# Architectural Synthesis: Integrating Multiple Architectural Perspectives

*Robert Waters and Gregory D. Abowd*
*College of Computing*
*Georgia Institute of Technology*
*{ watersr, abowd }@cc.gatech.edu*

## Abstract

*Many tools and techniques are available to help understand and analyze a system at the architectural level. Each of these provides its own perspective of the key architectural elements of the system. We introduce the process of architectural synthesis to deal with the problem of integrating these different architectural perspectives. The four steps of the synthesis process form an iterative cycle and include acquiring different perspectives (extraction), grouping related perspectives (classification), combining all perspectives that apply to the same architectural view (union), and finally composing views to determine consistency (fusion). We apply the synthesis process to the architecture of a software visualization tool. Our preliminary investigation shows that synthesized architectural perspectives provide a more complete and consistent representation of a real system. We conclude with directions for future research in this area.*

## 1. Introduction

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. To perform enhancement and evolution activities effectively, an analyst must recover high-level architectural information so as to understand the best way to make changes to the legacy system under study. Kazman et al. [19] aptly describe this process of recovery as "playing detective".

There is no single reverse engineering tool or method that can give an analyst all the information they need to describe the software architecture of an existing system. Each method or tool provides instead valuable clues about the structure or behavior of the system which must be pieced together to find the ultimate answer [19].

Over the last seven years, the study of software architecture has been a focal point of many research efforts [10]. Several different methods and tools have been developed which analyze different aspects of a software system from the standpoint of its architecture. Some of these approaches include architectural definition languages (ADLs) [22], simulation tools such as Rapide [25], and analysis methods such as the Software Architecture Analysis Method (SAAM) [17] and the Architecture Tradeoff Analysis Method (ATAM) [4]. Unfortunately, most of these techniques are focused on new development activities. This makes them difficult to use while reverse engineering since they require one or more accurate architectural representations to work with. For legacy systems, where do these representations come from?

Like Kazman's detective, we have plenty of clues, a few suspects, but no complete case to take to the district attorney. Just as a detective takes fingerprints, shell casings and witness interviews to make his case, we take call-graphs, design documents and developer interviews to make ours. As shown in Figure 1, there is a void between the arbitrarily many architectural perspectives of a legacy system and the refined inputs which most software architectural analysis tools desire. Our work looks at the broad problem of developing a consistent, complete and useful set of representations for a legacy system's software architecture to enable developers to evolve the system in an efficient and effective manner. The architectural synthesis process fills the void between the inputs required for forward engineering and architectural analysis and the outputs produced by reverse engineering tools and techniques by providing a general repeatable process with integrated tool support. Although this statement gives the impression the problem is one solely of tools and techniques, we recognize that the problem is actually one of architectural information management. An inexperienced analyst can be quickly overcome by the volumes of information at multiple levels of abstraction that can be produced by many of today's sophisticated tools. We want to aid the analyst in managing and processing this information by helping her separate the wheat from the chaff.

Since the vocabulary for software architecture has not yet been standardized, we first present the definitions of key terms. An overview of the architectural synthesis process and finally a discussion of our experience using the synthesis process to extract the architecture of a non-trivial software system follow the definitions and a summary of related work.

