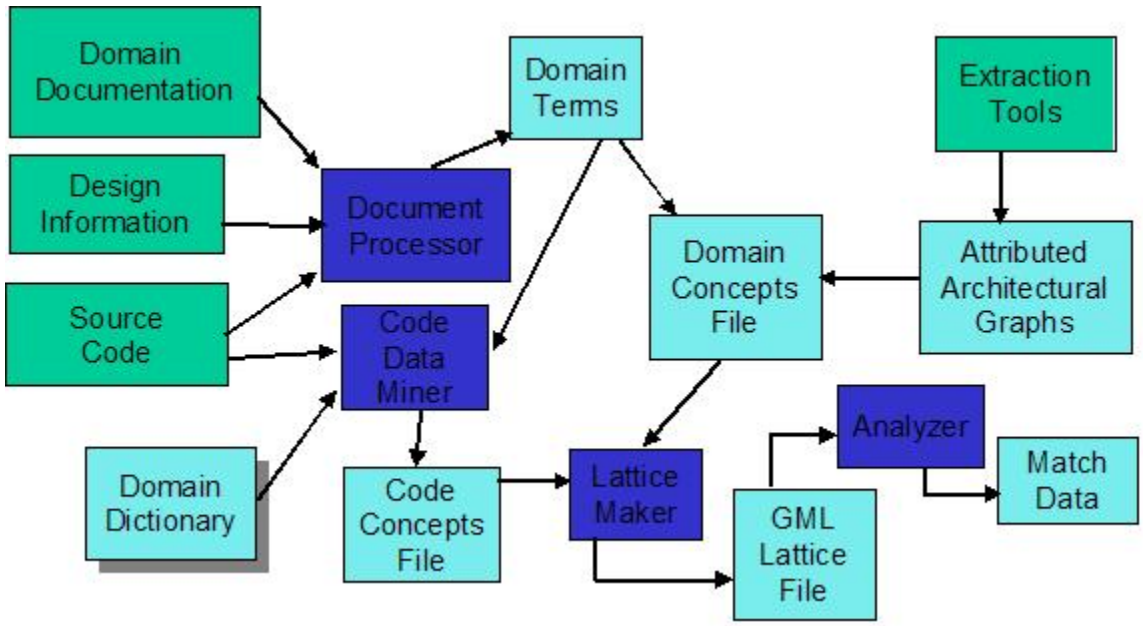

Figure 18: Semantic Approximation Information Flow

Figure 18 reviews the information flow for semantic approximation tools. We now look at typical outputs of the various tools that make up the semantic approximation portion of REMORA.

After mining the documentation for the domain terms (which produces a simple word list), those terms are associated with the appropriate architectural elements. This produces the domain concepts file for the first view (S1):

```
S1.Alphabatizer: Order Lexical;
S1.Shifter: Change Lexeme;
S1.Input: IO File Access;
S1.Database: Persistent;
```

This file would be combined with the domain concepts file for the second view of the architecture (S2) and the combined file run through the lattice builder. The output of the lattice builder is a *GML* file such as:

```
graph [
 version 2
 label "Context Lattice"
 directed 0
node [
  id 0
  label "S1.Alphabatizer S2.Sorter Order Lexical"
]
```

This GML file is then fed into the Analyzer which produces match data in the following format:

```
EXACT S1.Alphabatizer -> S2.Sorter Overlap=100
NOREL S1.Database -> Overlap=0
OVERLAP S1.Shifter -> S2.Rotate Overlap=33
SUBSUME S1.Input -> S2.Input Overlap=33
NOREL S2.Line -> Overlap=0
```

Finally, the match data is analyzed to produce the best matches available. The next chapter will give more detailed information on a non-trivial application of this tool set.

CHAPTER V

EVALUATING SEMANTIC APPROXIMATION FOR FUNCTIONAL VIEWS

The previous chapter presented the theory and concepts behind using semantic approximation to improve the quality and amount of information that is automatically integrated from legacy systems. This chapter describes the experiment that was conducted to validate semantic approximation when used for functional views. By functional views we mean specifically, the use of semantic approximation to look at integrating architectural information that could be localized to specific architectural elements.

The chapter is organized into three principle parts, a description of the experiment, presentation of the results, and a discussion of those results. The next chapter will provide more detailed cause analysis and conclusions drawn from this experiment.

5.1 Experimental Design

There are several questions whose answers will help in automating the software architecture recovery process and assist in overcoming the concept assignment gap between code and architecture. The specific questions underlying this experiment include:

1. What forms of documentation provide the most useful information in automated recovery and integration of architectural information?

2. How effective is the semantic approximation technique for assisting in the integration of multiple sources of information, as opposed to simply lexical or topological methods?

3. What techniques of document processing provide the best sets of components, connectors and properties to use for the recovery process?

To answer these questions we have the following informally stated hypotheses:

1. Documentation is most effectively used in three groupings:

   - Domain level documentation, which provides information about the highest-level abstractions in the architecture—specifically the top-level components, connectors and their properties.
   - Implementation level documentation, which provides information about the concrete realization of the abstract components and connectors into a specific system.
   - User level documentation, which provides a bridge between the abstract and the concrete elements of the architecture.

2. An intermediate level of text processing, which includes use of standard text preprocessing (tokenization, stemming) and use of a simple neural net, is more effective than simple statistical analysis (i.e. word frequency) but just as effective as more complex mechanisms such as Self-Organizing Maps (SOM) models.

3. Semantic approximation provides superior results in integrating and combining architectural information over using simple lexical matching or topological information.

To answer these questions we require several axes of evaluation. First, we need to vary the types of documents used in the experiment. Specifically we will treat the documentation as one complete group by itself and as three separate groups categorized into domain, user and implementation levels. Secondly, we need to vary the types of techniques used to combine architectural information. We will limit the experiment to the three previously discussed techniques: Lexical (L), Topological (T) and Semantic Approximation (S). These three techniques will be evaluated using 7 different combinations: L, T, S, LT, LS, TS, and LTS. Thirdly there is a difference in levels of abstraction (High (H) and Low (L)) that generally corresponds to the differences in concepts from domain to implementation. Finally we want to look at three methods for deriving information from documentation: Simple statistical, Simple Neural Net, and Self-Organizing Maps (SOM).

We impose the following limitations and assumptions to keep the experiment manageable:

1. We use the MORPH stemming algorithm. We assume that other stemming techniques (such as Porter [58] and Lovis [45]) will not have a significant effect on the results.

2. We use a domain synonym file to help detect multiple terms referring to the same architectural element. We also use the domain file to expand abbreviations (such as *sys* into *system*) based upon the domain of the recovery effort. We assume that this domain dictionary reduces the variation of terms caused by multiple terms being used to refer to the same item.

3. We use WordNet to assist in disambiguating synonyms and finding other relationships between words. We assume that this usage improves the quality of the terms and associations found in the document set.

5.1.1 Experiment Process

We followed the following steps in performing this experiment:

1. Gather and classify applicable documentation as presented in the previous section.

2. Extract architectural representations of the Linux Operating System Kernel 2.4 and the Interactive Scenario Visualization (ISVis) 1.1 reverse engineering tool at both a High and Low level of abstraction.

3. Create Truth Data by using previous peer-reviewed related work in recovering the architectures of these systems. By truth data, we mean an *oracle* by which we judge whether an answer obtained in the experiment is correct or not.

4. Process the documentation using three techniques:

   - Simple Statistical Processing
   - Use of Simple Neural Net (Bayesian Net using *libbow* [53])
   - Use of SOM model (using TextAnalyst [49] commercial software)

5. Build a Formal context for each of the representations extracted using the information extracted from the documentation.

6. Perform integration of the representations using the seven combinations of Lexical (L), Topological (T), and Semantic Approximation (S) techniques.

7. Analyze the results of the integration using the truth data categorizing the matches as false positive, false negative, true positive, partial positive and true negative where these terms are defined as:

    - False Positive: The technique identified two elements as matching when they were not matches.
    - False Negative: The technique identified an element in one perspective as not matching anything in the other perspective, when it should have matched at least one other element.
    - True Positive: The technique matched two elements, and this was a correct result.
    - Partial Positive: The technique matched two elements, but at a lower strength than what was true. For example, it claimed OVERLAP(e_1,e_2) when in fact the correct result was EXACT(e_1,e_2).
    - True Negative: The technique identified an element in one perspective as not matching anything in the other perspective and this was a correct result.

    Success for this experiment would be a 90% or greater correct match rate for the true postive and true negative results.

5.1.2 Documentation Used in Experiment

Appendix A provides a list of the documentation used for the ISVis and Linux systems. This appendix serves two purposes. First it allows the reader to repeat the experiment if desired using the same documentation base for mining activities. Secondly, it gives the reader an understanding of the types of documentation available for extraction.

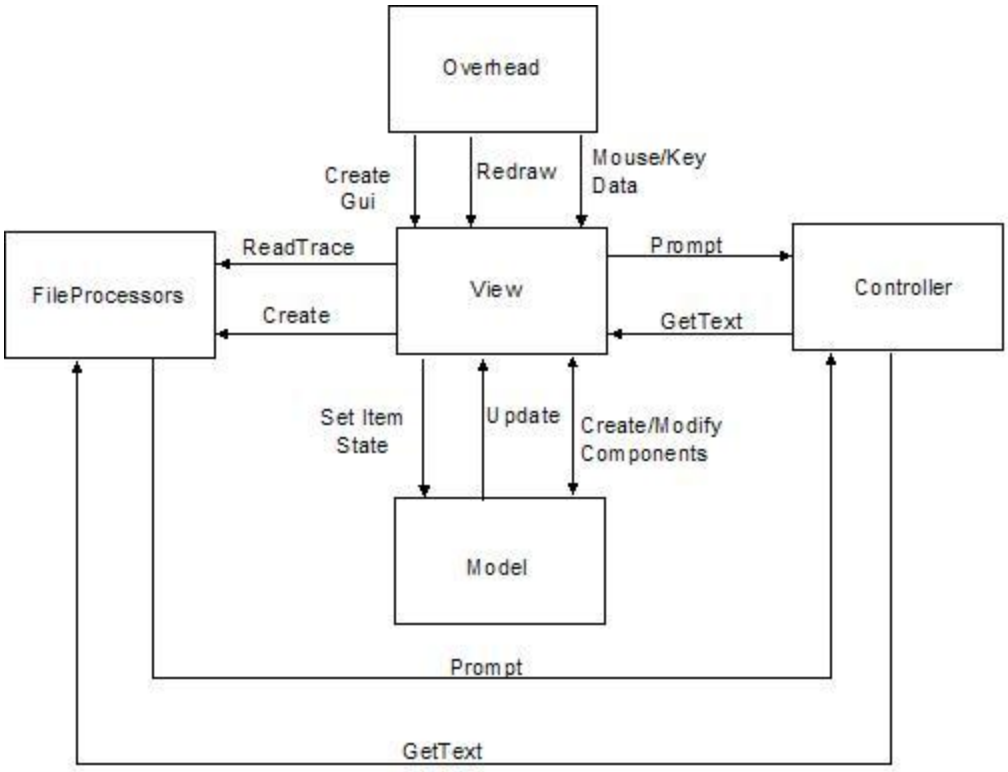

Figure 19: ISVis Dynamically Extracted Architecture

5.1.3 Architectural Perspectives Used in the Experiment

Figure 23 and Figure 24 depict the two top-level architectural perspectives of Linux used for the experiment. Figure 24 comes from an extraction effort by Bowman [11]. Figure 23 comes from Maxwell's Linux kernel commentary [48]. Figure 19, Figure 20 and Figure 21 depict the high-level views of the ISVis System. Figure 19 and Figure 21 was obtained by running ISVis on itself to extract the architecture, while Figure 22 came from the original design documentation. Figure 21 and Figure 22 are the subcomponent architectures and are provided to help the reader understand the process of building up a final architectural view.

