A Process for Obtaining Architectural Descriptions from Legacy Systems: the Architectural Synthesis Process (ASP)

Robert Waters
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

Ph.D. Thesis Proposal
September 20, 2000

Thesis Proposal Committee:
Gregory Abowd (Advisor)
Spencer Rugaber
Colin Potts
Mike McCracken
Rick Kazman (SEI)

Version 0.9: This is a final draft for review by committee members prior to the formal proposal. This proposal is a minor rewrite of version 0.6 which expands upon the use of IEEE standard P1471 on Architectural Descriptions. Changed emphasis on repeatable process to a defined process.

Abstract

The 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 architectural description simply does not exist. This work proposes a defined process to obtain a complete, consistent and useful description of a legacy system's software architecture so these descriptions can be used for evolution or maintenance tasks. This process is called the architectural synthesis process (ASP) and consists of four stages: *extraction* (obtaining opinions or perspectives about what the architecture might look like), *classification* (deciding which aspect (view) of a system the perspective describes), *union* (combining all perspectives describing a single aspect of the system to obtain a complete viewpoint) and *fusion* (comparing different viewpoints to insure consistency). This work further introduces the use of domain terms as an approximation to semantic descriptions of architectural elements. To aid in establishing process repeatability, a supporting toolkit called REMORA is provided. REMORA uses lexical analysis, topological features of the representation and concept analysis to aid in matching elements from different perspectives. This work will show that the proposed process is produces complete, consistent and useful descriptions of a software architecture. Validation will involve recovery of the architectural description of the LINUX operating system kernel and a comparison of the results of ASP to the common understanding of the LINUX community and other published architectural descriptions.

Table of Contents

Table of Figures

Index of Tables

1. INTRODUCTION AND MOTIVATION

There is a need for a defined process for obtaining software architectural descriptions from legacy systems. Although the exact percentage varies, most researchers agree that somewhere between 40 and 80 percent of development activities are focused on maintenance, enhancement or evolution of existing systems[16]. 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. Usually these changes are made with minimal understanding of the impacts they have on system quality over time. This led to the characterization of most 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].

Coincident with this shift of emphasis, the area of software architecture research has enjoyed a resurgence of interest [39]. Researchers have focused on improving systems development by providing techniques for analysis of system's designs at the architectural level. Examples of these techniques include Rapide[45] that provides event simulations based upon a system's architectural description and the software architecture analysis method (SAAM) [34], which performs impact analysis on similar architectural descriptions. While each architectural analysis technique differs in its benefits and goals, they all have one thing in common, the need for an accurate architectural description as input.

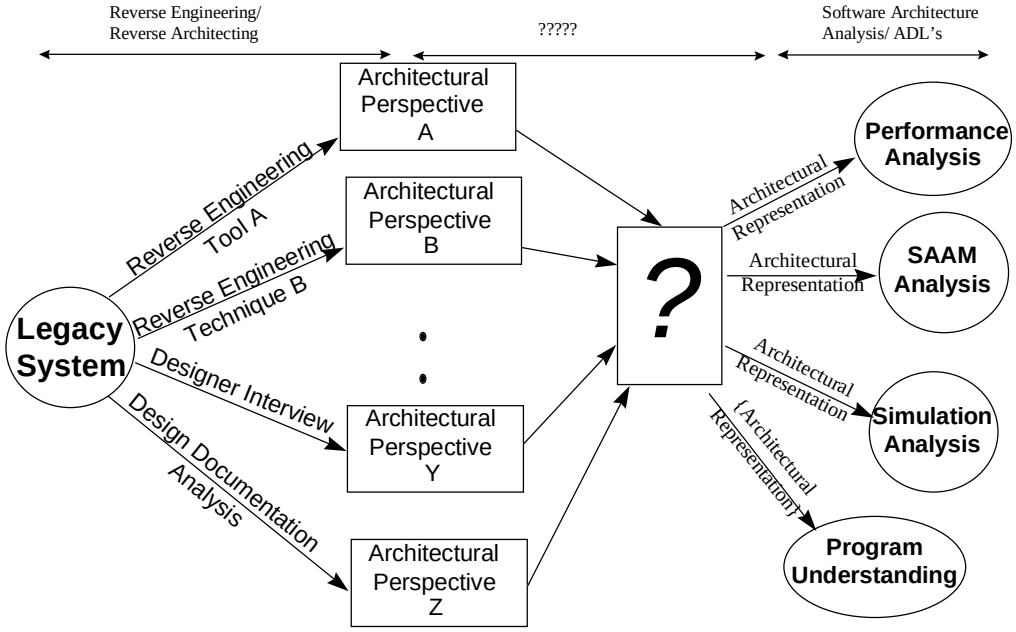

Figure 1: The Multiple Perspective Problem

The importance of using architecture as a vehicle for guiding a reengineering effort has recently been emphasized by the SEI. 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].

Unfortunately, software engineers have a real problem in focusing on the architecture of a legacy system. The bulk of forward engineering analysis tools and techniques usually produce a software architectural description, while the legacy system to be evolved usually has none. The problem of how to get there from here is shown graphically in Figure 1. There are many ways to get an opinion (or perspective) of what the architecture of a system might look like. These are shown on the left side of the figure. On the right side of the figure is a sample of analysis techniques used by an analyst in evolving a legacy system. The big question mark represents the void that currently exists: The need for a repeatable process that produces complete and consistent architectural representations from raw architectural information. This need is reiterated in section 4.3.3 of IEEE P1471[3], which states: "…a process for reverse engineering an architecture from an implementation is needed." It is meeting this need that motivated the research for ASP.

One might be tempted to assert that developing a software architecture is already a solved problem. After all, do not all projects develop an architecture at one time or another? Alexander Ran, project architect of the ARES project summed up the state of the art when he said[19], "…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 develop anything actually useful.

The fundamental thesis of this work is:

> *A defined process for obtaining a software architecture description (AD) from a legacy system can be specified in sufficient detail to be usable by practicing software engineers, and the products produced by the process are fully informed, consistent, useful, and conform existing standards (such as IEEE P1471).*

We use the terminology from the Capability Maturity Model [53], which specifies five levels of process definition. Any type process (including chaotic or random) is at the *initial* stage. The next step is to have a specification of how the basic tasks are to be performed. This is the *defined* level. Once the process is defined, it can be further enhanced to become *repeatable*. Once a process is defined and repeatable, it can be *measured* and *optimized*. For purposes of this proposal the initial research goal is to put the process at the defined level. Later research will move the process further towards the optimizing level by making it repeatable and finding appropriate metrics to measure its accomplishment.

An architectural description is *fully informed* if it uses all available information about the legacy system and its domain. An AD is consistent if it has accounted for conflicting or inaccurate information. An AD is *useful* if it provides the necessary information for the analyst to effectively use it for his objective. Using terms from the finance and accounting domain can also help define usefulness of the developed architectural description. In the financial domain, various artifacts are used to describe the business state of a corporation. These artifacts include the balance sheet, cash flow statement and income statement. When determining which artifacts to use to describe a business, accountants use the criteria of usefulness that they define as having four components: relevance, reliability, comparability and consistency. Relevance implies the artifacts are timely and are necessary to influence decision-making. Reliability means they are verifiable and neutral (that is they are not slanted to favor a single set of stakeholders). Comparability means that artifacts are prepared using a common standard while consistency means the same method of preparation is used each time. These same four factors should guide the development of any defined process so that its products will be useful to the people who invest the time and resources to use that process.

With such a large percentage of software development focused on reengineering and evolution efforts, defining this process will make a significant impact on the software engineering field by dramatically helping practicing software engineers.

The major contribution of this work is development of a process for developing conforming architectural descriptions from legacy systems. This process involves:

- An overarching conceptual model into which all current reverse architecting research can be placed.
- A practical methodology through which practitioners can recover a useful set of representations of the architecture of an existing system.
- A definition of the extraction information space and a description of its use in guiding choice of information extraction methods.
- A technique for semantic approximation of architectural elements using domain terms and concept analysis.
- A toolkit to support the process.
- Validation of the process through a detailed case study of a non-trivial system.

Other supporting contributions are discussed in section 4 of this proposal.

2. GENERAL APPROACH

The first step in formulating an overall solution to developing conforming architectural descriptions for legacy systems is to develop a conceptual model of architectural recovery onto which the process details can be mapped.

Up until now, we have used the term software architecture without providing a definition. In section 3,

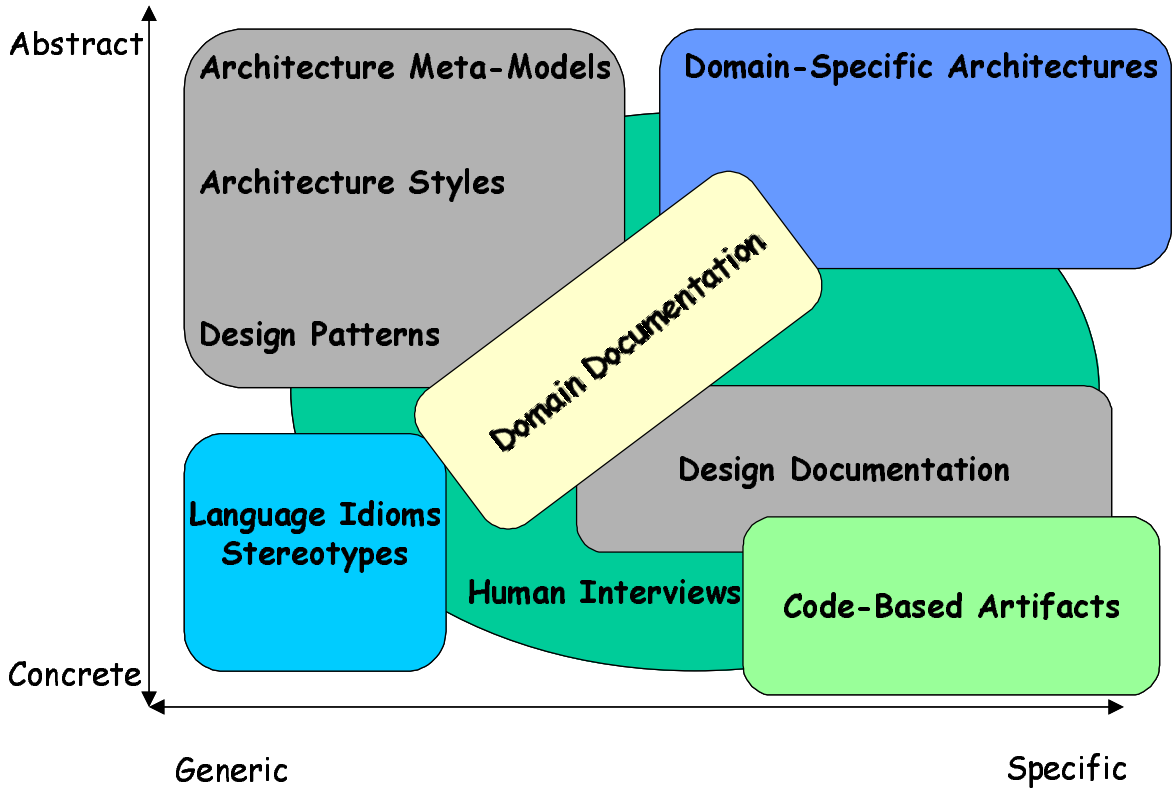

Figure 2: The Extraction Information Space

several formal definitions are provided. For purposes of this proposal, we use the P1471 description that states that architecture is the highest-level conception of a system in its environment. This conceptualization of a system includes consideration of both its functional and non-functional requirements.

Architectural information exists in many locations and can be derived from many sources. Unfortunately, current reverse engineering and architectural recovery techniques focus mainly 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. This problem is commonly referred to as the concept-assignment problem. Figure 2 depicts what we call the extraction information space. This labels on the graph come from a paper by Egyed[23]. General 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, which are also generic across domains. On the other hand, interviews with human experts about the system can span a variety of topics and thus might cover both generic and specific topics about the system at an abstract of concrete level. Examining the information space graph 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 information is in the lower half. Spanning that gap is the concept-assignment problem.

IEEE Standard P1471 states that any conforming architectural description must contain one or more *views*. 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 is called a view template (or in P1471 a *viewpoint*). The

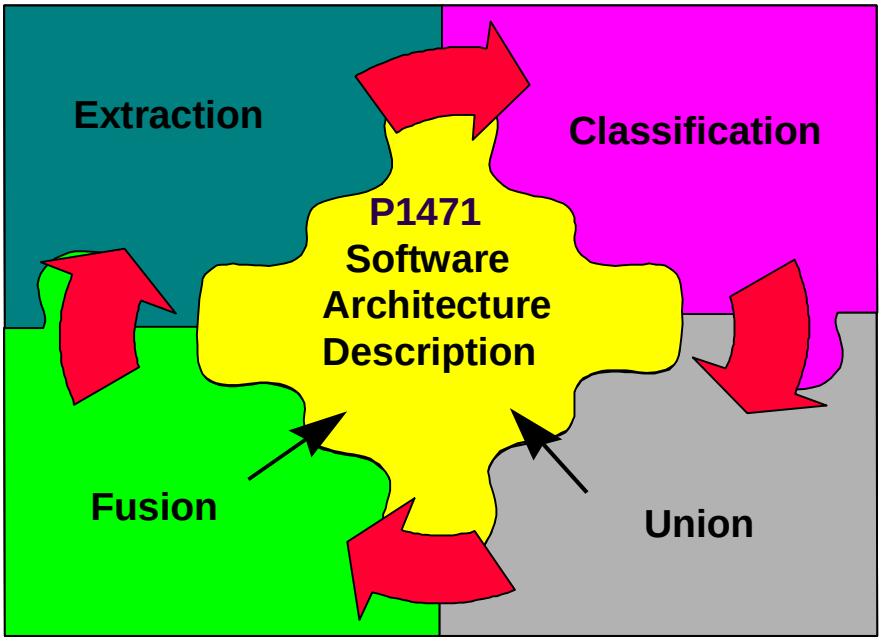

Figure 3: Architectural Synthesis Process (ASP)

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 a single recovery technique would adequately address multiple sets of concerns.

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 conforming architectural description to contain an analysis of the consistency across all views provided. This places the requirement on any process description that it addresses some type of consistency checking for the recovered description. Its important to note that P1471 is similar to the CMM in that it states what has to be in a conforming AD, but not how to derive or even to 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[47]. 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 3. 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 conforming architectural description resolve inconsistencies that are detected, only that they are detected and identified in the description.

The process is shown as a cycle (following the red arrows), since inconsistencies that must be acted upon typically require obtaining more information and incorporating it into the existing architectural description. The central part of the puzzle is the Software Architectural Description. This represents the conforming P1471 AD which is being produced by the synthesis process.

We now have defined the overarching conceptual model and the general process phases forming the methodology to be followed. Before detailing the specific process steps, we need to examine what other researchers have accomplished that supports the ASP.

3. RELATED WORK

To understand the theoretical underpinnings of ASP, it is first necessary to understand principles of software architecture and reverse engineering, other approaches to recovering architectures from legacy systems and finally supporting techniques for the various phases of the process. One can think of this related work discussion as flowing from the most abstract to the most concrete information needed to understand the ASP approach.

3.1 Software Architecture

To understand both the benefits of architectural synthesis and the complexities of architectural recovery, one must first understand what a software architecture is. Two seminal papers were published in the early eighties that attempted to define the phrase *software architecture*. Garlan and Shaw [29] define a *software architecture* as comprising *components* (elements which provide computation services), *connectors* (elements which provide interactions between the components) and *configuration* (the topology of the system). The authors also introduce the ideas of *architectural styles*. Styles are commonly occurring configurations in which the components and connectors interact according to a set of constraints. Common styles include *pipe and filter*, *implicit invocation* and *layered*.

Perry and Wolf [54] take a slightly different approach to the definition of a software architecture. Their definition couches software architecture as a triple 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 generally accepted one. Most author's today use the Bass et al. [6] definition (which is based on 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."

Just as a building architect needs multiple diagrams to describe the structure of a complex building, a software architect needs multiple descriptions of a software system. *Architectural views* reflect a set of specific interests that concern a given group of stakeholders[18, 43] Perry and Wolf also discuss the need to provide multiple views of an architecture. Typical views include:

- 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.
- Logical (Conceptual, Functional): This view depicts the software as a set of cooperating components 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.
- 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.
- 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.