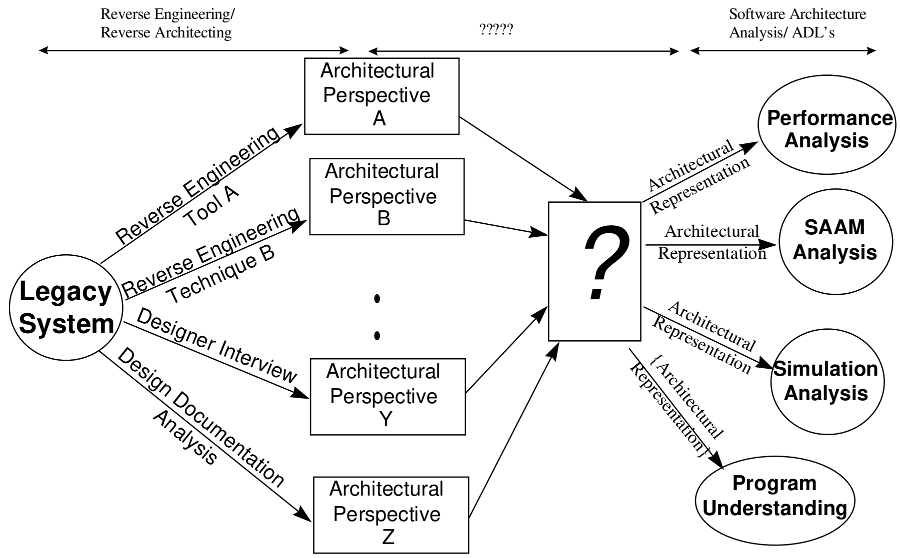


**Figure 1: The Multiple Perspective Problem**

## 2. Definitions

Before discussing the synthesis process in detail, we define the terms *architecture*, *representation*, *view* and *perspective*. These terms are used throughout software architecture literature, but with overloaded or conflicting meanings.

**Architecture.** When we discuss software architecture and the synthesis process, we are focusing primarily on the structural aspects of the architecture. We use the following definition from Bass, Clements and Kazman [5]: "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." The relationships are commonly called *connectors*. A similar definition can be found in ISO 10746 [15], defining an architecture as: "a set of rules to define the structure of a system and the interrelationships of its parts." So architecture is fundamentally information about the system at a high level of abstraction that describes how the system is designed to meet its functional and non-functional requirements. This information includes syntactic information such as component names and semantic information such as descriptions of component functionality and interfaces.

For generality, we refer to components and connectors collectively as *elements*. Any element may be further decomposed into another collection of elements that represent lower levels of abstraction. We refer to a decomposition of a component or connector as a *subsystem*.

**Representation.** Software architectures can be textually or graphically described. We refer to any of these descriptions as a *representation*. In our work we frequently use an attributed graph, where nodes denote components and edges denote connectors, as a common representation. If a representation is purely textual, we convert it to an attributed graph. We will present the attributes for these elements when discussing the union phase of the synthesis process.

**Perspective.** An architectural *perspective* is a representation of an architecture created directly by some method or tool. Perspectives vary in their degree of precision and include both unsubstantiated opinion, such as a simple box and arrow diagram drawn on a napkin, and precisely defined analyses, such as a call graph generated from source code. If we continue the detective analogy, perspectives are the individual clues on which we will build the case.

**View.** Researchers disagree about whether a single architectural representation can accurately portray a complete system. The existence of some ideal single representation is still an open question [10]. The current consensus is that a variety of representations is necessary to fully describe the architecture of any non-trivial system. We use the definition of *view* as presented by Clements and Northrup [9] to describe this set of representations. A *view* is a representation of an architecture that reflects a specific set of concerns that are of interest to a given group of stakeholders in the system. There is no generally accepted required set of representations that might be necessary to fully describe a system. Kruchten [24] proposes that such a set should include four different views. These include logical (focusing on software functionality), process (focusing on concurrency, fault-tolerance, and synchronization), module (focusing on division of the software into subsystems assigned to different programmers) and physical (focusing on assignment of functionality to specific hardware devices) views. Likewise Perry and Wolf [30] describe at least three separate views including processing (focusing on the flow of control through processing elements), data (focusing on the flow of data through the system) and connections (focusing on interactions between elements). For complex systems, there may even be views associated with specific non-functional requirements, such as a security view of the architecture. Our use of the phrase *architectural view* is consistent with all these uses. For any given software architecture there are potentially multiple views (some overlapping and some disjoint) that exist to describe it.

If we use an analogy from the requirements engineering world, it might make the perspective–view relationship clearer. When gathering requirements, we obtain raw information from different sources about what a specific requirement might be. These raw elements are like perspectives. By using different requirements analysis techniques we can refine this raw information into a final specification. These refined elements are equivalent to views.