For the low-level perspectives, we used a static call graph generated by *cflow* and the output of the *ctags* program. *Ctags* provides information about functions, the files they are in, and the line numbers they are defined in. *Ctags* would be a typical supporting tool for browsing code. Since there are 49,816 functions in the Linux kernel, full graphical representations of the low-level perspectives are not presented. Part of the automated

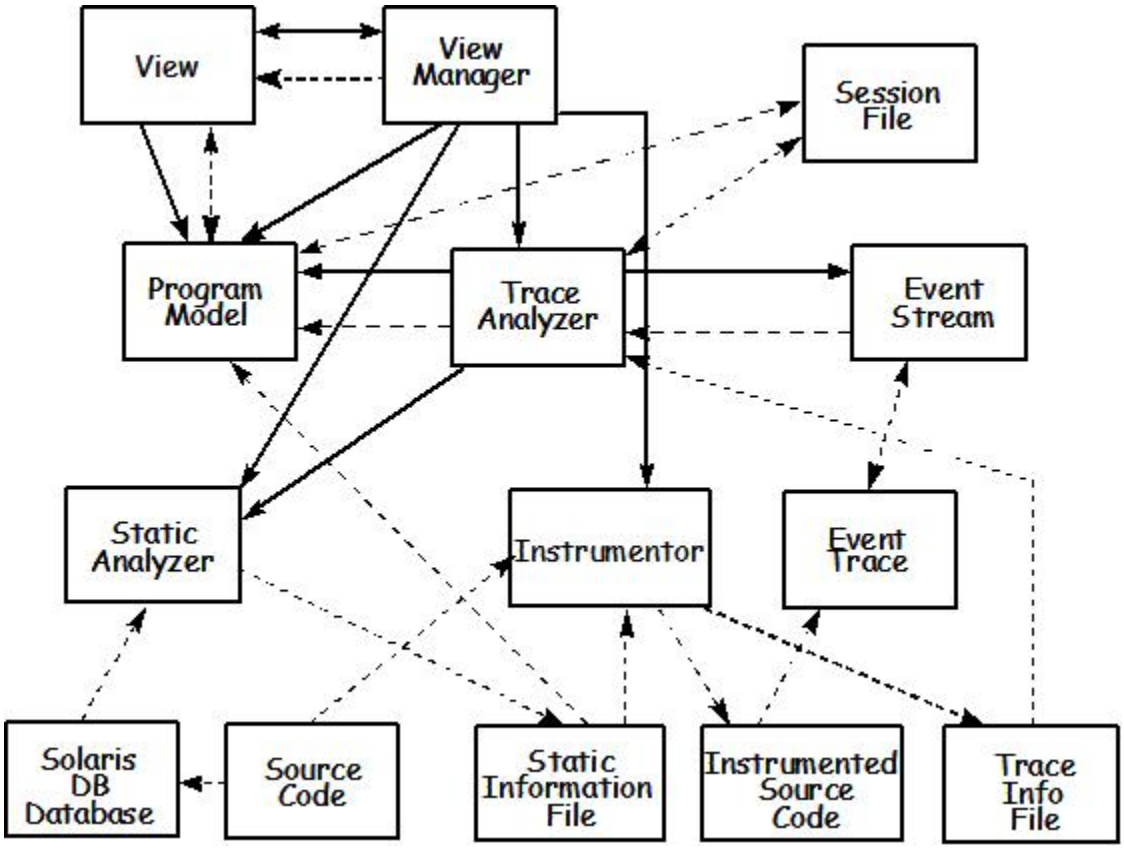

Figure 20: ISVis Design Architecture

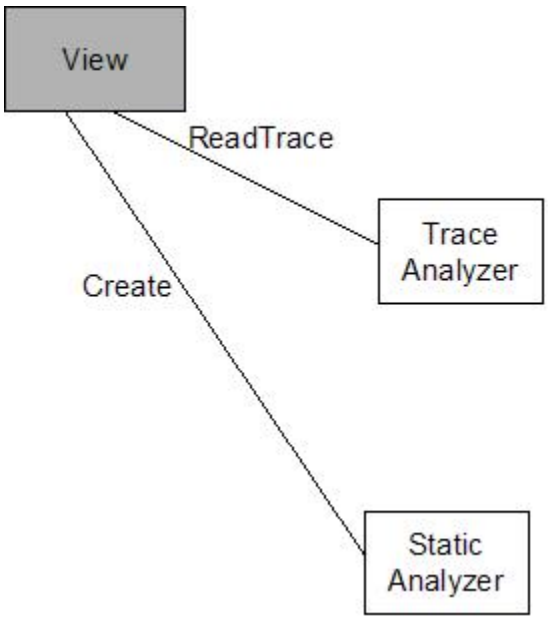

Figure 21: ISVis Dynamic Recovery, File Processor Subcomponents

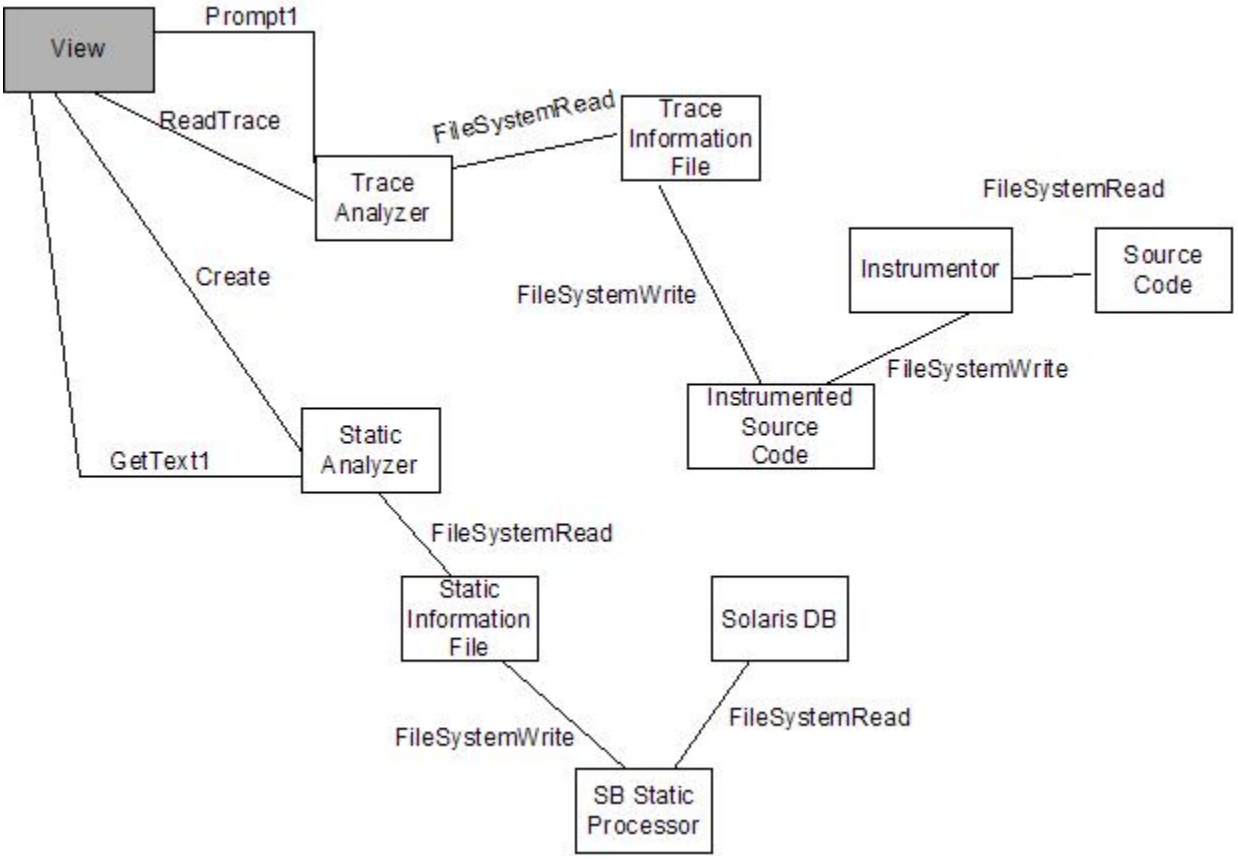

Figure 22: ISVis File Processors Subcomponents, Actual Architecture

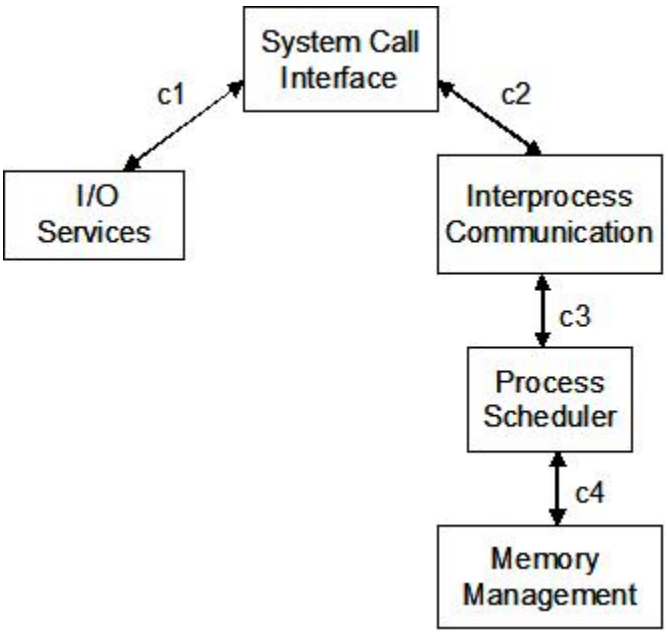

Figure 23: Linux High-Level Domain Architecture

support provided in REMORA is a component that takes *ctags* and *cflow* output and converts that output to a *gml* file so the information can be manipulated as an attributed graph. Figure 26 and Figure 27 depict the partial perspectives for the low-level Linux architectural information. For reference, Figure 25 depicts a snapshot of the raw *ctags* information from which Figure 27 is derived. The ISVis low-level perspectives are similar presentation and are thus omitted from this dissertation as figures.

5.1.4 Truth Data

Detailed truth data for the architecture element matching is shown in Appendix B, Table 11 through Table 15.

5.1.4.1 High-Level Truth Data

High-level truth data was derived by manual analysis of the components in the different perspectives and using expert analysis to decide the appropriate matchings. This expert analysis included Linux kernel developers and on-line documentation for the kernel design.

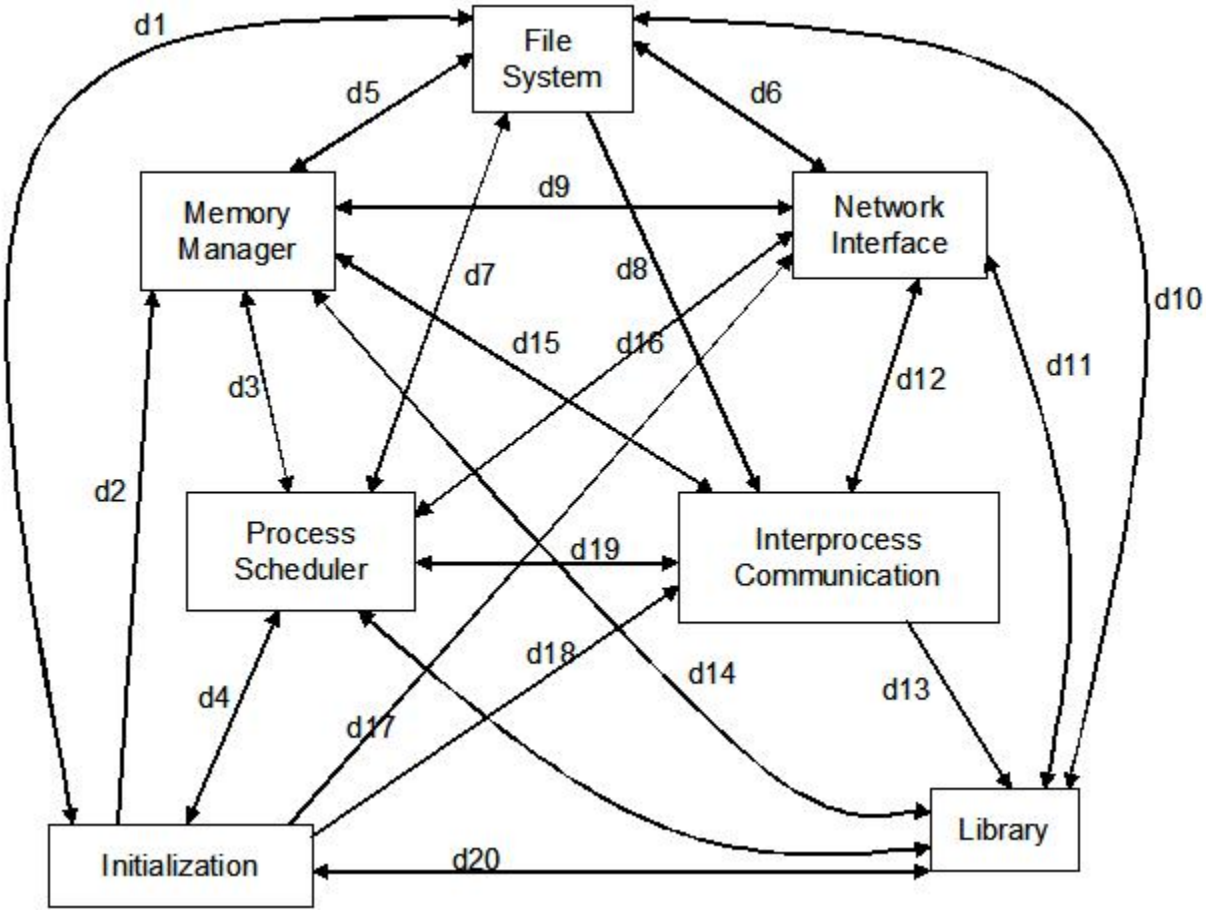

Figure 24: Linux Conceptual Architecture (High)

```
printk      h:\linuxdocs\source\linux\kernel\printk.c      /^asmlinkage int printk(const char *fmt, ...)$/;"       f
start_kernel        h:\linuxdocs\source\linux\init\main.c       /^asmlinkage void __init start_kernel(void)$/;"       f
setup_arch  h:\linuxdocs\source\linux\arch\i386\kernel\setup.c      /^void __init setup_arch(char **cmdline_p)$/;"       f
setup_memory_region  h:\linuxdocs\source\linux\arch\i386\kernel\setup.c      /^static void __init setup_memory_region(void)$/;"      f file:
memset      h:\linuxdocs\source\linux\arch\i386\boot\compressed\misc.c       /^static void* memset(void* s, int c, size_t n)$/;"      f file:
trap_init   h:\linuxdocs\source\linux\arch\i386\kernel\traps.c      /^void __init trap_init(void)$/;"   f
initialize_secondary    h:\linuxdocs\source\linux\arch\i386\kernel\smpboot.c     /^void __init initialize_secondary(void)$/;"   f
```

