Architectural Synthesis: Integrating Multiple Architectural Perspectives

Bob Waters **Spencer Rugaber** **Gregory Abowd**
College of Computing
Georgia Institute of Technology
{ watersr, spencer, abowd }@cc.gatech.edu

ABSTRACT
There are many tools and techniques available to understand and analyze a system at the architectural level. Each of these may provide 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. We emphasize the utility of the synthesis process in supporting the evolution of legacy software systems. The four steps of the synthesis process form a cycle and include acquiring different perspectives (generation), grouping common perspectives (classification), combining all perspectives that apply to the same view (union), and finally composing views to determine consistency (fusion). We give an example of the application of the synthesis process applied to the architecture of a software visualization tool. Our preliminary investigation shows that synthesized architectural perspectives provide a more complete representation of a real system. Use of the synthesis process uncovers inconsistencies pointing to areas needing further analysis or explanation. We conclude with directions for future research in this area and a description of the REMORA synthesis environment we are building to support architectural synthesis.

KEYWORDS
Software Architecture, Architectural Synthesis, Architectural Description, Reverse Engineering

1 INTRODUCTION

A newly appointed technical lead on the extension of a large company's software product is pondering where to begin the development effort. The requirements have been gathered and analyzed, and now the decision is made that doing an architectural analysis would be the best place to start the design phase. In the manual on architectural analysis, the first step is simply stated as "Generate an architecture." Lacking any other guidance, the technical lead peruses the existing design documentation and finds the original architecture which she brings to the first meeting. As soon as the viewgraph is placed on the screen, several developers begin talking at once, pointing out the changes over the years and arguing over what the architecture really looks like. The remainder of the meeting is wasted with bickering over what label should be in which box and where the lines between boxes should go. The technical lead sinks into her chair and thinks, "There must be a better way."

Over the last seven years, the study of software architecture has been a focal point of many research efforts[8]. Several different methodologies and tools have been developed which provide analysis of different aspects of a software system. Some of these approaches include architectural definition languages (ADLs)[18], simulation tools such as Rapide[21], and analysis methods such as SAAM[14] and ATAM[3]. Although these tools have diverse goals and different levels of formality, they have one thing in common—they all require a single architectural representation as input. When conducting new development (forward engineering), one or more representations are normally available as a by-product of development life cycle activities. For evolution of legacy systems (reengineering) accurate representations are not so readily available. There may be little or no documentation and the documentation that does exist probably does not resemble the current system due to drift and erosion[25]. Many of the architectural recovery techniques used in reverse engineering an existing system produce architectural representations which are incomplete. As shown in Figure 1, our work examines the broad problem of developing an effective representation of a legacy system's software architecture so developers can use that representation with other analysis tools to evolve legacy systems in an efficient and effective manner.

2 DEFINITIONS

Architecture - View - Perspective

Before discussing the synthesis process, it is necessary to establish the definitions of the terms architecture, view and perspective. These terms are used in one way or another in most of the software architecture literature, but often with overloaded or conflicting meanings.

When we discuss software architecture and the synthesis process, we are focusing primarily on the structural aspects of the architecture, therefore we use the following definition from Bass, Clements and Kazman[4]: "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 [connectors]."

The question of whether a single architectural description can accurately portray a complete system is still an open question[9]. A view addresses the need to present different aspects of an architecture in order to fully describe it. In our terminology, a *view* is a representation of an architecture used to describe a particular aspect of a system's structure or behavior and providing coverage of the entire system. There is no canonical list which represents a minimal set of views. Some authors such as Kruchten [20] describe such a set comprising five views. These include a logical (focusing on software functionality), process (focusing on concurrency, fault-tolerance, and synchronization), development (focusing on division of the software into subsystems assigned to different programmers) and physical (focusing on assignment of functionality to specific hardware devices) views, plus a set of scenarios to tie them together. Likewise Perry and Wolf [25] describe at least three separate views including processing (focusing on the flow of data through processing elements), data (focusing on the process flow of the system) and connections (focusing on interactions between elements). Our use of the phrase architectural view is consistent with these usages. For any given software architecture there are potentially several views (some overlapping and some disjoint) that may exist to fully describe it. For pure forward engineering work, the concepts of architecture and views are generally adequate—but this is not the case in the reverse engineering world. Here we also need the concept of perspectives.

