View Extraction and View Fusion in Architectural Understanding

Rick Kazman, S. Jeromy Carrière
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

{kazman, sjc}@sei.cmu.edu

ABSTRACT

When performing architectural analysis on legacy systems, it is frequently necessary to extract the architecture of the system, because it has not been documented, or because its documentation is out of date. However, architectural information does not exist directly in the artifacts that we can extract. The architecture exists in abstractions; compositions of extracted information. Thus extracted artifacts must be able to be flexibly aggregated and combined. We call this process view refinement and fusion. This paper presents a workbench for architectural extraction called *Dali*, and shows how Dali supports flexible extraction and fusion of architectural information. Its use is described through two extended examples of architectural reconstruction.

Keywords: Software Architecture, Source Model Extraction, Architectural Views

EXTRACTING ARCHITECTURAL INFORMATION

It is frequently necessary to extract architectural information from source artifacts. Why? Because some systems have never had their architecture documented, or have only documented one view of the architecture, or the system's documentation has become out of date with respect to the implementation. Each of these cases poses a potential risk to the system's stakeholders, for if an architecture has not been properly documented, it cannot be analyzed and understood with confidence.

It has been shown that an architecture, in addition to serving as a vehicle for stakeholder communication, is the embodiment of the earliest—and hence most deeply ingrained and farthest reaching—design decisions in a system [1]. For this reason, it is crucially important to be able to analyze an architecture; that is, to assess the degree to which an architecture supports a desired set of quality attributes or scenarios [4]. However, this is a hopeless task if the as-implemented architecture does not match the as-designed architecture, or if the documented information does not include what is needed to support the analysis. As a consequence, we believe that, over time, maintaining architectural understanding by maintaining its documentation becomes just as important as having had an architecture in the first place.

How is this maintenance to be done? Typically it is too time consuming and error prone to do it by hand, so some tool support is necessary. However, tools alone cannot understand the abstractions through which humans structure and communicate their systems, so human intervention and interpretation of tool-extracted artifacts is crucial. This is because architectural mechanisms are seldom to be found directly in artifacts that developers develop, compilers compile, and tools extract. For example, there is no *layer* construct in a programming language; there is no simple language-level way of determining whether an element is a *client* or a *server*; and a *subsystem* is often determined by naming and calling conventions. But we regularly use all four of these abstractions in *defining* a software architecture.

Source Views Alone Are Insufficient

There are mechanisms—for example, modules or inheritance with accessibility constraints—that can *support* layering or subsystem encapsulation, but they can not *enforce* these abstractions. Thus the abstractions live in the designer's head, or in usage conventions or naming conventions. These, however, are typically not extractable by a tool: they are too idiosyncratic.

Furthermore, existing tools that promise to aid in recovering architectural information from source artifacts are insufficient. This is because they rely on a single source of information such as an AST (Abstract Syntax Tree) [12] or a source model [8]. Our claim, that we will substantiate through examples in this paper, is that the information derivable from any single extraction technique is insufficient for general architecture recovery and reconstruction.

Why? For two reasons: the use of *late binding* in programming languages, and the need to extract *system topology information*. The term late binding refers to important architectural relations that are not bound at compile time. Late binding is found in many places in complex software systems, such as: polymorphism, function pointers and parameters provided by the user at invocation or run time. In each case, the structure and complete function of the system can not be determined until system initialization or run time. The other reason for needing multiple extraction techniques is for the capture of system topology information. This includes relations such as allocation of software to processes or to processors. These relations might be specified when a system is built or when it is running, but they are typically *not*

specified in any source artifact that is compiled. Thus, a source view alone is insufficient, and a dynamic view alone is insufficient. We need multiple kinds of extraction to support disparate information needs, and we need to be able to fuse these views, as will be discussed.

Fortunately, late binding and topology information can be recovered by examining artifacts other than just an AST or source model. But this is more than just extracting information from other sources like execution traces and build files.

View Fusion

To create a complete view of the architecture requires that disparate views be *fused*. Fusing architectural views means establishing connections between them. This is important for several reasons:

- Different views provide complementary information. For example, a class hierarchy view describes information sharing relationships among classes, whereas a process view indicates potential parallelism and resource contention.
- Users need to be able to navigate among views to completely understand an architecture (to know what classes, in the class hierarchy view comprise what functions, in a derived functional view, for example).
- One view can be improved with information from another. Murphy *et al* have shown that source extraction can be error prone [7]. One way of improving the quality of extracted information is by cross-checking it with information from other views, as we will show.

This paper describes our approach to view extraction and fusion using *Dali*, a workbench for architectural extraction, manipulation, and conformance testing [6]. In particular, we concentrate on how view fusion allows us to capture architectural information that we could not have acquired from a single view, or from a single extraction technique. We illustrate these points using experience gleaned from extracting software architectures of systems written in C and C++.