Figure 25: Raw Ctags Output (Partial) (Low)

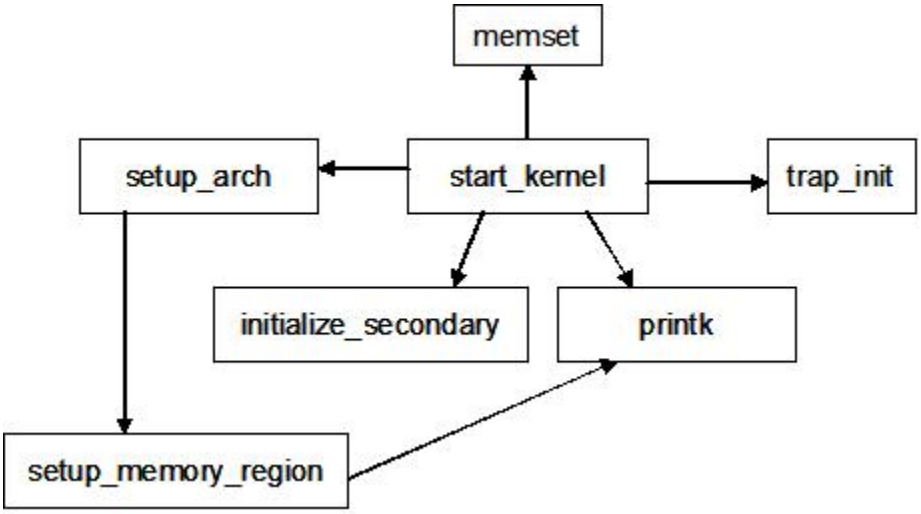

Figure 26: Linux Call-Graph (Partial) (Low)

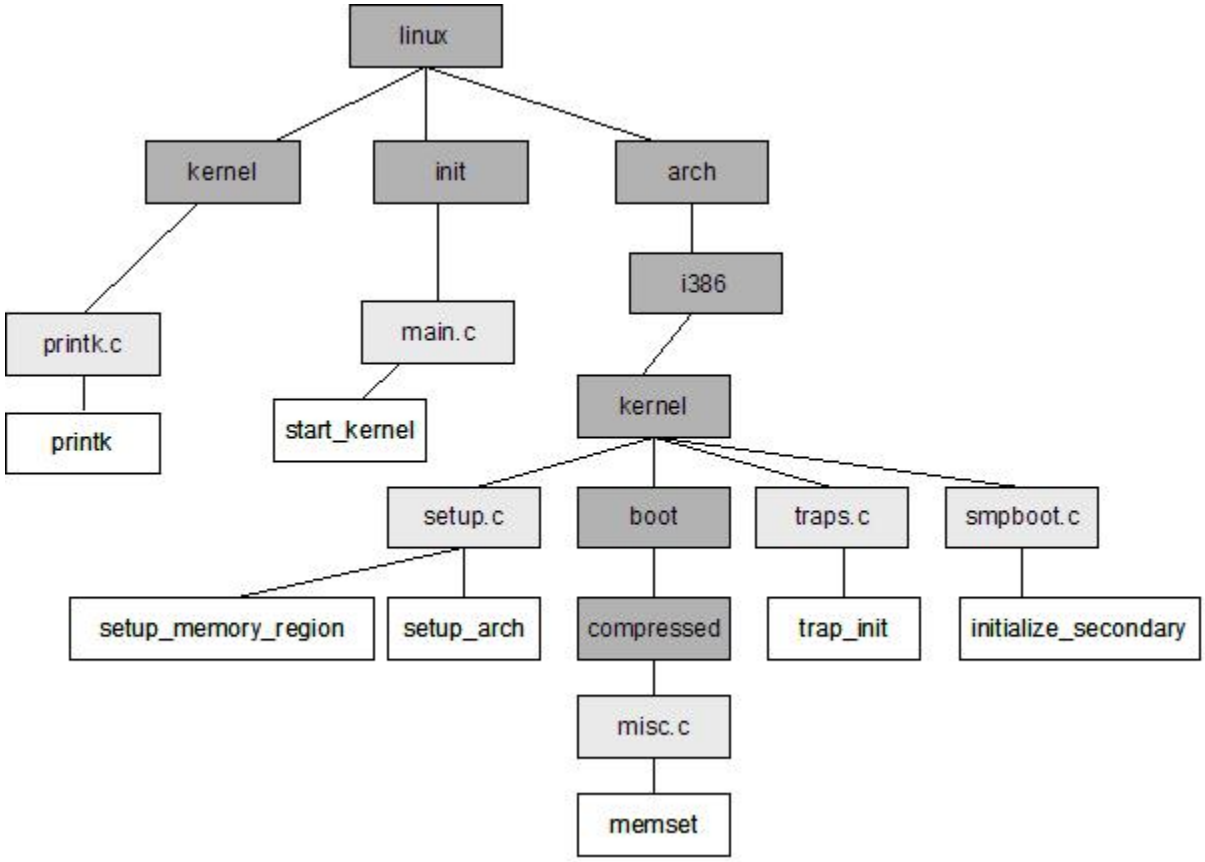

Figure 27: Linux Ctags Perspective (Partial) (Low)

5.1.4.2 Low-Level Truth Data

Since low-level data maps code to code, the component truth data is fairly trivial. Function names map directly to function names as EXACT matches, while directory and file names disambiguate similar function names. For connectors, the semantics of the two perspectives are completely different. In the call graph view, connectors represent calls relations between functions. In the *ctags* view, connectors represent contain relations between functions, directories and files. This makes all connector matching in the low level perspectives NOREL.

5.1.4.3 High-Low Truth Data

For the high-low integration experiment, we are attempting to combine information from the Linux domain-level architecture (Figure 25) and the Linux call graph (Figure 26). For the ISVis system, we combine the dynamically extracted representation (Figure 19) and the ISVis call graph. Because we are matching lower-level code entities to high-level abstractions, the majority of relations are CONTAIN. Because some specific implementation concerns like booting and general library functions are not represented in the domain-level architecture, there are few NOREL relations for Linux.

5.2 Experimental Results

Appendices C through J contain the detailed results of the experiment. The following sections contain a discussion of those results.

5.2.1 Lexical Matching (L)

Lexical matching was first accomplished between each of the architectural perspectives at the same level of abstraction.

Table 17 shows the results for the ISVis system components, while Table 21 shows the results for the LINUX operating system components. Connectors present a difficult situation for lexical matching because of their second-class status. Frequently, and as is the case in all the perspectives except Figure 19, Figure 21, and Figure 22, they have no labels or

information other than perhaps they are control, data or depend relationships. For this reason, they have arbitrary labels and thus, lexical matching is not effective. Any true negative results (NOREL) are coincidental. For ISVis, there were 21 true negatives (coincidentally) and 18 false negatives. For Linux, there were 20 true negatives (all coincidence) and 4 false negatives.

For the low level perspective matching, the lexical technique produced 100% of the true positive results for the components and 100% of the true negative results for the connectors. This is to be expected since both perspectives were derived from low-level implementation data and thus their lexical vocabulary was fairly well constrained. For low-level matching, it is understandable why researchers using code-based approaches rely on lexical information as a principle mechanism for combining source code derived information. The excellent results for the connectors were coincidental. Since the connectors on the call-graph carried the semantics of "function call" and the connectors in the ctags perspective had the semantics of "contains", there was no relation between them in a matching sense.

For the mixed perspective matching (High-Low), the effectiveness of the matching between abstraction levels was highly dependant on the coding discipline of the developers. If function, file and directory names were meaningful, then the matching results were fairly good. For ISVis components, the results of component matching are found in Table 18. Of the false negatives in the View component, the majority were due to the ViewManager class that should have mapped to the controller, but of course had no lexical information to support that match. Since the ideas of Controller and Application-level support functionality (Overhead) is rarely used in low-level function names, none of the ISVis functionality could be mapped to these high-level components.

For Linux, the results of the high-low matching are presented in Table 22. Probably the two most striking results are the high number of false negatives (over 22,000). This is to be expected since it supports the premise for the concept assignment problem and our contention that some type of semantic-based matching is needed to support information integration for perspectives (or sets of architectural information) at different levels of abstraction. The high-low matching results for ISVis point out the same conclusions.

5.2.2 Topological Matching (T)

For LINUX, using topological matching alone was virtually useless. Since the Concrete architecture (Figure 24) is basically a $K7$ completely connected graph, there were 2040 matches of the subgraph in Figure 23 onto it. Table 23 shows the best case mapping, but what constituted a "best case" was not derivable from topological information alone.

For ISVis the results were more constrained, but still there were many possibilities that were developed. To begin with, the File Processors component was expanded in the parent graph since the design graph was one layer. As in the LINUX example, Table 19 shows the best case of all the topological matchings.

Connectors were matched by comparing edges in the graphs after nodes had been matched successfully. Table 24 and Table 19 present the results of topological matching for the high-level perspectives.

For the low-level perspectives, the sheer size of the graphs caused the automated tool to fail. A manual attempt at building graphs of this size was not feasible. We can make some reasoned approximations however, at what the results would be. First we know that the connectors have no relation to each other in the graphs, since they have different semantics (one is function call, the other contains), therefore any matches are false positives and any NOREL results are true negative (but coincidental). Likewise, looking at the components, any positive match results will be coincidental.

For the high-low matching, topological matching ran into the same problems as both the previous cases. First, since the high-level graph was relatively small compared to the low-level graph, there are a very large number of possible projections of the high-level perspective onto the lower-level perspective. In reality, the high-level nodes (and possibly edges) might themselves be representations of graphs whose nodes might also be representations of graphs. In order for topological matching to be used with any success in the high-low scenario, the low-level graph would need to be preprocessed using some type of clustering or k-cut algorithm to convert it from a flat graph into a set of hierarchical graphs which could then be more accurately matched to the higher-level representation. Developing the algorithms necessary to convert the low-level perspectives from flat graphs to hierarchical graphs is

beyond the scope of this experiment, but is a necessary component of future work so as to achieve our ultimate goal of fully automated information integration. There is also the problem of scalability of the graph matching software currently available because of the large number of nodes in the low-level perspectives.

5.2.3 Semantic Approximation Matching (S)

Table 30 shows a comparison of the top twenty-five terms obtained by mining the Linux documentation, while Table 25 shows the results for ISVis. The number twenty-five is an arbitrary number chosen to make the lattice more manageable.

These words were mined from the documentation using three different techniques. First, a simple word frequency analysis was used. These results are shown in the Raw word column. Second, the *libbow* library was used to develop a simple neural network which was trained using a subset of the documentation. This collection of words is shown in the Bayesian Net column of the tables. Finally, *TextAnalyst* was used to generate self-organizing maps (SOM) of the documentation and key words were extracted. The results of this application are not shown, but were a combination of the words in the Raw and Bayesian columns. In preliminary tests, the Raw words from the all column produced the best results and these words were used for the bulk of the experiment as the domain terms.

These terms were then associated with the components in the various architectural elements in the perspectives to create the formal contexts to be used in developing the concept lattices for the semantic approximation. For the initial test, these 25 top terms were used to build the formal contexts.

For low-level semantic approximation, since the objects and attributes were derived from the same source (automatic processing of the source code), they had the exact same formal contexts (for the components) and thus all the matches were EXACT. For the connectors, as in the high-level experiment, semantic approximation was not usable for the same reasons.

Results of semantic approximation for combining two perspectives at a high level of abstraction are presented in Table 33 and Table 28. For the high-low semantic approximation, the results are presented in Table 29 and Table 34.

Here is the output for the Linux high-level architecture (Domain Concepts File).

```
d.IOServices: write network print read disk device directory file;
d.SystemCallInterface: interface software program version systems command mode;
d.InterprocessCommunication: pipe socket remote procedure call
                             message mailbox communication;
d.ProcessScheduler: time process thread schedule;
d.MemoryMangement: block page memory data access free allocate pointer;
```

Here is a code snippet with the corresponding entry in the Code Concepts File.

```
static unsigned inptr = 0;   /* index of next byte to be processed in inbuf */
static void *malloc(int size)  {
  void *p;
  if (size <0) error("Malloc error\n");
  if (free_mem_ptr <= 0) error("Memory error\n");

  free_mem_ptr = (free_mem_ptr + 3) & ~3; /* Align */

  p = (void *)free_mem_ptr;
  free_mem_ptr += size;

  if (free_mem_ptr >= free_mem_end_ptr)
     error("\nOut of memory\n");
  return p;
}
Code above produces this entry:

linux.arch.i386.boot.compressed.misc.malloc : pointer process memory allocate free  ;
```

After the concepts files are combined and run through the lattice maker, a graph markup language (gml) file is produced. The following is an excerpt from the lattice definition in gml.

```
node [
  id 264
label "linux.arch.i386.boot.compressed.misc.malloc "
]
```

Finally, the lattice definition is processed by our semantic approximation algorithm and the following excerpt would be produced.

```
OVERLAP linux.arch.i386.boot.compressed.misc.malloc>->
            d.ProcessScheduler             Overlap =12
OVERLAP linux.arch.i386.boot.compressed.misc.malloc>->
            d.MemoryManagement             Overlap =16
```

Given this output, REMORA would classify the *malloc* function as belonging to the *MemoryManagement* component.

5.2.4 Lexical and Topological Matching (LT)

If we combine lexical and topological techniques, we can gain some optimization. Rather than blindly matching nodes based solely on graph information, we can use lexical information to match as many nodes as possible, and then use topological information to match the remaining nodes.

For ISVis, we match View with View, and then Program Model with Model based upon the edges extending from View and the lexical information available. This leaves three options for the node mapping to View Manager: Controller, Overhead or File Processors. If we choose Controller, we can duplicate the best-case results of topological matching as shown in Table 19. Again, by using Lexical information, we are able to eliminate the many incorrect subgraph mappings that are possible, and choose something that is equal to the best case available for topological matching alone.

For the low-low matching, we are able to get all exact matches again for the nodes (components) since the lexical matching is totally accurate. For the connectors, the matching using topological information gives us all false positives because the semantics of the edges in the two graphs are totally different.

For high-low matching, we have no exact matches to get us started in the combination technique. For code organized like Linux into directories that match the conceptual architecture, we can match directory names to high-level components (i.e. mm = Memory Manager) and the cluster of functions in that directory as a subgraph of the Memory Manager node. Unfortunately, code is seldom organized as neatly as that, and in fact for ISVis, all code is in the *src* directory, which gives us no clustering clues at all.

In the general case, the performance of the lexical and topological techniques is so poor for high-low matching that their combination fails to give us any improvement over their individual use.

5.2.5 Lexical and Semantic Approximation (LS)

For ISVis, the combination of Lexical and Semantic Approximation allows us to clarify most of the partial positive results and improve our confidence. In applying the techniques, we

first use Semantic Approximation to obtain possible matches, then used lexical information to further refine our choices.

For example, using lexical matching alone, we had an EXACT match between Design.View and Dynamic.View, but also had a CONTAIN between Dynamic.View and Design.View Manager. Using Semantic Approximation, we can refine the OVERLAP to an EXACT and we can identify that Design.View Manager is not related to Dynamic.View, but is instead related to the Dynamic.Controller. Overall results are shown in Table 28.

Linux's results are similar to ISVis's and are shown in Table 38. Since semantic approximation was not useful for connectors, the connector results were identical to the lexical results alone.

Recall that we perform Semantic Approximation first, then apply lexical analysis to the choices that we have. For instance, a successful application of the LS approach would look like:

```
OVERLAP linux.arch.i386.kernel.io_apic.MPBIOS_trigger>->
        d.SystemCallInterface Overlap =11
OVERLAP linux.arch.i386.kernel.io_apic.MPBIOS_trigger>->
        d.ProcessScheduler Overlap =16
OVERLAP linux.arch.i386.kernel.io_apic.MPBIOS_trigger>->
        d.IOServices Overlap =10
```

In this case, semantic approximation is used to narrow our selection choices to three. Rather than use the Overlap measure as we do in just the semantic technique (which would give us an incorrect match of ProcessScheduler), we use lexical information if available to make the final selection. In this case the lexical match chooses correctly since *io_apic* expands to input output, which matches the IOServices component.

In cases where there is no additional lexical information provided in the selections, then the technique reverts back to just Semantic Approximation and the Overlap measure is used to choose the match.

It is also possible of course for LS to produce worse results than just S alone. Consider this example:

```
OVERLAP linux.drivers.net.strip.process_ARP_packet->
        d.IOServices  Overlap=10
```

```
OVERLAP linux.drivers.net.strip.process_ARP_packet->
        d.ProcessScheduler  Overlap= 5
```

In this case, semantic alone would choose the correct match to IOServices, however, the process in process_ARP_packet matches ProcessScheduler so an incorrect match is then chosen. This happens because the word *process* has a different sense in each usage, but our tool is unable to differentiate between those senses.

5.2.6 Lexical, Topological and Semantic Approximation (LTS)

A combination of all three techniques applied to the Linux and ISVis systems yielded the results in Table 43 and Table 42. The techniques were applied in order: Semantic Approximation, Lexical, Topological. This order was chosen as the experiment progressed, and it became evident that Topological information was least useful, and needed information from the other techniques to produce reliable results.

Since high-low matching with topological information is useless, the results here were the same as LS alone.

5.2.7 Summary

Recall that our goal was to show that semantic approximation provided a significant improvement in accuracy for automatic integration of architectural information, especially when crossing the concept assignment boundary (the high-low abstraction matching). The following tables (Table 7 to Table 10 display a summary comparison of the techniques. The percentages in the tables for true results are calculated from the truth data. The percentages for the false results are calculated based upon the total component count evaluated.

It is evident that semantic approximation does improve information for integration of components, but unfortunately not as significantly as we had hoped.

5.3 Discussion of Results

5.3.1 Text Data Mining

For combining information based upon functional views of the architecture, almost all of our hypotheses about text data mining were not true. In fact, simple text frequency analysis

Table 7: ISVis High-High Summary

	Truth	L	T	S	LT	LS	LTS
True Positive	20	8 (40%)	10 (50%)	20(100%)	10 (50%)	20 (100%)	20 (100%)
True Negative	1	1 (100%)	1 (100%)	1 (100 %)	1 (100%)	1 (100%)	1 (100%)
False Positive	0	4 (19%)	0	0	0	0	0
False Negative	0	8 (38%)	10 (50%)	0	10 (50%)	0	0

Table 8: ISVis High-Low Summary

	Truth	L	T	S	LT	LS	LTS
True Positive	774	269 (35%)	0	130 (17%)	269 (35%)	120 (9%)	120 (9%)
True Negative	0	0	0	0	0	0	0
False Positive	0	27 (3%)	0	130 (17%)	27 (3%)	173 (22%)	173 (22%)
False Negative	0	490 (63%)	0	526 (68%)	490 (63%)	485 (63%)	485 (63%)

Table 9: Linux High-High Summary

	Truth	L	T	S	LT	LS	LTS
True Positive	4	0	4 (100%)	0	0	4 (100%)	4 (100%)
True Negative	20	20 (100%)	16 (80%)	0	20 (100%)	16 (80%)	20 (100%)
False Positive	0	0	4 (17%)	0	0	4 (17%)	0
False Negative	0	4 (17%)	0	0	4 (17%)	0	0

Table 10: Linux High-Low Summary

	Truth	L	T	S	LT	LS	LTS
True Positive	49326	10306 (21%)	0	17364 (35%)	10306 (21%)	15678 (32%)	15678 (32%)
True Negative	490	119 (24%)	0	360 (73%)	119 (24%)	360 (73%)	360 (73%)
False Positive	0	16882 (34%)	0	14660 (29%)	16882 (34%)	16351 (33%)	16351 (33%)
False Negative	0	22509 (45%)	0	17432 (35%)	22509 (45%)	17427 (35%)	17499 (35%)

was just as effective as the more sophisticated and computationally complex techniques that used neural net technology. This is because the neural net based processing algorithms were optimized for information retrieval tasks. Since these are focused primarily on discerning keywords which will help discriminate between documents, they did not produce the kind of domain concepts we needed (i.e. what are the things most common across multiple documents). Semantic approximation requires that we develop of set of key descriptive terms for the particular domain that the legacy system we are analyzing is supporting. The results indicate that for architectural recovery tasks, the terms most frequently used in the documentation (after removal of stop words) are the ones that are most important in the domain.

There was also no significant benefit to dividing the documentation into three categories. Processing all the documents as a group seemed to provide just as much benefit as dividing them into subsets and processing them in different combinations. It was the case that whether a writer was producing a high-level manual for a systems administrator, or a lower-level manual aimed at a developer (such as a kernel hackers guide), the words and terms chosen were still oriented to the domain under study.

SOM-based semantic nets—like *TextAnalyst* described earlier—may however prove useful in automating the task of building the context matrix for high-level views. The ability to associate concepts with other concepts from a free-text document is a powerful feature. Adaptation of these neural networks would be an area of collaboration for future work.

5.3.2 Semantic Approximation

Our hypothesis about semantic approximation was somewhat true for component integration at the high-high level, but not connector integration. In fact, the most accurate results were obtained with a combination of lexical and semantic matching. Topological matching was not that useful for many reasons. First, the architectural perspectives being combined often were only partially complete, thus missing edges between perspectives caused problems for accurate topological comparisons. Secondly, differences in perspective abstraction levels sometimes made it impossible to use topological matching at all. The Linux call-graph

edges had no real correlation to the edges in the domain-specific architecture for operating systems. Lastly, and most significantly, there was no way to data mine information about connectors. Connectors did not explicitly exist in the source code, and thus they could not be identified or have attributes automatically assigned to them. Even more problematic is the absence of explicit connector discussion in any of the other documentation (such as domain or user).

For low-level data mining, the quality of the attributes mined is dependent on the quality of variable naming and comments in the various code modules. Clearly, if the developer does not use domain-oriented names or comments, the miner cannot detect the proper attributes. This is the reason that so many functions are in the false negative category for NOREL in the low-high matching.

There were a few "special" considerations in both the Linux and ISVis code base that made the lexical technique appear more effective than it might otherwise be. For Linux, the developers had grouped the source code into directories that matched many of the architectural domain concepts. Names of functions were fully qualified by the directory name, for example linux.mm.filemap_sync_pte_range which expands to the word list: linux memory manager file map synchronize page table range. Since "Memory Manager" is a domain element, an exact lexical match occurs. Most older legacy systems do not have the luxury of having code organized quite this well. Most are more like ISVis where everything is in the *src* directory.

Of course, the question then is, "if all of ISVis is in the *src* directory, why was lexical matching so effective?" The answer is that ISVis is a graphical intensive visualization program. Many of the functions deal with manipulation of views in the user interface. Most of the class names have the word *view* in them resulting in a large number of exact matches with the "View" component. Many of the remaining functions happen to be in class Program Model, which of course matches the Model high-level element.

Ultimately, the results of the experiment disproved our thesis statement. It was not possible to significantly improve the quality and quantity of architectural information developed in a recovery effort by using our semantic approximation technique. The next

chapter of this dissertation presents a basic root-cause analysis and a discussion of possible adjustments to the current algorithm.

CHAPTER VI

CONCLUSION

In the previous chapter, our validation experiment showed that our thesis statement was untrue in a practical sense. That is, although we were able to improve the quality and quantity of information automatically recovered, we were not able to do it in a consistent manner, or at a level that was reliable enough to be used by practitioners in the field.

This chapter will first present a root cause analysis of why our technique failed followed by a discussion of our analysis of the overall efficacy of this approach. The chapter will conclude with opportunities for future work and collaboration in this area.

6.1 Root Cause Analysis

In the previous chapter's summary, we saw that the principal problem with our technique was the excessive number of false negatives and false positives produced by our algorithm. We turn our attention first to the false negatives. What could cause so many elements to be classified NOREL when in fact they should have been matched to existing elements?

An examination of the concepts file shows that most of the NOREL's were caused by a failure of the source code data miner to assign any domain terms to the component. This in turn resulted in that component being considered no relation to any of the existing components, all of which had domain terms assigned. We then have to ask the question, "Why were there no domain terms assigned to the components?" An examination of the code reveals two primary factors.

First, there were no comments or meaningful names (methods, functions, classes, variables) from which to mine any meaningful terms. Secondly, there were comments and variable names that could have given information, but those comments did not contain any of the terms the data miner was looking for. Further examination of the code used in the experiment reveals that the former case was most prevalent. In the case of ISVis, a research

tool developed primarily by a single developer, there were often no meaningful comments for many of the functions. There is nothing our technique can do to overcome this problem. If the code has no comments or meaningful variable names, then our entire technique fails.

For Linux, there was a slightly different reason. Many of the functions in a particular file lacked any comments because there was an extensive comment in the header of the file which explained how the set of functions worked. Since the source code data miner assumes a comment describes the immediately following function, then only the first function in the file received the benefit of the comments, and the others got no attributes. For instance, in many of the networking files, the initial comment explained how the overall component worked, and thus obviated the need for further individual function comments.

What then can we do to address this root cause? The most obvious solution would be to consider the affinity of the header comment to the other functions in the file. Any concepts that are mined in the header comment will be assigned to any functions in the file which do not have their own comments or have no attributes mined from their comments. This would reduce, but not eliminate, the number of false negatives.

We then come to the second symptom, a large number of false positives. Examining the algorithm and the outputs, we can see that the false positives are caused under two situations: overlapping attributes between components and conceptual distance of the overlap in the lattice. Overlapping attributes, which are domain terms represented by the same word but which have slightly different meanings, caused many of the false positives to occur. Consider for instance consider the terms "read" and "write." They could be associated with the *Memory Manager* since we read and write memory, but they also could be associated with the *Network Interface* since we read and write to the network, or to *Interprocess Communication* since we read and write information between processes. In each of these cases the actual meaning of read and write is different, but there is no method currently available to automatically differentiate between them. During affinity analysis (the time where we assign domain terms to components), if we assign the terms "read" and "write" to all three top level components, we are guaranteed to have overlap results on all three components. Trying to disambiguate the shades of meaning automatically is beyond the capabilities of

the current text processing libraries. This relegates us to a compromise solution. If we require that all assignments of domain terms to architectural elements be disjoint, then we can eliminate this cause of overlap. In the example above, the domain terms "read" and "write" would be assigned to the component they have the greatest affinity with. This technique, however, only partially addresses the problem.