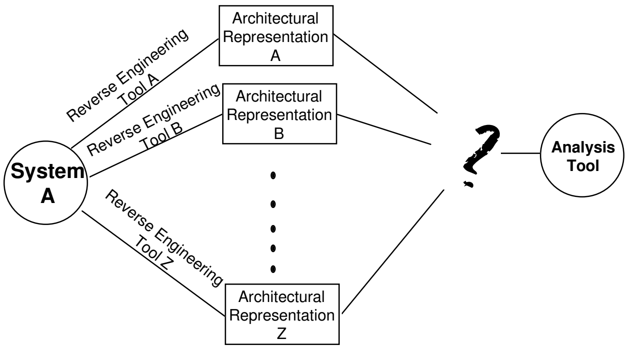

Figure 1: The Multiple Perspective Problem

An architectural *perspective* is a potentially incomplete representation of an architecture generated either by an automated tool or a human analyst and providing a subset of system coverage. Perspectives 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.

The architectural perspectives of a single system differ for a variety of reasons.

- Completeness: Perspectives may disagree as to the parts of the system or views they represent.
- Precision: Perspectives may differ as to the level of abstraction they provide.
- Aspect: Perspectives contribute to specific views of a system, such as its performance characteristics or how it provides reliable services. Consequently, they may ignore parts of the system not relevant to that view. Hence, different views may provide quite different coverages of a system.
- Version: Perspectives may represent a system as it evolves over time. Consequently, architectural perspectives go through versions, just as code does[28].
- Consistency: Perspectives are inherently approximations. Perspectives constructed by different people or different tools may disagree.
- Accuracy: Perspectives may be limited by the accuracy of the information used to construct them. This is especially true of manually-generated perspectives.
- Rigor: Perspectives differ in the degree and type of formality with which they express system characteristics.

Simply stated, an architecture may have from one to n views, with each view having potentially one to m perspectives associated with it. The essence of a view is the use to which it will be put, while the essence of a perspective is the mechanism used to generate it.

Architectural Synthesis

The architectural synthesis process consists of a cycle of activities that synthesize perspectives into a relevant set of views that together form a complete architectural representation. Synthesis is a human-centric task (shown graphically in Figure 2) consisting of the following four steps:

- Generation: The first and most obvious step 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 the legacy system's architecture.
- 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 usually a one-to-one

function, as the initial perspectives may overlap two or more views.

- Union: This step analyzes all perspectives that represent a specific view and develops a single view from them that incorporates structural information from all of them.
- Fusion: We adopt this term from DALI[16] and use it to represent analysis across multiple views to check for commonality, consistency and to create compositions of different views.

The remainder of the paper focuses on the practical application of the architectural synthesis process to a real system. By generating a typical set of architectural perspectives, uniting them into a coherent view, and fusing them for consistency, we show the overall utility of the process as a means for generating an accurate architectural representation suitable for use by other analysis techniques. We first compare the synthesis process to other work similar in concept.

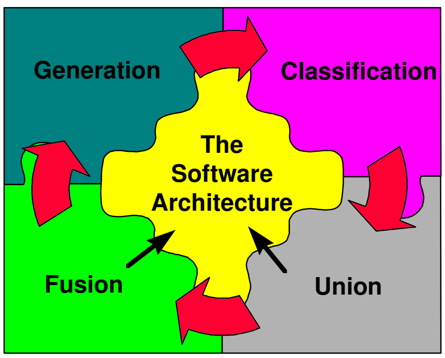

Figure 2: The Architectural Synthesis Process

3 RELATED WORK

Some work in architectural synthesis is already being conducted by other researchers. DALI [16][17] and ManSART[33] both look at synthesis activities for artifacts derived from the source code of a software system. DALI and ManSART overload the term view to be both what we call a view and what we term a perspective. DALI uses the term *view fusion* for what we call both union and fusion, while ManSART uses the term *view manipulation*. Our work is complementary to the work done in DALI and ManSART. In our work on legacy system evolution, we have architectural perspectives derived not only from source code, but from requirements engineering activities[26], program adaptation analysis[23], architectural analysis[14] and domain analysis[7]. These sources may create perspectives at different levels of abstraction. While pure source-code derived perspectives may at least have some naming consistency and comprehensive theme, we can expect no such help when resolving other more diverse perspectives.

The concept of multiple perspectives of requirements has been studied by the requirements engineering community for some time. Easterbrook and Nuseibeh[10] introduce the idea of resolution of conflicting requirements perspectives (which they call views) through the use of viewpoints. They define a *viewpoint* as "A loosely coupled, locally managed, distributable object which encapsulates partial knowledge about a system and its domain, specified in a particular, suitable representation scheme, and a partial knowledge of the process of development." Not coincidentally, this captures our idea of a perspective almost perfectly.

