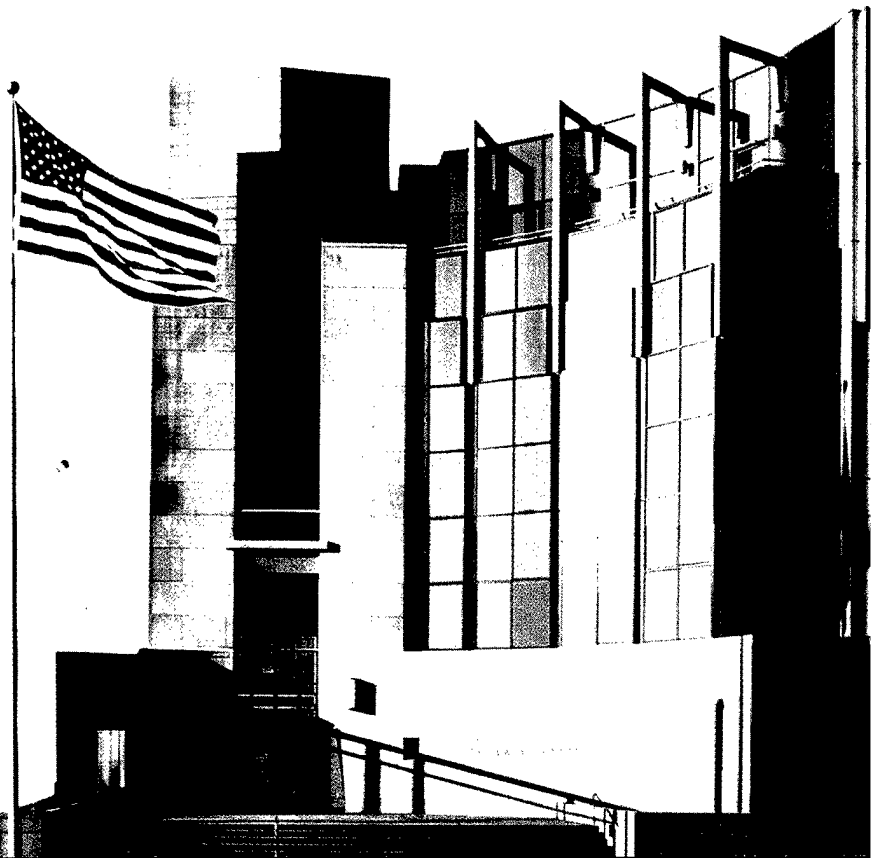
Carnegie Mellon
Software Engineering Institute

Architecture Reconstruction Guidelines

Rick Kazman
Liam O'Brien
Chris Verhoef

August 2001



TECHNICAL REPORT
CMU/SEI-2001-TR-026
ESC-TR-2001-026

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Architecture Reconstruction Guidelines

CMU/SEI-2001-TR-026
ESC-TR-2001-026

Rick Kazman
Liam O'Brien
Chris Verhoef

August 2001

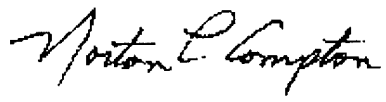
Architecture Tradeoff Analysis Initiative

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

Architecture reconstruction is the process where the "as-built" architecture of an implemented system is obtained from the existing legacy system. This is done through a detailed analysis of the system using tool support. The tools extract information about the system and aid in building and aggregating successive levels of abstraction. If the reconstruction is successful, the end result is an architectural representation of the system that aids in reasoning about the system. In some cases, it may not be possible to generate a useful representation due to the system.

Architecture reconstruction generates an architectural representation that can be used in several ways. One of the main uses is for documenting the existing architecture. If no documentation exists or it is out of date, the recovered architectural representation can be used as a basis for redocumenting the architecture. The recovered "as-built" architecture of the system can be used to check conformance against an "as-designed" architecture. The architectural representation can also be used as a starting point for reengineering the system to a new desired architecture. Finally, the representation can be used to help identify components for reuse, or to help establish a software product line.

In this report, we describe the process of architecture reconstruction using the Dali architecture reconstruction workbench. We outline guidelines for reconstructing the architectural representations of existing systems. The process that is undertaken to reconstruct an architecture can be supported by other tools and in fact can be done manually.

1 Introduction

Architecture reconstruction is the process where the "as-built" architecture of an implemented system is obtained from an existing legacy system. This is done through a detailed system analysis using tool support. The tools extract information about the system and aid in building and aggregating successive levels of abstraction. If the reconstruction is successful, the end result is an architectural representation that aids in reasoning about the system. In some cases, it may not be possible to generate a useful representation due to the system.

Architecture reconstruction generates an architectural representation that can be used in several ways. One of the main uses is for documenting the existing architecture. If no documentation exists or it is out of date, the recovered architectural representation can be used as a basis for redocumenting the architecture. The approach can be used either during development or when development has been completed to recover the "as-built" architecture of the system, so that it can be used to check conformance against an "as-designed" architecture. The architectural representation can also be used as a starting point for reengineering the system to a new desired architecture. Finally, the representation can be used as a means for identifying components for reuse, or for establishing an architecture-based software product line.

Architecture reconstruction has been used in a variety of projects ranging from Magnetic Resonance Imaging (MRI) scanners to public telephone switches, and from helicopter guidance systems to classified National Aeronautics and Space Administration (NASA) systems. The SEI has used architecture reconstruction to

- Redocument architectures for physics simulations.
- Understand architectural dependencies in embedded control software for reengineering.
- Evaluate conformance of a satellite ground station system's implementation to its reference architecture.
- Reconstruct three automobile systems and evaluate their potential for conversion to a product line.
- Recover the architecture of several network management systems.

Architecture reconstruction is a complex task that requires a range of activities and skills. Software engineers familiar with compiler construction techniques and Unix environments (especially utilities such as grep, sed, awk, perl, python, lex/yacc, etc.) have the necessary skills to undertake architecture reconstruction. However, with the large amount of software in most systems, it is impossible to undertake all architecture reconstruction activities manually.