INTRODUCTION TO DALI

The view extraction and fusion discussed in this paper is performed in the context of Dali. Dali is a workbench focused on easing the integration of a wide variety of extraction, manipulations, analysis and presentation tools, as discussed in [6]. Dali's architecture is shown in Figure 1. There are two important points to stress about Dali: it is open; and it incorporates view extraction, manipulation, and fusion as the cornerstones of architectural understanding.

- openness: Dali is a *workbench*, not a single tool. At its core is a repository that holds architectural information as a set of relations. All tools incorporated into Dali can add to or extract information from the repository, manipulate this information, or modify it. The various tools that live in the workbench extract architectural information from source artifacts, visualize this information to a user for viewing or manipulation, organize the information, or analyze it and provide reports to a user. Because the only communication path for the tools is through the database, it is easy to add news ones, as the need arises.
- recognition of fusion: the raw materials for creating architectural views in Dali are extracted from source artifacts. These are called *extracted views*. However, these views are primitive, consisting of two source elements related by a single relation (e.g. function1 calls function2). To create views that are more representative of an architect's understanding, extracted views can be manipulated, creating *refined views*. However, these manipulations still reflect the fruits of only a single form of extraction. To reflect a more complete, holistic understanding of an architecture, it is necessary to create *fused views*, which result when elements of two or more extracted views are combined together. Describing the reasons for creating and fusing views occupies the bulk of this paper.

HOW DALI WORKS

Dali supports a model of interpretive, interactive architecture reconstruction. Reconstruction is performed by a person familiar with the system, who iteratively manipulates architectural views into a desired form. This process provides a mechanism for the application of an *interpretation* of the system's architecture. The interpretation reflects how those involved perceive and understand the system. This is a key point: no system has *an* architecture; it has many potential views of its many structures, each of which is appropriate for different analysis activities.

Figure 1 depicts the components and activities of the Dali workbench, as follows.

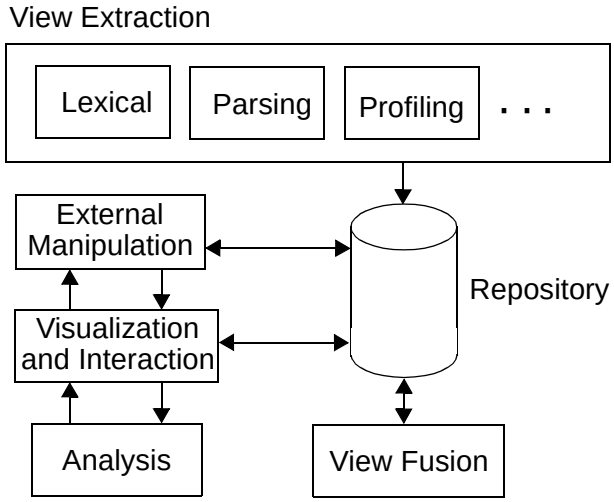

Figure 1: The Dali workbench

View Extraction

The first step in the reconstruction process is the extraction of static and dynamic elements from the system under examination. Examples of static elements are inheritance hierarchies, build dependencies, call graphs, variable accesses, and so forth. In short, static elements can be identified by exam-

ining source artifacts as they reside in a file system.

Dynamic elements include process spawning, instances of interprocess communication (IPC) and run time procedure invocation. It is important to note that there will be overlap between static and dynamic information; for example, a system's run time procedure invocation is obviously related to its static call graph. In Dali, extraction may apply any number of techniques and tools, such as parsing, lexical analysis, profiling and build-dependency analysis.

The results of extraction—the *extracted views*—are stored in the repository, currently an SQL database.

View Fusion

Once a collection of elements have been extracted, *fused views* may be defined over the extracted elements. For example, we may want to fuse an extracted view such as a static "calls" relation and a file/function containment relation to create a fused view representing calling relationships at the file level.

View fusion is the process by which views are defined and manipulated; it is discussed in depth below.

Visualization and Interaction

Dali is focused on aiding a user in reconstructing software architectures, and so effective visualization and user interaction facilities are crucial. Dali's user interaction is provided by Rigi [11], which we have found to provide an acceptable balance between generality (via a user programmable command language, RCL) and domain applicability. Rigi also provides mechanisms for graph layout, annotation and direct manipulation of the elements of a view.[1]

The user's primary task in interacting with and manipulating architectural structures is to organize extracted information into views that are more meaningful to them (i.e. more abstract, and more representative of their architectural understanding of the system). For example, a user might group all windowing system calls together into a "user interface" layer. The results of this process are the creation of *refined views*.

External Manipulation

Dali's goal of openness is facilitated by the programmable nature of Rigi. To integrate other tools into the Dali workbench, appropriate "glue" code is implemented in RCL. One example is Dali's primary facility for model manipulation, which is based on the execution of queries over the SQL repository. A query is performed, the results are processed externally using script-based tools, the output is interpreted by RCL code and applied to the model. These queries provide a pattern-matching facility which is central to Dali's interpretive approach. The result of the application of a pattern-matching query is a refined view that reflects the analyst's understanding of the realization of architectural constructs within the system. An example is a pattern that determines which classes have member functions that make calls to windowing system primitives, and identifying these classes as a "display" layer. Of course, this is not a universally applicable example: it is a part of a particular interpretation of the system's architecture.