4 CASE STUDY

We present a case study of architectural synthesis which looks at evolving a medium-complexity system called ISVis (Interaction Scenario Visualizer)[13]. ISVis is a three-year-old C++ /X-Motif application consisting of 27 separate source files containing 23,333 lines of commented source code. Functionally, ISVis is a reverse engineering tool which can be used by an analyst to help abstract an architectural perspective based upon both the source code static information and a behavioral trace of program execution. (In this particular case, the analysis gives rise to the interesting situation where the legacy system is used to analyze itself for evolution).

We are in the process of preliminary design for the next version of ISVis and wish to perform an ATAM analysis. Like the technical lead in the introduction, we had many perspectives of ISVis available to us, but no one representation to use to begin the process. We decided to do an architectural synthesis to arrive at an accurate architectural representation to begin the ATAM session.

Generation

An analyst first generated several perspectives of the ISVis architecture which are described below. A subset of the representations actually generated are included in this paper, but all are available in the technical report[1]. This subset was chosen because it demonstrates the issues involved in performing the synthesis process. The perspectives generated range from a generic, abstract Domain-Specific Software Architecture to a concrete code-level call-graph.

Domain-Specific Software Architecture. For many legacy systems, a Domain-Specific Software Architecture (DSSA)[29] 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[2]. For this case study, a simple reference architecture for the reverse engineering domain, (the domain of the ISVis tool) was constructed (Figure 3) using Tilley's reverse engineering framework[30] and Rugaber's Synchronized Refinement process[7]. (Note that this architecture is used as a representative

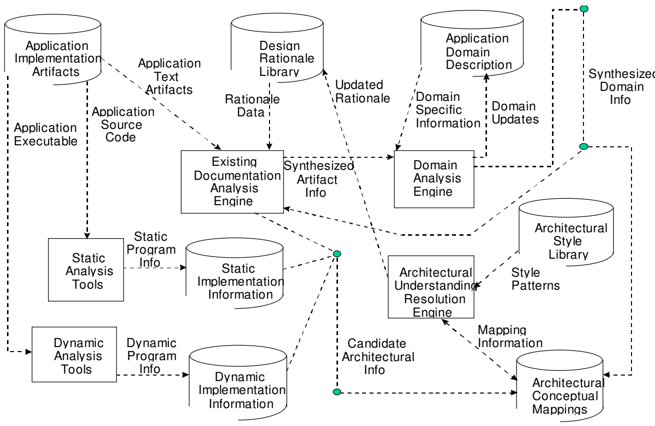

Figure 3: Domain Specific Architecture for ISVis

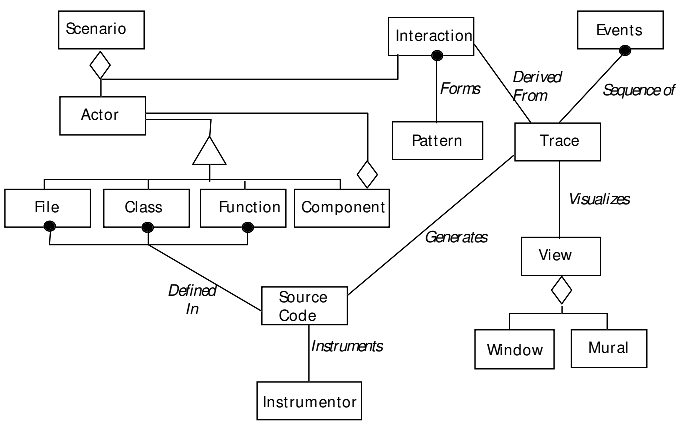

Figure 4 : DARE Domain Model

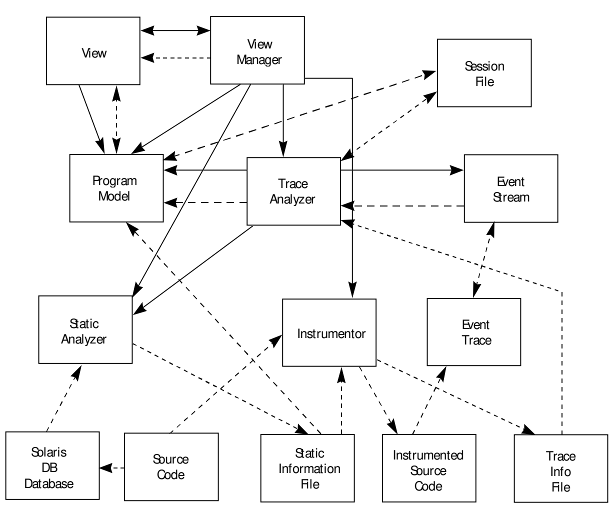

Figure 5 : ISVis Design

architecture for this level of abstraction and does not purport to be the definitive domain-specific