Tool support for these activities is needed, and in general, no single tool or set of tools is adequate. There is often diversity in the number of implementation languages and dialects in which a software system is implemented. For example, a mature MRI scanner easily contains software written in 15 different languages. During fixes applied to solve the Y2K problem, each additional language was estimated to add 5% to repair costs. Given such diversity, we cannot hope to have a full, universally applicable tool set that can operate with the push of a button. Instead we are led to a particular design philosophy for a tool set to support architecture reconstruction activities: the *workbench*.

An architecture reconstruction *workbench* should be open (easy to integrate new tools as required) and provide a lightweight integration framework whereby new tools that are added to the tool set do not impact the existing tools or data unnecessarily. The Software Engineering Institute (SEI) has developed *Dali*, which is such a workbench [Kazman 99]. Other examples include Sneed's reengineering workbench [Sneed 98], the software renovation factories of Verhoef and associates [Brand 97], and the rearchitecting tool suite by Philips Research [Krikhaar 99].

Using the tool support provided by the Dali workbench, the software architecture reconstruction process comprises the following five phases:

1. View Extraction

   In the View Extraction phase, information is obtained from various sources.

2. Database Construction

   The Database Construction phase involves converting the extracted information into the Rigi Standard Form [Müller 93] (a tuple-based data format in the form of "relation <entity1> <entity2>") and an SQL database format from which the database is created.

3. View Fusion

   The View Fusion phase combines various views of the information stored in the database.

4. Architecture Reconstruction

   In the Architecture Reconstruction phase, the main work of building abstractions and representations and generating an architectural representation takes place.

5. Architecture Analysis

   The Architecture Analysis phase involves analyzing the resulting architecture.

All five phases are highly iterative. Figure 1 depicts the structure of the Dali workbench and situates the tasks of architecture reconstruction within it.

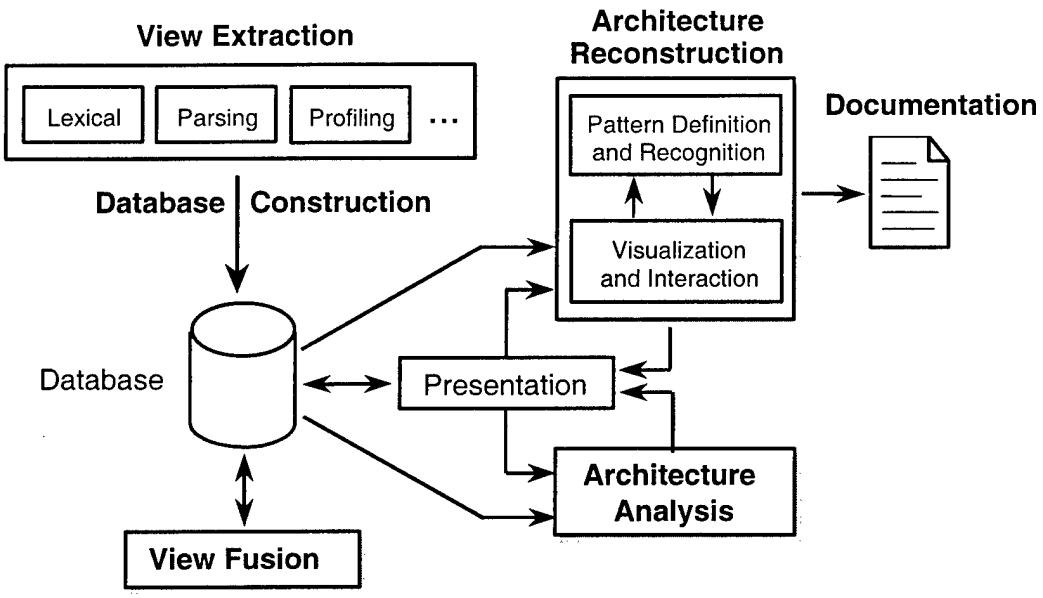


Figure 1: Outline of the Dali Workbench and Its Phases

Several people are required to carry out the reconstruction process. Those who should be involved include the person doing the reconstruction (reconstructor) and one or more people who are familiar with the system being reconstructed (e.g., the architect and software engineers familiar with the system).

The reconstructor extracts the information from the system and, either manually or with the use of tools, abstracts the architecture. First the reconstructor generates a set of hypotheses about the system. These hypotheses reflect the set of inverse mappings from the set of source artifacts to the design (ideally the opposite of the design mappings). The hypotheses are then tested by generating and applying these inverse mappings to the extracted information and validating the result. In order to generate these hypotheses and validate them, the reconstructor needs the support of people who are familiar with the system, including the system architect or engineers who initially developed or currently maintain it.

The following sections describe the architecture reconstruction process in more detail. They also present guidelines that can be used to carry out each phase. We do not discuss the Architecture Analysis phase in this particular report. Architecture Analysis is the topic of a separate report [Kazman 00]. Most of these guidelines are not specific to the Dali tool and could be applied if other tools were used, even if the architecture reconstruction was carried out manually.

2 View Extraction Phase

The View Extraction phase involves analyzing the existing design and implementation artifacts of a system to construct a model based upon multiple views. From the source artifacts (e.g., code, header files, build files) and other artifacts (e.g., execution traces) of the system, you can identify and capture the elements of interest and their relations to extract several fundamental views of the system. Table 1 shows a list of typical elements and several relations among these elements that might be extracted from a system.

Table 1: A Typical Set of Source Elements and Relations

Source Element	Relation	Target Element	Description
File	includes	File	A C preprocessor #include of one file by another
File	contains	Function	A definition of a function in a file
File	defines_var	Variable	A definition of a variable in a file
Function	calls	Function	A static function call
Function	access_read	Variable	A read access on a variable
Function	access_write	Variable	A write access on a variable

Each of the relations between the elements constitutes a different view of the system. The "calls" relation between the functions yields the call graph of the system. This shows how the various functions in the system interact. The "includes" relation between files shows us a dependence view between files in the system. The "access_read" and "access_write" relation between functions and variables, show how data is used in the system. Certain functions may write a set of data and others may read it. This information is used to determine how data is passed between various parts of the system. For example, we can determine whether or not a global data store is used (similar to a blackboard architectural style) or whether most information is passed through function calls.