## 3. Related Work

Several other research projects relate to the area of architectural recovery and development of multiple views of a system's architecture. These include DALI [18], CANTO [3] and MANSart [40]. Of these approaches, CANTO and MANSart are tied to specific extraction methods that focus on analysis of code-based artifacts to find architectural styles or patterns. DALI is more general in that it tries to incorporate multiple extraction methods into a tool framework—however it still relies primarily on code-based artifacts. Information extracted is placed into a database and then views can be built via queries created by the analyst. Recovery of design patterns [13, 40] by automated searching as well as classification of architectural components based upon matching their characteristics [21] has also been a focus of preliminary research.

Eixelsberger et al. [11] have a philosophy similar to ours, but with a different approach to realizing it. They recognize the need to use non-code-based information in the recovery process, but they emphasize recovering a single representation focused on a specific view of the system. For instance the safety architecture might be the focus of the extraction effort. While this single-aspect recovery has the advantage of requiring less time than our process, it does not attempt to ensure completeness or consistency of the produced view.

## 4. Architectural Synthesis

Architectural synthesis consists of a cycle of activities that integrates the raw information in perspectives into a synthesized set of views. The synthesized views form a complete and consistent architectural representation of the legacy system under study. Architectural synthesis is a semi-automatic task (shown graphically in Figure 2) consisting of the following four steps: Extraction, Classification, Union and Fusion. As we combine the various perspectives of the system, we create better and more complete representations. We may also find incompleteness and inconsistency that needs further resolution, thus the synthesis process is shown as a cycle. We do not claim that this process creates "the" software architecture, but rather that it creates a reasonably complete and consistent set of representations describing the architecture through various views of the system.

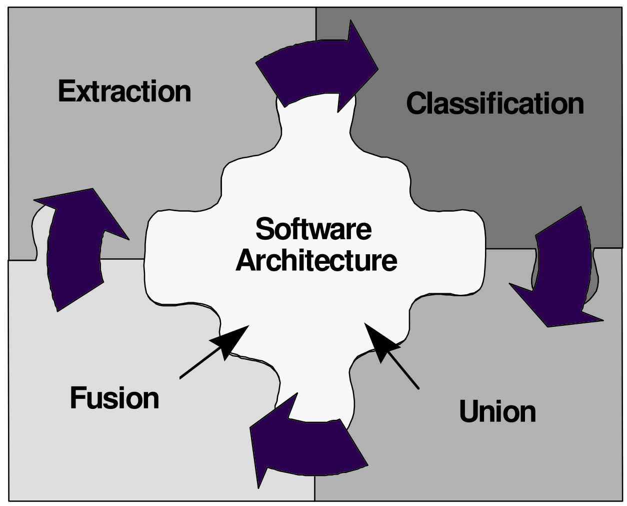

**Figure 2: The Architectural Synthesis Cycle**

### 4.1. Extraction

The first step in 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 [19], 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.

### 4.2. Classification

The next step is for the analyst to group perspectives into their respective views. This helps an analyst to focus initially on reconciliation of perspectives that are intended to describe the same aspects of a system. This is not necessarily a one-to-one function, as an individual perspective may map to two or more views.

We generally limit the classification step to the four major views (physical, logical, process and module). In larger systems classification may need to be expanded to include other views. For instance, the logical view might be separated into a data and control view if this is necessary to better understand a complex system. The main purpose of classification is to group perspectives in such a way that a common aspect of the system being analyzed relates the elements being combined.

### 4.3. Union

The union phase analyzes and combines all perspectives representing a specific view. During this step we manipulate a perspective as a graph where nodes denote components (boxes) and edges denote connectors (lines). Each element within a perspective has some set of attributes 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. General attributes consist of (attribute-value) pairs. Some common examples of these general attributes are shown in Table 1.