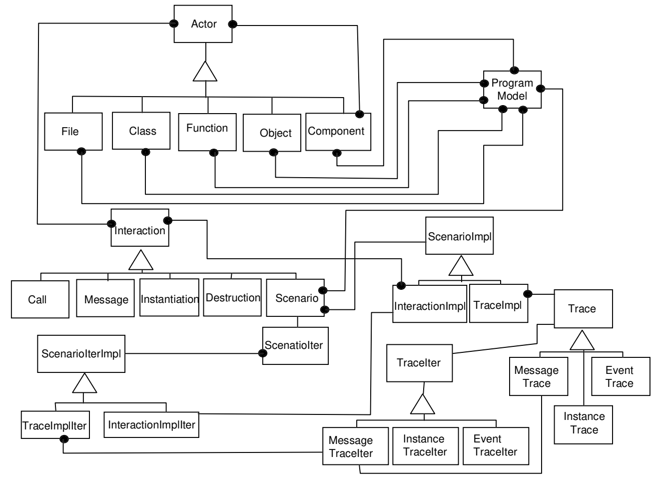

Figure 6 : ISVis Object Diagram

architecture for reverse engineering.) The DSSA portrays the active components (rectangles), passive components (cylinders) and connectors (dotted lines) that make up the reference architecture. Where connectors are used by more than one component, the small circles represent roles so multiple attachments can be described.

DARE (Domain Analysis for Reverse Engineering) Model. This model (Figure 4) is derived from textual analysis created by using the DARE process[6]. This gives another domain-oriented view derived directly from the ISVis documentation. The DARE tool first analyzed

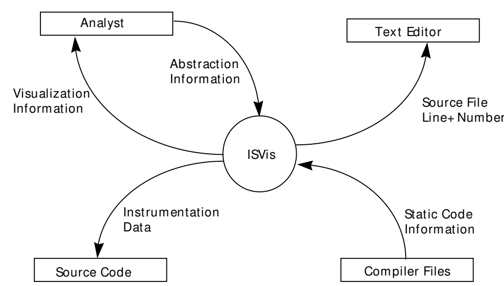

Figure 7: ISVis Context Diagram

the ISVis user's manual and tutorial extracting all unique words. 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.

ISVis Documented Architecture (Figures 5, 6 and 7). Figures 5 and 6 represent a part of the original developer's view of the architecture along with an OMT object model representation and are typical of the box and arrow diagrams available for most legacy systems. Figure 7 is also typical of many legacy system architectures and represents a context diagram showing ISVis' interaction with external entities.

ISVis Derived Architecture. This diagram (Figure 8) 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 [13]. The analyst first instrumented the ISVis source code and then performed two usage scenarios. From those two event traces, the analyst created abstracted components and scenarios which were manually translated into two different architectural representations.

RMTool Representations (Not Shown). These representations were the output of the Murphy, Notkin and Sullivan Reflexion model[24] applied to the ISVis source code. The high-level model of components used was identical to the one derived from the ISVis analysis. The mapping of classes and functions to components was based upon ISVis information as to the actors contained in the component. The Reflexion tool helped to provide a measure of the relative accuracy of the ISVis-derived architecture and thus was a complementary perspective.

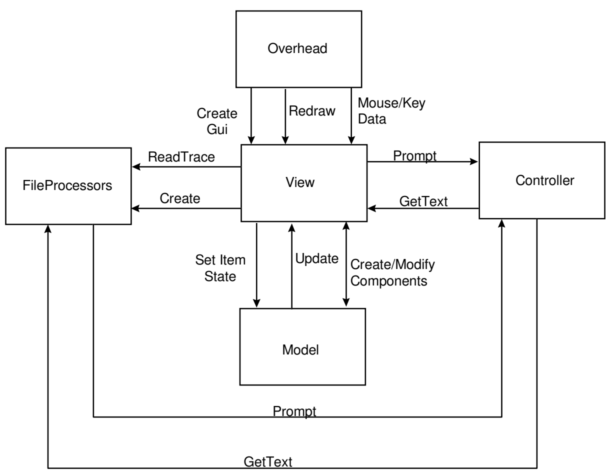

Figure 8: ISVis Extracted Architecture – Scenario 2

Call Graph (Not Shown). This type of diagram is a basic "who-calls-who" analysis typical of many reverse engineering static analysis tools such as cflow. To obtain the call graph, we wrote simple filter programs to act on the Solaris C++ compiler browser files and produce dot, gml, vcg and rigi 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[17] to aggregate the call information into an architectural perspective.