If the system being analyzed is large and is divided into a particular directory structure on a file system, capturing that directory structure may be important to the reconstruction process. Certain components or subsystems may be stored in particular directories and capturing relations such as "dir_contains_file" and "dir_contains_dir" would be useful in trying to identify components later in the reconstruction process.

The set of elements and relations that are extracted will depend on the type of system that is being analyzed and the extraction support tools that are available. If the system to be reconstructed is object-oriented, classes and methods would be added to the list of elements to be extracted, and relations such as "Class is_subclass Class" and "Class contains Method" could be extracted and used in the reconstruction process.

Extracted views can be categorized as either static or dynamic. Static views are those obtained by observing only the artifacts of the system, while dynamic views are those that are obtained by observing the system during execution. In many cases, static and dynamic views can be fused to create a more complete and accurate representation of the system. (This will be discussed in Section 4.) If the architecture of the system changes at runtime, for example, a configuration file is read in by the system and certain components are loaded at runtime, then that runtime configuration should be captured and used when carrying out the reconstruction.

To extract a source view, you can apply whatever tools are available, appropriate, or necessary for a given target system. The types of tools that we have used regularly in our extractions include

- parsers (e.g., Imagix, SNiFF+, CIA, rigiparse)
- abstract syntax tree-based (AST-based) analyzers (e.g., Gen++, Refine)
- lexical analyzers (e.g., LSME)
- profilers (e.g., gprof)
- code instrumentation
- ad hoc (e.g., grep, perl)

These tools are applied to the raw source code. Parsers analyze the code and generate internal representations from it (for the purpose of generating machine code). Typically, it is possible to save this internal representation to obtain a source view. AST-based analyzers do a similar job, but they build an explicit tree representation of the parsed information. One can build analysis tools that traverse the AST and output selected pieces of architecturally relevant information in an appropriate format.

Lexical analyzers examine source artifacts purely as strings of lexical elements or tokens. The user of a lexical analyzer can specify a set of patterns to be matched and the elements output. An example of a lexical pattern would be a pattern that recognizes the "#include <filename>" directive in source files and the output elements would be the source file in which the "#include" appeared and the file within the "< >". Applying this pattern yields the dependencies that exist between files.

Similarly, we have used a collection of ad hoc tools such as grep and perl to carry out pattern matching and searching within the code in order to output some required information. All of

these tools—code-generating parsers, AST-based analyzers, lexical analyzers, and ad hoc pattern matchers—are used to output purely static information.

Profilers and code coverage analysis tools can be used to output information about the code as it is being executed. They usually do not involve adding any new code to the system. On the other hand, code instrumentation, which has wide applicability in the field of testing, involves adding code to the system to output some specific information (such as what processes connect with each other at runtime) while the system is executing [McCabe 00]. These tools generate dynamic views of the system.

Tools to analyze design models, build files, makefiles, and executables can also be used to extract further information as required. For instance, build files and makefiles include information on module or file dependencies that may not be reflected in the source code.

Much architecture-related information may be extracted *statically* from source code, compile-time artifacts, and design artifacts. However, this may not be enough for the architecture recovery process. Some architecturally relevant information may not exist in the source artifacts, due to *late binding*. Examples of late binding include

- polymorphism
- function pointers
- runtime parameterization

There are other reasons why the precise topology of a system may not be determined until runtime. For example, multiprocess and multiprocessor systems, using middleware such as Common Object Request Broker Architecture (CORBA), Jini, or Component Object Model (COM), frequently establish their topology dynamically, depending on the availability of system resources. The topology of such systems does not live in its source artifacts and hence cannot be reverse engineered using static extraction tools.

Therefore, it may be necessary to use tools that can generate dynamic information about the system (e.g., profiling tools). In some instances, this may not be possible, because tools that can obtain this dynamic information may not be available on the system platform. Also, there may be no way to collect the results from code instrumentation. This usually occurs with embedded systems, where there is no means to output the information generated from code instrumentation.

2.1 Guidelines

The following guidelines apply to the View Extraction phase:

- Use the "least effort" extraction. Consider what information you need to extract from a source corpus and choose the most appropriate tool. Is the information lexical in nature? Does it require the comprehension of complex syntactic structures? Does it require some

semantic analysis? In each of these cases, a different tool could be applied successfully. In general, lexical approaches are the cheapest to use, and they should be considered if your reconstruction goals are simple.

Guiding Principles	Type of Extraction Required
The information that is to be extracted is lexical in nature. A set of patterns can be written that allows one to extract that information.	Lexical Analysis (You may be able to use simple lexical analysis utilities such as perl and grep.)
The information that needs to be extracted cannot be identified lexically. Through the use of a grammar for a language, it is possible to identify elements and relations.	Parsing
More contextual information (semantic information) must be available to clearly identify certain elements and relations.	AST-based analyzers (These allow for an AST to be built and updated after parsing with semantic information.)

- Validate source views. Before starting to fuse or manipulate the various views that have been obtained, make sure that the correct information has been captured in the view. It is important that the tools being used to analyze the source artifacts are carrying out their job correctly. A detailed manual examination and verification of a subset to the elements and relations against the underlying source code should be carried out to establish that the correct information is being captured. The precise amount of information that needs to be verified manually is up to the individual. Assuming that this is a process of statistical sampling, the reconstructor can choose a desired confidence level. In general, the more information that is validated manually, the higher the confidence in the results.
- Extract dynamic information where required. If there is a lot of runtime or late binding and the architecture is dynamically configurable, dynamic information about system runtime is essential and should be extracted using whatever technique is most appropriate. If a profiler is available, then use it to extract runtime information. If the system runs on a platform where no profiler is available, it may be necessary to instrument the code to obtain the runtime information. When it is not possible to extract the dynamic information, only static information may be available for architectural representations.

3 Database Construction Phase

The set of extracted views are converted into the Dali format and stored in a relational database during the Database Construction phase. Several tools and techniques have been incorporated into the Dali workbench to assist in this process. These mainly consist of perl scripts that read the data and convert it into a file in the Rigi Standard Format. The extracted views may be in many different formats depending on the tools used to extract them. For example, an extraction tool like Imagix-4D can be used to load the source code of a system into its internal representation and this information is dumped to a set of flat files indexed by file or by function. These files have a uniform structure, and tools can be developed in perl to read these files and output information about elements and relations.