IEEE Standard P1471, Recommended Practice for Architectural Description, defines architecture as the highest-level conception of a system in its environment. This definition is also the one that we use for our work. P1471 avoids using the traditional Garlan and Shaw definition—considering it to be too structurally biased. The standard also refines the idea of Krutchen's view into a view and viewpoint. A viewpoint is the same as the idea of views described above. A view in P1471 is the representation of a specific viewpoint. For example, a data viewpoint might be supported with an ER diagram as a view. So the viewpoint is basically a model that can be visualized through one or more views. Appendix E provides a sample viewpoint definition and a supporting view. We will use terminology consistent with the standard for the remainder of the proposal. Table I provides a quick reference for term equivalences.

Table I: Term Equivalences

Traditional	P1471	SEI
View	Viewpoint	View Template
View (Overloaded meaning)	View	View

3.2 Reverse Engineering

Chikofsky and Cross [16] define hardware reverse engineering as: "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system." Unlike hardware reverse engineering, the main objective of the software reverse engineering process is not to duplicate the system under study but to gain sufficient design-level understanding for redocumentation, design recovery or restructuring. For the remainder of this proposal, unless otherwise noted, the phrase *reverse engineering* will refer specifically to software reverse engineering.

The implementation of a software system is the embodiment of some set of human-oriented concepts that represent the original specification. Biggerstaff et al. [10] describe the problem of discovering the original human-oriented concepts and assigning them to their realizations within a specific program as the *concept-assignment problem*. They argue that code analysis alone cannot get at these human concepts. There must also be some way to get at the domain concepts and relationships that exist at abstraction levels higher than the source code. Since software architecture exists at a level of abstraction just below specifications, the concept-assignment problem is especially pronounced during architectural recovery activities. These findings support our assumption that code analysis alone is insufficient for developing an architectural description.

3.3 Architectural Recovery Strategies

We now examine 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, visualization support, sources of information, and ability to make claims about completeness and consistency. These approaches can be classified into three broad categories: Pattern-based recovery, Visualization-based recovery and Process-based recovery.

3.3.1 Pattern Based Architectural 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)[4, 26, 27] looks at the detection of clichés in the 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 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 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 conforming 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 considered. Third, the use of an AST as an intermediate representation necessarily limits CANTO's analysis engine to opera-

tions 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)[14, 15, 67] 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. Some combination of these last three would provide 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 then redisplayed. The intermediate representation for ManSART is called analysis module interfaces (AMI). AMI is an 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 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 [36, 37] (which by the way doesn't stand for anything in particular) takes a different tack than either CANTO or ManSART. DALI recognizes the limitations of tying yourself to a specific tool for extracting architectural information and thus uses the philosophy of an open workbench that can possibly integrate any tool into its functionality. DALI uses a database as the integration mechanism—and thus its intermediate representation is the database tables that record the information from the tools it uses. Standard DALI extraction tools include the Lightweight Source Model Extraction (LSME) [51], gprof, Reflexion Model Tool (RMTool) [50], and IAPR [35]. The Rigi environment provides DALI visualizations. Unlike CANTO and ManSART, Rigi allows an analyst to directly manipulate the architectural visualization and creates 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 primarily provides logical views of the architecture, but this is driven primarily 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 is able to at least allow consistency verification 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, it is tied completely to source-code derived information. The effectiveness of DALI is further 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)[31] is an extension of DALI by an integrated methodology. The open workbench concept of DALI is still there, but is supplemented by prepared scripts and heuristics that allow a more inexperienced analyst 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, however is still source-code bound and thus does not take advantage of all available architectural information.

If we were to project these techniques onto ASP, they all share some commonality. For extraction, they use a limited set of tools that emphasize various types of code analysis. Classification is inherent in

the different tools rather than being explicit. This limits the number and types of views that can be supported. Consistency checking is also implicit in the use of the tools. They are consistent primarily because they obtain information from only one source (the source code of the system) and use that to develop only one or two views.

3.3.2 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 feels visualization of the information is critical. These authors feel that if information is presented in the proper format, a human is the best pattern detector possible. FEPPS, ISVis and the Software Bookshelf best typify this category of tools.

The Flexible and Extensible Program Comprehension and Support System (FEPPS)[44] provides a three-dimensional (3D) viewing and manipulation interface to architectural information. Using this system, an analyst recovers the architecture by viewing complex relationships between elements of architectural information. This information comes from AST's, program slicing, control-flow graph (CFG) and data-flow graph (DFG) analysis. 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 appears to support 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 use any information other than code-based information in the recovery process.

The Interaction Scenario Visualizer (ISVis)[33] 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 final visualization-intensive technique is the Software Bookshelf (SBS)[59]. 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 in relational tables, which is 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.

3.3.3 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 essentially the sole source of information for architectural recovery. Process-based approaches try to incorporate explicitly information from other sources in Figure 1. The principle examples in this category are Krikhaar's Reverse Architecting Approach, SAAM, ARF and Hybrid.

Krikhaar's Reverse Architecting Approach (RAA) [41], consists of three phases, extraction, abstraction and presentation. 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 obtain information focused on code relations such as imports, part-of and uses. This information is then stored and manipulated in

the relation partition algebra. This algebra then 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) 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 most often 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, 24] attempts to build a single view representation of an architecture by integration of information from multiple sources. It represents information internally using the DARWIN ADL. The view 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 tries to build a single view of the architecture normally expressed in terms of some 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 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.

Table II: Comparison of Architectural Recovery Strategies

Recovery Strategy	Information Sources	Internal Representation	Viewpoints / Views Supported	Visualization Support	Consistency Mechanism
CANTO	Source Code	Annotated AST	Module Task	Dotty	Implicit
ManSart	Source Code	Analysis Module Interfaces (AMI)	Task Spawning Repository Service-Invocation ADT	Static Graph	Implicit
DALI ARM	Source Code	Database	Logical	Rigi	Patial
FEPPS	Source Code	Multi-Layer Multi-Graph (MLMG)	Module Logical	Custom 3D Display	Implicit
ISVis	Source Code Execution Traces	Internal OO	Logical Runtime	Information ` ` ` ` ` ` ` `	Implicit
SBS	Source Code Some Documentation	GCL Relations Tuple Algebra	Module	Browsing Graph	Implicit
RAA	Source Code Human Experts	Relational Partition Algebra	Module	Teddy Graph	Implicit
SAAM	Human Experts	Paper / White Board	Any based on Participants	Paper Graph	Implicit
ARF	Source Code Documentation Human Experts	DARWIN ADL	Centered around Specific NFR	Darwin render	Implicit
Hybrid	Source Code Human Experts	See SBS	Module	See SBS	Developer Interviews

The Hybrid process[62] 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:

- Choose a domain model. This step in the language of P1471 is to determine which viewpoint you are interested in recovering.
- Extract facts from source code. This corresponds to the extraction phase of our conceptual model.
- 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.
- Refine clustering using information derived from developer interviews.
- Refine the layout to accommodate new information.

Since the tool support appears to be based off of SBS (described in 3.3.2), the internal representation is also the TA. Consistency in the Hybrid approach is obtained manually by discussing the emerging architectural description with the human experts on the system.

3.3.4 Summary

Table I summarizes each of the recovery strategies based upon the elements of our conceptual framework. All have an equivalent 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 such as ASP. In this case, the designer of the recovery strategy has performed the classification task by designing the extraction task to get information focused on particular views. 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 is either implicitly accomplished by the tool by constraining the types of views and information sources or it is accomplished by simply discussion the views with human experts.

Architectural recovery strategies are therefore 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 conforming architectural descriptions becomes constrained. Almost none of architectural recovery strategies reviewed support explicit techniques for consistency checking or development of arbitrary views/viewpoints. Likewise, very few provide explicit coverage of the extraction information space. This explains why some people liken architectural recovery to an art form. Those who are good at creating architectural descriptions usually have knowledge and experience that cover the three quadrants that do not involve specific code-based knowledge. ASP seeks to make this knowledge use explicit within the process to remove architecture recovery from an art form and make it a standard task.

3.4 Supporting Process Technologies

We now break from the discussion of architectural recovery to consider supporting elements for ASP. These research areas impact directly on ASP, but are not normally associated with software architecture research.

3.4.1 Inconsistency Management

In the early stages of architectural recovery, it is unlikely that the information an analyst initially obtains is consistent. Drift and erosion [54] result in the current implementation differing—sometimes significantly—from the original design documents. Any process developed, must be able to move forward in the face of inconsistency. The requirements engineering field has coped with this problem for years. During early analysis, it unlikely that stakeholders will provide unambiguous, consistent requirements, yet the requirements process must proceed forward.

The ViewPoint framework of Easterbrook and Nuseibeh [22] provide a prototypical approach to handling inconsistency. Nuseibeh proposes the following set of activities to manage inconsistency:

- Detection: This activity involves processes that help determine if and when an inconsistency occurs. An inconsistency occurs in specifications when both X and not X are found to hold. Any synthesis process developed for software architectures must also be capable of detecting inconsistencies.

- Classification: This activity tries to determine what type of inconsistency has occurred. The classification may involve categorizing inconsistency based upon its cause or by a domain-specific classification scheme.
- Handling: This activity tries to deal with inconsistency. Finkelstein [Finkelstien, 1994 #171] describes the following taxonomy for handling inconsistency:
- Ignore: If the inconsistency is minor or isolated, it may be possible to simply ignore it. The author's recommend it be tracked, but no corrective effort be expended.
- Delay: If there is further information required to understand and resolve the inconsistency, then its resolution may be delayed. Two different situations may occur because of delayed handling. Either the inconsistency will resolve itself as more information is developed and other inconsistencies are eliminated or it will interfere with other aspects of the development and it can no longer be delayed, but must be resolved.
- Circumvent: If there is a specific rule that was broken, it may be possible to disable that rule thereby removing the inconsistency. It does not appear there is a parallel to this option when handling architectural inconsistency.
- Ameliorate: If there are steps which can be taken to improve the situation while not fully removing the inconsistency then the overall situation can be improved. This is similar to an incremental resolution process.
- Resolve: If necessary, actions can be taken to immediately repair and remove the inconsistency.

ViewPoints then "… are loosely coupled, locally managed, distributable objects which encapsulate partial knowledge about a system and its domain, specified in a particular, suitable representation scheme, and partial knowledge of the process of development." There is a close analogy between the idea of a ViewPoint in requirements and an architectural perspective.

3.4.2 Conflict Resolution

Like all other aspects of software development, architectural recovery is a task that intimately involves human beings. Anytime humans are involved in a process, there is the opportunity for conflict. In an architectural recovery task, analysts may disagree with not only concrete artifacts such as design and code documentation, but with each other. The requirements engineering community has been struggling with this problem for some time. One promising solution derived from management theory is called "Win-Win" [11]. In Win-Win, negotiations are conducted which try to satisfy each participant's "win" conditions. Certain architectural artifacts and views may have significant organizational impacts and thus an easy resolution may not be achievable. This implies the need to incorporate negotiation activities in any complete recovery process.

3.5 Supporting Integration Technologies

The previous sections have looked at research supporting the overall process for ASP. This final section will look at specific research technologies that might be applied to the automated matching of architectural elements in different perspectives. These techniques include matching on lexical information like names, on topological information based on configuration, using type-theoretic approaches or using mathematical approaches like concept analysis. Before discussing these three technologies in detail, it would be useful to discuss the reasoning behind why these technologies might be applicable.

There are many underlying characteristics of software architectures that lead to the consideration of these three technologies. First consider the characteristics of software architectures from Bass et al [6]:

- Architecture defines components. At its topmost level, an architecture is an abstraction of the major components (and connectors) which make up the system. Bass et al also describe the architecture as the earliest embodiment of system design decisions. All these characteristics imply that at its topmost level the architecture contains information about how the architecture relates to its problem space (or domain). Each element has some relationship to specific concepts in the problem domain. This leads to the possibility that concept analysis is an appropriate technology to examine. The fact that architectures form hierarchical abstractions creates some challenges for the use of concept analysis. In traditional concept analysis there is generally one level of abstraction that is being examined. This means we will have to extend concept analysis somewhat to allow it to address the idea that some elements are contained inside another.

• Systems may have more than one structure. The topology (or structure) of the architecture typically forms a graph. Using some type of topological matching is a logical approach to comparing two architectures. One the other hand, there are complexities that complicate the use of topological matching. First, the information in each perspective is potentially incomplete. This leads to problems using techniques such as sub-graph isomorphism since connectors (edges) might be present in one perspective, but not in another causing false positive or negative indications. Secondly, architectures may represent different aspects of the system. For instance, if we compare a graph representing the physical aspect with one that represents the logical aspect we may have a difficult time understanding the results using topological information alone. Different aspects of the same system can have very different graphs (Consider the degenerative case where a complex processing system runs on a single machine. The physical graph has a single node and no edges, while the logical graph may have many nodes and edges in a complex layout.). This particular problem can be overcome at the process level by comparing only graphs representing the same aspect. Finally, architectures have multiple levels of abstraction. This introduces the problem of multiple graphs representing not only different aspects, but also different levels of abstraction of the same system.

• Every software system has an architecture. This characteristic actually helps us in that no matter how badly designed the legacy system is, there is some architecture to recover. This idea supports the use of almost any technology that can help in the recovery effort—not just the three detailed here.

• The behavior of each component is part of the architecture. This implies that elements (components and connectors) do things (and therefore have a semantic as well syntactic property). Behavior is usually described in terms of the problem domain. This supports the idea that by relating terms in the problem domain to elements, we are accounting for the description of the basic behavior of the element.

We also know that elements in the architecture have properties [54]. One such property is the lexical name of the element. Names are typically given to elements to reflect their function or purpose. This implies that doing some type of lexical analysis would aid in the comparison process.

3.5.1 Lexical Analysis

The idea of matching differing elements by their names is nothing new. In database schema integration, name matching can be a primary technique for combining various schema descriptions [25]. Within the reverse engineering community, recent work has focused on matching names in code libraries to promote reuse [48]. Michail's approach involves three different steps:

- Changing names in different libraries to a common name. This involves making the following names equivalent: TopWindow, top_window, and topWindow.
- Calculating metrics to help find naming matches when the names don't reduce to a common term. Michail refers to this as similarity matching. These metrics include:
- Inverse Document Frequency (idf): This measure relates how important a term is in the library. It is computed by the equation:

$$idf(t)=log_2((N/df(t)) - 1)$$

where t is a term, N is the number of components in the library, and df(t) represents the number of components which have term t in them.

- Within Document Weight (wdw_i): This measure describes how important a term is within a particular component.

$$wdw_i(t)=\sum (dc_i(t)+ic_i(t))$$

where $dc_i(t)$ is the direct contribution calculated by:

$$dc_i(t)=a_1(i,t)+ a_2(i,t)+ a_3(i,t)$$

where each of the a1, a2, and a3 terms represent three sources: the name, class/function, or comments. Each $a_k(i,t)$ is defined by:

$$a_k(i,t) = \{ A_k/2^{b_k(i,t)-1} \text{ if term t is in source k or 0 otherwise } \}$$

and $ic_i(t)$ is the indirect contribution calculated by:

$$ic_i(t)= \sum_{D_j \in R_i} dc_j(t)/2^{d(D_i, D_j)+1}$$

where $d(D_i,D_j)$ is the shortest distance between two classes or functions in a call-graph or class inheritance graph.

These formulae should be adaptable with minimal effort to architectural name comparisons as opposed to library name comparisons.

3.5.2 Topological Analysis

Most architectural perspectives can be represented as a graph. Matching elements then involves trying to determine if part of one perspective is a subgraph of another. Unfortunately, this problem—also known as subgraph isomorphism—is NP-complete [21]. Kazman and Burth [35] developed a technique called Interactive Architecture Pattern Recognition (IAPR). By annotating the architectural graphs, they turned sub-graph isomorphism into a constraint satisfaction problem as presented in Woods and Yang [66]. IAPR tries to match predefined architectural patterns to a given architecture. This technique should be adaptable to matching the elements in different perspectives by substituting an architectural perspective for the pattern.