**Table 1: Sample Attribute/Value Pairs**

| Attribute | Values |
|---|---|
| Abstraction Level | *composite* or *atomic* |
| Behavior Type | *passive* or *active* |
| Component Type | *function* or *procedure* |
| Connector Type | *shared_memory, socket* or *file* |

While the previously described attributes focus on syntactic information about the architecture, we use domain terms to approximate semantic information about the various elements in an architecture. Domain terms are obtained by dowsing [8] the available system artifacts. Dowsing is a technique that scans textual artifacts such as source code, design documents and user's manuals and extracts frequently occurring key words or phrases (n-grams). A set of these n-grams is chosen by the analyst to represent the key terms in the domain. A subset of these terms forms the set of domain terms associated with each element. It is important to note that any duplication of terms (or synonyms) amongst the domain terms is handled during the dowsing process. The set of domain terms to be associated with each element are thus unique. Since architecture addresses how the system solves the problem within a specific domain, we feel the use of these domain terms is an adequate, lightweight approximation to the semantics of the various elements.

The union phase derives its name from 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 view into which we classified perspectives during the classification phase. We discuss this matching process in detail in the case-study section.

### 4.4. Fusion

We adopt this term from DALI [19] and use it to represent analysis across different views of the system. Fusion serves two purposes: first it provides a check on the consistency of the views [18, 19] and secondly, it provides additional information about the architecture through the composition of related views.

For instance, consider the fusion of the process and physical views as a mapping of runtime processes onto the hardware elements on which they run. Potential inconsistencies include processes that have no hardware or hardware that supports no processes. Inconsistencies might be caused by either a lack of information, i.e. the extraction phase failed to elicit enough information, or because there is an erroneous element in one of the representations. The following case study focuses on the union phase of the synthesis process.

## 5. Case Study

We present a case study of architectural synthesis which looks at evolving a medium-complexity system called ISVis (Interaction Scenario Visualizer) [16]. ISVis is a three-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. Many perspectives of ISVis were obtainable, but no single perspective 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.

### 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 paper, but all are available in the technical report [37]. The perspectives generated ranged from a generic, abstract reference architecture to a concrete, code-level call-graph.

**5.1.1. Domain-Specific (Reference) Software Architecture.** For many legacy systems, a Domain-Specific Software Architecture (DSSA) [35] or a reference architecture describing it 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 [36] and Rugaber's Synchronized Refinement process [31].

**5.1.2. DARE (Domain Analysis for Reverse Engineering) Model.** This model is derived from textual analysis created by using the DARE process [7]. 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 by dowsing. 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 [32] 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 terms included such words as *Disk File, Actor, Scenario, Utility, Source Code, Event, Trace, Visual, Mural* and *Static*. In all, the analyst selected 18 domain terms.

**5.1.3. ISVis Documented Architecture.** Figure 3 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.

**5.1.4. ISVis Derived Architecture.** Figure 4 presents 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 [16]. 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. The analyst first instrumented the ISVis source code and then executed 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.