Once the elements and relations (Extracted View) file is converted to Rigi Standard Format, it is read by another perl script. The data is output in a format that includes the necessary SQL code to build and populate the relational tables with the extracted information. Figure 2 depicts this process.

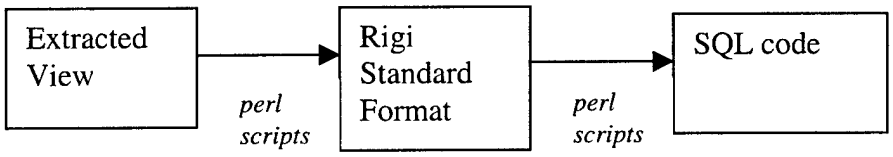


Figure 2: Conversion of the Extracted View to SQL Format

Figure 3, next page, shows a typical example of the SQL code that is generated.

```
create table calls( caller text, callee text );
create table accesses( func text, variable text );
create table defines_var( file text, variable text );
...
insert into calls values( 'main', 'control' );
insert into calls values( 'main', 'clock' );
...
insert into accesses values( 'main', 'stat1' );
...
```

Figure 3: Example of SQL Code Generated in Dali

Dali currently uses the PostgreSQL[1] relational database. When the data is entered into the database, two additional tables are generated: *components* and *relationships*. The *components* table lists the set of source and target elements that has been extracted from the system, and the *relationships* table lists the set of relations that has been extracted from the system.

It is possible to create new tools and techniques other than those currently available in Dali, to carry out the conversion from whatever format(s) an extraction tool uses. For example, if a tool is required to convert the output from a tool not currently supported, it can be built. Then the output from the new tool can be converted into Rigi Standard Format and converted to SQL code. The conversion tool that does this can become part of the Dali workbench.

In the current version of the Dali workbench, the PostgreSQL relational database provides functionality through the use of SQL and perl for generating and manipulating the architectural views [Stonebraker 90] (examples are shown in Section 5). Changes could easily be made to the SQL scripts to make them compatible with other SQL implementations.

[1] http://www.postgresql.org

3.1 Guidelines

The following guidelines apply to the Database Construction phase:

- Build database tables from the extracted relations to make processing the data views easier. For example, create a table that stores the results of a particular query such as grouping the files into components or subsystems. Then you don't have to run that query again. If the results of that query are required in building further queries, you can access them easily through that table.
- As with any database construction, carefully consider the database design before you get started. What will the primary (and possibly secondary) key be? Will any database joins be particularly expensive, spanning multiple tables?
- Use perl, awk, and other similar lexical tools to change the format of data extracted using any tools into the Rigi Standard Format so that the Dali workbench can use the data. These tools are less expensive in terms of development time and resource utilization than writing more complex tools using other languages.

4 View Fusion Phase

The View Fusion phase involves defining a set of queries that manipulate the extracted views to create *fused* views. For example, a static call view may be fused with a dynamic call view. As we said earlier, a static view may not give us all of the architecturally relevant information. In the case of late binding in the system, some function calls may not be identifiable until runtime, so there is a need to generate a dynamic call view. These two views need to be reconciled and fused to produce the complete call graph for the system.

The View Fusion phase reconciles and establishes connections between views that provide complimentary information. Fusion is illustrated using the following examples. The first shows the improvement of a static view of an object-oriented system with dynamic information. The other shows the fusion of several views to identify function calls in a system.

4.1 Improving a View

Consider the two excerpts shown in Figure 4. They are from the sets of methods extracted from a system implemented in C++.

Static Extraction

```
InputValue::GetValue
InputValue::SetValue
List::[]
List::length
List::attachr
List::detachr
PrimitiveOp::Compute
```

Dynamic Extraction

```
InputValue::GetValue
InputValue::SetValue
InputValue::~InputValue
InputValue::InputValue
List::[]
List::length
List::getnth
List::List
List::~List
ArithmeticOp::Compute
AttachOp::Compute
 . . .
StringOp::Compute
```

Figure 4: Static and Dynamic Data Views

These views include a static view and a dynamic view of an object-oriented piece of code. The differences between them are shaded in Figure 5.

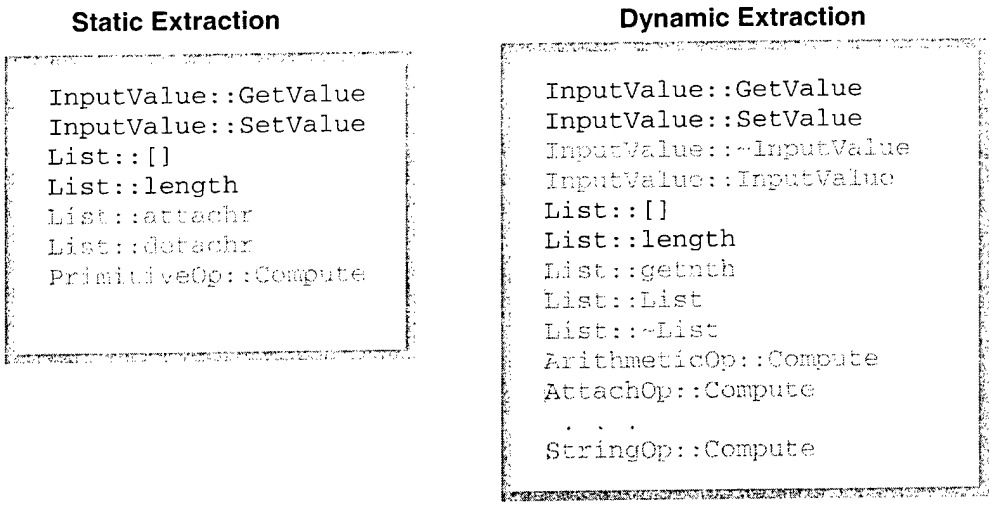


Figure 5: The Difference Between Static and Dynamic Views

We can see from an examination of the dynamic view that, for example, `List::getnth` is called. However, this method is not included in the static analysis view. Also, the calls to the constructor and destructor methods of `InputValue` and `List` are not included in the static view. These missing methods must be added to the overall (reconciled) architecture view.