Summary. By the end of the generation phase, 12 perspectives had been generated. These perspectives are summarized in Table 1. 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 1
Generation 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

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

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. Even if all the perspectives map to the same view, there is still a useful function served by the classification process. It forces the analyst to begin thinking about the levels of abstraction represented by each perspective. Remember the principle purpose of the classification task is to avoid wasting time unioning mutually disjoint perspectives and arguing over whether a perspective properly belongs to one view or another. We developed the following guidelines to aid in the classification task:

- 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 the ISVis, RMTool or Call Graph diagrams, 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.

It may be the case that some perspectives contain pieces belonging in multiple views. In these cases an analyst can use a subset of the perspective during the union process for each view that the perspective involves. These types of perspectives frequently come from the expert-opinion generation effort and can be very valuable later during the fusion process because they provide information on how different views might be fused.

At the end of the classification phase of the case study, there were 11 perspectives grouped into the logical view, and one in the process view. From Table 1, only the make file analysis and interviews yielded a process view. All the others were classified in the logical view.

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 in the union process is to choose a perspective to act as the base representation for the remainder of the process. This base representation should be the perspective which represents the highest level of confidence and the highest level of abstraction for the view under consideration. For the case study, the ISVis-derived architecture of Figure 8 was selected. We rejected the basic ISVis design (Figure 5) because it mixed abstraction levels and the ISVis object design (Figure 6) because it did not address the user interface components—a strange design oversight since ISVis is a visualization tool! The RMTool components and mappings were generated from the ISVis-extracted architecture and thus were complementary to the representation chosen for the base. The context diagram (Figure 7) and domain-specific architecture (Figure 3) were too generically abstract to form the base representation, while the DARE model (Figure 4) and call-graph diagram (not shown) provided too low an abstraction level.

The base representation formed our initial top-level architectural logical view. This base representation was then extended by performing the following steps (these steps are reminiscent of set union, hence the name):

- Pick a perspective classified into the view undergoing the union process that has not yet been considered and look for a component with a name lexically identical to one in the base representation, or one which is a synonym in the domain of the application. An example of a domain synonym is the resolution of the Compiler Files component (Figure 7) with the more specific implementation-oriented name Solaris DB Database component (Figure 5) found in the original design document.
- Overlay the selected component with the one in the base representation. For example Program Model in Figure 5 overlays Model in the base representation (Figure 8). For each connector on the overlaying component, attempt to match it with an existing connector. For instance the control connections shown in Figure 5 overlaid the Update and Set Item State connectors between the Model and View, already in the base representation. If a connector cannot be matched, it is added to the representation as a new connector.
- If a component cannot be matched lexically, then determine if it is a subcomponent of one of the components in the current base representation. If it is, then add it to the subsystem representation of the containing component using the following procedure: If this is the first subcomponent, draw a bounding box to represent the subsystem boundary and place the new subcomponent inside. If this is not the first component in the subsystem, then simply add the component inside the bounding box. For example, the components Static Analyzer, Trace Analyzer and Instrumentor of Figure 5 were all subcomponents of the File Processors component of the base representation (Figure 8). Resolve the connectors so they are either part of the subsystem (that is, they run between components inside the bounding box) or they are bindings to the parent system (they run from a component inside the bounding box to a component at the next higher level of abstraction). As shown in Figure 10 for example, components at the next higher level of abstraction are shown outside the bounding box to make the bindings more explicit. For instance, Figure 5 shows a control connection from the Program Model to the Trace Analyzer. To resolve this we need a binding from the Trace Analyzer component up to the Model Component which also then requires a connector at the top-level between File Processors and Model. In the base representation of Figure 8, there is no such connection shown. We must add a connector which represents this control. We also need to label that

connector – and this can be a challenge. By looking at the other perspectives we determine that the connector represents Model Information.