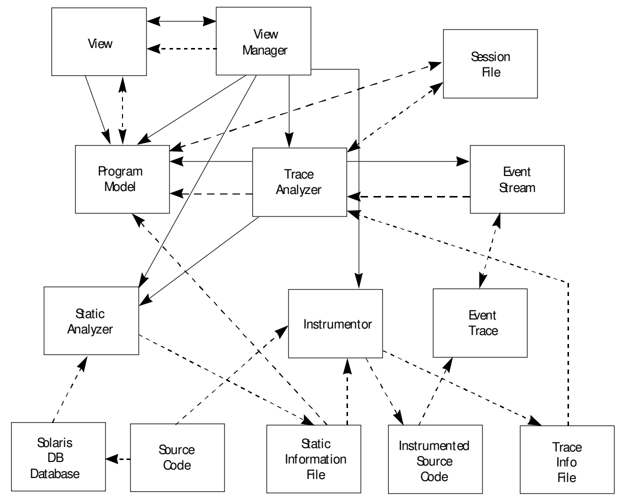


**Figure 3: ISVis Design**

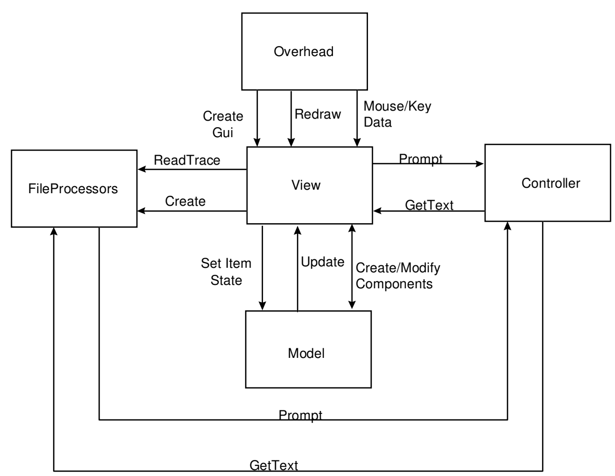


**Figure 4: ISVis Extracted Architecture**

**5.1.5. RMTool Representations.** RMTool perspectives were the output of the Murphy, Notkin and Sullivan Reflexion tool [28] applied to the ISVis source code. The input models used by RMTool were based upon the information derived from the ISVis analysis. The high-level model of components was based upon the abstract components 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 tool helped to provide a measure of the relative accuracy of the IS-

Vis-derived architecture and thus was a complementary perspective rather than one which 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 to produce *dot* [23], *gml* [14], *vcg* [33] and rigi [38] graphics files. This approach allowed us to use a wide variety of visualization tools to view and refine the graphs. The raw ISVis graph had over 800 nodes and well over one thousand edges. This information was used much as it was in DALI [18] to aggregate the call information into an architectural perspective.

**5.1.7. Make Analysis.** The *Makefiles* [29] 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 derived from the dependency analysis.

**5.1.8. Summary.** By the end of the generation phase, 13 perspectives had been generated. These perspectives are summarized in Table 2 (note that some lines of the table represent more than one perspective.) 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.

**Table 2. 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 |
| Designer/ Interview 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 |
| Make | 3 | 3 | 1 |
| Make Depend | 30 | 83 | 1 |

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

## 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. Primarily this was because ISVis runs on a single machine with three easily identified processes greatly reducing the complexity of the physical and process views.

Classification uses not only the content of a perspective, but also its source to determine the view to which it pertains. Some general (but certainly not all-inclusive) 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 dependency sections of the make file it is a module view.

At the end of the classification phase of the case study, there were 11 perspectives grouped into the logical view, and one each in the process and module views.

## 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. 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. [11] for example recommend the design documentation be used as the starting point. For this case study, we selected the ISVis-derived architecture of Figure 4. We did not chose the design perspective as recommended by other researchers because we felt in this case that the code-derived perspective was more accurate. This base representation is then extended by unioning it with the other perspectives in the logical view. We briefly discuss issues in the union of the design perspective (Figure 3) with the base representation in this section. More details can be found in the technical report [37].

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. Equality of lexical names is determined by either string equality or by determining the names are 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 e. $D_e \subseteq D$
- A: the full set of general attributes for the legacy system
- $A_e$: the set of attributes associated with the element e. $A_e \subseteq A$
- $V_a$: the value of the attribute associated with a specific element
- $\{(A_e, V_a)\}$: 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 view 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 $\mathrm{EXACT}(e1,e2) \equiv (N_{e1}=N_{e2}) \wedge (D_{e1}=D_{e2}) \wedge (\{(A_{e1}, V_a)\}=\{(A_{e2},V_a)\})$. EXACT is a symmetric relation. Name equality includes the idea of synonym comparison.
- SUBSUME: A node in one perspective is a more detailed description of a node in a different perspective and therefore is subsumed by that node. Thus $\mathrm{SUBSUME} \equiv ((N_{e1}=N_{e2}) \wedge (D_{e1} \subseteq D_{e2}) \wedge (\{(A_{e1},V_a)\} \subseteq \{(A_{e2}, V_a)\})$. 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. $\mathrm{OVERLAP}(e1, e2) \equiv (D_{e1} \cap D_{e2} \neq \emptyset) \wedge (D_{e1} - D_{e2} \neq \emptyset) \wedge (D_{e2} - D_{e1} \neq \emptyset)$. OVERLAP 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 in a different perspective. This indicates discovery of a new element and thus this node should be treated as a new node in the result. $\mathrm{NOREL}(e1,e2) \equiv (N_{e1} \neq N_{e2}) \vee ((D_{e1} \cap D_{e2}=\emptyset) \vee (\{(A_{e1},V_a)\} \cap \{(A_{e2},V_a)\}=\emptyset)$. If a node cannot satisfy the SUBSUME, OVERLAP, CONTAIN or EXACT relation with any other node in another perspective then it is treated as a new node.