In addition, in this example we have a situation where the static extraction shows that the `PrimitiveOp` class has a method called `Compute`. The dynamic extraction results show no such class, but does show classes such as: `ArithmeticOp`, `AttachOp`, `StringOp`, each of which has a `Compute` method and is in fact a subclass of `PrimitiveOp`. `PrimitiveOp` is purely a superclass; it is never actually called in an executing program. But it is the call to `PrimitiveOp` that a static extractor sees when scanning the source code, since the polymorphic call to one of `PrimitiveOp's` subclasses occurs at runtime. So, to get an accurate view of the architecture, we need to reconcile the static and dynamic views of `PrimitiveOp`. To do this, we perform a fusion using SQL queries over the extracted *calls, actually_calls,* and *has_subclass* relations. In this way, we can see that the calls to `PrimitiveOp::Compute` in the static view and to its various subclasses in the dynamic view are really the same thing.

The lists in Figure 6 show the items that would be added to the fused view (in addition to the methods that the static and dynamic views agreed upon) and those that are removed from the fused view (even though one of the static or dynamic views included them).

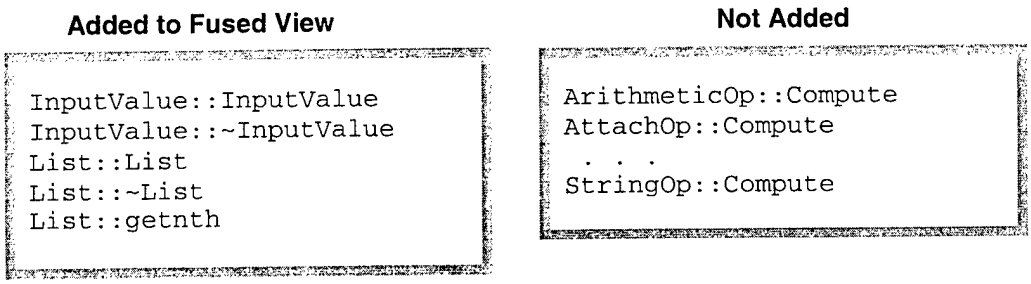


Figure 6: Items That Were Added to and Omitted from the Overall View

4.2 Disambiguating Function Calls

In a multiprocess application, name clashes are likely to occur. For example, several of the processes might have a procedure called "main." It is important to identify and disambiguate these name clashes within the extracted views. Once again, by fusing information that can be extracted easily, we can remove this potential ambiguity. In this case, we would need to fuse the static "calls" view with a "file/function containment" view (to determine which functions are defined in which source files) and a "build dependency" view (to determine which files are compiled together to produce which executables). The fusion of these three information sources makes potentially ambiguous procedure or method names unique, and hence unambiguously referred to in the architecture reconstruction process. Without the view fusion, this ambiguity flaw would persist, and the reconstruction results would be ambiguous.

4.3 Guidelines

The following guidelines apply to the View Fusion phase:

- Fuse views when no single view provides the needed information for architecture reconstruction. For example, we need the calls view to show the functional decomposition of the system. If we have a static calls view and a dynamic call view, these are fused to produce a single calls view that shows the decomposition.
- Fuse views when there is ambiguity within a view, and it is not possible to disambiguate using a single view.
- Consider different extraction techniques to extract different view information. For example, you can use different extraction techniques, such as dynamic and static. Or you might want to use different instances of the same kind of technique, if you feel that a single instance might provide erroneous or incomplete information. For example, you might use different parsers for the same language if each provides different information.

5 Architecture Reconstruction Phase

The Architecture Reconstruction phase consists of two primary activity areas: *visualization and interaction* and *pattern definition and recognition.*

Visualization and interaction provides a mechanism by which the user may interactively visualize, explore, and manipulate views. Rigi is used to present views to the user as a hierarchically decomposed graph [Wong 94]. An example presentation of an architecture view is shown in Figure 7.