Analysis

Closely related to the techniques for external manipulation are those for analysis. Many tools exist for performing architectural analyses, such as conformance testing and pattern-based complexity measurement. To leverage these tools, Dali applies an import/export model: the existing Rigi model is exported, the appropriate tool is applied and the results are visualized, either in Rigi or using other external tools.

USING DALI

To get a better understanding of the process of using Dali, consider Figure 2, which shows the raw extracted views for a 50 KLOC C++ application called VANISH (this application is discussed in more detail below). The figure displays the files, functions, classes and objects of the system, along with relationships between the elements (such as "file contains function", "class defines function" and "function calls function"). This view is obviously unsuitable for human consumption. However, using a series of view fusions, pattern applications and direct manipulations, the high-level architecture for the system was reconstructed.

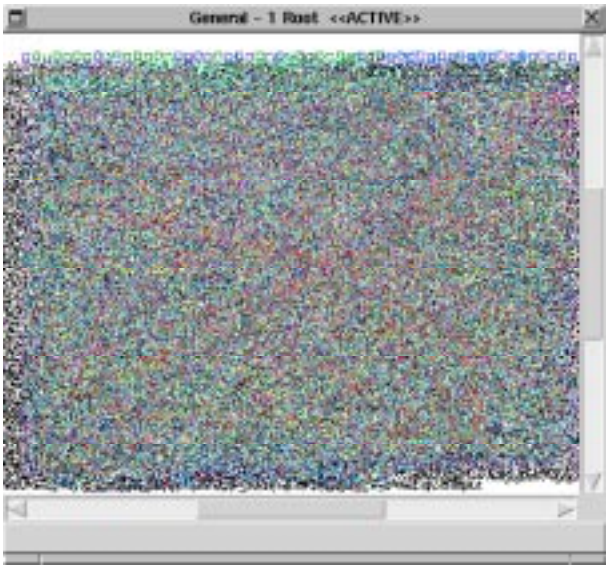

Figure 2: Raw extracted views

The view fusions applied to VANISH, and how they combine to transform the raw data of Figure 2 into a *usable* model of the architecture are discussed in the following section.

1.In this paper, images captured from Rigi will be shown with their window borders. Also note that an arc in Rigi is drawn *from* the bottom of a node *to* the top of a node.

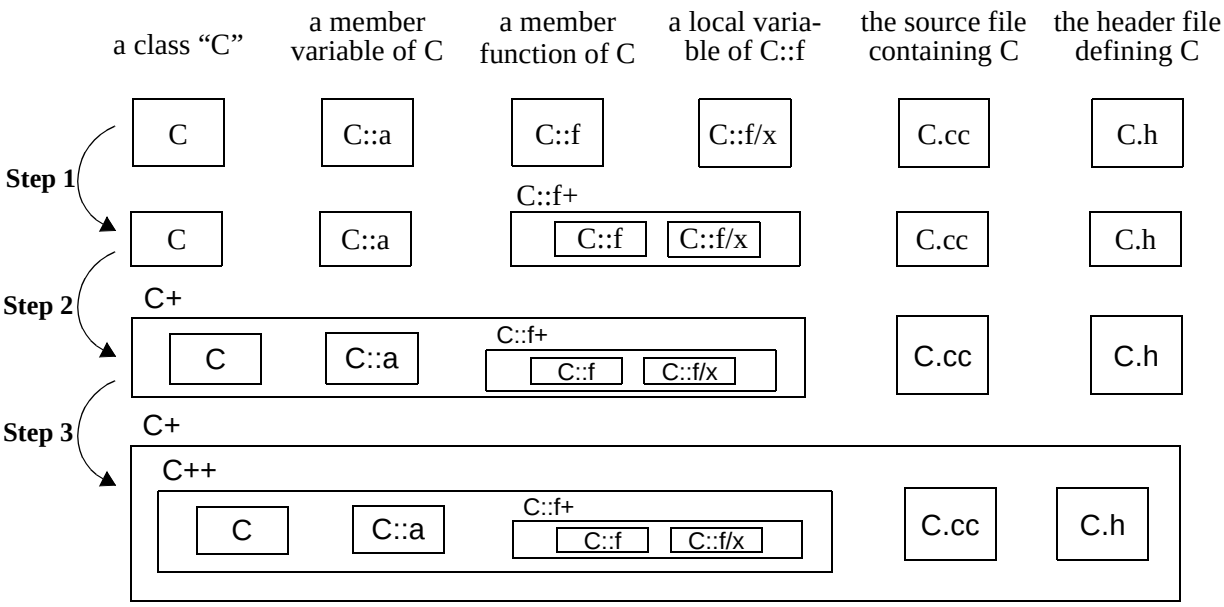

Figure 3: The application of aggregation patterns

The first thing that transforms raw extracted data into a suitable architectural view is to “tame” the complexity through *aggregation*. Figure 3 shows how the elements of the raw extracted views, shown in the top row, are aggregated to generate higher-level (more abstract) components. The patterns perform the following transformations:

Step 1: aggregate functions (e.g. C::f) with their local variables (e.g. C::f/x)

Step 2: aggregate classes (e.g. C), their member variables (e.g. C::a), and their member functions (e.g. C::f+)

Step 3: aggregate classes (e.g. C+) and the files in which they are declared and defined (e.g. C.cc and C.h)

Figure 4: VANISH after application of aggregations

These patterns are language-specific but application-independent. In conjunction with an additional application-specific pattern Figure 4 is generated from the raw data in Figure 2. The application-specific pattern identifies the “Graphics” component as an architectural structure: it comprises all functions known to be windowing system primitives: i.e. those called by subclasses of the Presentation class, plus those function names beginning with X, fl_ or gl_ .

The final step in the reconstruction of VANISH’s software architecture of is the application of another application-specific pattern that identifies the further aggregation of classes into architectural components**.** The result is shown in Figure 5. This representation, which corresponds to the documented architecture of VANISH, is the result of repeated view refinements and fusions. The means of doing these fusions are discussed next.

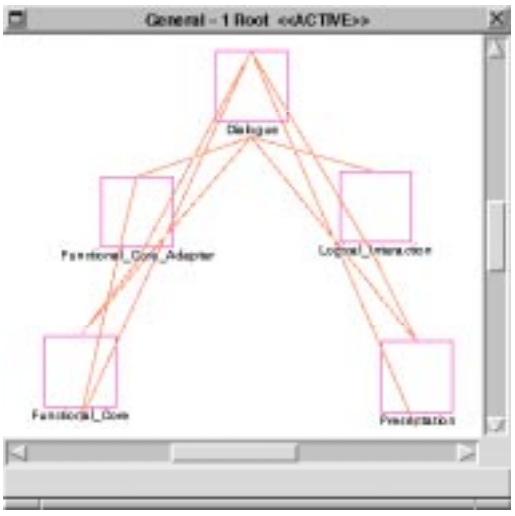

Figure 5: The VANISH software architecture

VIEW EXTRACTION AND FUSION

In Dali, views can be extracted using a wide variety of techniques. These techniques include lexical analysis tools such as LSME [8], parsers (e.g. Imagix and SNiFF+) and profilers (e.g. gprof). We can also extract views from utilities such as *make* and through direct instrumentation of the system under study, as we will show.

We maintain that a combination of these techniques is *necessary* for the process of architecture reconstruction. For example, the static information derivable from a parser can not identify IPC in a system that has no fixed topology until run time. But examining only the dynamic elements of such a system will also provide insufficient information, since constructs such as inheritance hierarchies and file/function containment are hidden or absent in a compiled and linked application.

Once a collection of views has been extracted from a system, by whatever means, a necessary next step in the architecture reconstruction process is an appropriate reconciliation of the views. This step is required because the information from different extractors frequently overlaps, depicting different aspects of the same elements of a system. For example, a parser and a profiler both produce information concerning functions and calls between them, but the former identifies *potential calls* while the latter identifies *actual calls*.

To facilitate exposition, we will present two systems to which the reconstruction process has been applied using the Dali workbench. The first is VANISH, a C++ application for prototyping visualizations [5], and the second is Serpent, a user-interface management system comprising approximately 80 KLOC of C [2]. Because the architecture reconstruction process is an interpretive one, a person with expert knowledge of the system under study plays an important role in reconstruction. In the case of VANISH, the experts were the authors of this paper; in the case of Serpent, a key development team member was available for consultation.

The extraction techniques applied to the two systems are as follows. For VANISH, LSME patterns were developed to extract a view comprising several relations, including “function calls function”, “class inherits from class” and “class defines member variable”. The gprof profiling tool was used to extract a view capturing run time function call information. In addition, scripts were written to extract a “file depends on file” relation from make input files (*makefiles*).

For Serpent, CIA [3] was used to extract a “function calls function” view, LSME was used to extract file inclusion (`#include`) information and scripts were used to capture the structure of the directory tree of the source repository. Build dependencies were extracted from makefiles. Also, several functions within the source code were instrumented to generate a view identifying IPC and file access.

Table 1 shows the extracted relations, numbered by view, for VANISH and Serpent. We will show how each of these views is critical to the successful reconstruction of the architectures of these systems.

VANISH		Serpent	
1	calls	1	calls
	defines_var		contains
	defines_fn	2	includes
	has_instance	3	depends_on
	has_member	4	writes
	has_subclass		read_by
	has_friend		communicates_with
	contains	5	contains_file
	defines_global		
	defines		
2	actually_calls		
3	depends_on		

Table 1: Extracted relationships colored by view

The extracted views for the example applications were stored in an SQL database [9], one table per relation. The SQL database provides more than a view repository: it provides the machinery necessary to perform view fusion. Because relations are stored as database tables, standard relational database operations such as joins, selects and projections may be used to combine the tables. We have found these operations to provide sufficient expressive power to perform all view fusions of interest in our example applications.

We will now explore several instances of view fusion, discussing in particular the implications for the process of architecture reconstruction.

VANISH Layering

VANISH was developed following the Arch metamodel of interactive software [10]. The Arch metamodel is intended to promote integrability and modifiability within a software system and to this end specifies five application layers. The VANISH implementation *realizes* these layers via several different programming language constructs, such as constrained calling relationships and class inheritance. Thus, the simplest example of a view fusion is the selection of a set of complementary relations that together realize an architectural feature or features.

In reconstructing the layered architecture of VANISH, it was necessary to fuse the calls relation with the class inheritance relation to properly reflect the system’s architecture. Figure 5 depicts the five components of the VANISH architecture and their interrelationships. The arcs between the Dialogue, the Functional Core Adapter and the Logical Interaction components comprise primarily function (method) calling relations, while the arcs between the Functional Core Adapter and the Functional Core and those between the Logical Interaction and Presentation components comprise inheritance relations.

Without the understanding that a *combination* of views is necessary, and without the means for fusing these views, this

representation would have been extremely difficult to produce.

Creating this view also had some unexpected side benefits. Once the view was created, we noted that the arcs between the Dialogue, the Functional Core and the Presentation were *not* specified by the Arch metamodel, and were not supposed to be in the software architecture. The representation derived in Figure 5 thus shows *deviations* of the VANISH's *as-implemented* architecture from its *as-designed* architecture [6]. While some of these deviations are necessary due to the nature of the programming language, others are simply deficiencies of the implementation that, having been identified, should be fixed. Without the appropriate fused view, these deviations would not have been apparent, limiting our opportunities to leverage architecture reconstruction for the improvement of implemented systems.

Elimination of Unnecessary Information

The extracted views that contribute to the reconstruction of VANISH's software architecture are made up of relationships over many types of components, including functions, classes and files. As part of the process of architectural reconstruction, appropriate aggregation relations are derived between classes and files (in VANISH, a file contains functions for at most one class, allowing a unique containment) to allow construction of higher-level architectural elements from classes. After applying these aggregations, we noted that several files remained. There were two reasons for this: some files were not associated with any class, such as the file containing event processing code; and some files were "leftovers" from previous versions, no longer contributing to the application. Because we're interested in those files in the former category, but not those in the latter, it was necessary to systematically separate them.

This pruning was accomplished using another instance of view fusion, combining the LSME-extracted view with the extracted build dependencies. This fusion identifies results in a view derived from the LSME view, but with files that do not contribute to the application removed. Figure 6 shows a

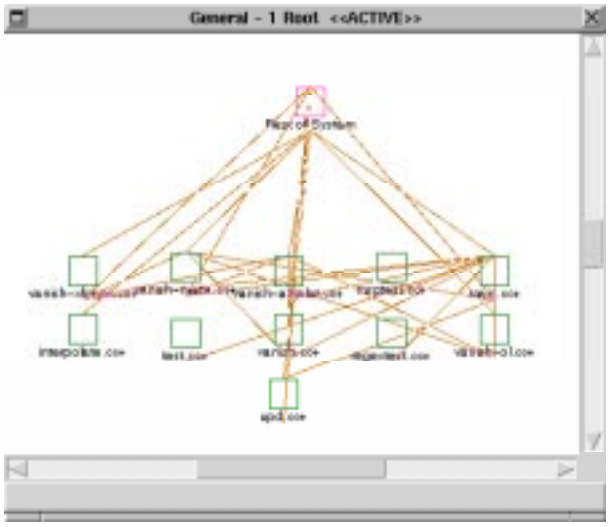

Figure 6: VANISH before removal of unused files

simplified view of VANISH (in which all other components have been elided as "Rest of System") before removal of the unused files and Figure 7 shows the same view after the

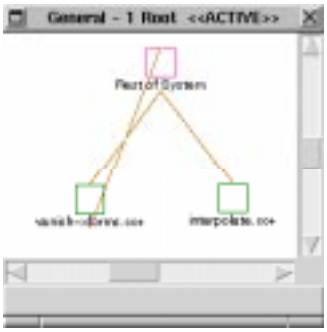

Figure 7: VANISH after removal of unused files

unused files have been removed. This example of view fusion is particularly important to the process of architecture reconstruction as it provides a mechanism for systematic elimination of unnecessary information that could otherwise hamper the process.

Improving One View With Another

Static extraction tools are error prone, frequently producing false negatives (missing elements in the source corpus) and false positives (identifying elements that do not actually appear). On the other hand, dynamic extraction tools, such as profilers, rarely (if ever) produce false positives. Unfortunately, these tools are limited by the completeness of the scenarios for which they are executed, and so they typically produce many false negatives, representing call paths that were not traversed during execution. Here we have an opportunity to leverage the high quality (but incomplete) views generated by profiling tools to improve the lower quality of the static views generated by source code analyzers.