Girard and Koschke [30] present a method to extract components from call graphs via dominance analysis. This technique might also be useful in detecting abstractions within a dense perspective.

3.5.3 Unification

Knight [38] gives a excellent overview of unification theory as applied to many different disciplines. If one considers elements as some type about which we know partial information, then unification might provide a mechanism similar to type-inference [49]. Unification from computational linguistics might also be applicable. If we considered perspectives as sentences about which we know certain words (elements), then we might be able to use feature-set unification to match the elements.

3.5.4 Concept Analysis

A very promising technology for matching elements amongst perspectives is concept analysis [64]. Concept analysis allows groupings to be formed based upon maximal sets of common attributes. Several other software engineering researchers are looking into concept analysis for module identification [58] and configuration management [42].

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, a subset of O X A that relates objects to the attributes they possess such that (o,a)∈R if object o has attribute a. We can then produce two important functions:

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

$\sigma(X) = \{ a \in A \mid \forall o \in X: (o,a) \in R \}$ (The set of common attributes of X)

$\tau(Y) = \{o \in O \mid \forall\ a \in Y: (o,a) \in R \}$ (The set of common objects of Y)

These functions lead to the fundamental theorem of concept lattices:

$$\bigcup_{i \in I}(X_i, Y_i) = \left(\tau\left(\bigcap_{i \in I} Y_i\right), \bigcap_{i \in I} Y_i\right)$$

If we assign architectural elements of different perspectives as objects and develop a set of common attributes across all perspectives, then it might be possible to compare two different perspectives based upon their concept lattices.

4. DETAILED APPROACH

This section presents the details of a proposed solution to the problem of obtaining a complete and consistent set of architectural representations from a legacy system using a repeatable process. The overall process is called *architectural synthesis*. This section is divided into two parts—a more detailed discussion of ASP, introduced initially in section 2, and a description of the tool that will provide automated support. Implementation details of the process will be illustrated in the context of concrete case studies detailed in the sections 5 and 6. Figure 4 presents the overall context of the ASP process.

4.1 The Architectural Synthesis Process (ASP)

ASP is fundamentally an information-processing task. Information is obtained about the architecture using some set of tools and techniques. We refer to a collection of information obtained from a single source or technique as a *perspective*. 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 not possible. Every legacy system varies as to its size, complexity and architecture. Most legacy systems are poorly documented, or the system has been modified so often that the existing documentation is inaccurate. 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.

It was stated earlier that a goal of this research was to produce complete and consistent representations. How then do we make a claim about completeness? 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 complete is to think about the

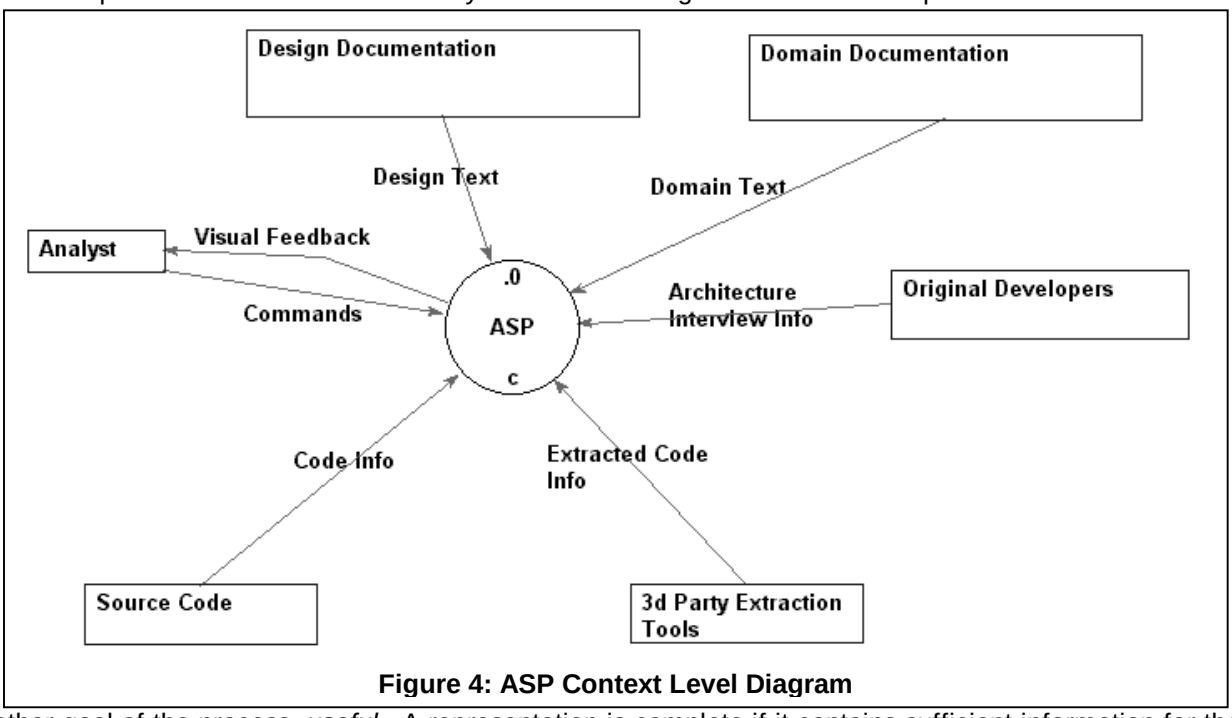

Figure 4: ASP Context Level Diagram

other goal of the process, *useful*. A representation is complete if it contains sufficient information for the analyst to accomplish his goal. This leads to the situation where a representation considered complete for a Rapide simulation may not be complete for a SAAM evaluation and vice versa.

This notion of completeness is consistent with current thought in the software architecture community. It is an accepted principle that there is no one true software architecture. Rather we can prepare an architectural description that supports some goal of the preparer. This matches our intuition since if I were going to discuss the performance of the system; I would sketch a representation that emphasized the architectural elements that deal with performance.

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. Secondly, we want to identify information that is in conflict with other information that has been collected. This 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. 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 sync 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, 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[69] on composing these specifications, the authors present three primary goals that their solution must meet. These goals are adapted here for ASP:

- ASP should accommodate a wide variety of architectural recovery paradigms and techniques.

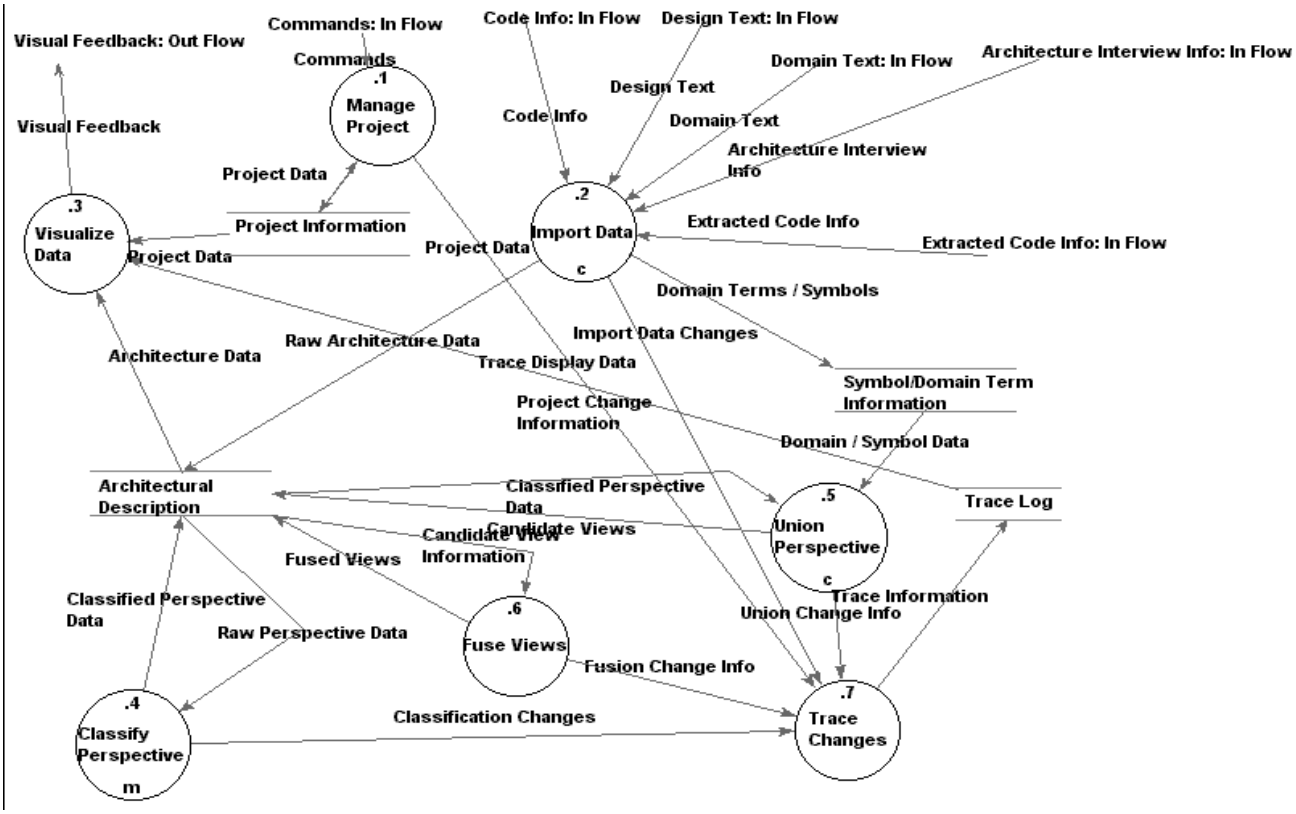

Figure 5: ASP Top-Level Process Diagram

- 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 5 presents the top-level process diagram for ASP.

4.1.1 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[34], 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 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 general 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 (Figure 2) to obtain the widest range of data sources for the architecture.

The principle requirement, and thus limitation, of ASP extraction is that the output of any extraction technique must be expressible as an attributed graph. Components become nodes and connectors become edges. Any other architectural information extracted is attached to the nodes, edges or to the graph as a whole as attributes. This is not however 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.

Architectural elements (components and connectors) can be analyzed and compared to one another using either syntactic or semantic information. Syntactic information about an architecture includes such superficial items as the element name and topological information. Semantic information about an architecture specifies the function of the element. Traditionally, semantic specification has been done using formal methods such as Z or VDM. For ASP however, a more lightweight specification needs to be found. I coin the term *semantic approximation* to denote a lightweight, easy to use method that provides enough information about the semantic meaning of elements to make matching decisions about them. (Lightweight implies both minimal effort and minimal training required to use the technique.)

One informal, lightweight method of recording the semantics of an element would be to write a short description of its function in plain text. While lightweight, this method does not easily lend itself to automation and matching. Since an architecture is a high-level description of the design solution to a problem in a specific domain, we can make a simple approximation to the semantics of an element by using a subset of the description in the form of key domain terms. By associating domain terms with the elements which have some relation to them we can form an approximation of what the full semantic description of an element might be.

We can then use these semantic approximations as attributes so that later in ASP we can consider not just structural but behavioral characteristics of graph elements when processing architectural information. Our approach to semantic approximation using concept analysis is presented in section 4.1.3.3. The use of concept analysis for semantic approximation of architectural elements is a supporting contribution of this work.

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.

Table III: 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

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

- A component in the process view maps to a component in the physical view via the runs-on relation. (We must determine which processors a given process or thread runs on). A con-

nector 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 the fusion phase.

4.1.1.1 Preconditions :

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

4.1.1.2 Process Steps:

- Determine the purpose of the architectural description that ASP is being used to develop. The purpose will drive many of the subsequent decisions that must be made. Write a short statement of the purpose to focus and guide the effort.
- 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.
- Determine the mappings (relationships) between the viewpoints that have been selected in the previous step.
- Gather Legacy System Information: Stable Code Base, Design and Domain Information and a list of original designers and current maintainers for potential interviews.
- 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 II presents a summary of some possible sources of perspectives based upon the viewpoints chosen.
- 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 will be in paper form)
- Convert the information in the previous step into an attributed graph, with components being nodes, connectors being edges and associated information being attributes.
- Dowse the available design and domain information to obtain a set of domain terms.
- Assign a subset of these domain terms to the elements in the perspective graph as another attribute.

4.1.1.3 Postconditions:

- Short statement of purpose of architectural recovery effort.
- List of Viewpoints that are relevant for the recovery effort.
- List of mapping relationships between the viewpoints.
- Set of perspectives (attributed graphs) representing the architectural data that has been extracted.

4.1.2 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 reconciliation 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:

- If the perspective came from an object-model diagram (such as an OMT diagram) it is a logical view.

- If the perspective came from a call-graph representation then it is a logical view.
- If the perspective contains hardware elements then it is a physical view.
- If the perspective contains component names that can be matched to static source code entities, it is a logical view.
- If the perspective contains names that can be matched to *make* targets it is a process view.
- If the perspective came from a legacy "box-and-arrow" diagram, it is a logical view.
- If the perspective contains information derived from the source code directory structure or depends sections of the make file it is a module view.

4.1.2.1: Preconditions:

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

4.1.2.2: 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.

4.1.2.3: Postconditions:

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

4.1.3 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 a set of domain terms that have been dowsed [14] from the available system artifacts. Dowsing is a technique that scans textual artifacts such as source code, design documents and user's manuals and extracts key words or n-grams. A set of these is chosen by the analyst to represent the key terms in the domain. Then a subset of these terms forms the set of domain terms to be associated with each element. These n-grams act as our semantic approximation method for the synthesis process.

General attributes consist of (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 basic 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.

4.1.3.1 MATCHING USING LEXICAL ANALYSIS

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[48] and function name analysis[13]. 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.

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.”

4.1.3.2 MATCHING USING TOPOLOGICAL ANALYSIS

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-Complete. The second however is the more serious. Since each perspective is potentially only partially complete, there may be nodes and edges missing between the two graphs.
There is a potential for these two problems to be overcome using Kazman’s IAPR technique to solve the sub-graph isomorphism problem using constraints. This might allow parts of different perspectives to be matched based upon their topological characteristics.

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, then the edges can be used to select the next node to match. Node matching also helps resolve the edge matching.

4.1.3.3 MATCHING USING CONCEPT ANALYSIS

Each architectural element has some set of attributes or properties that are associated with it as previously discussed. Using context analysis terminology (introduced in section 3.5.4), we establish the following:

- P1, P2 are perspectives
- O = { elements from P1 and P2 }
- A = { domain concepts }
- $R \subset (O \times A)$
- The projection of all common attributes of some set of objects is known as the sigma function. Sigma is defined as: for $X \subseteq O$, $\sigma(X) = \{\forall a \in A \mid o \in X: (o,a) \in R\}$

We can use this information to help in determining what relationships exist between an element in one perspective and an element in a different perspective. We can first note that there are several possible relations that might exist between the elements of perspectives P1and P2. These relations are presented from the strongest to the weakest confidence levels:

- EXACT(e1, e2). This relation is true if e1 is an exact match for e2—that is all attribute values of e1 are also attribute values of e2 and vice versa. Using our previous notation, EXACT(e1, e2)$\equiv \sigma(e1)=\sigma(e2)$. Note this relation is symmetric, i.e., EXACT(e1,e2) = EXACT(e2,e1).
- SUBSUME(e1,e2). This relation is true if e1 subsumes the description of e2. This occurs when the set of attribute values of e2 is a proper subset of e1. More formally, SUBSUME(e1,e2)$\equiv \sigma(e2) \subset \sigma(e1)$. Note this relation is asymmetric, that is SUBSUME(e1,e2) $\neq$ SUBSUME(e2,e1).

- CONTAIN(e1,e2). 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 subsystem of the component or connector that was decomposed. The CONTAIN relation is true if e2 is part of the subsystem of e1. This occurs when we can match e2 using EXACT, SUBSUME or OVERLAP to an element in the subsystem of e1. This relation is also asymmetric such that CONTAIN(e1,e2) ≠ CONTAIN(e2,e1).
- OVERLAP(e1,e2). This relation is true when e1 overlaps the description of e2. This occurs when e1 has attribute values in common with e2, but has other attribute values that are different. OVERLAP(e1, e2) ≡ $(\sigma(e1)\cap\sigma(e2)\neq\emptyset) \wedge (\sigma(e1) - \sigma(e2)\neq\emptyset \wedge \sigma(e2) - \sigma(e2)\neq\emptyset)$. OVERLAP is symmetric thus OVERLAP(e1, e2) = OVERLAP(e2, e1).
- NOREL(e1,e2). This relation is true if e1 and e2 have no apparent commonality—that is they are not related by the EXACT, SUBSUME, OVERLAP or CONTAIN relations. NOREL(e1,e2) ≡ $\sigma(e1) \cap \sigma(e2) = \emptyset$. NOREL is symmetric, that is NOREL(e1, e2) == NOREL(e2, e1).