Currently the algorithm treats all concepts with the same level of confidence, regardless of how far those concepts may be separated in the lattice. If instead, we place more emphasis on concepts which are closer in the lattice (and thus closer conceptually) we should further improve the performance of our algorithm in the reduction of false positives.

6.2 Efficacy of Approach

We now analyze the overall approach for its appropriateness for further research. Some of the root cause problems we discussed above are insurmountable. Even after we apply all the improvements described above to our algorithm, there are problems that cannot be corrected by any alterations of the algorithm. For instance, we cannot control the quantity or quality of comments that the developers provide with their code.

This then leaves us to decide whether we can ever get this approach to work on legacy systems where there may be sparse commenting and poor documentation. Unfortunately, it seems that given the limitations discussed previously, this technique is not suited to poorly documented code. This then limits our approach to use on systems like Linux, where comments and documentation do exist.

We still have the problem that our sub 40% results for matching—while still an overall improvement on existing techniques—fall far short of the 90% or more reliability required for an actual practioner to effectively use our algorithm in a recovery effort. Even after applying all our root cause analysis corrections to the algorithm, it is doubtful we would approach 90% reliability.

So now we must ask the crucial question, "Should this dissertation serve as a warning to avoid a *dead end* research path?" I do not believe so. There is an interesting alternative view we could take. What if the truth data used in our validation experiment is itself wrong? For

this work, we felt it was important to validate our results against accepted, peer-reviewed work. It is important to note that in architectural recovery, where it is difficult to make the claim that an architecture is "right" or "wrong", there is the possibility that even the peer-reviewed work might contain conceptual errors.

With over 49,000 individual functions in the Linux kernel, it is doubtful that previous analysts agonized over every function when assigning them to components. It appears more likely that the analysts used the file level as a primary mechanism for making code organization decisions. This problem reinforces our entire motivation discussion from Chapter 1 which emphasized the need for an automated solution to the recovery problem. The sheer volume of information facing an analyst during recovery of a nontrivial system is staggering.

Consider as an example the function linux.arch.i386.boot.compressed.misc.malloc. This function is assigned to the *Initialization* component in the literature, but conceptually it allocates memory during the decompression of the kernel during the boot process. Our algorithm matched this to the *Memory Manager* component and conceptually we would state that this is probably correct. Although temporally, this function does have an affinity with the initialization process, its actual conceptual function is memory management. This suggests that our algorithm would be useful for conceptual code organization where we were not trying to force the functions into an existing structure.

Where else might our technique be useful? Anywhere that a first-order approximation of code organization needs to be made. For instance, consider Murphy's reflexion technique[54]. One of the first steps is to generate a mapping file that provides an initial "guess" at the relationship between a high-level architecture and the code. This step is currently done manually, and obviously can be time-consuming for large systems. Our algorithm can write out a mapping file with minimal effort and remove a large labor-intensive operation in starting the reflexion process.

This technique should also be useful in a forward engineering setting. Consider the problem of generating an architecture from a set of requirements. It should be possible to mine the requirements document for key concepts and group those concepts together in the lattice to produce a first approximation at a conceptual architecture for the system.

6.3 Future Work

Besides the obvious work on improving our algorithm in response to our root cause analysis described in the previous section, there are also many opportunities for collaborative and future work which this research has uncovered.

There were several shortcomings of many of the tools which were used in the experiment. We detail here a short list of opportunities for collaboration:

- The concept lattice algorithm requires an excessive amount of computation time. For Linux, which had over 50,000 objects and 25 attributes, the computation time required to compute the full concept lattice was measured in days rather than minutes. This provides the opportunity for collaboration with mathematics and algorithm researchers to find more efficient ways to compute the lattice.

- The concept lattice and call graphs are very dense graphs with several thousand nodes and edges. Visualizing and filtering these graphs is both difficult and important to architectural analysis activities. This provides the opportunity to work collaboratively with members of the visualization research community on ways to effectively manipulate and view large dense graphs.

- There are many areas in text analysis, natural language processing and data mining which would also improve our recovery approach. Better summarization and affinity analysis methods would improve the assignment of domain terms to objects. It would also improve the selection of the key domain terms used in the recovery.

- Finally, working with other architectural researchers to discover better ways to find connector information from existing documentation would improve the technique. Use of domain models[16] as a basis for determining high-level connectors is one example. As we pointed out in the previous chapter, the current technique is limited to component comparisons.

6.4 Conclusion

In our work, we have emphasized the necessity of achieving an automated architectural recovery solution as a means to overcome the large amounts of information an analyst is faced with. We have presented an analysis of current tool suites and techniques along with a recovery framework (ASP) within which an automated solution can be developed.

Like the sound barrier was to aviation in the mid-twentieth century, the concept assignment barrier looms as a major obstacle to further automation of the recovery effort. Our work on semantic approximation, while not totally successful, does improve the ability of automated tools to cross that barrier and improve the architectural recovery effort. Unfortunately, it was not reliable enough to declare that the concept assignment problem has been solved.

APPENDIX A

DOCUMENTATION USED IN EXPERIMENTS

A.1 ISVis Documentation Sources

A.1.1 Domain-Level Documentation

Visualizing Interactions in Program Executions(Jerding, Stasko et al. 1987)

Using Visualization for Architectural Localization and Extraction(Jerding and Rugaber 1997)

A.1.2 User-Level Documentation

ISVis User's Manual version 1.1

ISVis Tutorial Pt I

ISVis Tutorial Pt II

ISVis Tutorial Pt III

ISVis Tutorial Pt IV

A.1.3 Implementation-Level Documentation

ISVis Design Notes

ISVis version 1.1 source code

A.2 Linux Documentation Sources

A.2.1 Domain-Level Documentation

Bootstrapping a Linux System (Ghosh)

Bringing SMP to Your Operating System (Cammeresi)

Comp.os.research FAQ Parts I - III

Comparison of Server-Based Operating Systems (Bullington)

Resource Management and Deadlocks (OS Course Notes)

File Systems (OS Course Notes)

Interprocess Communications (OS Course Notes)

I/O Management (OS Course Notes)

Memory Management (OS Course Notes)

Scheduling (OS Course Notes)

Processes and Threads (Rinard)

The Extended-2 Filesystem (Oxman)

Journal File Systems (Florido)

Master Boot Record Basics (Landis)

An Operating Systems Vade Mecum (Finkel)

Operating System Introduction (Stallings Chapter 1)

Operating System Threads (Stallings Chapter 2)

Operating System Deadlock (Stallings Chapter 3)

Operating System Memory Management (Stallings Chapter 4)

Operating System Power Management (Stallings Chapter 5)

Operating System File System (Stallings Chapter 6)

Operating System Multimedia File System (Stallings Chapter 7)

Operating System Multiple Processor Systems (Stallings Chapter 8)

Operating System Security (Stallings Chapter 9)

Operating System Processes in Unix (Stallings Chapter 10)

Operating System Win 2000 File System Case Study (Stallings Chapter 11)

Operating System Implementation (Stallings Chapter 12)

The Free BSD System (Appendix A of Stallings OS Text)

The Mach File System (Appendix B of Stallings OS Text)

The Nachos File System (Appendix C of Stallings OS Text)

Signals, Traps and Interrupts (Anonymous)

The Slab Allocator: An Object-Caching Kernel Memory Allocator (Bonwick)

A.2.2 User-Level Documentation

An Overview of the LINUX Proc Filesystem (Fink)

Compiling and Installing a Linux Kernel (Ghosh)

How to use a Ramdisk for Linux (Nielsen)

Linux Primer Series (Jenkins)

Linux Administration Made Easy (Frampton)

Linux From Scratch Version 3.1 (Beekmans)

The Linux Programmer's Guide (Goldt, van der Meer, Burkett, Welsh)

The Linux Users' Guide (Greenfield)

Mandrake Linux 8 Reference Manual

Mandrake Linux 8 Install and User Guide

Mandrake Security User Guide

Securing Linux: First Steps (Lukas)

Securing and Optimizing Linux (Mourani)

Security and the Linux Router Project (Fevola)

The Linux System Administrator's Guide Version 0.7 (Wirzenius)

Syslog (Scheidler)

A.2.3 Implementation-Level Documentation

Linux Allocated Devices (Anvin)

Kernel Support for Binary Formats v1.1

Creating a Kernel Driver for the PC Speaker (Mathew)

NASM Bootstrap Tutorial (Marjamki)

Design and Implementation of the Second Extended Filesystem (Card)

Dynamically Loadable Kernel Modules

Designing Hardware for Microsoft (r) Operating Systems FAT (Microsoft)

File Locking Release Notes (Walker)

Finding All Filenames with Identical I-Node Numbers (O'Neil)

First Attempt at Creating a Bootable Live Filesystem on a CDROM (Nielsen)

The Frame Buffer Device (Uytterhoeven)

Guide to x86 BootstrappingGuide to x86 Bootstrapping (and Partitioning)

Inside the High Performance File System Part 0: Preface (Bridges)

Inside the High Performance File System Part 1: Introduction (Bridges)

Inside the High Performance File System Part 2: The SuperBlock and the SpareBlock (Bridges)

How to Kick Out A Memory Manager (Anonymous)

The Enhanced IDE drive in Linux (Anonymous)

Ioctl Numbers (Chastain)

Linux IO Mappings (Anonymous)

Kernel Level Exception Handling in Linux (Pommnitz)

The Kernel Module Loader (Petersen)

Linux Kernel Parameters (Anonymous)

The Kernel and the VFS : A Filesystem Engineer's Perspective

Linux Kernel 2.4 Internals (Aivazian)

Linux Kernel Module Programming Guide (Pomerantz)

Mandatory File Locking For The Linux Operating System (Walker)

i386 Micro Channel Architecture Support

Network Block Device (TCP version)

Mounting the Root Filesystem via NFS (Kuhlmann)

Operating Systems: The Boot Sector

Parallel Port Code for Linux

Linux and Parallel Port IDE Devices (Guenther)

How it Works – Partition Tables Version 1c (Landis)

The PCI Subsystem (Mares)

Protected Mode: A More Detailed Approach (Stephen)

The /Proc Filesystem (Bowden)

Real Time Clock Driver for Linux

Linux's SCSI Driver

Linux Serial Console

Standalone Device Drivers in Linux (Ts'o)

The Linux Kernel (Rusling)

Video Mode Selection Support 2.13 (Mares)

Virtual DMA Services (VDS)

Linux Kernel Source Code, Version 2.4

APPENDIX B

EXPERIMENT TRUTH DATA

Table 11: ISVis High-Level Truth Data (Components)

Dynamically Extracted Architecture Components (Figure 19 & Figure 21)	Design Document Architecture Components (Figure 20)	Match Result
Overhead		NOREL
View	View	EXACT
File Processors	Event Trace Event Stream Trace Info File Instrumentor Session File Source Code Static Info File Solaris DB Database Instrumented Source Code	CONTAIN
FileProcessors.TraceAnalyzer	Trace Analyzer	EXACT
FileProcessors.StaticAnalyzer	Static Analyzer	EXACT
Model	Program Model	EXACT
Controller	View Manager	EXACT

Table 12: ISVis High Truth Data (Connectors)

Dynamically Extracted Architecture Connectors (Figure 19)	Design Document Architecture Connectors (Figure 20)	Match Result
CreateGui		NOREL
Redraw		NOREL
Mouse/Key Data		NOREL
Prompt	C9	CONTAIN
GetText	C9	CONTAIN
ReadTrace		NOREL
Create		NOREL
SetItemState	C10	CONTAIN
Update	C10	CONTAIN
Create/ModifyComponents	C10	CONTAIN
Prompt1	C5	CONTAIN
	C4	NOREL
GetText1	C8	CONTAIN
	C1, C2, C3, C6, C7, C11, C12, D2, D3, D4, D5, D6, D7, D8, D9, D10	NOREL

Table 13: ISVis High-Low Truth Data

Dynamic Extracted Elements(Figure 19)	Call Graph Elements	Match Result
Overhead	All functions in class Xapplication, RogueWave utilities, ACMEInterface, DiskFile, Hash	CONTAIN
View	All functions in class Mural, InteractionListSubView, MainView, View, ScenarioView, ScenarioListSubView, FunctionListSubView, ClassListSubView, FunctionListSubView, FileListSubView, ComponentListSubView, ActorListSubView	CONTAIN
File Processors	All functions in Instrumentor, EventReader	CONTAIN
FileProcessors.TraceAnalyzer	All functions in TraceAnalyzer	CONTAIN
FileProcessors.StaticAnalyzer	All functions in StaticAnalyzer	CONTAIN
Model	All Functions in class ProgramModel, Actor, ClassActor, FunctionActor, FileActor, DotActor, Trace, TraceIterator, Scenario, ScenarioIterator, MessageTrace,ClassInfo, FunctionInfo, Event, Interaction	CONTAIN
Controller	All functions in class IOShell and ViewManager	CONTAIN

Table 14: Linux System High-Level Truth Data (Components)

Concrete Architecture Components (Figure 24)	Domain Architecture Components(Figure 23)	Match Result
Memory Manager	Memory Management	EXACT
File System	IO Services	CONTAIN
Network Interface	IO Services	CONTAIN
Process Scheduler	Process Scheduler	EXACT
Interprocess Communication	Interprocess Communication	EXACT
Initialization		NOREL
Library		NOREL
System Call Interface		NOREL

Table 15: Linux High-Level Connectors, Truth Data

Concrete Architecture Connectors (Figure 24)	Domain Architecture Connectors(Figure 23)	Match Result
D1, D2, D4, D5,D6, D7, D8, D9, D10, D11, D12, D13, D14, D15, D16, D17, D18, D20		NOREL
D19	C3	EXACT
D3	C4	EXACT
	C1,C2	NOREL

Table 16: Linux High-Low Truth Data

Linux Domain Architecture Elements(Figure 23	Linux Call Graph Elements	Match Result
Memory Management	All functions in files in /arch/i386/mm,/mm, /kernel/dma.c, exec_domain.c, /arch/i386/kernel/mtrr.c, pci-dma.c,pci-i386.c, pci-pc.c,(530 functions)	CONTAIN
IO Services	All functions in files in /drivers, /fs, /net, /kernel/pm.c, /arch/i386/kernel/acpi.c, apm.c, apic.c, dmi_scan.c, io_apic.c, ioport.c, mca.c, mpparse.c, microcode.c ,msr.c, nmi.c (48243 Functions)	CONTAIN
Process Scheduler	/kernel/acct.c, capability.c, context.c, sched.c, fork.c, ptrace.c, resource.c, sys.c, user.c and exit.c, /include/linux/sched.h, /arch/i386/kernel/irq.c, ldt.c, process.c, ptrace.c (231 Functions)	CONTAIN
Interprocess Communication	All functions in files in /ipc; /include/asm-i386/signal.h, /include/linux/signal.h, /arch/i386/kernel/irq.c, irq.h, i8259.c, pci-irq.c, semaphore.c, signal.c, traps.c, /kernel/softirq.c (213 Functions)	CONTAIN
System Call Interface	/arch/i386/kernel/entry.S, head.S, trampoline.S, sys_i386.c, /kernel/sys.c, info.c, sysctl.c, uid16.c, and time.c (109 Functions)	CONTAIN
	(Functions related to library support of other modules) All functions in /lib directory except those in inflate.c; /kernel/itimer.c, panic.c, printk.c; /arch/i386/math-emu, /arch/i386/kernel/i387.c, time.c (297 Functions)	NOREL
	(functions related to startup and boot time functionality) All functions in /init; /lib/inflate.c, , /arch/i386/boot, /arch/i386/kernel/bluesmoke.c, cpuid.c, setup.c, smpboot.c (128 Functions)	NOREL
	(Functions related to dynamic kernel modifications) /kernel/kmod.c, ksyms.c, module.c (41 Functions)	NOREL
	(Functions related to smp processing) /arch/i386/kernel/smp.c (24 Functions)	NOREL

APPENDIX C

MATERIALS USED

C.1 Case Study Applications

C.1.1 ISVis

The Interaction Scenario Visualizer (ISVis) is a C++/Motif application that allows developers to view static program information and dynamic event traces of program executions. This program information is then used to create high-level components and interaction scenarios (connectors). The application has 24,333 non-comment lines of code distributed in 30 source files. It was written by a single developer to demonstrate the utility of an *Information Mural* to visualize large event traces. Since this was a proof-of-concept application written by one developer in a research environment, it represents a worse-case scenario in terms of program comments and documentation.

C.1.2 Linux

Linux is an open-source operating system similar to UNIX. The Linux kernel source code contains over 49,800 functions distributed in over 1000 separate files. The source code is developed and maintained by many different people, but it adheres to a prescribed coding standard and is relatively well commented. This software product represents a normative-case scenario in terms of program comments and documentation.

C.2 Software Tools used in Experiment

C.2.1 libbow[53]

This is a text-processing library written in C which supports statistical analysis and information retrieval tasks. Some of the specific features of the library include:

- Support for manipulation of N-grams.
- Creation and analysis of word vectors.

- Scoring queries for information retrieval.
- Clustering and Classification of documents.
- Maintenance of sparse matricies for document and token count correlation.
- Pruning vocabulary by information gain and word count.

C.2.2 WordNet[52]

WordNet is a lexical analysis tool developed by Princeton University. It consists of a database and access libraries written in C. These C libraries are are used in Tcl/Tk and Java-based access systems also. WordNet can analyze nouns, verbs, adjectives and adverbs to determine relationships such as synonym and antonym. Wordnet also ships with a `lex` specification for the Morph stemming algorithm which was used a separate component for the experiment.

C.2.3 TextAnalyst[49]

TextAnalyst is a commercial package that uses self-organizing maps (SOM) to perform sophisticated document summarization, word proximity analysis and other text processing tasks. TextAnalyst is available as a COM component with a C++ interface.

C.2.4 Link Grammar Parser[28]

The Link Grammar Parser (LGP) allows parsing and analysis of sentences according to their parts of speech. It is delivered with a C API, and has third-party support for Java and C#. By parsing the sentence using LGP, the parts of speech of each word can be determined and used for analysis by other tools.

C.2.5 GraphGrep[29]

GraphGrep is a tool for performing searches for a query graph in a database of graphs. This allows an application to find all occurrences of a pattern in each graph. It uses a unique algorithm to increase efficiency since graph matching is an NP complete problem. GraphGrep is a set of C programs which run from the command line.

C.2.6 Graphlet, GML Parser and Graph Template Library(GTL)[31]

Graphlet is a Tcl/Tk graph visualization and layout application. It is user-extensible via standard Tcl/Tk and a Tcl extension called Graphscript. The GML parser is a C library which allows parsing of graph specifications in the Graph Markup Language. The GTL is a library which emulates the C++ Standard Template Library(STL), except it provides data structures and algorithms for graphs. The core application of Graphlet was modified for this project by augmenting it with the ability to display and manipulate architectures specified in ACME.

C.2.7 AcmeLib[1]

AcmeLib is a library for applications wishing to use the ACME Architectural Description Language to describe software architectures. There is a C++ and a Java version of the library available.

C.2.8 Concepts[44]

Concepts is a Lindig's C library for computing concept lattices from formal contexts. This library was modified for this project by augmenting it with the capability to write lattice descriptions in GML.

APPENDIX D

ISVIS LEXICAL AND TOPOLOGICAL RESULTS

Table 17: ISVis Lexical Match Results (Components) (L)

Dynamically Extracted Architecture Components(Figure 19)	Design Document Architecture Components(Figure 20)	Lexical Match Result	Correct?
Overhead		NOREL	Yes (True Negative)
View	View	EXACT	Yes (True Positive)
File Processors	Trace Info File, Session File, Static Info File	CONTAIN	Yes (True Positive)
	Event Trace	NOREL	No (False Negative)
	Event Stream	NOREL	No (False Negative)
	Instrumentor	NOREL	No (False Negative)
	Source Code	NOREL	No (False Negative)
	Instrumented Source Code	NOREL	No (False Negative)
	Solaris DB Database	NOREL	No (False Negative)
Model	Program Model	CONTAIN	No (Partial Positive)
Controller		NOREL	No (False Negative)
View	View Manager	CONTAIN	No (False Positive)
FileProcessors. TraceAnalyzer	Trace Analyzer	EXACT	Yes (True Positive)
FileProcessors. StaticAnalyzer	Static Analyzer	EXACT	Yes (True Positive)

Table 18: ISVis Lexical Component Matching (High-Low)

Dynamically Extracted Architecture Components(Figure 19)	Ctags Function Match Summary	Match Result
Model	39 True Positive	CONTAIN
Controller	None (False negative)	NOREL
View	227 True Positive, 2 False Positive	CONTAIN
Overhead	None (False Negative)	NOREL
File Processors	3 True Positive, 25 False Positive	CONTAIN
	478 False Negatives	NOREL

Table 19: ISVis High Level Topological Matching (Components Best Case) (T)

Dynamically Extracted Architecture Components(Figure 19)	Design Document Architecture Components(Figure 20)	Topological Match Result	Correct?
Overhead		NOREL	Yes (True Negative)
FileProcessors. TraceAnalyzer	Trace Analyzer	EXACT	Yes (True Positive)
	Event Trace	NOREL	No (False Negative)
	Event Stream	NOREL	No (False Negative)
	Instrumentor	NOREL	No (False Negative)
	Solaris DB Database	NOREL	No (False Negative)
	Source Code	NOREL	No (False Negative)
FileProcessors. StaticAnalyzer	Static Analyzer	EXACT	Yes (True Positive)
	Session File	NOREL	No (False Negative)
	Static Info File	NOREL	No (False Negative)
	Trace Info File	NOREL	No (False Negative)
	Instrumented Source Code	NOREL	No (False Negative)
Model Program	Model	EXACT	Yes (True Positive)
Controller	View Manager	EXACT	Yes (True Positive)
View	View	EXACT	Yes (True Positive)
File Processors		NOREL	No (False Negative)

Table 20: ISVis High-Level Topological Matching (Connectors) (T)

Dynamically Extracted Architecture Connectors(Figure 19)	Design Document Architecture Connectors(Figure 20)	Topological Match Result	Correct?
CreateGui		NOREL	Yes (True Negative)
Redraw		NOREL	Yes (True Negative)
Mouse/Key Data		NOREL	Yes (True Negative)
Prompt	C9	EXACT	No (Partial Positive)
GetText	C9	EXACT	No (Partial Positive)
ReadTrace		NOREL	Yes (True Negative)
Create		NOREL	Yes (True Negative)
SetItemState	C10	EXACT	No (Partial Positive)
Update	C10	EXACT	No (Partial Positive)
Create/ ModifyComponents	C10	EXACT	No (Partial Positive)
Prompt1	C5	EXACT	Yes (True Positive)
GetText1	C8	EXACT	Yes (True Positive)
	C4	NOREL	Yes (True Negative)
	C1, C2, C3, C6, C7, C11, C12, D2, D3, D4, D5, D6, D7, D8, D9, D10	NOREL	Yes (True Negative)

APPENDIX E

LINUX LEXICAL AND TOPOLOGICAL RESULTS

Table 21: Linux High-Level Lexical Comparison (Components) (L)

Concrete Architecture Components(Figure 24)	Domain Architecture Components(Figure 23)	Lexical Match	Result Correct?
Memory Manager	Memory Management	CONTAIN	No (Partial Positive)
File System	System Call Interface	CONTAIN	No (False Positive)
Network Interface		NOREL	No (False Negative)
Process Scheduler	Process Scheduler	EXACT	Yes (True Positive)
Interprocess Communication	Interprocess Communication	EXACT	Yes (True Positive)
Initialization		NOREL	Yes (True Negative)
Library		NOREL	Yes (True Negative)
	IO Services	NOREL	No (False Negative)

Table 22: Linux Component High-Low Lexical Matching

Linux Domain Architecture Elements(Figure 23)	Linux Call Graph Elements	Match Result
Memory Management	459 True Positive, 2798 False Positive	CONTAIN
IO Services	13438 True Positive, 422 False Positive	CONTAIN
Process Scheduler	147 True Positive, 2971 False Positive	CONTAIN
Interprocess Communication	25 True Positive, 1034 False Positive	CONTAIN
System Call Interface	155 True Positive, 15000 False Positive	CONTAIN
	343 True Negatives, 29179 False Negatives	NOREL

Table 23: Linux High Level Topological Matching (Components Best Case) (T)

Concrete Architecture Components(Figure 24)	Domain Architecture Components(Figure 23)	Topological Match Result	Correct?
Memory Manager	Memory Management	EXACT	Yes (True Positive)
File System	System Call Interface	EXACT	No (False Positive)
Network Interface	IO Services	EXACT	No (Partial Positive)
Process Scheduler	Process Scheduler	EXACT	Yes (True Positive)
Interprocess Communication	Interprocess Communication	EXACT	Yes (True Positive)
Initialization		NOREL	Yes (True Negative)
Library		NOREL	Yes (True Negative)

Table 24: Linux High-Level Topological Matching (Connectors, Best Case) (T)

Concrete Architecture Components(Figure 24)	Domain Architecture Components(Figure 23)	Topological Match Result	Correct?
D3	C4	EXACT	Yes (True Positive)
D19	C3	EXACT	Yes (True Positive)
D8	C2	EXACT	No (False Positive)
D6	C1	EXACT	No (False Positive)
D1, D2, D4, D5, D7, D9, D10, D11, D12, D13, D14, D15, D16, D17, D18, D20		NOREL	Yes (True Negative)

APPENDIX F

ISVIS SEMANTIC RESULTS

Table 25: Top 25 Terms Mined from ISVis Documentation

Raw Word Count (All)	Raw Word (Domain)	Raw Count Impl.	Bayesian Net, All Docs	Bayesian Net, Domain	Bayesian Net, Implementation
scenario	program	scenario	analyst	engineer	analyst
interaction	interaction	actor	page	process	align
program	pattern	software	area	focus	area
actor	trace	interaction	mural	generation	mural
trace	event	mural	area interaction	detection	page
pattern	message	class	view	characterize	word
file event	actor	ioshell	generate	direct	derive
analyst	analyst	widget	analyze	conformance	select
message	execution	viewmanager	group	help representation	area interaction
mural	scenario	file scale	visualization	project	main
class execution	software	trace	dynamic	application	assign
main	class	scenarioview	sequence	recognition	mouse
understanding	understanding	stream	build	reference	performance
model	mural	size	temporal	compute	environment
software	view	functionactor	abstraction	visualize	trade
data	model	scenarioiterator	understanding	reflexion	usefulness
focus	data	draw	execution	model	behave
global	global	muralscrollbar	mouse	prototype sys	actor
function	architectural	forecolor	main	compare	define
view	visualization	event	word	map	understood
architectural	function	zoom	derive	overview	analyze
display	abstraction	programmodel	phase	occurrence	temporal
visualization	dynamic	program	validation	evidence	generate
	process	mainview	recovery		build
	file event				

Table 26: ISVis Domain Architecture Formal Context (Components)

	scenario	visualization	program	interaction	actor	trace	pattern	file	event	analyst	message	mural	class	execution	main	understanding	model	software	trace	focus	data	function	view	architectural	display
Program Model				X	X			X					X				X				X	X		X	
View Manager			X	X						X										X					
View		X					X			X		X				X							X	X	X
Trace Analyzer	X			X	X				X					X					X						
Event Stream								X	X					X				X			X				
Session File			X					X									X	X			X				
Event Trace				X	X	X		X	X					X							X				
Instrumented Source Code			X					X										X			X				
Trace Info File			X		X	X		X	X					X							X				
Source Code			X					X										X			X				
Solaris DB Database								X					X					X			X	X			
Static Analyzer								X					X									X			
Instrumentor			X			X				X				X				X							
Static Info File								X					X								X	X			

Table 27: ISVis Formal Context (Components Dynamically Recovered) (High)

	scenario	visualization	program	interaction	actor	trace	pattern	file	event	analyst	message	mural	class	execution	main	understanding	model	software	trace	focus	data	function	view	architectural	display
Model	X			X	X	X	X		X				X				X				X	X		X	
View		X								X		X				X				X			X		X
Controller			X	X						X	X														
File Processor			X			X		X		X				X				X				X			
Overhead															X										X

Table 28: ISVis High-Level Semantic Approximation (Components) (S)

Dynamically Extracted Architecture Components(Figure 19)	Design Document Architecture Components (Figure 20)	Semantic Approximation Match Result	Correct?
View	View	OVERLAP	No (Partial Positive)
Model	Program Model	OVERLAP	No (Partial Positive)
Overhead		NOREL	Yes (True Negative)
Controller	View Manager	OVERLAP	No (Partial Positive)
FileProcessors. Static Analyzer	Static Analyzer	SUBSUME	No (Partial Positive)
FileProcessor. TraceAnalyzer	Trace Analyzer	SUBSUME	No (Partial Positive)
File Processor Event Trace Event Stream Trace Info File Instrumentor Session File Source Code Static Info File Instrumented Source Code	Solaris DB Database	OVERLAP	No (Partial Positive)

Table 29: ISVis High-Level Semantic Approximation (Components) (S)

Dynamically Extracted Architecture Components(Figure 19)	Design Document Architecture Components (Figure 20)	Semantic Approximation Match Result	Correct?
View	View	OVERLAP	No (Partial Positive)
Model	Program Model	OVERLAP	No (Partial Positive)
Overhead		NOREL	Yes (True Negative)
Controller	View Manager	OVERLAP	No (Partial Positive)
FileProcessors. Static Analyzer	Static Analyzer	SUBSUME	No (Partial Positive)
FileProcessor. TraceAnalyzer	Trace Analyzer	SUBSUME	No (Partial Positive)
File Processor Event Trace Event Stream Trace Info File Instrumentor Session File Source Code Static Info File Instrumented Source Code	Solaris DB Database	OVERLAP	No (Partial Positive)

APPENDIX G

LINUX SEMANTIC RESULTS

Table 30: Top 25 Terms Mined from LINUX Documentation

Raw Word Count (All)	Raw Word (Domain)	Raw Count Impl.	Bayesian Net All Docs	Bayesian Net, Domain	Bayesian Net, Implementation
file process data time directory program root memory device disk command page read address access network system version block type software interface mode structure write	process file time page data memory system disk program directory block thread access read device address store virtual size write hardware resource cpu command main	file root directory command program network time configuration software version interface security disk type mode option fileutils access default data boot change read process ip	system program environment disk process application multiprogramming job login directory scheduling administrator performance file editor database management time password backup service execution communicate dynamically security	hda filename lilo filesystem modem raid loopback loadable administrator fs integrity log udp ide icmp elf nic xterm consoles backup host thread root bootup audit	program boot disk file root environment drive directory floppy system partition drive time access filesystem monthly pc file bios cylinder lilo subdirectory application performance read

Table 31: Linux Concrete Architecture Formal Context (Components)

	Write	Mode	Block	Network	Access	Read	Disk	Device	Directory	Time	Data	File	Structure	Interface	Software	Type	Version	Systems	Program	Command	Memory	Root	Process	Page	Address
Memory Manager	X	X	X		X	X		X			X										X			X	X
File System	X	X	X		X	X	X	X	X	X	X	X	X	X				X	X	X					
Network Interface	X			X	X	X		X			X			X											
Process Scheduler		X								X			X										X		
Interprocess Communication											X				X			X	X				X		
Initialization																	X	X	X			X			

Table 32: Linux Domain Architecture Formal Context (Components)

	Write	Mode	Block	Network	Access	Read	Disk	Device	Directory	Time	Data	File	Structure	Interface	Software	Type	Version	Systems	Program	Command	Memory	Root	Process	Page	Address
I/O Services	X	X	X	X	X	X	X	X	X	X	X	X													
System Call Interface													X	X	X	X	X	X	X	X		X			
Process Scheduler		X								X			X										X		
Interprocess Communication											X				X			X	X				X		
Memory Management	X	X	X		X	X		X			X										X			X	X

Table 33: Linux High-Level Semantic Approximation Results (Components) (S)

Concrete Architecture Components (Figure 24)	Domain Architecture Components(Figure 23)	Semantic Approximation Match Result	Correct?
Memory Manager	Memory Management	EXACT	Yes (True Positive)
File System	I/O Services	OVERLAP	No (Partial Positive)
Network Interface	I/O Services	OVERLAP	No (Partial Positive)
Process Scheduler	Process Scheduler	EXACT	Yes (True Positive)
Interprocess Communication	Interprocess Communication	EXACT	Yes (True Positive)
Initialization		NOREL	Yes (True Negative)
Library	System Call Interface	OVERLAP	No (False Positive)

Table 34: Linux High-Low Semantic Approximation Matching

Domain Architecture Components (Figure 23)	Semantic Approximation Match Result	Match Result?
Memory Management	203 Partial, 8322 False Positive	OVERLAP
I/O Services	10181 Partial Positive, 194 False Positive	OVERLAP
Process Scheduler	3 Partial Positive, 2661 False Positive	OVERLAP
Interprocess Communication	16 Partial Positive, 1232 False Positive	OVERLAP
System Call Interface	1 Partial Positive, 2265 False Positive	OVERLAP
	260 True Negative, 24478 False Negative	NOREL

APPENDIX H

ISVIS LEXICAL AND SEMANTIC APPROXIMATION RESULTS

Table 35: ISVis High-Level Lexical and Semantic Approximation Results (Components) (LS)

Dynamically Extracted Architecture Components(Figure 19)	Design Document Architecture Components(Figure 20)	Semantic Approximation Match Result	Correct?
Overhead		NOREL	Yes (True Negative)
View	View	EXACT	Yes (True Positive)
File Processors Trace Info File Session File Static Info File Event Trace Event Stream Instrumentor Source Code Instrumented Source Code	Solaris DB Database	CONTAIN	Yes (True Positive)
FileProcessors. TraceAnalyzer	Trace Analyzer	EXACT	Yes (True Positive)
FileProcessors. StaticAnalyzer	Static Analyzer	EXACT	Yes (True Positive)
Model	Program Model	CONTAIN	No (Partial Positive)
Controller	View Manager	OVERLAP	No (Partial Positive)

Table 36: ISVis High-Level Lexical and Semantic Approximation Results (Components) (LS)

Dynamically Extracted Architecture Components(Figure 19)	Design Document Architecture Components(Figure 20)	Semantic Approximation Match Result	Correct?
Overhead		NOREL	Yes (True Negative)
View	View	EXACT	Yes (True Positive)
File Processors Trace Info File Session File Static Info File Event Trace Event Stream Instrumentor Source Code Instrumented Source Code	Solaris DB Database	CONTAIN	Yes (True Positive)
FileProcessors. TraceAnalyzer	Trace Analyzer	EXACT	Yes (True Positive)
FileProcessors. StaticAnalyzer	Static Analyzer	EXACT	Yes (True Positive)
Model	Program Model	CONTAIN	No (Partial Positive)
Controller	View Manager	OVERLAP	No (Partial Positive)

Table 37: ISVis High-Low Lexical and Semantic Approximation (LS)

Dynamically Extracted Architecture Components(Figure 19)	Ctags Semantic Approximation Match Summary	Match Result?
View	37 True Positive, 3 Partial Positive, 14 False Positive	CONTAIN
Model	25 True Positive, 25 Partial Positive, 76 False Negative	CONTAIN
Overhead	36 False Positive	CONTAIN
Controller	11 False Positive	OVERLAP
File Processor	9 Partial Positive, 36 False Positive	CONTAIN

APPENDIX I

LINUX LEXICAL AND SEMANTIC APPROXIMATION RESULTS

Table 38: Linux High-Level Lexical and Semantic Approximation Results (Components) (LS)

Concrete Architecture Components(Figure 24)	Domain Architecture Components(Figure 23)	Lexical Match Result	Correct?
Memory Manager	Memory Management	EXACT	Yes (True Positive)
File System	IO Services	OVERLAP	No (Partial Positive)
Network Interface	IO Services	OVERLAP	No (Partial Positive)
Process Scheduler	Process Scheduler	EXACT	Yes (True Positive)
Interprocess Communication	Interprocess Communication	EXACT	Yes (True Positive)
Initialization		NOREL	Yes (True Negative)
Library	System Call Interface	OVERLAP	No (False Positive)

Table 39: Linux High-Level Lexical and Semantic Approximation Results (Components) (LS)

Concrete Architecture Components(Figure 24)	Domain Architecture Components(Figure 23)	Lexical Match Result	Correct?
Memory Manager	Memory Management	EXACT	Yes (True Positive)
File System	IO Services	OVERLAP	No (Partial Positive)
Network Interface	IO Services	OVERLAP	No (Partial Positive)
Process Scheduler	Process Scheduler	EXACT	Yes (True Positive)
Interprocess Communication	Interprocess Communication	EXACT	Yes (True Positive)
Initialization		NOREL	Yes (True Negative)
Library	System Call Interface	OVERLAP	No (False Positive)

Table 40: Linux High-Low Semantic/Lexical Technique

Domain Architecture Components(Figure 23)	Semantic Approximation Match Result	Match Result?
Memory Management	184 True Positive, 24 Partial, 7701 False Positive	CONTAIN
I/O Services	869 True Positives, 8137 Partial Positive, 442 False Positive	OVERLAP
Process Scheduler	3 Partial Positive, 2678 False Positive	OVERLAP
Interprocess Communication	0 True Positive, 13 Partial Positive, 1193 False Positive	OVERLAP
System Call Interface	3 True Positive, 0 Partial Positive, 4578 False Positive	OVERLAP
	260 True Negative, 23730 False Negative	NOREL

APPENDIX J

ISVIS LTS RESULTS

Table 41: ISVis High-Level Lexical, Topological and Semantic Results (Components) (LTS)

Dynamically Extracted Architecture Components(Figure 19)	Design Document Architecture Components(Figure 20)	Semantic Approximation Match Result	Correct?
Overhead		NOREL	Yes (True Negative)
View	View	EXACT	Yes (True Positive)
File Processors Trace Info File Session File Static Info File Event Trace Event Stream Instrumentor Source Code Instrumented Source Code	Solaris DB Database	CONTAIN	Yes (True Positive)
FileProcessors. StaticAnalyzer	Static Analyzer	EXACT	Yes (True Positive)
FileProcessors. TraceAnalyzer	Trace Analyzer	EXACT	Yes (True Positive)
Model	Program Model	EXACT	Yes (True Positive)
Controller	View Manager	EXACT	Yes (True Positive)

Table 42: ISVis High-Level Lexical, Topological, Semantic Approximation (Connectors)

Dynamically Extracted Architecture Connectors(Figure 19)	Design Document Architecture Connectors(Figure 20)	Semantic Approximation Match Result	Correct?
CreateGui		NOREL	Yes (True Negative)
Redraw		NOREL	Yes (True Negative)
Mouse/Key Data		NOREL	Yes (True Negative)
Prompt	C9	EXACT	No (Partial Positive)
GetText	C9	EXACT	No (Partial Positive)
ReadTrace		NOREL	Yes (True Negative)
Create		NOREL	Yes (True Negative)
SetItemState	C10	EXACT	No (Partial Positive)
Update	C10	EXACT	No (Partial Positive)
Create/ Modify-Components	C10	EXACT	No (Partial Positive)
Prompt1	C5	EXACT	Yes (True Positive)
GetText1	C8	EXACT	Yes (True Positive)
	C4	NOREL	Yes (True Negative)
	C1, C2, C3, C6, C7, C11, C12, D2, D3, D4, D5, D6, D7, D8, D9, D10	NOREL	Yes (True Negative)

APPENDIX K

LINUX LTS RESULTS

Table 43: Linux High-Level Lexical, Topological and Semantic Approximation (Components) (LTS)

Concrete Architecture Components(Figure 24)	Domain Architecture Components(Figure 23)	LTS Match Result	Correct?
Memory Manager	Memory Management	EXACT	Yes (True Positive)
File System	IO Services	CONTAIN	Yes (True Positive)
Network Interface	IO Services	CONTAIN	Yes (True Positive)
Process Scheduler	Process Scheduler	EXACT	Yes (True Positive)
Interprocess Communication	Interprocess Communication	EXACT	Yes (True Positive)
Initialization		NOREL	Yes (True Negative)
Library	System Call Interface	OVERLAP	No (False Positive)

Table 44: Linux High-Level Lexical, Topological and Semantic Approximation (Connectors) (LTS)

Concrete Architecture Components(Figure 24)	Domain Architecture Components(Figure 23)	LTS Match Result	Correct?
D3	C4	EXACT	Yes (True Positive)
D19	C3	EXACT	Yes (True Positive)
	C1,C2	NOREL	Yes (True Negative)
D1, D2, D4, D5,D6, D7, D8, D9, D10, D11, D12, D13, D14, D15, D16, D17, D18, D20		NOREL	Yes (True Negative)

REFERENCES

[1] *The Java AcmeLib Programmers Manual*. Avaliable electronically at: www-2.cs.cmu.edu/ãcme/acme_documentation.html.