We now give concrete examples of these relations as applied to the union of Figures 3 and 4. We initially select a node in Figure 3, *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 *Mode*, 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 5 shows the result view 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 or NOREL.

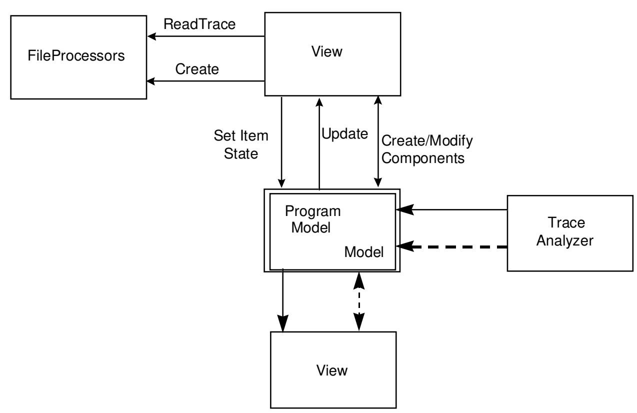


**Figure 5: EXACT match for Model, Resolving Edges**

We do 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. We choose NOREL since we cannot find a match for this element using any other relation and therefore assume this is a newly identified element. Later by using the perspective produced by the Reflexion Model, we find that the control connector is an error 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 Model 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 additional attributes, we might make an incorrect match. Using our relation rules, we find it is in the CONTAIN relation and therefore a subcomponent 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 subcomponents in the File Processors component via a CONTAIN relation. As shown in Figure 6, the analyst actually placed them at the top level rather than as a subcomponent 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 equivalent representations that might be developed. The important thing for the analyst is to develop a complete, consistent and useful set of views.

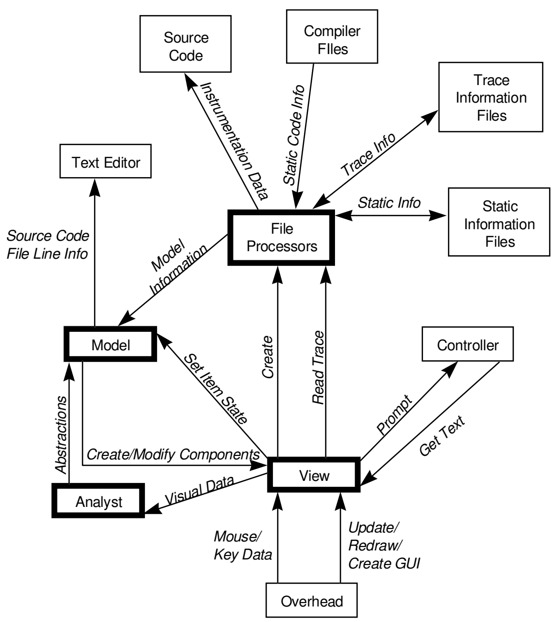


**Figure 6: Final ISVis Top-Level Logical View**

Figure 6 presents the final top-level logical systems view obtained 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 subsystem representation also! This occurs because the analyst has significant computation and data responsibilities within the ISVis architecture. For instance, from the DSSA, we know there exists both 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 style-related information comes from. If we had a code-extracted perspective alone this type of information would not be available.

Handling of connectors is one of the more difficult parts of the union process. The reasons for this are twofold. First, connectors are usually second-class citizens in the world of legacy architectures. As the ISVis design in Figure 3 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 tool. 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* (determining which 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. A call-graph perspective can also help with this task when developing the logical view.

At the end of the union process, the 11 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. The union of 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 [4]. 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.

## 6. Future Work And Conclusions

The case study described here was primarily conducted manually to develop ideas for what can and cannot potentially be automated and to refine the synthesis process itself. We believe the processes of union and fusion described in this paper can never be totally automated, but there are possibilities for performing many of these tasks in a semi-automated fashion.

Automating much of the union and fusion process requires that elements in different perspectives be matched and the EXACT, CONTAIN, SUBSUME, OVERLAP, and NOREL relations be determined for the analyst. These relations provide a mechanism for conveying matching information to an analyst. We are looking at four primary technologies to perform these actions.

The first is the application of type-inferencing [27], commonly used in compiler construction to suggest matches among the components and connectors that make up the architectural structure. If we consider elements to be types then the partial knowledge we have about their attributes at any stage of analysis is analogous to the partial knowledge a compiler has about a variable's type in languages without explicit type declarations. We have developed a prototype of this technique using the unification features of Prolog.

The second technique focuses on assisting with the lexical matching activities of the element names. Matching these names lexically is similar to traditional database schema integration activities [12]. If we treat components as entities and connectors as relations we can mirror some of the well-developed techniques already existent in the database world. Other metrics for disambiguating lexical names have been developed for assisting in reuse of code libraries by Michail and Notkin [26]. We are currently considering adapting these metrics to the synthesis process to assist the dowser in determining domain terms of extracted elements.

We have already mentioned that we use a graph to represent the different perspectives during analysis. There exist many algorithms for graph manipulation such as sub-graph isomorphism. These might be useful during perspective manipulation, but many are computationally expensive. Kazman has developed a technique called IAPR [20] that uses a constraint algorithm to limit the search space and allow graph matching to find architectural patterns. This technique holds promise to allow graph-level matching of perspectives.

Finally, the area of concept analysis has received much attention of late in reverse engineering [34]. If we use elements as objects and domain terms as attributes, we can build a concept lattice representing the perspectives under union. We are presently refining an algorithm that traverses the lattice and detects our five relations. The analyst then uses this information to combine the perspectives.

One potentially significant issue that did not arise in the case study, primarily because one person conducted the activities, is that of human conflict resolution. Clearly, when multiple analysts are involved, there will be human issues that must be resolved in addition to simply technical ones. We plan to incorporate some of the lessons learned by Win-Win [6] and other conflict-resolution strategies into the synthesis process.

To address these open issues, our future research will center on refining the architectural synthesis process and its supporting toolkit REMORA (Resolution of MORALE Architectures). REMORA provides a graphical environment where the different representations can be visualized and manipulated. Many of the lexical matching, overlaying and binding tasks can be done semi-automatically so that an analyst is free to concentrate on the difficult parts of the synthesis process that require human reasoning.

Performing architectural recovery and synthesis using a repeatable process helps analysts to produce usable architectural products for evolving legacy systems. Some form of automated support, however, is required to make the process feasible for large-scale industrial systems.

## Acknowledgements

This work supports MORALE (Mission-Oriented Architectural Legacy Systems Evolution) a part of the DARPA EDCS project and was sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-96-2-0229. We are also indebted to Rich Clayton and Dean Jerding whose work on ISVis and Dowsing was critical to the success of this study.

## 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 available at: http://www.paranoia.com/~vax/cflow/cflow.html.

[3] 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*, IEEE, 1997.

[4] 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.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*: Addison Wesley Longman, 1998.

[6] 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.

[7] R. Clayton, S. Rugaber, L. Taylor, and L. Wills, "A Case Study of Domain-Based Program Understanding," *Workshop on Program Comprehension*, 1998.

[8] R. Clayton, S. Rugaber, and L. Wills, "Dowsing: A Tool Framework for Domain-Oriented Browsing of Software Artifacts," *13th International Conference on Automated Software Engineering*, Honolulu, Hawaii, 1998.

[9] P. Clements and L. Northrop, "Software Architecture: An Executive Overview," Carnegie Mellon University, Technical Report CMU/SEI-96-TR-003, 1996.

[10] 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.

[11] 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, Springer-Verlag, 1998.

[12] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, Second ed. New York: Addison-Wesley, 1994.

[13] 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, 1996.

[14] M. Himsolt, "GML: Graph Modeling Language", available electronically from http://www.fmi.uni-passau.de/Graphlet GML/index.html, 1996.

[15] *ISO 10746, Basic Reference Model of Open Distributed Processing*, available electronically at http://archive.dstc.edu.au/AU/staff/kerry-raymond/rmodp/P0.html

[16] D. Jerding and S. Rugaber, "Using Visualization for Architectural Localization and Extraction," *Working Conference on Reverse Engineering*, 1997.

[17] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-Based Analysis of Software Architecture," *IEEE Software*, vol. 13, 1996, pp. 47-56.

[18] R. Kazman and S. Carriere, "View Extraction and View Fusion in Architectural Understanding," *Fifth International Conference on Software Reuse*, 1998.

[19] 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.

[20] R. Kazman and M. Burth, "Assessing Architectural Complexity," *2nd Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '98),* 1998

[21] R. Kazman, P. Clements, L. Bass, and G. Abowd, "Classifying Architectural Elements as a Foundation for Mechanism Matching," *COMPSAC*, Washington, DC, 1997.

[22] P. Kogut and P. Clements, "Features of Architecture Description Languages," *Software Technology Conference*, Salt Lake City, 1995.

[23] E. Koutsofis and S. C. North, "Drawing graphs with dot", available electronically from http://www.research.att.com/sw/tools/graphviz/refs.html, 1996.

[24] P. Krutchen, “The 4+1 View Model of Architecture,” *IEEE Software*, vol. 12, 1995.

[25] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann, “Specification and Analysis of System Architecture Using Rapide,” *IEEE Transactions on Software Engineering, Special Issue on Software Engineering*, vol. 21, pp. 336-355, 1995.

[26] A. Michail and D. Notkin, “Accessing Software Libraries by Browsing Similar Classes, Functions, and Relationships,” *21st International Conference on Software Engineering*, Los Angeles, 1998.

[27] J. Mitchell, “Chapter 11 Type Inferencing,” in *Foundations of Programming Languages*, MIT Press, 1998.

[28] G. Murphy, D. Notkin, and K. Sullivan, “Software Reflexion Models: Bridging the Gap between Source and High-Level Models,” *ACM SIGSOFT*, vol. 1995, 1995.

[29] R. Necaise, "gmake: The GNU Make Utility", available electronically from http://www.cs.wm.edu/~necaise/refs/unix/gmake.html, 1996.

[30] D. Perry and A. Wolf, “Foundations for the Study of Software Architecture,” i*ACM SIGSOFT Software Engineering Notes*, vol. 17, pp. 40-52, 1992.

[31] S. Rugaber, “MORALE METHODOLOGY GUIDEBOOK: Methodology Guidebook for Synchronized Refinement,” Georgia Institute of Technology, 1998.

[32] J. Rumbaugh, Blaha, M. , Premerlani, W., Eddy, F. , Lorensen, B., *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ, Prentice Hall, 1991.

[33] G. Sander, "Visualization of Compiler Graphs", available electronically from http://www.ca.uni-sb.de/RW/users/sander/html/gsvcg1.html, 1995.

[34] I. Schmitt and G. Saake, “Merging Inheritance Hierarchies for Schema Integration based on Concept Lattices,” Universitat Magdeburg, Magdeburg, Germany, Technical Report 1997.

[35] R. Taylor, W. Tracz, and L. Coglianese, “Software Development Using Domain-Specific Software Architectures,” *ACM Software Engineering Notes*, vol. 20, 1995.

[36] S. Tilley, “A Reverse Engineering Environment Framework,” Carnegie Mellon University, Technical Report CMU/SEI-98-TR-005, 1998.

[37] 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, 1998.

[38] K. Wong, "Rigi User's Manual : Version 5.4.4", available electronically from http://www.rigi.csc.uvic.ca/rigi/rigiframe1.shml, 1998.

[40] A. Yeh, D. Harris, and M. Chase, “Manipulating Recovered Software Architectural Views,” *19th International Conference on Software Engineering*, 1997.