Prima facie, it appears that we could blindly merge the information from the two types of views, resulting in an improved

fused view. However, this will not work because of inconsistencies between the views. In particular, certain critical information, such as the polymorphism attained through class inheritance, or the realization of a function pointer as a specific function, is absent from the static views. So, for example, a call apparently to some method *X* of class *C* might be realized, at run time, as a call to methods X_1, X_2, or X_3, of subclasses of *C*, called C_1, C_2, or C_3 respectively.

Because of this, a call to some function in the static view might turn into any number of different calls (one per subclass) in the dynamic view. We therefore must *reconcile* the two views if we are to benefit from the different kinds of information that each can uniquely contribute. We can accomplish this reconciliation by translating elements of the dynamic views to make them compatible with the static views with which they are being fused.

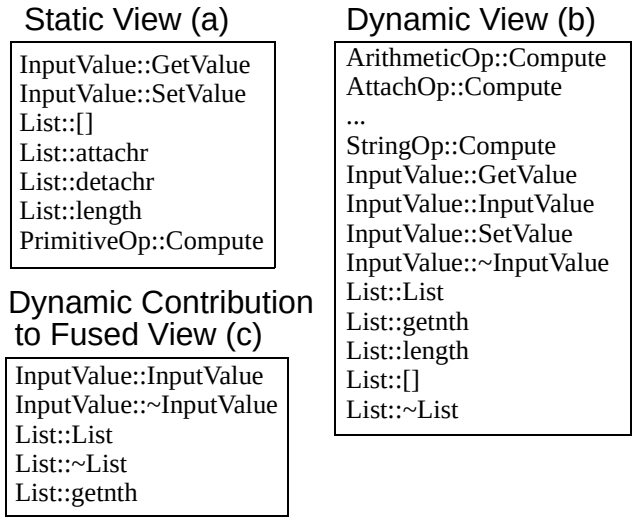

Figure 8: Combining calls from static and dynamic views

Consider Figure 8, which shows calls made by a function, from both a static view (a) and a dynamic view (b). The various "`Op`" classes (e.g. `ArithmeticOp, StringOp`) inherit from the `PrimitiveOp` class and the calling function calls the `Compute` method in these classes via their superclass. We see that several calls appear in both views, such as `InputValue::GetValue`. In addition, there are calls that appear in the dynamic view that do *not* appear in the static view, such as `List::getnth`.

A simple-minded fusion of these two views, with the objective of improving the error prone static view, would end up including calls to each of the `PrimitiveOp` subclasses. This could potentially be a view of interest, but the purely static view should include only the single call to the superclass, because this is all that one would find upon examining the source code. To leverage the accuracy of the dynamic view to improve the static view, we must therefore explicitly make the mapping between the calls to the subclass functions and the call to the superclass function.

Fortunately, we can make this mapping using other information stored in the repository. By incorporating statically extracted class inheritance information we can make these associations and thus identify the contribution of the dynamic view to the fused view (c). When this contribution is combined with the original static view, the fused view has fewer false negatives. For example, in the context of a single thousand-line VANISH source file, the application of this view fusion reduced the number of false negatives from 45 in the raw static view to 18 in the fused view.

There is a potential weakness in this technique however: if the static extractor had not identified the call to `PrimitiveOp::Compute`, each of the calls to the subclass `Compute` functions would have been included in the fused view, generating a large number of false positives, since these calls to the subclasses do not properly belong in a static view of the system. Although we have not yet seen this weakness materialize, it must be kept in mind as a possibility.

Disambiguation of Function Calls

Consider a multi-process application in which each process is built from many source files. There is no constraint (at least not in C or C++) that names be unique across different executables. Thus, in performing a static analysis of function calls, we will not be able to identify which specific function is being called if there are multiple functions in the source corpus with the called function's name. On first blush, it might appear that this is a rare situation. Actually, it is not, for there are, at a minimum, multiple "main" functions in a typical multi-process application.

Fortunately, a fusion of the build dependency view, as extracted from system makefiles, and the static view allow us to disambiguate function calls. This fusion is simply a matter of formulating an SQL query that, for each function, determines which executable(s) it is built into, and uses this information to uniquely identify the functions that it calls.

This view fusion proved invaluable during the reconstruction of the Serpent software architecture. Serpent comprises several executables, some of which participate in the construction of the others. In addition, there are many name conflicts throughout the system, making necessary the disambiguation of function names by pre-pending to the name of a function the name of the file that contains it. During extraction, file names can easily be prepended to calling function names, but view fusion is necessary to disambiguate the names of called functions.