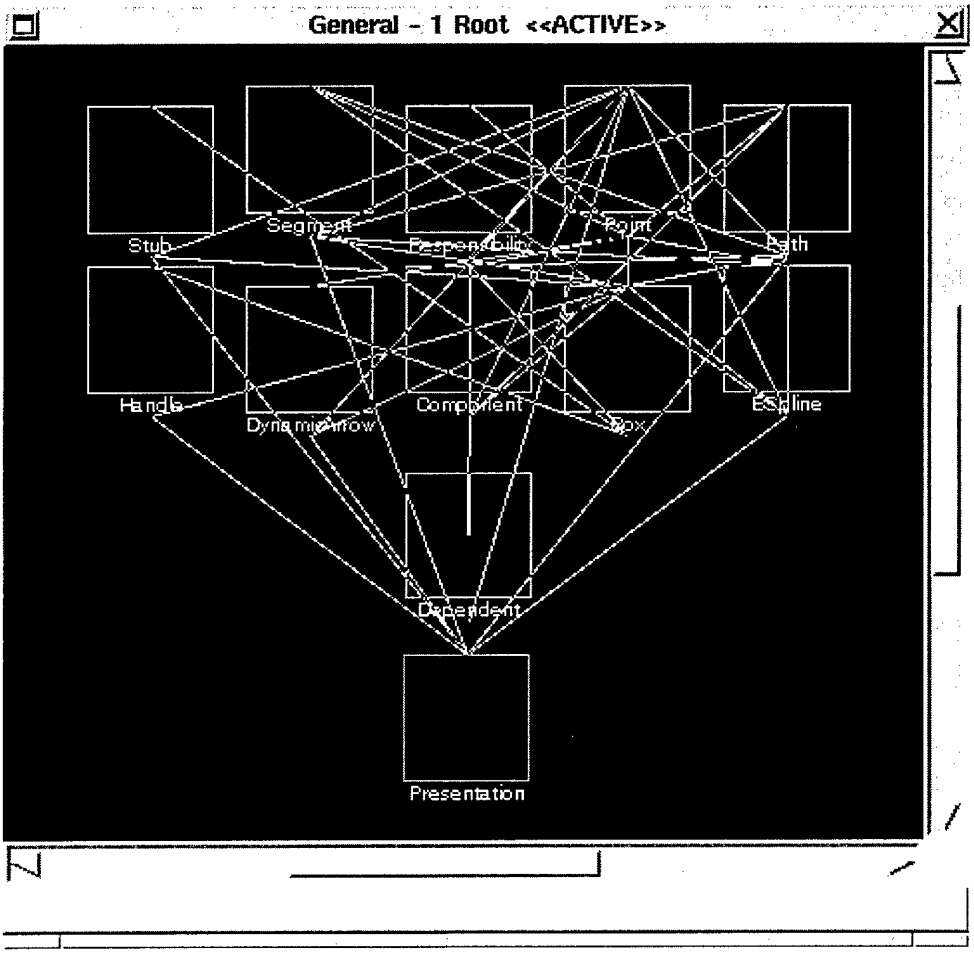


Figure 7: An Architecture Represented at the Highest Hierarchical Level

Pattern definition and recognition provides facilities for architectural reconstruction: the definition and recognition of architectural patterns. Dali's architecture reconstruction facilities allow a user to construct more abstract views from more detailed ones by identifying aggregations of elements. Patterns are defined in Dali using a combination of SQL and perl patterns. An SQL query is used to identify elements from the Dali repository that will contribute to a new aggregation and perl's expressions are used to transform names and perform other manipulations of the results of the query. Patterns are captured in a patterns file and users can selectively apply and reuse various patterns.

Architecture reconstruction is not a straightforward process. For one thing, architectural constructs are not represented explicitly in the source code. Additionally, architectural constructs are realized by many diverse mechanisms in an implementation. Usually these are a collection of functions, classes, files, objects, and so forth. When a system is initially developed, its high-level design/architectural elements are *mapped* to implementation elements. Therefore, when we "reconstruct" architectural elements, we need to apply the inverses of the mappings.

Architecture reconstruction is an interpretive, interactive, and iterative process; it is not an automatic process. It requires the skills and attention of both the reverse engineering expert and the architect (or someone who has substantial knowledge of the architecture). Based upon the architectural patterns that the architecture expert expects to find in the system, the reverse engineer can build various queries using the Dali tool. These queries result in new aggregations that show various abstractions or clusterings of the lower level elements (which may be source artifacts or abstractions). By interpreting these views and actively analyzing them, it is possible to refine the queries and aggregations to produce several hypothesized architectural views of the system. These views can be interpreted, further refined, or rejected. There are no universal completion criteria for this process; it is complete when the architectural representation is sufficient to support the analysis needs of Dali users.

Suppose we have the subset of elements and relations shown in Figure 8.

Element	Relation	Element
f	defines_var	a
f	defines_var	b
g	calls	f
f	calls	h

Figure 8: Subset of the Elements and Relations

In this example variables a and b are defined in function f, that is, they are local to f. We can graphically represent this information as shown in Figure 9.

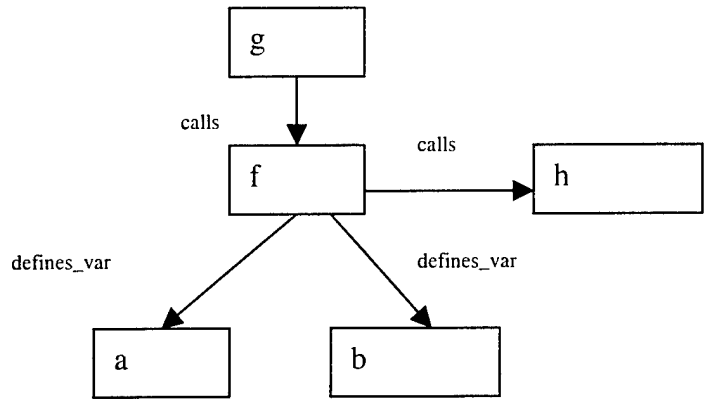


Figure 9: Graphical Representation of Elements and Relations

When carrying out an architecture reconstruction we are not interested in the local variables because they lend very little insight into the architecture of the system. Therefore we can aggregate instances of local variables to the functions in which they occur. We can write two patterns that do this. An example of the patterns that can be written is shown in Figure 10.

```
#Local Variable aggregation
SELECT tName
     FROM Components
     WHERE tType='Function';
print ''$fields[0]+ $fields[0] Function\n'';

SELECT d1.func, d1.local_variable
     FROM  defines_var d1;
print ''$fields[0] $fields[1] Function\n'';
```

Figure 10: Patterns to Aggregate Local Variables to the Function in Which They Are Defined

The first pattern updates the visual representation in Dali by adding a "+" after each function name, which means that the function is now an aggregate of the function and the local variables defined within it. The SQL query selects functions from the components table. The perl expression d is executed for each line of the result of the SQL query. The $fields array is automatically populated with the fields resulting from the query. In this case, only one field is selected (tName) from the table, so $fields[0] will store the value of this field for each tuple selected. The expression generates lines of the form:

<function>+ <function> Function

This line specifies that the element <function> should be aggregated into <function>+, which will have the type Function.

The second pattern hides the local variables from the visualization. The SQL query will identify the local variables for each function defined by selecting each tuple in the defines_var table. Thus in the perl expression $fields[0] corresponds to the func field and $fields[1] corresponds to the local_variable field. So the output is of the form

<function>+ <variable> Function

Each local variable for a function is to be added to the <function>+ aggregate for the function. The order of execution of these two patterns is not important as the final results of applying both of these queries is sorted.

The result of applying the pattern is represented graphically in Figure 11. Most patterns in Dali are developed in a similar manner.

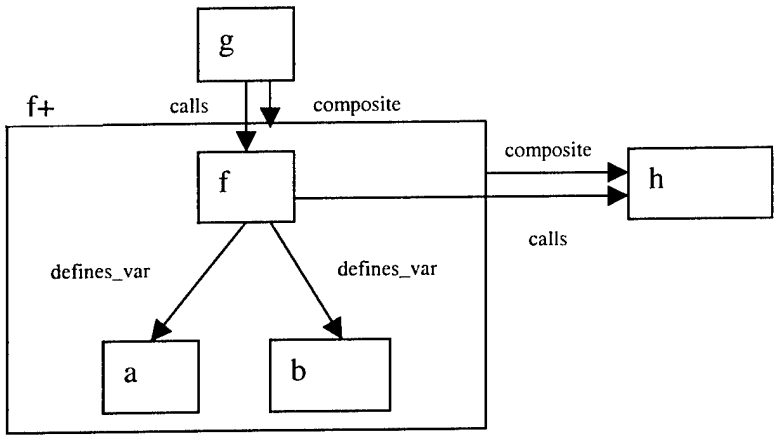


Figure 11: Result of Applying the Pattern

The primary mechanism for manipulating the views is the application of patterns (i.e., inverse mappings). Examples include patterns that

- Identify types.
- Aggregate local variables with functions.
- Aggregate members with classes.
- Compose architecture-level elements.