- If a component is not a subcomponent then add it to the top-level representation. For example, because the ISVis analysis is based upon program traces of call data, it does not currently have the ability to analyze passive data accesses. Other perspectives that contain passive data components (such as the design document in Figure 5) must be used to complete the view to form a complete picture. These data components were not subsystems of any of the base representation components, but were themselves components at the same level of abstraction, and thus had to be added. This is a good example of how different perspectives complement each other and how their union is often necessary to get a complete representation of the architectural view.

Following this procedure for Figures 3-8 yields the top-

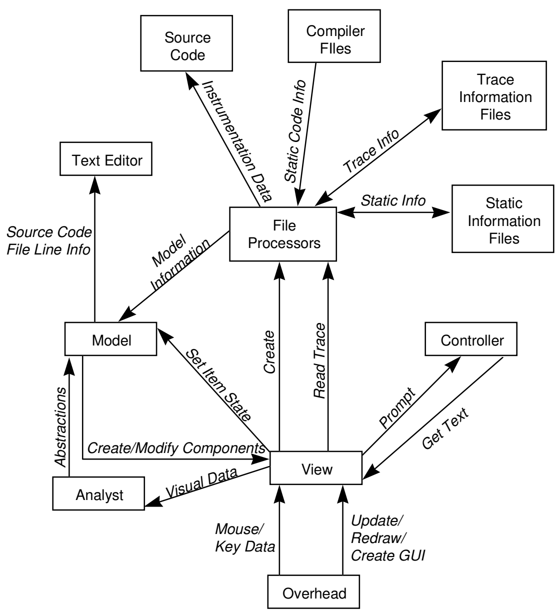

Figure 9: Final ISVis Top-Level Logical View

level logical systems view in Figure 9 and a representative subsystem view in Figure 10. 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 so 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 (Figure 3), 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 style-related information comes from. If we had a code-extracted perspective alone that information would be unavailable.

Another component-related decision made by the analyst was in which level of abstraction to place certain components when there was a choice. For instance, the information files shown in Figure 9 interact only with the File Processors component and could have been hidden inside that component to reduce the complexity of the top-level architecture. The analyst wanted to emphasize the data files during the ATAM analysis, so they were promoted to the top level. Under other circumstances, they could just as easily have been included inside the File Processors component.

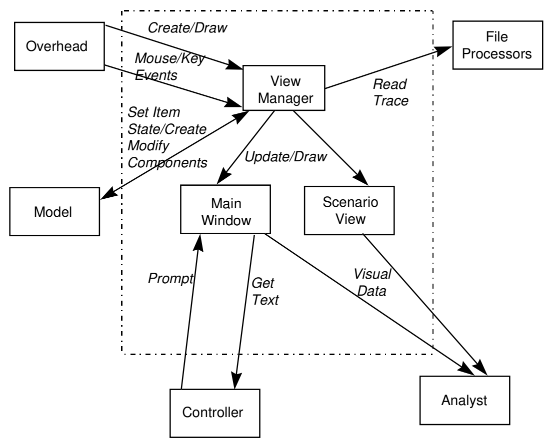

Figure 10 : Final View SubSystem

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 5 demonstrates, many documented legacy architectures do not even label the connectors. They may be annotated (as this one is) for differentiating control vs. data, but they have no precise meaning. Frequently they simply mean calls, uses, or talks-to. Promoting these ill-defined connectors to first-class elements of the architecture requires critical thinking and application 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 (Figure 3) 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 in Figure 7. The need for these types of complex transforms motivates our belief that the synthesis process can never be fully automated.

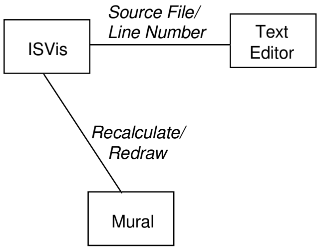

Figure 11
ISVis Process View

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 for 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 which provides the abstraction needed over sequences of program events. The call-graph perspective can also help with this task when developing the logical view.

At the end of the union process, the 12 perspectives representing the logical view--with their single level of abstraction, 758 potential components and 1601 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 three levels of abstraction that more accurately portrayed the actual ISVis architecture.

It is interesting to note that although the component count (disregarding the call-graph case) did not increase significantly, the connector count did. This is an indication that the union process aided in deriving an accurate architectural representation, and not just another box-and-arrow diagram.

The top level representation has only 10 components and 16 connectors, a configuration easily analyzed during an ATAM session[3]. If questions arise during the session, there are subsystem representations that clearly identify the functionality in each of the top-level components. Clearly this is better than using any of the initial perspectives by themselves.

There is still an open issue in the choice of the initial representation to form the base representation. Did the architecture fall into place because we were lucky and happened to pick a really good perspective to start with, or would the process have worked if we had picked some other perspective to start with? Does a code-based architectural abstraction tool like ISVis always provide the best base representation or would some other perspective have been a better base representation under different circumstances? We are looking into these questions as we continue to extend and refine the synthesis process.

Fusion

After individual views have been developed through the union process, the fusion process can begin. Fusion can occur between any two views that exist that are not disjoint. Frequently two views may overlap which can allow consistency checks to be performed. For instance ISVis has the logical view shown in Figure 9 and a process view shown in Figure 11. There are three concurrent processes: the ISVis main process, the Mural process and a text editor process. The text editor process is a COTS viewer for selecting source code and may be any available text editing program. Currently most people using ISVis use XEMACS. If we try to fuse the logical and process views, we can see matches between the Text Editor component of Figure 9 and the Text Editor process of Figure 11. Likewise the Mural process matches the Mural component (a part of the Scenario View component of Figure 11, but not shown in this paper). The ISVis process does not match directly a component, but by examining the make files and ISVis execution traces, the ISVis process includes the Overhead, Model, Controller, and other View components (less the Mural). We can check the connector information in the process view and evaluate its consistency with appropriate connectors in the logical view. In this case, the fusion process shows us that the views are consistent. We can match the connectors so that information flowing between processes in the process view is the same information flowing between components in the logical view.

In the event inconsistencies were detected, we would reenter the synthesis process to generate additional perspectives to help remove the inconsistency. This is why the synthesis process is depicted as a cycle.

In the case study, ISVis had only two required views and one of them (the process view) was trivial, so we were unable to do any extensive analysis of the fusion process and inconsistency detection between views. This is another area we intend to address in the next case study.

5 FUTURE WORK

The architectural synthesis process described here was conducted manually to develop ideas for what can and cannot potentially be automated and to refine the 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 some of these tasks in a semi-automated fashion.

One area for further research is the application of type-inferencing[22], commonly used in compiler construction to suggest matches among the components and connectors that make up the architectural structure. Many people have done research in recovery of design patterns[19][31] by automated searching as well as analysis of architectural components based upon matching their characteristics[15]. We intend to adapt these concepts to create pseudo-types which can allow automated matching activities to occur by using the type-inferencing previously mentioned.

One potentially significant issue that did not arise in the case study because the activities were conducted primarily by one person 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[5] and other conflict-resolution strategies into the synthesis process.

We also did not examine the role of architectural version management in this process. This is an area of interest to many software engineering practitioners. We intend to develop this area as an extension to the ACME interchange language[11] via the property mechanism. Some potential extensions can be extracted from Sommerville and Dean's PCL language[28].

To address these open issues, our future research will center around refining the architectural synthesis process and its supporting toolkit REMORA (Resolution of Mulitple Reverse Engineered Architectures). Not only will REMORA provide a graphical environment where perspectives can be synthesized, it will provide automated services for version management of architectures. Many of the lexical matching, overlaying and binding tasks can be done semi-automatically so that an analyst will be free to concentrate on the difficult parts of the synthesis process that require human reasoning.

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.

REFERENCES

1. Abowd, G., Rugaber, S., and Waters, R., "Using the Architectural Synthesis Process to Analyze the ISVis System – A Case Study", GT Technical Report GIT-CC-98-22, August 1998.

2. Architecture-Based Acquisition and Development of Software Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) Program. Technical report, Teknowledge Federal Systems, October 1994.

3. Barbacci, M., Carriere, S. J., Feiler, P., Kazman, R., Klein, M., Lipson, H., Longstaff, T. and Weinstock, C. "Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis", Technical Report CMU/SEI-97-TR-029, 1998.

4. Bass, L., Clements, P. and Kazman, R. Software Architecture in Practice, Addison-Wesley, 1998.

5. Boehm B., Bose, P., Horowitz, E., and Lee, M. J., "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach," Proceedings of the 17th International Conference on Software Engineering (ICSE-17), Seattle, April 1995.

6. Clayton, R., Rugaber, S., Taylor L. and Wills, L. "A Case Study of Domain-Based Program Understanding", Paper Presented to Workshop on Program Comprehension, 1998.