While our technique works equally well for components and connectors, for 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. We now have a set of objects and attributes that are used to build the concept lattice. What algorithm can we use to traverse the lattice and locate these relations in the lattice?

We provide a simple example to illustrate the principles of element matching using concept analysis. Figure 6 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. Using dowsing, we build the formal context shown in Table III. We then compute the concept lattice using the *concepts* software package [41] to produce Figure 7. For this simple example, we can examine the lattice and detect the relations manually as follows:

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 7 where Alphabetizer in system S1 and Sorter in system S2 map to the same concept in the lattice.

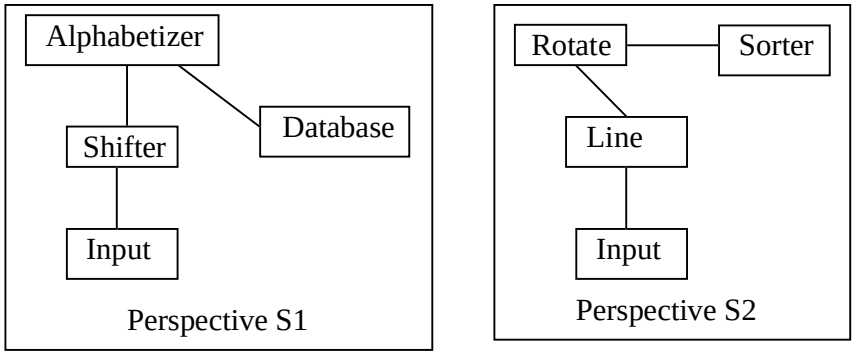

Figure 6: Simple Perspectives to Combine

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 7 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) [55]. 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. Run the depth-first search (dfs) algorithm provided by the GTL.

Table IV: Simple Architecture Formal Context

	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								

3. Iterate through the node list in dfs order

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 node is 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

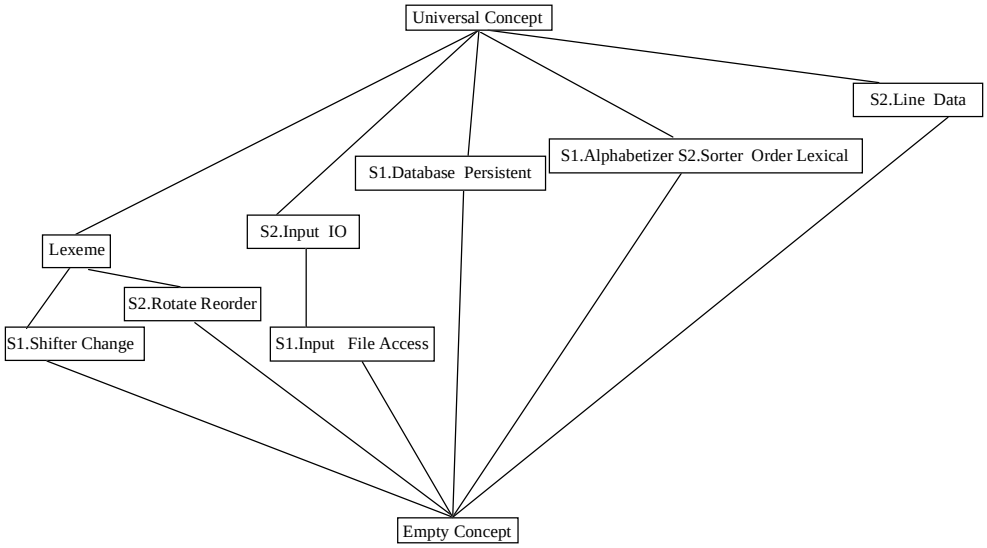

Figure 7: Simple Example Concept Lattice

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 outdegree>=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.

The CONTAIN relation is inferred from the EXACT, SUBSUME, or OVERLAP relation by examining the node labels and recognizing which are subsystem names and which are top-level system names.

We also compute two different metrics while checking for the OVERLAP relation. The first is an overlap measure that is the number of common attributes divided by the total number of attributes. For our example, the overlap measure for elements Shifter and Rotate would be .33. We also compute a concept distance by simply counting the intervening concepts between the nodes. Again for Shifter and Rotate the concept distance would be 1. These measures provide a way for the analyst to determine whether the OVERLAP relation is significant enough to warrant a match.

Unfortunately, in a real system the lattice is much more complex than the one shown in Figure 7. (For an example of a real-world lattice see section 5). There are two potential options for scaling the lattice. One is a mathematical approach using decomposition as proposed by Snelting [28]. The other is an in-

formation grouping and filtering strategy as proposed by [20]. Further study must be completed to determine which of these is best for complementing the synthesis 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.

4.1.3.4 Preconditions:

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

4.1.3.5 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 using the following technique:
      - Lexical Matches: First attempt to use the element name to match the first component in the union.
      - Topological: Use the connectors from this first match along with lexical information to continue the matching/union effort.
      - Semantic Approximation: Use concept analysis to complete the union effort and resolve any unknown elements that are not topologically or lexically matched.

4.1.3.6 Postconditions:

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

4.1.4 Fusion

We adopt this term from DALI [34] 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 [33, 34] 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 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. Figure 2 depicts sources of information that we use for fusion, mapped onto a portion of Egayed's conceptual framework. Information about what constitutes a viewpoint and general semantics is derivable from existing architectural metamodels. 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.

A complete description of our consistency checking approach and mappings using the Alloy language is presented in appendix A. The following paragraphs discuss mappings between the four basic viewpoints.

4.1.4.1 System->Physical

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.

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 a subset of the system view.

The actual process of mapping the system to the physical view is fairly trivial. There should be a correspondence between the elements in each view. Additionally, since these elements are of the same type, 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 is missing from the physical view.

4.1.4.2 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 interprocess 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.

4.1.4.3 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, which represent actual 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.

4.1.4.4 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[68] 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.

4.1.4.5 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 Figure 1, 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 presented in Figure 2, 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.

4.1.4.6 Preconditions:

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

4.1.4.7 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,

4.1.4.8 Postconditions:

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

4.2 REMORA Toolkit

Two things enhance developing a repeatable process. First we can define a standard process as in the previous discussion. Second, we can develop automated support that performs routine actions in a standardized way. This section details the proposed design for REMORA (Resolution of MORALE Architectures). REMORA has a conceptual architecture as shown in Figure 8.

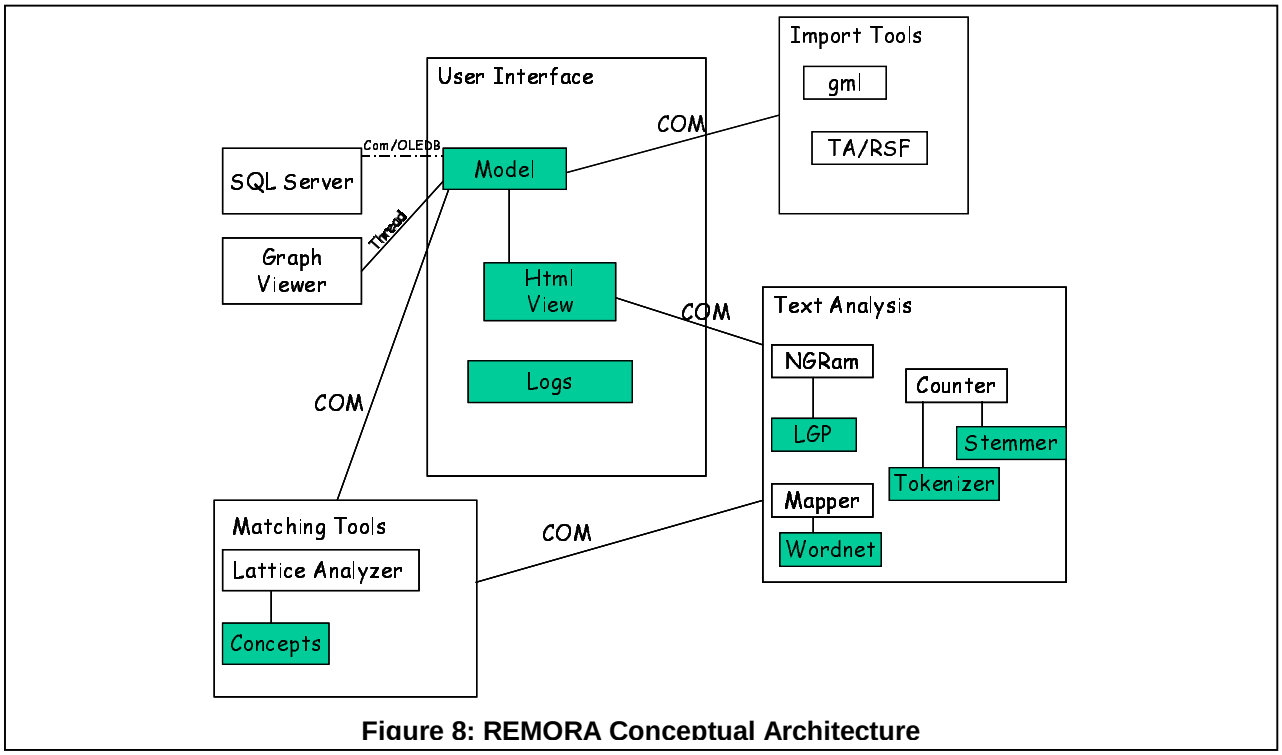

Figure 8: REMORA Conceptual Architecture

The conceptual architecture of REMORA represents a component-based approach to the development of the 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 remoting feature, workload-intensive components can run on different machines to accomplish their major tasks. In Figure 8, the shaded components are building block components which 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 [41] *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. This component also contains the WordNet software package, which performs significant computation services for synonym and semantic similarity detection.

4.2.1 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 9. 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 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

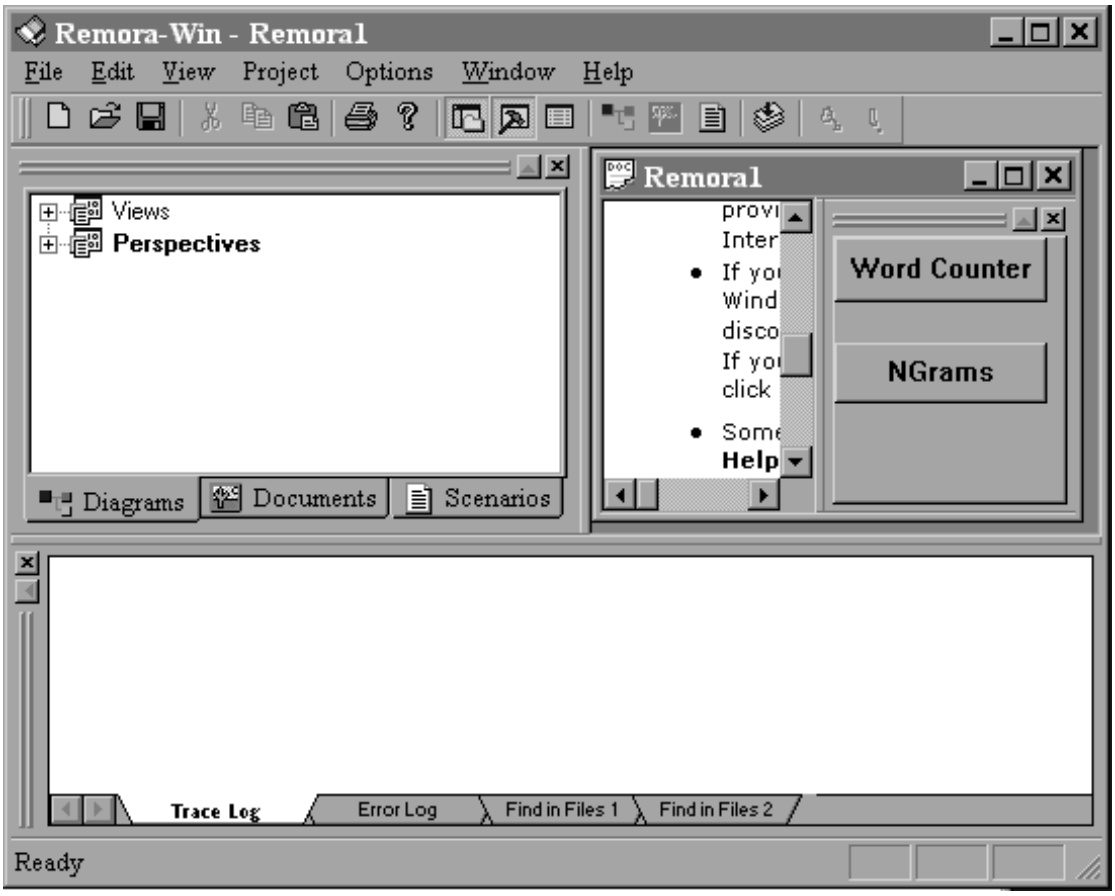

Figure 9: REMORA User Interface

grouped using the UI module.

The UI Module is implemented using 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 html, xml or MS Word documents. The view has an integral button bar which allows access to the supporting COM 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 an 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.

4.2.2 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 is via the 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.

4.2.3 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 in 4.1.3.3 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.

4.2.4 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.

4.2.5 SQL Server DB Module

Project information is persistently stored in a relational database. Currently, Remora uses SQL Server 7.0, 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 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. Data models are provided in Appendix C. There are two models, a high-level and low-level model. These models are currently correlated and integrated by the application code rather than by the data model itself.

4.2.6 Text Analysis Module

The text analysis module is the heart of the document processing capability in Remora. It provides several tools inspired by Dowser to analyze the project architectural documents. The shaded components in Figure 8 represent smaller building-block COM components that are not directly used by Remora. These are the Stemmer, Tokenizer, LGP and Wordnet 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 or Morph 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. Mapper takes a word and returns a set of related words based upon the user's request. It primarily uses wordnet, but may also use a user provided custom dictionary of domain synonyms.

4.3 Mapping P1471 to ASP

Table IV maps the requirements of a conforming P1471 architectural description to ASP.

Table V: ASP to P1471 Mapping

P1471 Requirement	ASP Phase
ID of Stakeholders/Concerns	Extraction: Identify interview candidates
ID and Define Viewpoints	Extraction: Identify viewpoints for classification
Representation of Architecture through views	Union: Develop set of views by combining classified perspectives
Record Inconsistencies	Fusion: Identifies and deals with inconsistencies
Rationale for selection of architecture	Extraction: Use of human interview information

5. Case Study 1 : Applying ASP to recover the ISVis architecture

We now present a case study of a manual use of ASP that supports the evolution of a medium-complexity system called ISVis (Interaction Scenario Visualizer)[33]. ISVis is a five-year-old C++ /X-Motif / Perl application consisting of 30 separate source files containing 24,333 lines of commented source code. Functionally, ISVis is a reverse engineering tool used to abstract an architectural perspective based upon both source code static information and a behavioral trace of program execution.

We are in the process of preliminary design for the next version of ISVis and wish to perform an ATAM analysis. We had many perspectives of ISVis available, but no one representation was accurate enough to begin the process. We decided to do an architectural synthesis to obtain an accurate set of architectural representations to begin the analysis.

A fundamental research purpose for this case study was to examine the requirements of ASP, refine the rough process, and document the principle steps required to accomplish the architectural recovery. This case study serves to illustrate many of the theoretic techniques (such as lexical, topological and semantic matching) previously presented in a more practical context.

5.1 Extraction

An analyst first generated several perspectives of the ISVis architecture, which are briefly described below. Only two of the graph representations generated are presented as figures in this proposal, but all are available in the technical report [63]. The perspectives generated ranged from a generic, abstract reference architecture to a concrete code-level call-graph. We selected as appropriate viewpoints the standard four—physical, conceptual(logical), runtime(process), and module(code).

5.1.1 Domain-Specific (Reference) Software Architecture.

For many legacy systems, a Domain-Specific Software Architecture (DSSA)[60] or a reference architecture describing may exist. A DSSA can be thought of as "an assemblage of software components, specialized for a particular type of task (domain), generalized for effective use across that domain, composed in a standardized structure (topology) effective for building successful applications" [1]. For this case study, a simple reference architecture for the reverse engineering domain, (the domain of the ISVis tool) was constructed using Tilley's reverse engineering framework [61] and Rugaber's Synchronized Refinement process[56].

5.1.2 DARE (Domain Analysis for Reverse Engineering) Model

This model is derived from textual analysis created by using the DARE process[17]. This gives another domain-oriented view derived directly from the ISVis documentation. The DARE tool first analyzed the ISVis user's manual and tutorial extracting all unique words using the dowsing technique. A filter then removed words of no interest to the analysis and the remaining words were counted to produce a frequency list. The most common domain-significant words were then analyzed and an OMT model was produced. The dowsed word list also formed the basis for the set of domain terms assigned to the various extracted architectural elements. For the case study, these domain term attributes included domain significant words such as *Disk File, Actor, Scenario, Utility, Source Code, Event, Trace, Visual, Mural* and *Static*. In all, 18 domain terms were identified.

5.1.3 ISVis Documented Architecture

Figure 10 represents a part of the original developer's view of the architecture typical of the box and arrow diagrams available for most legacy systems. Also available in this category were context diagrams and OMT object models of the legacy system.

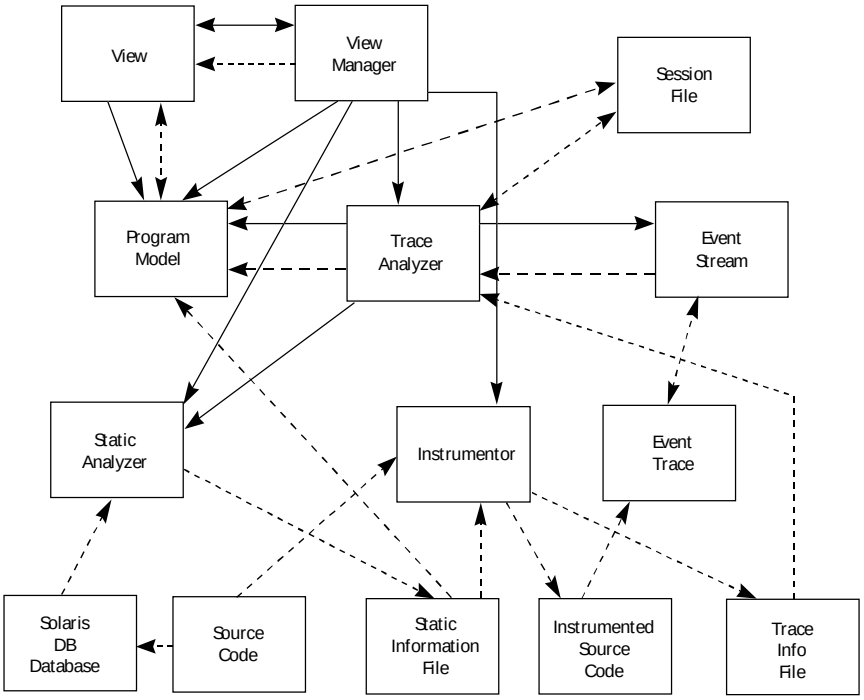

Figure 10: ISVis Design

5.1.4 ISVis Derived Architecture

This diagram (Figure 11) represents one of the architectures derived by using the ISVis tool on itself to create an architectural perspective from the source code. This process, which uses architectural localization and visualization, is described in [33]. In this particular case, the use of the ISVis tool gives rise to the interesting situation where the legacy system is used to analyze itself for evolution. The analyst first instrumented the ISVis source code and then performed two usage scenarios. From the two generated event traces, the analyst created abstracted components and scenarios which were manually translated into two different architectural perspectives.

5.1.5 RMTool Representations

These perspectives were the output of the Murphy, Notkin and Sullivan Reflexion model [50] applied to the ISVis source code. The input models used by RMTool were based upon the information derived from the ISVis analysis(5.1.4). The high-level model of components was based upon the abstract com-

ponents identified by the analyst. The mapping of classes and functions to components was based upon ISVis information as to the actors contained in each component of the high-level model. The Reflexion

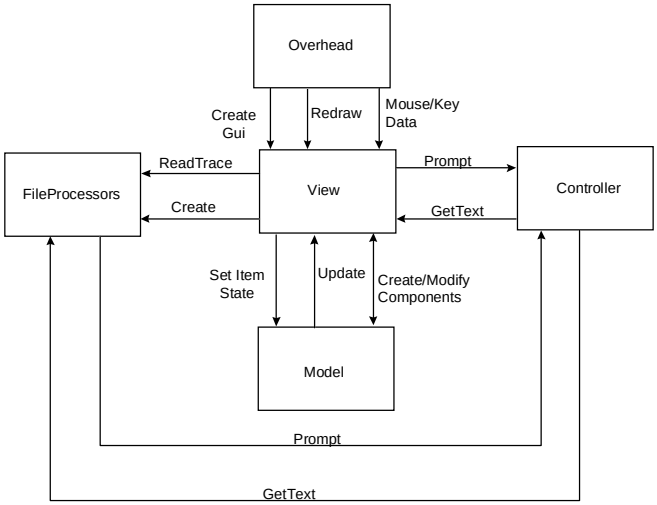

Figure 11: Dynamic Trace Extracted Architecture

tool helped to provide a measure of the relative accuracy of the ISVis-derived architecture and thus was a complementary perspective rather than one that added totally new information.

5.1.6 Call Graph

This type of diagram is a basic "who-calls-who" analysis typical of many reverse engineering static analysis tools such as *cflow* [2]. To obtain the call graph, we wrote simple filter programs to act on the Solaris C++ compiler browser files and produce *dot*[40], *gml* [32], *vcg* [57] and rigi [65] graphics files. This allowed the use of a wide variety of visualization tools to view and refine the graphs. The raw ISVis graph had over 820 nodes and well over one thousand edges. This information was used much as it was in DALI[36] to aggregate the call information into an architectural perspective.

5.1.7 Make Analysis

The Makefiles [52] for the application were analyzed and two perspectives were created. One dealt with the process view that resulted from make target analysis and the other was a module view that was

Table VI
ISVis Extraction Phase Results Summary

Source	Component Count	Connector Count	Levels
DSSA	12	15	1
ISVis Design	14	26(1)	1
OMT Design	28	0	1
Context Diagram	5	5	1
ISVis Derived (2 models)	39	38	2
Reflexion (3 models)	5	9(1)	1
DARE	15	7(2)	1
Call-Graph	820	1500(3)	1
Interview/Make	3	3	1
Make Depend	30	83	1

(1) Denoted as control or data only
(2) Labeled associations only
(3) Function calls only

derived from the dependency analysis.

5.1.8 Summary

By the end of the extraction phase, 13 perspectives had been obtained. These perspectives are summarized in Table VI and plotted against the information space in Figure 12. For each perspective generated from a given source, the table shows the number of components and connectors in that perspective and the number of levels of abstraction in the perspective. It is immediately evident that most sources produce a relatively flat representation. Referring back to section 4, at the end of the extraction phase we had completed all the tasks in section 4.1.1.2. We had established a purpose for the recovery

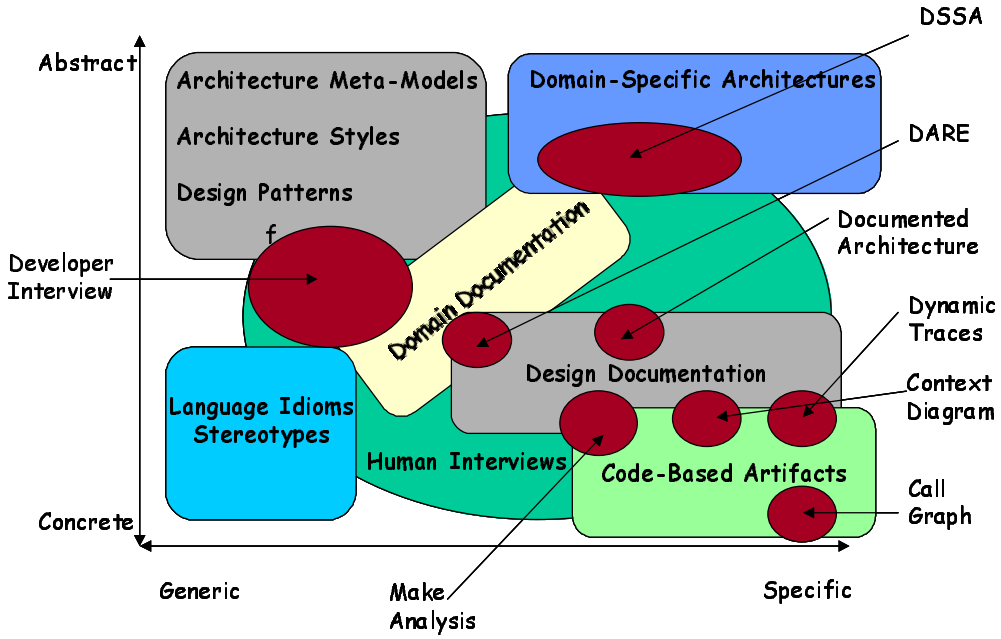

Figure 12: Information Space Coverage for ISVis Extraction

effort, we had selected viewpoints supporting that purpose, and we had obtained a set of perspectives that represented an adequate coverage of the extraction information space.

5.2 Classification

The classification task for this case study was fairly simple because most of the perspectives dealt with a logical view of the system. This resulted primarily because ISVis runs on a single machine with three easily identified processes greatly reducing the complexity of the physical and process views. At the end of the classification phase of the case study, there were 11 perspectives grouped into the logical view, one in the process view and two in the module view. Since ISVis is not a distributed program, but runs on a single processor, the physical view was a trivial single component. Further use of ASP focused on the conceptual, module and process viewpoints only. At the end of the classification phase, we had completed the steps of section 4.1.2.2.

5.3 Union

With the perspectives generated and classified, the real work of uniting these multiple perspectives to obtain a single set of views to describe the architecture can begin. Referring to section 4.1.3.5, we see the same set of steps is repeated for each viewpoint therefore this section will concentrate on the application of union to the conceptual viewpoint. The same procedures would be followed for each viewpoint which we felt was significant for the AD being recovered.

5.3.1 Choosing the Base Representation

The first step during union is to choose a perspective to act as the base representation for the remainder of the process. This base representation should be the perspective that represents the highest level of confidence and a high level of abstraction for the view under consideration. Eixelsberger et al.[24] for example recommend the design documentation be used as the starting point. For this case study, we selected the ISVis-derived architecture of Figure 11. We did not chose the design perspective as recommended by other researchers because we felt in this case the code-derived perspective was more accurate. In the general case, we would like to choose a perspective for the base representation that uses a source from Figure 2 that is located slightly above the midway point of the abstract-concrete vertical axis and clearly in the specific range of the horizontal axis. At first glance then, it might seem we violated our own advice since ISVis might be considered a source from the code-based artifacts region of Figure 2. In ISVis, the analyst must assign code components to higher-level abstractions by grouping them into components. In this particular case, the high-level components came from interview information with the original developer. By discovering that the developer had tried to use the standard Model-View-Controller design pattern, information from higher in the source graph was used to improve the more specific information from the code. This is an example of promoting artifacts up the abstraction graph through complementary sources of information. This also explains why ISVis could successfully serve as a base representation.

We briefly discuss issues in the union of the design perspective (Figure 11) with the base representation in this section. More details can be found in the technical report [63]. This base representation is then extended by unioning it with the other perspectives in the logical view.

5.3.2 Using the Union Algorithms

We briefly discuss issues in the union of the design perspective (Figure 11) with the base representation selected in the previous section. This section focuses on the conceptual viewpoint, however more details can be found in the technical report [63] on other viewpoints. This base representation is then extended by unioning it with the other perspectives in the logical view. This section illustrates the three principle algorithms described in sections 4.1.3.1, 4.1.3.2 and 4.1.3.3. Since the use of the algorithms is synergistic, their descriptions are interleaved in this section.

We begin by picking an element in the design representation and attempt to match it to an element in the base representation. Elements are described in a perspective with the following characteristics:

- N : the lexical name of the element. Lexical names may be the same or may be domain synonyms.
- D: the full set of domain terms dowsed from the legacy system's artifacts.
- D_e: the set of domain terms associated with the element. $D_e \subseteq D$
- A: the full set of general attributes for the legacy system
- A_e: the set of attributes associated with the element
- V_e: the value associated with a specific attribute of an element
- $\{(A_e, V_e)\}$: the set of all general attribute-value pairs associated with the element

We begin element matching by looking at nodes in the graphical representation of the perspectives. Let P1 and P2 be two perspectives pertaining to the same viewpoint and let element $e1 \in P1$ and $e2 \in P2$. We have five possibilities for the comparison of e1 and e2 (listed from highest confidence to lowest confidence):

- EXACT: A node in one perspective has the exact same general attribute-value pairs, domain terms and lexical name as a node in another perspective and is therefore the same element. Thus $EXACT(e1,e2) \equiv (N_{e1}=N_{e2}) \wedge (D_{e1}=D_{e2}) \wedge ((A_{e1}, V_{e1})=(A_{e2},V_{e2}))$. EXACT is a symmetric relation.
- SUBSUME: A node in one perspective is a more detailed description of a node in a different perspective and therefore is subsumed by the node. Thus $SUBSUME \equiv ((N_{e1}=N_{e2}) \wedge (D_{e1} \subseteq D_{e2}) \wedge ((A_{e1},V_{e1}) \subseteq (A_{e2},V_{e2}))$. SUBSUME is an antisymmetric relation.
- CONTAIN: A node in one perspective is a component (node) in the subsystem of the node in another perspective. We can determine the CONTAIN relation by finding an EXACT, SUBSUME or OVERLAP relation between the contained node and a node in the subsystem of the containing node. CONTAIN is an antisymmetric relation.

- OVERLAP: A node in one perspective has some domain terms in common with a node in a different perspective, but other domain terms are different. OVERLAP(e1, e2)≡($D_{e1} \cap D_{e2} \neq \varnothing$)∧($D_{e1} - D_{e2} \neq \varnothing$)∧($D_{e2} - D_{e1} \neq \varnothing$). OVER-LAP is the weakest of the 3 matches and the most prone to false positives. When determining an OVERLAP relation, we use only the domain concepts to reduce false positives. If we did not, then every composite element that was an abstraction of other atomic elements would satisfy the OVERLAP relation. OVERLAP is a symmetric relation.
- NOREL: A node in one perspective does not match any node and therefore is a new node. NOREL(e1,e2)≡ ($N_{e1} \neq N_{e2}$)∨(($D_{e1} \cap D_{e2} = \varnothing$)∨(($A_{e1}, V_{e1}$)∩($A_{e2}, V_{e2}$)=∅). If a node cannot satisfy SUBSUME, OVERLAP, CONTAIN or EXACT with any other node in another perspective then it is treated as a new node.

We should mention here that there is a sixth type of relation that might exist—the ERROR relation. It might be the case that an element in perspective is in error. This type of relation frequently occurs in human interviews or domain architectures where an element is claimed to exist, but does not actually exist in the real system. ERROR(e1) implies that element e1 should be removed from the unioned view.

We now give concrete examples of these relations as applied to the union of Figures 10 and 11. We initially select a node in Figure 11, *Program Model*, and try to match it to a node in the base representation using one of our five relations. Searching the nodes in the base representation, we first look for lexical name matches and find *Model*, which is a domain synonym. We now compare domain terms as dowsed from the ISVis textual artifacts and the general attribute-value pairs. Comparing the domain terms we find that they all match, therefore we have an EXACT relation. Figure 13 shows a partial view

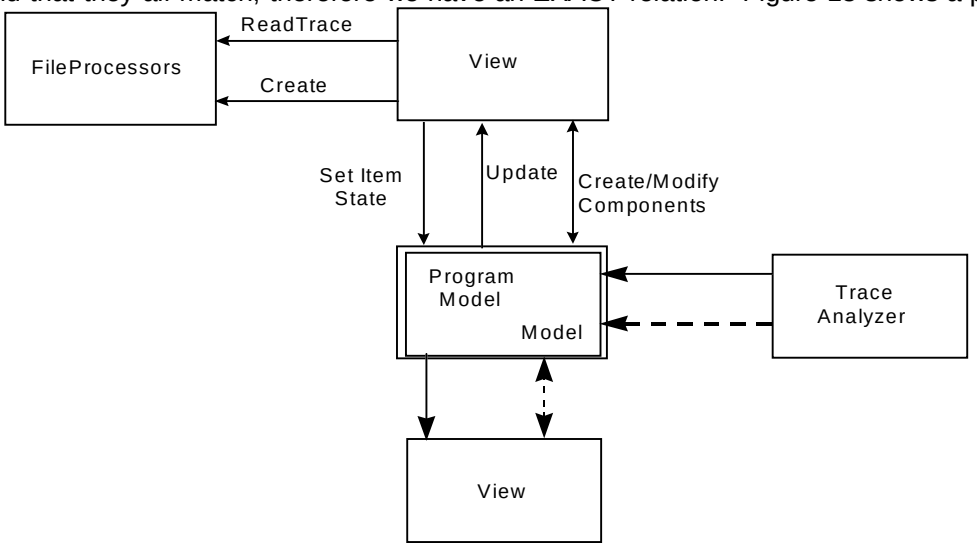

Figure 13: EXACT match for Model, Resolving Edges

after the Model node is matched. We now try to match the edges flowing from *Program Model* to those in the base representation. We generally refer to the process of edge matching after a node match as *resolving* the edge.

We see there are control and data connectors between *Program Model* and *Trace Analyzer*. We first have to find *Trace Analyzer* in the base representation. Doing the initial lexical comparison we do not find a match so our choices are narrowed down to CONTAIN, NOREL or ERROR. We can get an EXACT match to the *TraceAnalyzer* subcomponent of the *File Processors* node. This produces a CONTAIN relationship between *Trace Analyzer* and *File Processors*. Looking at the base representation, there is no connector between the *Program Model* and the *File Processors*, so we must add one for control and one for the data connectors in effect giving us NOREL relations for these connectors. Later by using the Reflexion Model, we find that the control connector is an ERROR relation and should not exist in the final logical view. We also record a binding in the *FileProcessors* subsystem so that we know that the *Trace Analyzer* sub-component has a connector that reaches the *Mode*l component in the top-level diagram.

We now resolve a new edge leading out of *ProgramModel* and choose the control and data edges connecting to the *View* component. When we match the *View* component in the design representation to the base representation, the initial lexical evaluation points us to the View node. If we did not have addi-

tional attributes, we might make an incorrect match. Using our relation rules, we find it is in the CONTAIN relation and therefore a sub-component of the View element in the base representation. This is why the domain terms and attribute-value pairs are so important. They help to prevent false matches that might otherwise occur if we depended solely on lexical names.

The design representation has several passive elements that are not present in the base representation. This is because the ISVis perspective is generated from dynamic event traces that do not do a good job of identifying passive data elements. For all the file-type components we have a NOREL with the elements in the base representation. Based on the attributes, they could be placed as sub-components in the *File Processors* component via a CONTAIN relation. As shown in Figure 14, the analyst actually placed them at the top level rather than as a sub-component of *File Processors* because modification of these files was projected for the new version of ISVis and we wanted to emphasize them for the analysis. This is a good example of how the use to which the representation will be put influences the content. It also demonstrates that there is no one "right" answer when someone asks to see an architecture. Rather there are many representations that are equivalent and might be developed. The important thing for the analyst is to develop a complete, consistent and useful set of views.

Figure 14 presents the final top-level logical systems view attained after unioning all perspectives classified to the logical view. The components with thick outlines have subsystems associated with them. One might be struck immediately by the inclusion of the analyst as a component in the top-level system. Normally the human user is not explicitly modeled in an architectural representation – yet in this case, the analyst not only is a top-level component, but has a subsystems view also! This occurs because the analyst has significant computation and data responsibilities within the ISVis architecture. For instance, from the DSSA, we know there is an architectural style library and a component that uses the library to understand an architecture. In ISVis these functions are performed manually by the analyst making them significant enough to include in the architectural diagram. Later, if we need to use this top-level representation for impact analysis using either SAAM or ATAM, we can better understand where

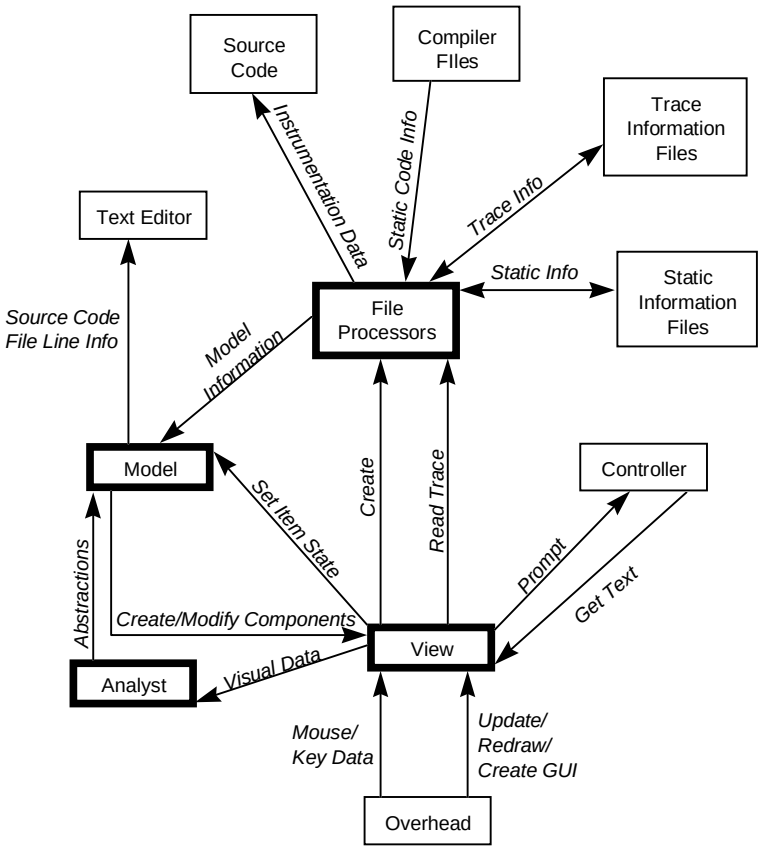

Figure 14: Final ISVis Top-Level Logical View

style-related information comes from. If we had a code-extracted perspective alone this type of information would not be available. This situation also illustrates the importance of obtaining information from sources in Figure 2 that are in the abstract-specific quadrant of the graph. No amount of clustering, k-cuts or remodularization can mitigate the concept-assignment problem as well as understanding the specific domain abstractions for the legacy system being analyzed.

Handling of connectors is one of the more difficult parts of the union process. The reasons for this are two-fold. First, connectors are usually second-class citizens in the world of legacy architecture. As the ISVis design in Figure 10 demonstrates, many documented legacy architectures do not even label the connectors. They may be annotated (as this one is) for differentiating control versus data, but they have no precise meaning. For example they might mean calls, uses, or talks-to. Promoting these ill-defined connectors to first-class elements of the architecture requires critical thinking and use of domain and application knowledge by the person doing the synthesis.

A principle function of an analyst in connector resolution is looking for name changes from the generic to the specific. For instance, in the DSSA mapping information connectors indicate architectural information that has been synthesized by an analyst or system. In ISVis these mappings correspond to the visual information provided the analyst. The need for these types of complex transforms motivates our belief that the synthesis process can never be fully automated.

The second complexity for connectors is resolving their bindings (that is what components are attached to each end of the connector). This is especially challenging when placing connectors in subsytems, and determining how these connect back to the upper-level system. This again requires the analyst to have an understanding of how the components communicate. Some of this understanding comes from observing the contained interactions that can be represented through an analysis/visualization tool like ISVis that provides the abstraction needed over sequences of program events. The call-graph perspective can also help with this task when developing the logical view.

5.3.3 Summary of Union Phase

At the end of the union process, the 13 perspectives representing the logical view—with their single level of abstraction, over 750 potential components and 1600 potential connectors—were reduced to three levels of abstraction with a total of 26 components and 40 connectors. By following the union process, different perspectives comprising very flat information (at most two levels of hierarchy), were refined into a single perspective, with multiple levels of abstraction that more accurately portrayed the actual ISVis architecture. We felt that this reduction made the representations more understandable and usable for other analysis activities.

It is interesting to note that although the component count (disregarding the call-graph case) did not increase significantly, the connector count did. Many reverse engineering tools do well at identifying components, but do not fare so well at finding connectors. By unioning several different perspectives we were able to find additional relationships between components that might have been lost if a single perspective had been used.

The top level representation has only 11 components and 16 connectors, a configuration easily analyzed during an ATAM session [5]. If questions arise during the session, there are subsystem representations that clearly identify the functionality in each of the top-level components. Again this is better than using any of the initial perspectives by themselves.

5.4 Fusion

The fusion phase was fairly straightforward for ISVis. Mapping the Process->Physical views was trivial since ISVis ran on a single processor. The Conceptual->Module mapping was also simply accomplished—primarily because one of the extraction tools used (ISVis itself) provided a mapping of code modules to conceptual elements as a by product of its maintenance of static and dynamic architectural information.

5.5 Summary of Lessons Learned

This case study illustrates several important concepts pertaining to practical uses of ASP including:

- Extraction techniques should be chosen so that some information comes from each of the four quadrants of Figure 2.

- Choice of the base representation should be made using a source from the abstract-specific quadrant of Figure 2. This helps mitigate the concept assignment problem by providing the high-level abstractions for the system.
- Connector resolution is difficult for legacy systems because they are frequently unlabelled and poorly documented in existing legacy documentation.
- The use to which the architectural description will be put influences the content of the developed views.
- Lightweight semantic approximation mitigates false positives caused by similar naming schemes, but different levels of abstraction.
- Provision of explicit matching relations (EXACT, SUBSUME, CONTAIN, OVERLAP and NOREL) help an analyst reason about the combinations of two different perspectives.

6. Case Study 2: LINUX Kernel recovery using ASP

The LINUX operating system kernel architectural description was developed using ASP and following the process steps outlined in section 4 of the proposal. In order to evaluate the effects of purpose on the resolution of inconsistencies and the selection of viewpoints, the architecture was recovered for two purposes. First as a common program understanding and modification task, we wanted to understand the security structure of kernel so as to modify the kernel to use access control lists rather the more simplistic Unix permission structure. This is a realistic task and was derived from the developers todo list at the Linux web site. Secondly, so as to address the most general case, we recovered the architecture for the purpose of developing a conforming P1471 AD. This would correspond to the general task of supporting the Linux documentation project with a full architectural description of the Linux kernel.

There have been several other attempts made to recover various aspects of the LINUX kernel, but none have assumed either of the purposes that were presented in the introduction to this case study. We can, however, import their data and use it to provide a comprehensive set of perspectives from which to work with. We can also use this data as "truth" data to compare the derived results of ASP to their conclusions so that an assessment of "value-added" can be made.

NOTE TO COMMITTEE:

This section is a placeholder for emerging results in a major case study to validate the process definition. For now you can skip over the rest of section 6 until I get some results to place in here.

6.1 Extraction

The following perspectives were obtained using various tool suites.

6.1.1 Domain Specific Software Architectures (DSSA)

There are many DSSA's available for the domain of operating systems. Appendix D, Figures 1 and 2 present two common ways of looking at an operating system. These figures are adapted from [46]. Figure 1 is based on the concept of a non-portable abstraction layer that interfaces directly with the hardware and a portable layer that remains constant regardless of the platform that the OS is being ported to. Figure 2 on the other hand, emphasizes the major functional groupings that exist in an operating system. We would expect then that if the final AD conceptual view conforms closely to Figure 1, then portability was a major nonfunctional requirement. Likewise, if we find the architecture conforms more closely to Figure 2, we might imply that modifiability or extensibility were major considerations. In any event, neither of these architectural views helps us much with the task of modifying the security features of Linux, since security does not seem to be addressed at this level of abstraction.

6.1.2 Portable Bookshelf

The portable bookshelf (PBS) research group (described in 3.3.2) used their technique to recover the LINUX Kernel. This perspective can be accessed and viewed at http://www.turing.toronto.edu/~brewste/bkshelf/linux/V2.0.27a/index.html. This particular perspective, because it is itself synthesized, provides a rich set of information about the module view of LINUX.

6.1.3 Rigi

There is also available a detailed set of RSF extracted raw facts about the LINUX kernel. This information was downloaded and imported into Remora. It basically provides the equivalent of static call-graph level and data access information.

6.1.4 DALI

The DALI team also provided a set of information extracted using the DALI techniques. This information was in the form of a Tuple Attribute (TA) file. This file was also imported into Remora and used to generate perspectives.

6.1.5 Text Documentation

Documentation was available from the LINUX Documentation Project. This documentation was imported into Remora and used for the text processing tasks. We also used electronic information on the general domain of operating systems.

6.1.6 Call Graph Data

6.1.7 RMTool

6.2 Classification
6.3 Union
6.4 Fusion
6.5 SUMMARY
6.6 CONCLUSIONS

7. CONCLUSIONS

This work has demonstrated a conceptual model into which all architectural recovery work can be placed. Further, a defined process has been specified that produces P1471 compliant AD's. By using domain information to help guide aggregation of information, ASP mitigates the concept assignment problem. By providing a method for checking different views and cross-mapping information, inconsistencies can be identified. By providing the concept of an extraction information space and information space coverage, the analyst can plan extraction so that redundant information is not gathered, and fully informed views can be developed. Finally, a tool supporting an analyst using ASP has been presented.

Table VII: Case Study Plan

Case Study	Domain	Languages	Validation	Perspectives Developed	Views Created (Union)	Views Fused	Automation Used	Results
ISVis	Reverse Engineering	Perl, C, C++ Single Process, Single Processor 24 KLOC		13 RMTool Make Analysis Cflow ISVis Design Docs Dowser Interview	5	2	None	Proof of Concept see TR GIT-CC-98-22 Shortcomings - limited opportunity for fusion Usefulness:
Linux	Operating Systems	C, ASM,	Compare to other LINUX case studys	PBS DALI Rigi	TBD	TBD	Remora	TBD

8. VALIDATION

Validation of the technique will be conducted in two phases. In phase one, the overall conceptual model is validated through a survey of existing architectural recovery techniques and mapping these

onto the ASP. This validates the thesis that the conceptual model is able to accept and organize all different recovery techniques.

Phase 2 involves use of ASP case studies to determine adequate definition, completeness and consistency of the AD. Table VI summarizes the planned case studies. The first study (ISVis) was completed using primarily manual techniques. Its purpose was to develop and validate the synthesis process. There were several limitations discovered during this case study. The first was that ISVis is essentially a single-process/single-processor application. This limited the physical viewpoint of the architecture to a trivial representation. The lack of multiple views resulted in a limited chance to develop and refine the fusion phase of the synthesis process. The manually developed views of the AD were then used to validate the ASP algorithms by using those views as truth data.

The primary validation case study will involve extraction of the LINUX operating system kernel architecture. This system provides an "industrial-strength" size application of non-trivial complexity. It also has the advantage of being open-source and having an accessible developer community. The LINUX kernel has also been designated as an architecture recovery exemplar, and thus has been the object of several recovery efforts including DALI, PBS and a manual recovery by Bowman et al. [12]. These recovered architectures will be used as baselines to show the additional value-added by ASP. Also, the AD produced by ASP can be critiqued by the LINUX developer community for additional validation of its "quality." In a sense the LINUX development community can be used as an oracle that "knows" the true architecture and can provide a standard against which the ASP-derived products can be compared. The idea of quality of the AD of LINUX principally focuses on the latter half of the thesis statement. While we could make the statement that a defined process was specified and usable at the end of the first case study, we cannot make any claims about the fully informed, consistency, usefulness or conformance of the AD. Table IV demonstrates that the product is conforming by showing the mapping of the ASP process onto P1471 (which in turn could be mapped onto other emerging standards for architectural description). That leaves the more subjective elements of fully informed, consistent and useful.