An example of a pattern that identifies an architectural level component is shown in Figure 12. This query identifies the *Logical_Interaction* architectural component. The query says that if the class name is *Presentation*, *Bspline*, or *Color* or the class is a subclass of *Presentation*, it belongs in the *Logical_Interaction* component.

```
SELECT tSubclass
     FROM has_subclass
     WHERE tSuperclass='Presentation';
print ''Logical_Interaction $fields[0]'';

SELECT tName
     FROM components
     WHERE tName='Presentation'
     OR tName='BSpline'
     OR tName='Color';
print ''Logical_Interaction $fields[0]'';
```

Figure 12: Query to Identify the Logical_Interaction Component

Patterns are written in this way to abstract from the lower level information to generate architecture-level views. The reconstructor builds these patterns to test hypotheses about the system. If a particular pattern does not yield useful results then it can be discarded. The reconstructor iterates through this process until useful architectural views have been obtained.

5.1 Guidelines

These guidelines apply to the Architecture Reconstruction phase:

- Be prepared to work with the architect closely and to iterate several times on the architectural abstractions that you create. This is particularly so in cases where the system has no explicit, documented architecture. In such cases, you can create architectural abstractions as hypotheses, and test these hypotheses by creating the views and showing them to the architect and other stakeholders. Based upon the false negatives and false positives found, the architect may decide to create new abstractions, resulting in new Dali patterns to apply (or perhaps even new extractions that need to be done).
- When developing patterns, try to build ones that are succinct and do not list every source element. The pattern shown in Figure 12 is an example of a good pattern; an example of a bad pattern is shown in Figure 13. The source elements that comprise the component are simply listed. This makes the pattern difficult to use, understand, and reuse.
- Patterns can be based on naming conventions, if the naming conventions are used consistently throughout the system. An example of a naming convention is where all functions, data, and files that belong to the Interface component have names that begin with "i_".
- Patterns can be based on the directory structure where files and functions are located. Component aggregations can be based on these directories.
- As architecture reconstruction is the effort of re-determining architectural decisions, given only the result of these decisions in the actual artifacts (i.e., the code that implements the decisions). As the reconstruction process proceeds, information must be added to re-introduce the architectural decisions. This process introduces bias from the reconstructor, thus reinforcing the need to have an architecture expert involved.

```
SELECT tName
     FROM components
     WHERE tName='vanish-xforms.cc'
     OR tName='PrimitiveOp'
     OR tName='Mapping'
     OR tName='MappingEditor'
        .    .    .
     OR tName='InputValue'
     OR tName='Point'
     OR tName='VEC'
     OR tName='MAT'
     OR ((tName ~ 'Dbg$' OR tName ~ 'Event$'
         AND tType='Class');
print ''Dialogue $fields[0]'';
```

Figure 13: Example of a Bad Pattern

6 Other Architecture Reconstruction Approaches

There have been several other efforts in architecture analysis and reconstruction.

6.1 Bowman and Associates

Bowman and associates outline a similar method to that of Dali for extracting architectural documentation from the code of an implemented system [Bowman 99]. In one example, they reconstructed the architecture of the Linux system. They analyzed source code using the cfx program (c-code fact extractor) to obtain symbol (elements in Dali) information from the code and generated a set of relations between the symbols. Then, they manually created a tree-structured decomposition of the Linux system into subsystems and assigned the source files to these subsystems. Next, they used the grok fact manipulator tool to determine relations between the identified subsystems, and the lsedit visualization tool to visualize the extracted system structure. Refinement of the resulting structure was carried out by moving source files between subsystems.

The difference between this approach and that used in Dali is that this approach is mainly a manual approach, where the reconstructor carries out subsystem and component identification by manually selecting source file elements to belong to these views. Dali is more automated in that queries can be written to carry out these tasks. The first step in Bowman and associates' approach was to develop a conceptual architecture. This is not done in the phases outlined earlier using Dali but developing a conceptual architecture view with the help of the developers, maintainers, or architecture is certainly part of the overall approach when Dali is used. This helps to guide the reconstruction effort in the generation and testing of hypotheses. The visualization using Rigi allows for more interaction by the reconstructor. By selecting a particular component in Dali, one can see the lower level elements that comprise those components; and by selecting a link between two components, one can see the relations represented. This level of interaction does not seem to be provided in Bowman's approach.

6.2 Harris and Associates

Harris and associates outline a framework for architecture reconstruction using a combined bottom-up and top-down approach [Harris 95]. The framework consists of three components: the architectural representation, the source code recognition engine and supporting library of recognition queries, and a "bird's eye" program overview capability. The bottom-up analysis uses the bird's eye view to display the system's file structure and components, and to

reorganize information into more meaningful clusters. The top-down analysis uses particular architectural styles to define components that should be found in the software. Recognition queries are then run to determine if the expected components exist.

Harris's approach is based upon a set of implementation language independent queries that are applied to an Abstract Syntax Tree (AST). Parsing the source code of a system generates the AST, which is specific to a particular programming language. The application mechanism of the queries is specific for each programming language (AST specific). Thus if a new language needs to be handled, then a new AST has to be developed, a parser has to be written, and a new application mechanism has to be derived. This is not the case in Dali. There, views can be extracted from different languages using the appropriate tools and the development of queries to generate architectural representations does not depend on any particular programming language. In fact, Dali can be used on code that cannot be parsed. Thus Dali is more easily applicable across a wider set of programming languages. Harris' approach does provide some metrics information about the amount of code that is covered by particular architectural styles in the system. This information may be useful for maintenance and reengineering purposes. For example, when one has to change or reimplement a particular architectural style in the system, one has an idea as to the magnitude of the problem. This type of information is not provided in the Dali workbench.