7. Clayton, R., Rugaber, S., Wills, L., "Dowsing: A Tool Framework for Domain-Oriented Browsing of Software Artifacts," in Proceedings of Automated Software Engineering, 13th IEEE International Conference, Honolulu, Hawaii, October 13-16 1998.

8. Clements, P. and Northrop, L. "Software Architecture: An Executive Overview", CMU Technical Report CMU/SEI-96-TR-003, February 1996.

9. Clements, P. and Weiderman, N. "Report on the Second International Workshop on Development and Evolution of Software Architectures for Product Families", Technical Report CMU/SEI-98-SR-003, 1998.

10. Easterbrook, S. and Nuseibeh, B. "Using Viewpoints for Inconsistency Management", *IEE Software Engineering Journal*, November 1995.

11. Garlan, D., Monroe, R. T., Wile, D., "ACME: An Architecture Description Interchange Language," *Proceedings of CASCON 97*, November 1997, pp. 169-183.

12. Garlan, D. and Shaw, M. "An Introduction to Software Architecture." In V. Ambriola and G. Tortora (ed.), Advances in Software Engineering and Knowledge Engineering , Series on Software Engineering and Knowledge Engineering, Vol 2, World Scientific Publishing Company 1993, pp.1-39.

13. Jerding, D. and Rugaber, S. "Using Visualization for Architectural Localization and Extraction", WCRE 97.

14. Kazman, R., Abowd, G., Bass, L. and Clements, P. "Scenario-Based Analysis of Software Architecture",

IEEE Software, 13(6), November 1996, pp. 47-56.

15. Kazman, R., Clements, P., Bass, L. and Abowd, G. "Classifying Architectural Elements as a Foundation for Mechanism Matching", Proceedings of COMPSAC, Washington, D.C., August 1997, pp. 14-17.

16. Kazman, R. and Carriere, S. J. "Playing Detective: Reconstructing Software Architecture from Available Evidence", CMU Technical Report CMU/SEI-97-TR-010, 1997.

17. Kazman, R. and Carriere, S. J. "View Extraction and View Fusion in Architectural Understanding", Paper submitted to Fifth International Conference on Software Reuse 1998.

18. Kogut, P. and Clements, P. "Features of Architecture Description Languages", Paper presented at Software Technology Conference, Salt Lake City, April 1995.

19. Kramer, C. and Prechelt, L. "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software", *Proceedings of WCRE 96*, pp. 208-215.

20. Krutchen, P. "The 4+1 View Model of Architecture", IEEE Software, 12 (6), 1995 pp. 42-50.

21. Luckham, D. "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events", Paper presented at DIMACS Partial Order Methods Workshop IV, Princeton University, July 1996.

22. Mitchell, J., "Type Inferencing," in chapter 11 of *Foundations of Programming Languages*, MIT Press, 1998.

23. Murdock, J. and Goel, A. "A Functional Modeling Architecture for Reflective Agents." AAAI-98 workshop on Functional Modeling and Teleological Reasoning, Madison, WI, July, 1998.

24. Murphy, G., Notkin, D. and Sullivan, K. "Software Reflexion Models: Bridging the Gap between Source and High-Level Models", ACM SIGSOFT, 1995.

25. Perry, D. and Wolf, A. "Foundations for the Study of Software Architecture*", ACM Software Engineering Notes*, 17(4), (1992).

26. Potts, C. "Scenario-Based Inquiry Cycle (ScenIC)", MORALE Methodology Guidebook, 1998.

27. Rugaber, S. "Program Comprehension", in Encyclopedia of Computer Science and Technology, 35(20), Marcel Dekker, Inc:New York, 341-368,1995.

28. Sommerville, I. and Dean, G. "PCL: a language for modeling evolving software architectures", *Software Engineering Journal*, March 1996, pp. 111-121.

29. Taylor, R., Tracz, W., Coglianese, L., "Software Development Using Domain-Specific Software Architectures," Software Engineering Notes, Vol 20, No. 5, December 1995

30. Tilley, S. "A Reverse Engineering Environment Framework", Technical Report CMU/SEI-98-TR-005, 1998

31. Tonella, P., Fiutem, R., Antoniol, G. and Merlo, E. "Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic – A Case Study", *Proceedings of WCRE 96*, 198-207.

32. Weiderman, N., Tilley, S. and Smith, D. "Approaches to Legacy System Evolution", Technical Report CMU/SEI-97-TR-014, 1997.

33. Yeh, A., Harris, D. and Chase, M. "Manipulating Recovered Software Architecture Views", Proceedings 19th International Conference on Software Engineering, May 1997.