As we defined earlier, fully informed implies the use of all available information about the legacy. To ensure that a user of ASP uses all available information, we introduced the idea of the extraction information space. By obtaining information from all quadrants of the information space, we can gain enough information to make fully informed descriptions. How then do we prove that the information space represents the necessary range of extraction information? We can map existing taxonomies of information onto the information space and provide a reasoned argument that this information provides an adequate basis to develop architectural descriptions.

Likewise, we previously defined consistent as having accounted for all conflicting or inaccurate information. How do we show that the LINUX ASP is consistent? First we have the other recovered architectures to compare it to. Secondly, and most valuable there is a rabid core of developers for the kernel who would be only too anxious to find errors and inconsistencies in the produced AD.

Finally usefulness has to be shown. Can the AD be used for its intended purpose? Again the entrenched LINUX community can be extremely helpful. For the group looking into extending the security features of LINUX, we can ask if the AD was helpful in performing architectural-level understanding of the security architecture of the kernel. Likewise, for the general architectural description, we can query the LINUX documentation project to determine whether the total AD is valuable for understanding the architecture and structure of the kernel across multiple viewpoints.

Completing these steps, we will have shown that the thesis:

> *A defined process for obtaining a software architecture description (AD) from a legacy system can be specified in sufficient detail to be usable by practicing software engineers, and the products produced by the process are complete, consistent, useful, and conform to P1471.*

is true.

9. SCHEDULE

The schedule below relates the principle research questions of this proposal to project delivery dates.
Thesis restatement: The fundamental thesis of this work is that a repeatable process for obtaining a software architecture from a legacy system can be defined in sufficient detail to be usable by practicing software engineers, and that its products are complete, consistent and useful.
Research Questions:

1. What process can be defined to synthesize architectural information obtained from any potential source? (See section 2 ASP)
2. What methods are appropriate for manipulating and representing architectural information? (see section 4 on Remora DB data model)
3. What is an appropriate data model for storing architectural information? (see section 4 on Remora DB model)
4. Given two different sets of architectural information, how can that information be combined? (see section 4 on lexical, topological and semantic approaches)
    4.1 What syntactic information is important for comparing two sets of architectural information? (see section 4, lexical matching)
    4.2 How can semantic information be represented in a lightweight, easy to automate fashion? (see section 4 on sematic approximation using concept analysis)
    4.3 What methods are most effective for matching architectural elements in different perspectives? (see section 4 on lexical, topologic and semantic approaches)
5. What views or view sets are most useful for practitioners? (Open, seems to depend on domain and purpose for which architecture is being recovered)
    5.1 What rules are effective for determining which information elements refer to which views? (see section 4 on classification)
6. What rules can be developed to provide guidelines for determining the completeness of architectural information? (see section 4 on details of union and section 2 on the extraction information space)
7. What mappings exist between different architectural viewpoints and their corresponding views? (see section 4 on fusion)

20 Sep 200	Complete draft Proposal, gain preliminary approval from committee
Fall (Oct) 2000	Formal Proposal Presentation
Fall (Dec) 2000	Complete Remora Toolkit
Fall (Dec) 2000	Complete LINUX Case Study (Start-Up and extraction phase underway)
Spring 2000	Defense of Research
Summer 2000	Graduation

10. DELIVERABLES

The following three items constitute the deliverables for this research:

- ASP User's Guide
- REMORA Toolkit
- Dissertation

11. FUTURE WORK

Future work in ASP can be focused on moving from a defined to a repeatable process. This would involve observation-type experiments on several groups, each using ASP to recover the same legacy system and analyzing where variances occurred. This would also allow further refinement of the mechanics of the ASP process.

There are several automatic text processing algorithms using neural networks and self-organizing maps that hold promise to further refine the semantic approximation algorithm. This would reduce variability in assignment of domain terms to the appropriate architectural elements.

12. REFERENCES

[1] "Architecture-Based Acquisition and Development of Software Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program," Teknowledge Federal Systems 1994.

[2] "cflow homepage," , 2.1 ed: public domain, 1997.

[3] "Draft Recommended Practice for Architectural Description," IEEE, Piscataway, NJ IEEE P1471/D4.1, December 1998.

[4] G. Antoniol, G. Fiutem, G. Lutteri, P. Tonnella, S. Zanfei, and E. Merlo, "Program Understanding and Maintenance with the CANTO Environment," *International Conference on Software Maintenance*, 1997, pp.
[5] M. Barbacci, S. Carriere, P. Feiler, R. Kazman, M. Klien, H. Lipson, T. Longstaff, and C. Weinstock, “Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis,” Carnegie Mellon University, Technical Report CMU/SEI-97-TR-029, 1998.
[6] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*: Addison Wesley Longman, 1998.
[7] B. Bellay and H. Gall, "Reverse Engineering to Recover and Describe a System's Architecture," *Proceedings : Development and Evolution of Software Architecture for Product Families*, Las Palmas de Gran Canaria, Spain, 1998, pp. 115-122.
[8] J. Bergey, D. Smith, S. Tilley, N. Weiderman, and S. Woods, “Why Reengineering Projects Fail,” Carnegie Mellon University, Software Engineering Institute, Pittsburg, PA CMU/SEI-99-TR-010, April 1999.
[9] J. K. Bergey, L. M. Northrop, and D. B. Smith, “Enterprise Framework for the Disciplined Evolution of Legacy Systems,” Software Engineering Institute, Pittsburg, PA, Technical Report CMU/SEI-97-TR-007, October 1997.
[10] T. Biggerstaff, B. Mitbander, and D. Webster. "Program Understanding and the Concept Assignment Problem," *Communications of the ACM*, vol. 37, pp. 72-83, 1994.
[11] B. Boehm, P. Bose, E. Horowitz, and M. Lee, "Software Requirements Negotiation and Renegotiation Aids: A Theory-W based Spiral Approach," *17th International Conference on Software Engineering (ICSE-17)*, Seattle, 1995, pp.
[12] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a Case Study: Its Extracted Software Architecture," *International Conference on Software Engineering*, Los Angeles, CA, 1999, pp. 555-563.
[13] B. Caprile and P. Tonella, "Nomen Est Omen: Analyzing the Language of Funtion Identifiers," *Sixth Working Conference on Reverse Engineering*, Atlanta, Georgia, 1999, pp. 112-122.
[14] M. Chase, Christey, S., Harris, D. and Yeh, A., "Managing Recovered Function and Structure of Legacy Software Components," *Working Conference on Reverse Engineering*, Hawaii, 1998, pp. 79-88.
[15] M. P. Chase, S. M. Christey, D. R. Harris, and A. S. Yeh, "Recovering Software Architecture from Multiple Source Code Analyses," *PASTE*, Montreal Canada, 1998, pp. 43-50.
[16] E. Chikofsky and J. Cross. "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, pp. 13-17, 1990.
[17] R. Clayton, S. Rugaber, L. Taylor, and L. Wills, "A Case Study of Domain-Based Program Understanding," *Workshop on Program Comprehension*, 1998, pp.
[18] P. Clements and L. Northrop, “Software Architecture: An Executive Overview,” Carnegie Mellon University, Technical Report CMU/SEI-96-TR-003, February 1996 1996.
[19] P. Clements and N. Weiderman, “Report on the Second International Workshop on Development and Evolution of Software Architectures for Product Families,” Carnegie Mellon University, Technical Report CMU/SEI-98-SR-003, 1998.
[20] R. Cole and P. Eklund. "Scalability in Formal Concept Analysis," *Computational Intelligence*, vol. 2, pp. 1-19, 1993.
[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill Book Company, 1994.
[22] S. Easterbrook and B. Nuseibeh. "Using Viewpoints for Inconsistency Management," *IEE Software Engineering Journal*, vol. November 1995, 1995.
[23] A. Egyed, “Automating Architectural View Integration in UML,” University of Southern California, Los Angeles, CA, Technical Report USCCSE-99-511, 1999.
[24] W. Eixelsberger, M. Kalan, M. Ogris, H. Beckman, B. Bellay, and H. Gall, "Recovery of Architectural Structure : A Case Study," *Proceedings : Development and Evolution of Software Architecture for Product Families*, Las Palmas de Gran Canaria, Spain, 1998, pp. 89-96.
[25] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, Second ed. New York: Addison-Wesley, 1994.

[26] R. Fiutem, M. Merlo, G. Antoniol, and P. Tonella, “Understanding the Architecture of Software Systems,” IRST, Povo, Italy, Technical Report 9510-06, October 1995 1995.
[27] R. Fiutem, P. Tonella, G. Antoniol, and E. Merlo, “A Cliche-Based Environment to Support Architectural Reverse Engineering,” IRST, Povo, Italy, Technical Report 9602-02, February 1996.
[28] P. Funk, A. Lewien, and G. Snelting, "Algorithms for Concept Lattice Decomposition and their Application," : Informatik-Bericht Nr. 95-09, 1995.
[29] D. Garlan and M. Shaw, “An Introduction to Software Architecture,” in *Advances in Software Engineering and Knowledge Engineering*, V. A. a. G. Tortora, Ed. Singapore: , World Scientific Publishing Company, 1993, pp. 1-39.
[30] J.-F. Girard and R. Koschke, "Finding Components in a Heirarchy of Modules: a Step towards Architectural Understanding," *Working Conference on Reverse Engineering*, 1997, pp. 58-65.
[31] G. Y. Guo, J. M. Altee, and R. Kazman, "A Software Architecture Reconstruction Method," *TC2 Working Conference on Software Architecture (WICSA1)*, San Antonio, TX, 1999, pp. 16-33.
[32] M. Himsolt, "GML: Graph Modeling Language," : University Passau, 1996.
[33] D. Jerding and S. Rugaber, "Using Visualization for Architectural Localization and Extraction," *Working Conference on Reverse Engineering*, 1997, pp.
[34] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-Based Analysis of Software Architecture," in *IEEE Software*, vol. 13, 1996, pp. 47-56.
[35] R. Kazman and M. Burth, "Assessing Architectural Complexity," *2nd Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '98)*, 1998, pp.
[36] R. Kazman and S. Carriere, "View Extraction and View Fusion in Architectural Understanding," *Fifth International Conference on Software Reuse*, 1998, pp.
[37] R. Kazman and S. J. Carriere, “Playing Detective: Reconstructing Software Architecture from Available Evidence,” Carnegie Mellon University, Technical Report CMU/SEI-97-TR-010, 1997 1997.
[38] K. Knight. "Unification: A Multidisciplinary Survey," *ACM Computing Surveys*, vol. 21, pp. 93-123, 1989.
[39] P. Kogut and P. Clements. "The Software Architecture Renaissance," *The Journal of Defense Software Engineering*, vol. 7, 1994.
[40] E. Koutsofis and S. C. North, "Drawing graphs with dot," : AT&T Bell Laboratories, 1996.
[41] R. Krikhaar, "Reverse Architecting Approach for Complex Systems," *International Conference on Software Maintenance*, 1997, pp.
[42] M. Krone and G. Snelting, "On the Inference of Configuration Structures from Source Code," *International Conference on Software Engineering (ICSE 16)*, 1994, pp. 49-57.
[43] P. Krutchen. "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, 1995.
[44] T. Lin and O. L., "FEPSS: A Flexible and Extensible Program Comprehension Support System," *Working Conference on Reverse Engineering*, Hawaii, 1998, pp. 40-49.
[45] D. Luckham, "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events," *DIMACS Partial Order Methods Workshop IV*, Princeton University, 1996, pp.
[46] S. Maxwell, *Linux Core Kernel Commentary*, 1st ed. Scottsdale, AZ: CoriolisOpen Press, 1999.
[47] N. C. Mendonca and J. Kramer, "Requirements for an Effective Architecture Recovery Framework," *SIGSOFT*, 1996, pp. 101-105.
[48] A. Michail and D. Notkin, "Accessing Software Libraries by Browsing Similar Classes, Functions, and Relationships," *21st International Conference on Software Engineering*, Los Angeles, 1999, pp. 463-472.
[49] J. Mitchell, “Chapter 11 Type Inferencing,” in *Foundations of Programming Languages*: MIT Press, 1998.
[50] G. Murphy, D. Notkin, and K. Sullivan. "Software Reflexion Models: Bridging the Gap between Source and High-Level Models," *ACM SIGSOFT*, vol. 1995, 1995.
[51] G. C. Murphy and D. Notkin. "Lightweight Lexical Source Model Extraction," *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 262-292, 1996.
[52] R. Necaise, "gmake: The GNU Make Utility,” 1996.
[53] M. C. Paulk, “A Capability Maturity Model for Software,” Software Engineering Institute SEI SEI-93-TR-24, 1993.

[54] D. Perry and A. Wolf. "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, pp. 40-52, 1992.
[55] M. Raitner, M. Forster, and A. Pick, "The Graph Template Library," 0.2.6 ed. University of Passat, 1999.
[56] S. Rugaber, "MORALE METHODOLOGY GUIDEBOOK: Methodology Guidebook for Synchronized Refinement," : Georgia Institute of Technology, 1998.
[57] G. Sander, "Visualization of Compiler Graphs," , 1995.
[58] M. Siff and T. Reps, "Identifying Modules Via Concept Analysis," *ICSM' 97: IEEE Conference on Software Maintenance*, Bari, Italy, 1997, pp. 170-179.
[59] S. E. Sim, C. L. A. Clarke, R. C. Holt, and A. Cox, "Browsing and Searching Software Architectures," *International Conference on Software Engineering (ICSE)*, Los Angeles, CA, 1999, pp. To Appear.
[60] R. Taylor, W. Tracz, and L. Coglianese. "Software Development Using Domain-Specific Software Architectures," *ACM Software Engineering Notes*, vol. 20, 1995.
[61] S. Tilley, "A Reverse Engineering Environment Framework," Carnegie Mellon University, Technical Report CMU/SEI-98-TR-005, 1998 1998.
[62] V. Tzerpos and R. C. Holt, "A Hybrid Process for Recovering Software Architecture," *CASCON*, Toronto, Canada, 1996, pp. 1-6.
[63] R. Waters, S. Rugaber, and G. Abowd, "Using the Architectural Synthesis Process to Analyze the ISVis System -- A Case Study," Georgia Institute of Technology, Technical Report GIT-CC-98-22, August 1998 1998.
[64] R. Wille, "Restructuring Lattice Theory: an approach based on hierarchies of concepts," in *Ordered Sets*, I. Rival, Ed. Dordrecht-Boston: Reidel, 1982, pp. 445-470.
[65] K. Wong, "Rigi User's Manual : Version 5.4.4," , 1998.
[66] S. Woods and Q. Yang, "The Program Understanding Problem: Analysis and a Heuristic Approach," *International Conference on Software Engineering (ICSE 18)*, 1996, pp. 6-15.
[67] A. Yeh, D. Harris, and M. Chase, "Manipulating Recovered Software Architectural Views," *19th International Conference on Software Engineering*, 1997, pp.
[68] J. A. Zachman, "A Framework for Information Systems Architecture," in *IBM System Journal*, vol. 24, 1987.
[69] P. Zave and M. Jackson. "Composition as Conjunction," *ACM Transactions on Software Engineering and Methodology*, vol. 2, pp. 379-411, 1993.

Appendix A : Alloy Description

```
//alloy model of the architectural synthesis processs
model sa {
   //*****************************DOMAIN*********************************
  //The major objects in our domain of interest are:
  //   elements : These are the things that make up an architecture
  //   representation: This is a presentation of the architecture in some format
  //   constraints : This is a description of any constraints that apply to the architecture
  //   attributes : These are things about elements that  help us differentiate them
 domain {elements, representation, constraints, attributes, viewpoint}

 //**********************STATE*********************************************
 state {
   //******************SETS*************************************

   //Architectural elements can be divided into components and connectors
   partition components, connectors : elements

   //The representation of the architecture may be a view or a perspective
   //  Views are the final representation of architectural information describing a viewpoint
   //  Perspectives are raw information about an architecture derived from a single source
   partition view, perspective : representation

   //There are four canonical views and one category for specialization
   // system : This is the high level systems view of which software physical  is a subset
   // module : A code-based view of the architecture
   // logical : The conceptual or functional architectural view ( a static picture of the architecture)
   // runtime : A view focused on threads/processes existing at runtime and the ipc between them
   partition module, system, runtime, logical : viewpoint

   //Additionally we have some common subsets
   //The software physical view is a subset of the system architecture
   physical : system

   //Some NFR views are subsets of the logical
   security, performance : logical

   //Within each view the components have special semantic meaning
   // process : A runtime process or thread, dll or COM/CORBA component
   // processor : A physical device capable of hosting a process component
   // code_module : A source code file, directory or package
   // logical_component : An active or passive functional component
   partition process, processor, code_module, logical_component : components

   //Within each view the connectors also have special semantic meaning
   // ipc : Runtime interprocess communication such as RPC, RMI, COM/CORBA calls
   // comm_path : A physical communications path between processors such as ethernet
   // dependency : A relationship between code modules.  One may contain another or need
   //                  another for compilation
   //logical_connector : A relationship between two logical_components
   partition ipc, comm_path, dependency, logical_connector : connectors

   //******************RELATIONS**********************************
    //Now define the state relations
   //runsOn maps processes to the processor on which they run (Maps runtime to physical view)
   runsOn(~hosts) : process -> processor

   //travelsOn maps ipc mechanisms between processes on different processors (Maps runtime to physical view)
   travelsOn(~supports) : ipc -> comm_path

   //implements maps code modules to the components they implement
   implements(~implementedBy) : code_module -> logical_component

   //views are built up from individual perspectives, thus a view is derived from a set of perspectives
   derivedFrom :  view -> perspective
```

```
    //These relations get information about elements in a particular representation
    getComponent(~getCompRep)  : representation! -> components
    getConnector(~getConnRep) : representation! -> connectors
    getConstraints : representation! -> constraints
    getAttributes : elements -> attributes
    getViewpoint : representation -> viewpoint!

    from, to : connectors -> components!
  }

  //*******************INVARIANTS***********************************
  //define that for certain views, only certain types of components/connectors exist
  //Views are constrained to have a single general semantic meaning
  ///If the views have mixed information, we didn't do union right
  inv Affinity {
    all r : view | r.getViewpoint in module <-> (r.getComponent in code_module and r.getConnector in dependency)
    all r : view | r.getViewpoint in runtime <-> (r.getComponent in process and r.getConnector in ipc)
    all r : view | r.getViewpoint in system <-> (r.getComponent in processor and r.getConnector in comm_path)
    all r : view | r.getViewpoint in logical <-> (r.getComponent in logical_component and r.getConnector in logical_connector)
  }

  //for connector, it must have a component at each end
  //and they all have to be in the same rep
  inv connectivity {
      all c : connectors | some m,n : components | c.from=m and c.to = n and c.getConnRep in m.getCompRep and c.getConnRep in
n.getCompRep
  }

 //*********************OPERATIONS*****************************************
  //for the union process we add the elements in one perspective to the base_rep
  op union ( baseRep : perspective!, per : perspective! ) {
    //details not important for fusion
  }
  op classify (rep:perspective!) {
    //details not important for fusion
  }
  //For the system and physical views to be consistent all the processors and
 //comm paths in the physical have to have a match in the system
  op check_sys_phys_consistent (p : view!, s : view!) {
      p.getViewpoint in physical and s.getViewpoint in system
      all pp:processor | some ps:processor | pp in p.getComponent and ps in s.getComponent  and pp = ps
      all cp:comm_path | some cs:comm_path | cp in p.getConnector and cs in s.getConnector and cp = cs
  }
  //For the runtime and physical views to be consistent there must be a processor for each
 //process and a process on each processor
 //ipc connectors however may run on either processor or comm_path
op check_runtime_phys_consistent(r:view!, p:view!) {
     r.getViewpoint in runtime and p.getViewpoint in physical
    all pr:process | some ps:processor | pr in r.getComponent and ps in p.getComponent and  pr.runsOn = ps
    all ps:processor | some pr:process | pr in r.getComponent and ps in p.getComponent and ps.hosts = pr
    all i :ipc | some ps:processor |some c:comm_path | i in r.getConnector and ps in p.getComponent and c in p.getConnector
         and  (i.runsOn=ps or i.travelsOn=c)
  }
  //check the logical and code module views for consistency
  //every code module should implement an element in logical view
  op check_logical_module(l:view!, m:view!) {
    l.getViewpoint in logical and m.getViewpoint in module
   all cm:code_module | some lc:logical_component | some le:logical_connector |
       cm in m.getComponent and lc in l.getComponent and le in l.getConnector and (cm.implements = lc or cm.implements=le)
 }
 //*********************CONDITIONS*****************************************
  //this condition tries to force the system not to choose the simplest (NULL) sets
  cond PopulateSchema {
     some view
     some perspective
     some connectors
       }
}
```

APPENDIX B: Glossary

Architecture: (P1471) The highest level conception of a system in its environment.

Legacy System: An existing system which has a significant software component. Frequently, the system is mission critical, has been in place for several years, and is poorly documented and understood. Usually, the original developers have long since left the company.

Perspective: A single representation of a software architecture that is derived from a single source. It can be thought of as a human (or machine-derived) opinion of what some part of the software architecture looks like. A perspective can be thought of as a set of incomplete information.

Representation: A description of a part of an architecture. This description may be textual in the form of an unstructured language description, but is often expressed in an architectural description language (ADL). Representations are most often graphical. The most common graphical notation, especially for legacy systems, is the “box-and-arrow” diagram.

View: A representation of a specific viewpoint. Views are composites of a set of one or more perspectives. A view can be thought of as having a sense of completeness about it. This is what differentiates a view from a classified perspective.

Viewpoint: A specific set of interests that concern a group of stakeholders. Viewpoints are expressed using one or more views.

Appendix C: Remora Data Models

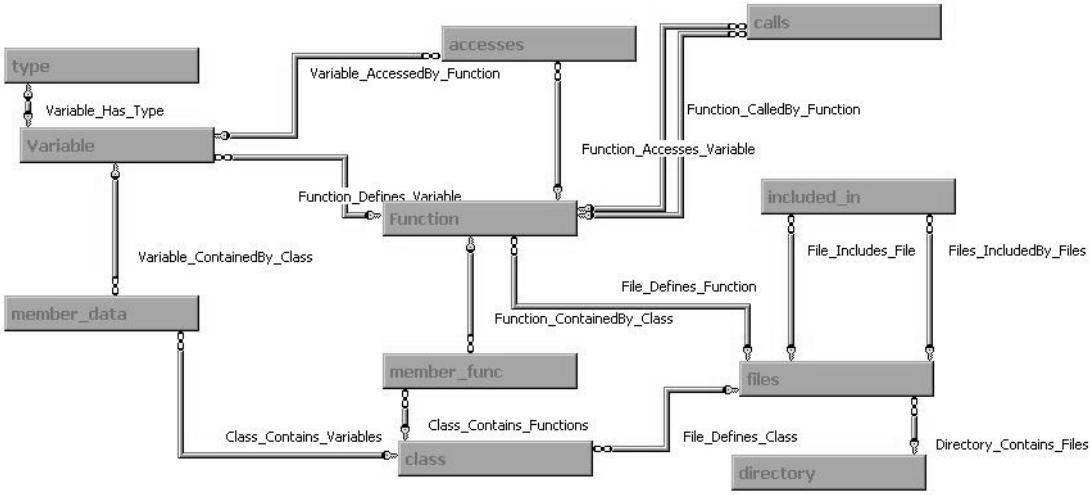

Figure C.1: Low-Level Data Model

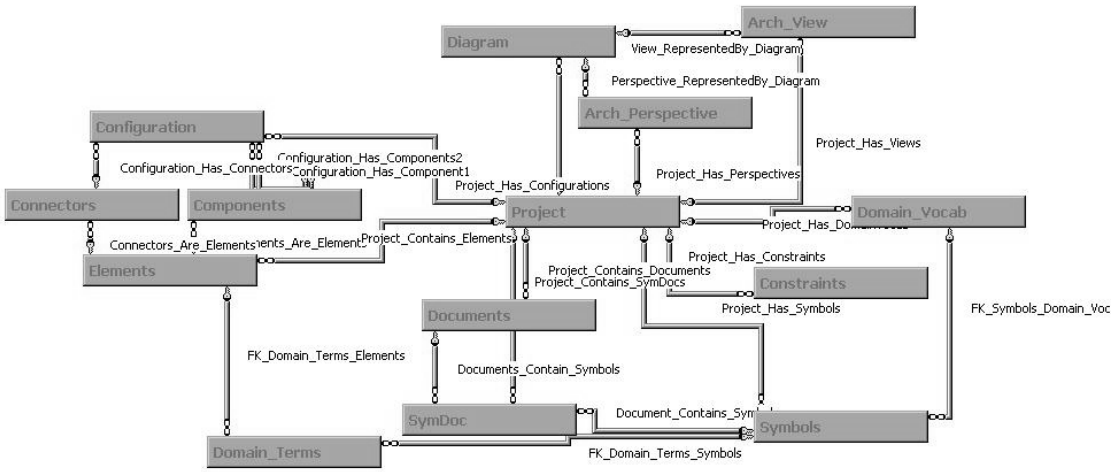

Figure C.2: High-Level Data Model

Appendix D : LINUX Perspectives

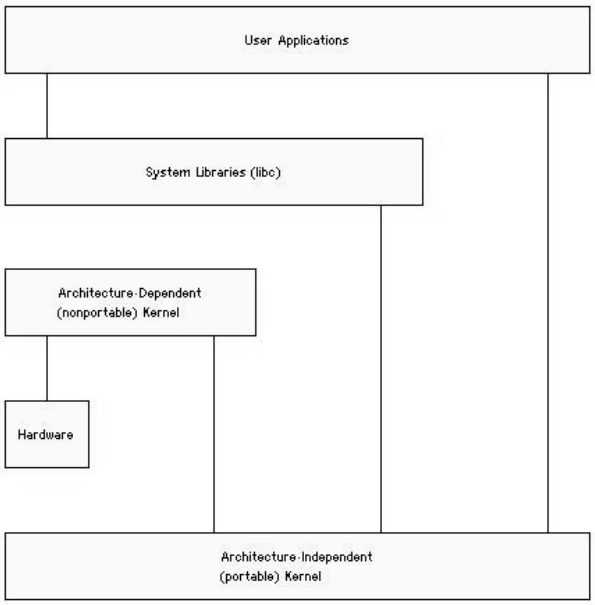

Figure D.1: A Domain Specific Architecture for Operating Systems

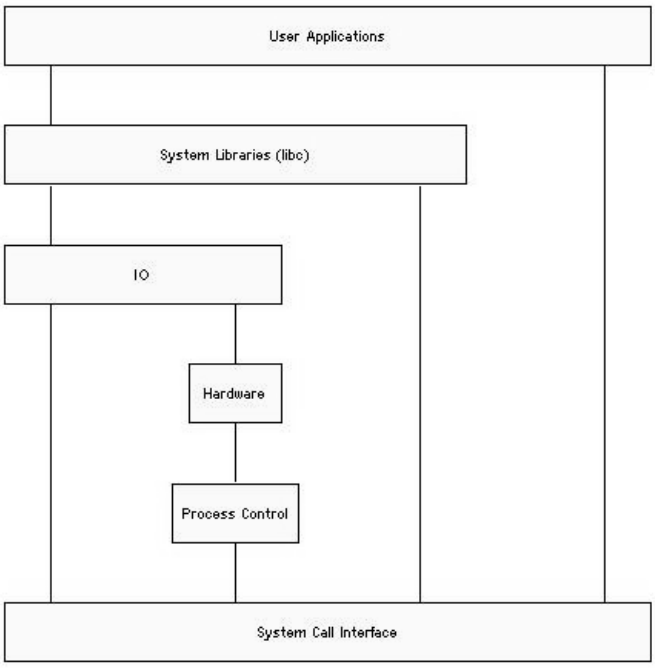

Figure D.2: Another Domain Specific Architecture for Operating Systems

Appendix E: Sample Viewpoint and View

The following viewpoint definition is taken from IEEE Draft Standard P1471 and represents a viewpoint description for the Air Force Command and Control Target Architecture(C2STA):

VIEWPOINT NAME: **Data**
Purpose: To establish how enterprise data is defined, organized, accessed and maintained.
Stakeholders: Data Administrators, producers, and composers of capabilities
Concerns: How is data accessed? How is data interpreted? What metadata is available about data?
Modeling Methods: data access capabilities, interfaces, data stores, data models, metadata, ownership and security, CRUD (Create, Read, Update and Delete) privileges, consistency information
Viewpoint Language: UML Class Diagrams, Entity-Relationship Diagram, CRUD Matrix

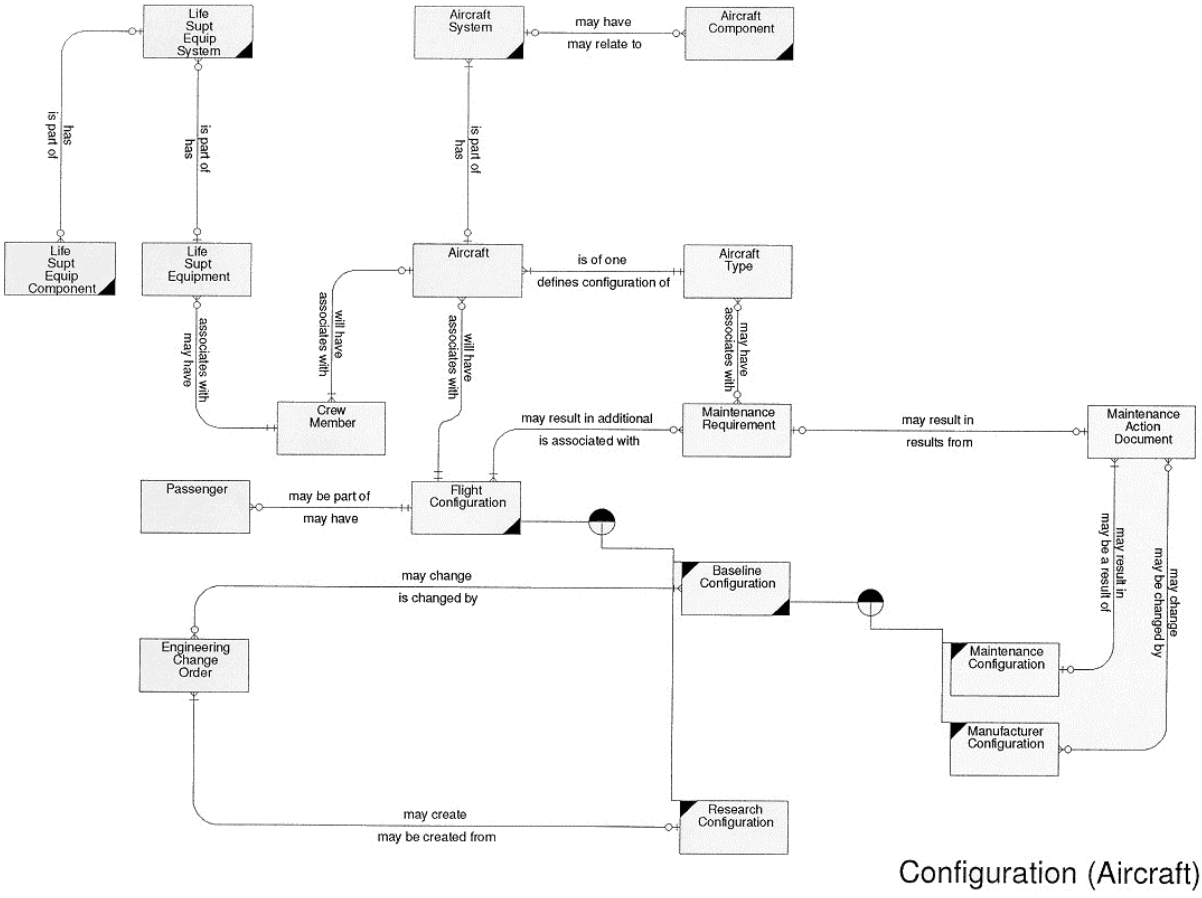

Figure E1: Sample View for the Data Viewpoint