[2] *Microsoft SQL Server Product Information*. Available electronically at: www.microsoft.com/sql/evaluate/overview/.

[3] Abowd, G., Kazman, R., and Clements, P., "Scenario-based analysis of software architecture," Tech. Rep. GIT-CC-95-41, Georgia Institute of Technology, October 29, 1995 1995.

[4] Antoniol, G., Fiutem, G., Lutteri, G., Tonnella, P., Zanfei, S., and Merlo, E., "Program understanding and maintenance with the canto environment," in *International Conference on Software Maintenance*, IEEE, 1997.

[5] Baniassad, E., Murphy, G., Schwanninger, C., and Kircher, M., "Managing crosscutting concerns during software evolution tasks: An inquisative study," in *1st International Conference on Aspect-Oriented Software Development*, (Enchede, The Netherlands), pp. 102–126, ACM Press, 2002.

[6] Bass, L., Clements, P., and Kazman, R., *Software architecture in practice*. Reading, Mass.: Addison-Wesley, 1998. 97013979 Len Bass, Paul Clements, Rick Kazman. Includes bibliographical references (p. 437-445) and index.

[7] Bellay, B. and Gall, H., "Reverse engineering to recover and describe a system's architecture," in *Proceedings : Development and Evolution of Software Architecture for Product Families* (Van Der Linden, F., ed.), vol. 1429, (Las Palmas de Gran Canaria, Spain), pp. 115–122, Springer-Verlag, 1998.

[8] Bergey, J., Smith, D., Tilley, S., Weiderman, N., and Woods, S., "Why reengineering projects fail," Tech. Rep. CMU/SEI-99-TR-010, Carnegie Mellon University, Software Engineering Institute, April 1999 1999.

[9] Bergey, J. K., Northrop, L. M., and Smith, D. B., "Enterprise framework for the disciplined evolution of legacy systems," Technical Report CMU/SEI-97-TR-007, Software Engineering Institute, October 1997 1997.

[10] Biggerstaff, T., Mitbander, B., and Webster, D., "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–83, 1994.

[11] Bowman, I. T., Holt, R. C., and Brewster, N. V., "Linux as a case study: its extracted software architecture," in *21st International Conference on Software Engineering*, (Los Angeles, CA), pp. 555–563, IEEE Computer Society Press, Los Alamitos, CA, 1999.

[12] Caprile, B. and Tonella, P., "Nomen est omen: Analyzing the language of funtion identifiers," in *Sixth Working Conference on Reverse Engineering*, (Atlanta, Georgia), pp. 112–122, IEEE Computer Society, 1999.

[13] Chase, M., C. S. H. D. and Yeh, A., "Managing recovered function and structure of legacy software components," in *Working Conference on Reverse Engineering*, (Hawaii), pp. 79–88, 1998.

[14] Chase, M. P., Christey, S. M., Harris, D. R., and Yeh, A. S., "Recovering software architecture from multiple source code analyses," in *PASTE*, (Montreal Canada), pp. 43–50, ACM Press, 1998.

[15] Chikofsky, E. and Cross, J., "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[16] Clayton, R., Rugaber, S., Taylor, L., and Wills, L., "A case study of domain-based program understanding," in *Workshop on Program Comprehension*, 1998.

[17] Clements, P. and Weiderman, N. H., *Report on the second international workshop on development and evolution of software architectures for product families.* Special report ; CMU/SEI-98-SR-003., Pittsburgh, Pa.: Carnegie Mellon University Software Engineering Institute, 1998. Paul C. Clements, Nelson Weiderman. Special report. Carnegie Mellon University. Software Engineering Institute ; CMU/SEI-98-SR-003 "May 1998." Includes bibliographical references.

[18] Egyed, A., "Automating architectural view integration in uml," Technical Report USCCSE-99-511, University of Southern California, 1999 1999.

[19] Eixelsberger, W., Kalan, M., Ogris, M., Beckman, H., Bellay, B., and Gall, H., "Recovery of architectural structure : A case study," in *Proceedings : Development and Evolution of Software Architecture for Product Families* (Van Der Linden, F., ed.), vol. 1429, (Las Palmas de Gran Canaria, Spain), pp. 89–96, Springer-Verlag, 1998.

[20] Elrad, T., Filman, R. E., and Bader, A., "Apsect-oriented programming: Introduction," *Communications of the ACM*, vol. 44, pp. 29–32, October 2001.

[21] Finkelstien, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B., "Inconsistency handling in multi-perspective specifications," *Transactions on Software Engineering*, vol. 20, no. 8, pp. 569–578, 1994.

[22] Fiutem, R., Merlo, M., Antoniol, G., and Tonella, P., "Understanding the architecture of software systems," Technical Report 9510-06, IRST, October 1995 1995.

[23] Fiutem, R., Tonella, P., Antoniol, G., and Merlo, E., "A cliche-based environment to support architectural reverse engineering," Technical Report 9602-02, IRST, February 1996 1996.

[24] Gall, H., Jazayeri, M., Klosch, R., Lugmayr, W., and Trausmuth, G., "Architecture recovery in ares," in *Joint Proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoint '96) on SIGSOFT '96 workshops*, (San Francisco, California), pp. 111–115, ACM Press New York, NY USA, 1996.

[25] Gansner, E. and North, S., "An open graph visualization system and its application to software engineering," *Software-Practice and Experience*, no. 30(11), pp. 1203–1233, 2000.

[26] Ganter, B. and Wille, R., *Formal Concept Analysis : Mathematical Foundations*. Berlin: Springer Verlag, 1999.

[27] Garlan, D. and Shaw, M., "An introduction to software architecture," in *Advances in Software Engineering and Knowledge Engineering* (Tortora, V. A. and G., eds.), pp. 1–39, Singapore: , World Scientific Publishing Company, 1993.

[28] Grinberg, Dennis, J. L. and Sleator, D., "A robust parsing algorithm for link grammars," in *Proceedings of Fourth International Conference on Parsing Technologies*, (Prague), 1995.

[29] Guingo, R. and Shasta, D., "Graphgrep: A fast and universal method for querying graphs," in *Proceedings of the International Conference in Pattern Recognition*, Quebec, Canada, 2002.

[30] Guo, G. Y., Altee, J. M., and Kazman, R., "A software architecture reconstruction method," in *TC2 Working Conference on Software Architecture (WICSA1)* (Donohoe, P., ed.), (San Antonio, TX), pp. 16–33, Kluwer Academic Publishers, 1999.

[31] Himsolt, M., "The graphlet system," in *Volume 1190 Lecture Notes in Computer Science* (North, S., ed.), (New York/Berlin), pp. 233–240, Springer-Verlag, 1996.

[32] Hipson, P. and Jennings, R., *Database Developers Guide with Visual C++*. Indianapolis, Indiana: SAMS Publishing, 1996.

[33] Holt, R. C., Malton, A. J., Davis, I., Bull, I., and Trevors, A., "Software architecture toolkit," 2003.

[34] Incorporated, R. S., "Refine user's guide," 1992.

[35] Jerding, D. and Rugaber, S., "Using visualization for architectural localization and extraction," in *Working Conference on Reverse Engineering*, 1997.

[36] Kazman, R. and Carriere, S., "View extraction and view fusion in architectural understanding," in *Fifth International Conference on Software Reuse*, 1998.

[37] Kazman, R. and Carriere, S. J., "Playing detective: Reconstructing software architecture from available evidence," Technical Report CMU/SEI-97-TR-010, Carnegie Mellon University, 1997 1997.

[38] Kazman, R. and Burth, M., "Assessing architectural complexity," in *2nd Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '98)*, IEEE Computer Society Press, 1998.

[39] Kpskimies, K., M. T. S. T. and Tuomi, J., "Automated support for modeling oo software," *IEEE Software*, vol. 15, pp. 87–94, January/February 1998.

[40] Krikhaar, R., "Reverse architecting approach for complex systems," in *International Conference on Software Maintenance*, IEEE, 1997.

[41] Krone, M. and Snelting, G., "On the inference of configuration structures from source code," in *International Conference on Software Engineering (ICSE 16)*, pp. 49–57, IEEE Computer Society Press, 1994.

[42] Krutchen, P., "The 4+1 view model of architecture," *IEEE Software*, vol. 12, no. 6, 1995.

[43] Lin, T. and OBrien, L., "Fepss: A flexible and extensible program comprehension support system," in *Working Conference on Reverse Engineering*, (Hawaii), pp. 40–49, 1998.

[44] Lindig, C., "Concepts source code," 1999.

[45] Lovis, C., M. P. B. R. S. J., "Word segmentation processing: A way to exponentially extend medical dictionaries," in *In proceedings 8th World Congress on Medical Informatics*, pp. 28–32, 1995.

[46] Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., and Mann, W., "Specification and analysis of system architecture using rapide," *IEEE Transactions on Software Engineering, Special Issue on Software Engineering*, vol. 21, no. 4, pp. 336–355, 1995.

[47] Magee, J., D. N. E. S. and Kramer, J., "Specifying distributed software architectures," *Proceedings of 5th European Software Engineering Conference (ESEC 95)*, 1995.

[48] Maxwell, S., *Linux Core Kernel Commentary*. Scottsdale, AZ: CoriolisOpen Press, 1st ed., 1999.

[49] Megaputer, "Textanalyst."

[50] Mendonca, N. C. and Kramer, J., "Requirements for an effective architecture recovery framework," in *SIGSOFT*, pp. 101–105, ACM, 1996.

[51] Michail, A. and Notkin, D., "Accessing software libraries by browsing similar classes, functions, and relationships," in *21st International Conference on Software Engineering*, (Los Angeles), pp. 463–472, IEEE Press, 1999.

[52] Miller, George, R. B. C. F. D. G. and Miller, K., "Introduction to wordnet: an on-line lexical database," in *International Journal of Lexicography*, vol. 3, pp. 235–244, 1990.

[53] Mitchell, T., "Bow: A toolkit for statistical language modelling, text retrieval, classification and clustering," 1997.

[54] Murphy, G., Notkin, D., and Sullivan, K., "Software reflexion models: Bridging the gap between source and high-level models," *ACM SIGSOFT*, vol. 1995, 1995.

[55] Murphy, G. C. and Notkin, D., "Lightweight lexical source model extraction," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 262–292, 1996.

[56] Paulk, M. C., "A capability maturity model for software," Tech. Rep. SEI SEI-93-TR-24, Software Engineering Institute, 1993.

[57] Perry, D. and Wolf, A., "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[58] Porter, M., "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[59] Rugaber, S., "Morale methodology guidebook: Methodology guidebook for synchronized refinement," 1998.

[60] Siff, M. and Reps, T., "Identifying modules via concept analysis," in *ICSM' 97: IEEE Conference on Software Maintenance*, (Bari, Italy), pp. 170–179, IEEE Computer Society, 1997.

[61] Sim, S. E., Clarke, C. L. A., Holt, R. C., and Cox, A., "Browsing and searching software architectures," in *International Conference on Software Engineering (ICSE)*, (Los Angeles, CA), p. To Appear, IEEE, 1999.

[62] Systa, T., *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. Dissertation, University of Tampere, 2000.

[63] Tilley, S., "A reverse engineering environment framework," Technical Report CMU/SEI-98-TR-005, Carnegie Mellon University, 1998 1998.

[64] Tzerpos, V. and Holt, R. C., "A hybrid process for recovering software architecture," in *CASCON*, (Toronto, Canada), pp. 1–6, 1996.

[65] Weiderman, N. Tilley, S. and D., S., "Approaches to legacy system evolution," Technical Report CMU/SEI-97-TR-014, CMU, 1997 1997.

[66] Wille, R., "Restructuring lattice theory: an approach based on hierarchies of concepts," in *Ordered Sets* (Rival, I., ed.), pp. 445–470, Dordrecht-Boston: Reidel, 1982.

[67] Wong, K., "Rigi user's manual : Version 5.4.4," 1998.

[68] Yeh, A., Harris, D., and Chase, M., "Manipulating recovered software architectural views," in *19th International Conference on Software Engineering*, 1997.

[69] Zachman, J. A., "A framework for information systems architecture," *IBM System Journal*, vol. 24, no. 3, 1987.

[70] Zaremski, A. M. and Wing, J. M., "Specification matching of software components," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1997.

[71] Zave, P. and Jackson, M., "Composition as conjunction," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 4, pp. 379–411, 1993.