Interprocess Communication and File Access

Figure 9 depicts one (simplified) interpretation of the as-designed software architecture of Serpent. Briefly, the Shared Data Description provides a specification for the data that will be interchanged between two run time components: the Application, which performs interaction-independent computation; and the Dialogue, which specifies the form that user interaction will take on. Saddle is a Serpent executable that generates a platform-specific representation for the platform-independent Shared Data Description. Slang is another executable that builds the Dialogue from a Dialogue Description. The Presentation is a toolkit-specific process

that realizes the actual user interface, based on interaction with the Dialogue.

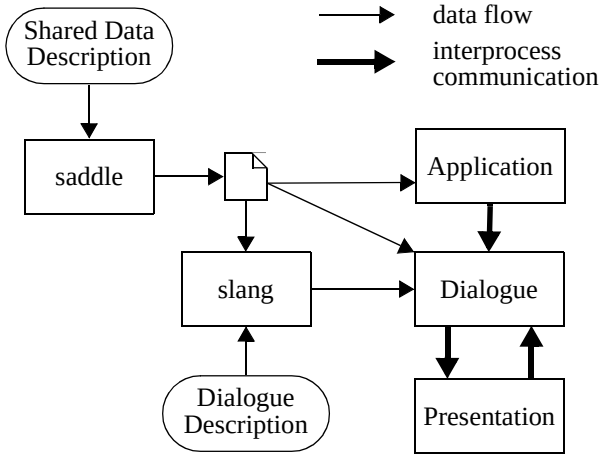

Figure 9: An interpretation of Serpent's architecture

It is important to note that the run time topology of a Serpent-based application (the Application, Dialogue and Presentation processes) is not established until run time. Various other data files (not shown) define the interaction relationships. Identifying these relationships statically was not possible: the information simply did not exist in any statically extractable corpus. Therefore, to achieve a useful reconstruction of the Serpent software architecture, it was necessary to *instrument* the Serpent source code.

As the primary interactions of interest were IPC and file access, two separate instrumentations were performed:

1. all calls to the `fopen` (file open) function in the system were replaced with calls to another function, called `fopen_instr`, that recorded the process identifier of the calling process, the name of the file being opened and the mode (read or write) of access
2. the functions providing Serpent's interprocess communication abstraction, called `ipc_send`, `ipc_receive` and `ipc_receive_nowait`, were modified to record the address of the sender, the address of the receiver and the process identifier of the caller

Data collected from these instrumentations was post-processed to determine the correlation between process identifiers, IPC addresses and process names.

The instrumented system was then compiled—the process of building slang, saddle, the Presentation, etc.—and used to build a Serpent application—the process of running saddle, slang and so forth to build the Application and the Dialogue. Next, the application (called "spider") was executed, exercised and terminated. Finally, the data recording all these activities, produced by the instrumentation, was post-processed and added to the Dali repository.

Figure 10 shows a view refined from the instrumented view, via the application of a small number of aggregating patterns similar to those described above. In the upper left corner, we

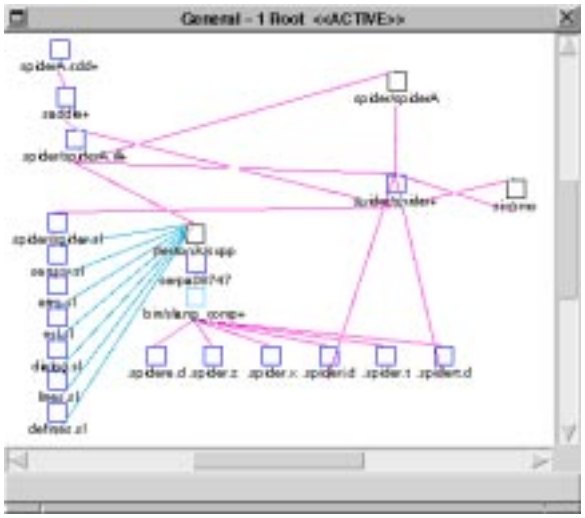

Figure 10: A view refined from the instrumented view

see the Shared Data Description (`spiderA.sdd+`) being transformed by saddle (`saddle+`) into `spider/spiderA.ill+`. In the lower left of the figure, we see the dialogue description (`spider/spider.sl`) being read by the slang preprocessor (`devtools/scpp`) and the resulting intermediate file (`serpa08747`) being read into the slang compiler (`bin/slang_comp+`). The slang compiler writes several files (e.g. `.spidere.d`) which will be used to determine the run time IPC topology. Finally, in the upper right corner, we see the application run time: the Application (`spider/spiderA`), the Dialogue (`spider/spider+`) and the Presentation (`six/smo`), along with their interactions. Also note that the Application and the Dialogue both read `spider/spiderA.ill+`, as expected.

This example demonstrates the importance of incorporation of a dynamically-extracted *instrumented* view to the reconstruction of the software architecture of Serpent. Without it, it would have been impossible to establish a relationship between the as-designed architectural representation of Figure 9 and the as-implemented system.

In addition, there is another important view fusion that can now be applied to realize another interpretation of Serpent's architecture: a fusion between the instrumented view, the build-dependency view and the static view. This fused view will thus represent the interactions between the executables in the system as well as their dependencies on the sub-

systems embodied in the source code. Figure 11 shows this

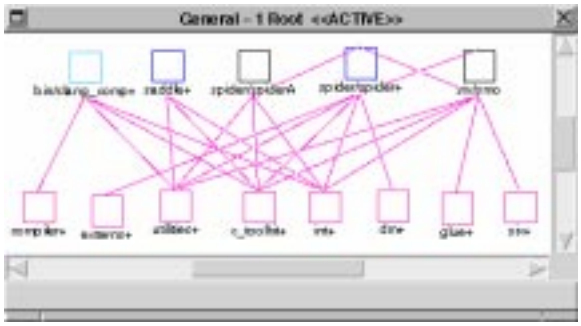

Figure 11: A fusion of static, instrumented and build-dependency views

fused view. Along the top of the figure are the Serpent processes, as shown in Figure 10; along the bottom of the figure are the Serpent source subsystems, as partitioned by the directory structure of the source repository. Arcs between the processes and the subsystems represent source dependencies (build dependencies or function calls) on the subsystems (interdependencies between the subsystems have been elided).

Together, Figures 10 and 11 represent two of the most common and important interpretations of the Serpent software architecture, or equivalently, two of the most important of Serpent's software architectures. Without the use of multiple extracted views and appropriate view fusions, these representations would have been unobtainable.

THE END

This paper has shown how the Dali workbench has supported the extraction and fusion of architectural views. We have tried to make 3 key points here: that view fusion is necessary to create appropriate views of a system's software architecture, that views from different source extraction techniques are necessary to properly support view fusion, and that an open approach to extraction is critical, for without it we could not opportunistically add new extraction techniques, or combine existing techniques in novel ways.

The fusion of views extracted through multiple different techniques distinguishes Dali from other architectural reconstruction approaches. We have seen a number of benefits here that it provided:

- the improvement of an error-prone static extraction by fusing it with dynamic information, as mediated by an inheritance view
- the disambiguation of function names in a multi-process system by fusing function call information with build dependencies
- the pruning of "dead" elements from the architecture, by combining static source elements with a build view
- the uncovering of the true layering of VANISH, by combining function call and inheritance information
- the establishment of the topology and dependencies of Serpent by fusing the instrumented run-time view with static and build dependency information

Architectural maintenance throughout the life cycle is of critical importance to the long-term health of a software intensive system. Without an understanding of the relationship between a system's as-designed and as-implemented architectures, we can have no confidence that desired properties embodied in the architecture are actually exhibited by the system.

We have argued that architectural maintenance, understanding, and conformance require the use of tools that support the reconstruction of architectural elements from the available artifacts of a software system. Such tools must provide facilities for the extraction of many architectural views and for the fusion of these views. Without these facilities, the process of software architecture reconstruction is severely limited in scope, to the extent that important architectural perspectives can not be recovered.

REFERENCES

[1] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1997, to appear.

[2] L. Bass, B. Clapper, E. Hardy, R. Kazman, R. Seacord, "Serpent: A User Interface Management System," In *The Proceedings of the Winter 1990 USENIX Conference*, January 1990, pp. 245-257.

[3] Y.-F. Chen, M.Y. Nisihmoto, C.V. Ramamoorthy, "The C Information Abstraction System", *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, March 1990, pp. 325-334.

[4] R. Kazman, G. Abowd, L. Bass, P. Clements, "Scenario-Based Analysis of Software Architecture", *IEEE Software*, Nov. 1996, pp. 47-55.

[5] R. Kazman, J. Carrière, "An Adaptable Software Architecture for Rapidly Creating Information Visualizations", *Proceedings of Graphics Interface '96*, (Toronto, ON), May 1996, pp. 17-27.

[6] R. Kazman, J. Carrière, "Playing Detective: Reconstructing Software Architecture from Available Evidence", CMU/SEI-97-TR-010, 1997.

[7] G. Murphy, D. Notkin, E. Lan, "An Empirical Study of Static Call Graph Extractors", *Proceedings of ICSE 18*, (Berlin, Germany), March 1996, pp. 90-99.

[8] G. Murphy, D. Notkin, "Lightweight Lexical Source Model Extraction", *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, July 1996, pp. 262-292.

[9] M. Stonebraker, L. Rowe, M. Hirohama, "The Implementation of POSTGRES", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 125-141.

[10] UIMS Tool Developers Workshop, “A Metamodel for the Runtime Architecture of an Interactive System”, *SIGCHI Bulletin*, 24(1), 32-37.

[11] K. Wong, S. Tilley, H. Müller, M. Storey. “Programmable Reverse Engineering”, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 4, No. 4, pp. 501-520, December 1994.

[12] A. Yeh, D. Harris, M. Chase, “Manipulating Recovered Software Architecture Views”, *Proceedings of ICSE 19*, (Boston, MA), May 1997, pp. 184-194.