6.3 Guo and Associates

Guo and associates outline the semi-automatic architecture recovery method called ARM, which assists in architecture recovery for systems that are designed and developed using patterns [Guo 99]. It consists of four major phases: 1) developing a concrete pattern recognition plan, 2) extracting a source model, 3) detecting and evaluating pattern instances, and 4) reconstructing and analyzing the architecture. Case studies have been presented showing the use of the ARM method to reconstruct systems and check the conformance of these systems against their documented architectures. Pattern rules are transformed into pattern queries, which can be applied automatically to detect pattern instances from the source model. Refinement of the pattern queries can help to improve the precision of pattern recognition. Visualizations of the recovered patterns are presented to the tool user and aligned with the designed pattern instances.

Guo and associates used the Dali workbench to perform the architectures recovery work. An abstract pattern rule was mapped into a concrete pattern rule and was converted into an SQL query. This query was then applied to the database to extract instances of the pattern. This method is aimed particularly at systems that have been developed using design patterns. This limits the applicability of the method so that it may only apply to systems developed using design patterns or in cases where one can be sure that design pattern implementations have not eroded over time.

7 Summary

In this report, we outlined the major phases in architecture reconstruction:

- View Extraction
- Database Construction
- View Fusion
- Architecture Reconstruction

We described the activities that are carried out to complete these steps and provided examples of tool support for each activity. We also outlined guidelines for carrying out these activities to obtain a satisfactory architecture representation from an existing system. Most of these guidelines are applicable even if other tools are used to support the reconstruction effort or even if the reconstruction is carried out manually.

In our work at the SEI, we have used Dali to support the reconstruction efforts on several systems in a wide variety of domains. One of the reasons why Dali has been very useful is because of its language independence. It can be used to analyze information from many different languages, systems and from many different domains. The Dali workbench continues to evolve and be applied on new projects.

References

[Bowman 99] Bowman, T.; Holt, R.C.; & Brewster, N.V. "Linux as a Case Study: Its Extracted Software Architecture." 555-563. *Proceedings of the 21st International Conference on Software Engineering*. Los Angeles, CA, May 16-22, 1999. New York, NY: ACM Press, 1999.

[Brand 97] van den Brand, M. G. J.; Sellink, M. P. A.; & Verhoef, C. "Generation of Components for Software Renovation Factories From Context-Free Grammars." 144-153. *Proceedings of the Fourth Working Conference on Reverse Engineering*. Amsterdam, The Netherlands, October 6-8, 1997. New York, NY: ACM Press, 1997.

[Guo 99] Guo, G.; Atlee, J.; & Kazman, R. "A Software Architecture Reconstruction Method." 225-243. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*. San Antonio, Texas, February 22-24, 1999. Norwell, Massachusetts: Kluwer Academic Publishers, 1999.

[Harris 95] Harris, D. R.; Reubenstein, H. B.; & Yeh, A. S. "Reverse Engineering to the Architectural Level." 186-195. *Proceedings of the 17th International Conference on Software Engineering (ICSE)*. Seattle, Washington, April 23-30, 1995. New York, NY: ACM Press, 1995.

[Kazman 99] Kazman, R.; & Carriere, S. J. "Playing Detective: Reconstructing Software Architecture from Available Evidence." *Journal of Automated Software Engineering 6*, 2 (April, 1999): 107-138.

[Kazman 00] Kazman, R.; Klein, M.; & Clements, P. *ATAM: Method for Architecture Evaluation* (CMU/SEI-2000-TR-004, ADA382629). Carnegie Mellon University, Pittsburgh, PA. WWW: <URL: http://www.sei.cmu.edu/publications/documents/00.reports/00tr004.html> (2000).

[Krikhaar 99] Krikhaar, R. *Software Architecture Reconstruction,* Ph.D. Thesis. University of Amsterdam, Amsterdam, The Netherlands, 1999.

[McCabe 00] McCabe IQ2 (an integrated set of products and processes). WWW: <URL: http://www.mccabe.com> (2000).

[Müller 93] Müller, H. A.; Mehmet, O. A.; Tilley, S. R.; & Uhl, J. S. "A Reverse Engineering Approach to System Identification." *Journal of Software Maintenance: Research and Practice 5*, 4 (December, 1993): 181-204.

[Sneed 98] Sneed, H. M. "Architecture and Functions of a Commercial Software Reengineering Workbench." 2-10. *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering.* Florence, Italy, March 8-11, 1998. Los Alamitos, California: IEEE Computer Society Press, 1998.

[Stonebraker 90] Stonebraker, M.; Rowe, L.; & Hirohama, M. "The Implementation of POSTGRES." 125-141. *IEEE Transactions on Knowledge and Data Engineering 2*, 1. March, 1990.

[Wong 94] Wong, K.; Tilley, S.; Müller, H.; & Storey, M. "Programmable Reverse Engineering." *International Journal of Software Engineering and Knowledge Engineering 4*, 4 (December 1994): 501-520.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (LEAVE BLANK)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 2001	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Architecture Reconstruction Guidelines	C — F19628-00-C-0003

6. author(s)
Rick Kazman, Liam O'Brien, Chris Verhoef

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2001-TR-026

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ESC-2001-TR-026

11. SUPPLEMENTARY NOTES

12.A DISTRIBUTION/AVAILABILITY STATEMENT	12.B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Architecture reconstruction is the process where the "as-built" architecture of an implemented system is obtained from the existing legacy system. This is done through a detailed analysis of the system using tool support. The tools extract information about the system and aid in building and aggregating successive levels of abstraction. If the reconstruction is successful, the end result is an architectural representation of the system that aids in reasoning about the system. In some cases, it may not be possible to generate a useful representation due to the system.

In this report, we describe the process of architecture reconstruction using the Dali architecture reconstruction workbench. We outline guidelines for reconstructing the architectural representations of existing systems. The process that is undertaken to reconstruct an architecture can be supported by other tools and in fact can be done manually.

14. SUBJECT TERMS	NUMBER OF PAGES
architecture representation, architecture reconstruction, architecture re-engineering	41

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL