Carnegie Mellon
Software Engineering Institute

Architecture Reconstruction Guidelines, Third Edition

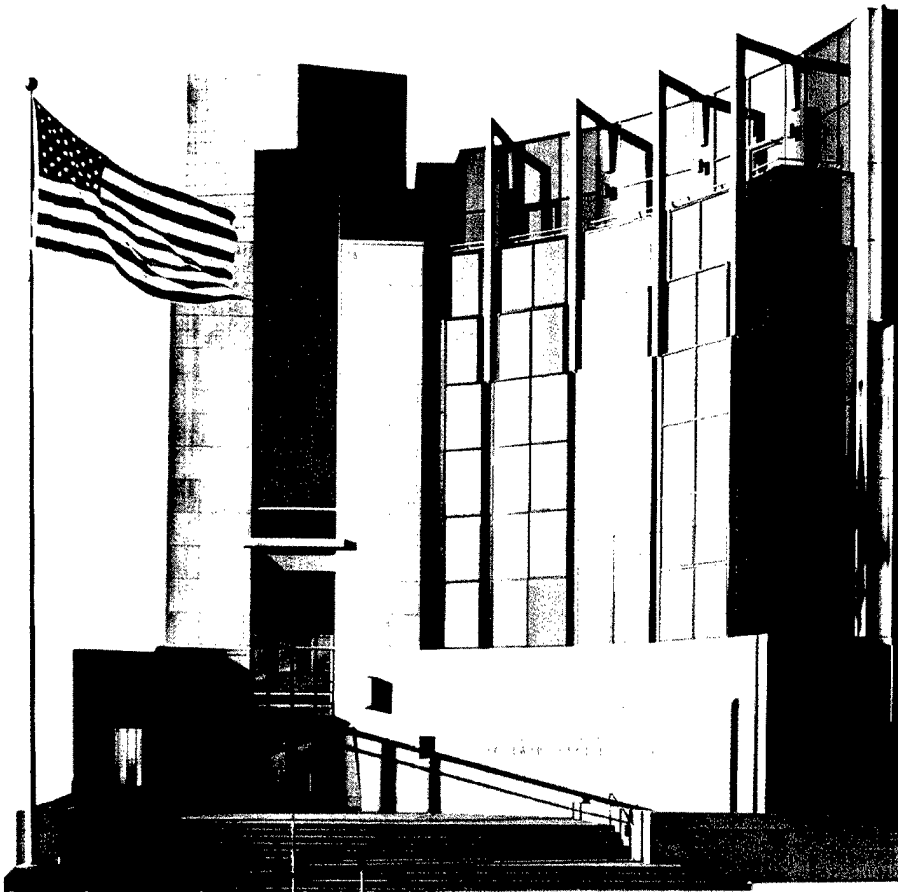
Rick Kazman
Liam O'Brien
Chris Verhoef

November 2003

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

20040412 008

TECHNICAL REPORT
CMU/SEI-2002-TR-034
ESC-TR-2002-034



Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Architecture Reconstruction Guidelines, Third Edition

CMU/SEI-2002-TR-034
ESC-TR-2002-034

Rick Kazman
Liam O'Brien
Chris Verhoef

November 2003

Architecture Tradeoff Analysis Initiative

Unlimited distribution subject to the copyright.

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

Christos Scondras
Chief of Programs, XPK

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.

Copyright 2003 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

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 252.227-7013.

Table of Contents

Preface **vii**

Abstract **ix**

1 Introduction **1**

2 Architecture Reconstruction **3**
2.1 Recommendations for Reconstruction Projects 5

3 Information Extraction Phase **7**
3.1 Guidelines 9

4 Database Construction Phase **11**
4.1 Guidelines 12

5 View Fusion Phase **13**
5.1 Improving a View 13
5.2 Disambiguating Function Calls 15
5.3 Guidelines 15

6 Architectural View Composition Phase **17**
6.1 Guidelines 21

7 Other Architecture Reconstruction Approaches **23**
7.1 Bowman and Associates 23
7.2 Harris and Associates 23
7.3 Guo and Associates 24

8 Summary **25**

References **27**

List of Figures

Figure 1: Conversion of the Extracted View to RSF 11

Figure 2: Excerpt from a Sample RSF File 11

Figure 3: Static and Dynamic Data Views 13

Figure 4: The Differences Between Static and Dynamic Views 14

Figure 5: Items That Were Added to and Omitted from the Overall View 15

Figure 6: An Architectural View of a System Presented in ARMIN 17

Figure 7: Graphical Representation of Elements and Relations 19

Figure 8: Script for Aggregating Local Variables to the Function in Which They Are Defined 19

Figure 9: Result of Applying the Script to Aggregate Local Variables 20

Figure 10: Query to Identify the Logical_Interaction Component 20

Figure 11: Example of a Bad Script 22

List of Tables

Table 1: A Typical Set of Source Elements and Relations 7

Table 2: Guiding Principles for Choosing Types of Extraction 10

Table 3: Subset of the Elements and Relations 18

Preface

The first edition of this report was published in August 2001 under a different document number: CMU/SEI-2001-TR-026. Because the second edition of this report contained significant changes from the first edition, it was assigned a new document number— the current one—and was published in December 2002.

Abstract

Architecture reconstruction is the process of obtaining the "as-built" architecture of an implemented system from the existing legacy system. For this process, tools are used to extract information about the system that will assist in building successive levels of abstraction. Although generating a useful representation is not always possible, a successful reconstruction results in an architectural representation that aids in reasoning about the system. This recovered representation is most often used as a basis for redocumenting the architecture of an existing system if the documentation is out of date or nonexistent, and can be used to check the "as-built" architecture against the "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.

This report describes the process of architecture reconstruction using the Architecture Reconstruction and Mining (ARMIN) tool developed by the Carnegie Mellon® Software Engineering Institute and the Robert Bosch Corporation. Guidelines are presented for reconstructing the architectural representations of existing systems. Most of these guidelines are not specific to ARMIN, can be used with other tools, and are useful even if the architecture reconstruction is carried out manually.

® Carnegie Mellon is registered in the U.S. Patent and Trademarks Office.

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 successive levels of abstraction. If the reconstruction is successful, the end results are architectural representations that aid in reasoning about the system. In some cases, however, generating useful representations is not possible due to the complexity and the lack of structure of the system involved.

This is the *third* edition of this technical report, which was originally published in August 2001. This new edition describes a new Architecture Reconstruction and Mining (ARMIN) tool developed by the Carnegie Mellon® Software Engineering Institute (SEI℠) and the Robert Bosch Corporation, and the slightly revised reconstruction process based on it. This edition contains new examples and reconstruction guidelines that reflect the use of this new tool. In addition, the name of the Architecture Reconstruction phase has been changed to the Architectural View Composition phase to better reflect what it involves and to avoid confusion between the phase and the overall architecture reconstruction process.

® Carnegie Mellon is registered in the U.S. Patent and Trademarks Office.
℠ SEI is a service mark of Carnegie Mellon University.

2 Architecture Reconstruction

Architecture reconstruction generates an architectural representation that can be used in several ways. The main use for this representation is to document the existing architecture of a system. If no documentation exists or the available documentation is out of date, the recovered architectural representation can be used as a basis for redocumenting the architecture. Reconstruction can be performed either during the development of an architecture or after the development has been completed to recover the "as-built" architecture of the system to check conformance against the "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 the conformance of a satellite ground station system's implementation to its reference architecture
- reconstruct three embedded automotive systems and evaluate their potential for conversion to a product line
- recover the architecture of several network management systems
- recover the architecture of a satellite-tracking simulation system
- recover the architecture of a Java2 Platform, Enterprise Edition (J2EE) banking application

Other organizations have used the SEI's architecture reconstruction methods as well. A technical note by Liam O'Brien provides details about various projects undertaken at Nokia, some of them using the SEI's methods [O'Brien 02].

Architecture reconstruction 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, and Lex/Yacc) have the necessary skills to undertake architecture reconstruction. However, with the large amount of software in most systems, it is nearly impossible to perform all architecture reconstruction activities manually.

Tool support is needed for these activities, and in general, no single tool or set of tools is adequate. Software systems are often implemented in many languages and dialects. For example, a mature MRI scanner easily contains software written in 15 different languages. Because of this diversity, there is no complete, universally applicable tool set that can operate with the push of a button. Instead, a tool set (workbench) is needed to support architecture reconstruction activities.

An architecture reconstruction workbench should be open (i.e., easily accommodate new tools as required) and provide a lightweight integration framework so that new tools added to the set do not impact the existing tools or data unnecessarily. Such a workbench exists in ARMIN, which replaces the Dali Architecture Reconstruction workbench [Kazman 99]. ARMIN still uses the workbench philosophy as it allows for other tools to be used to extract information from the source code and other system artifacts, and as it enables the loading of this information so it can be used in the reconstruction process.

Other examples of reconstruction workbenches 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 ARMIN, the software architecture reconstruction process comprises the following five phases:

1. Information Extraction

   In the Information 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 loading it into ARMIN.

3. View Fusion

   The View Fusion phase combines information stored in the ARMIN database to generate a set of low-level views.

4. Architectural View Composition

   In the Architectural View Composition 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. Architecture analysis is not addressed in this report, but is the topic of *ATAM: Method for Architecture Evaluation* [Kazman 00]. Current and complete details about architecture analysis can be found in the book *Evaluating Software Architectures: Methods and Case Studies* [Clements 02].

In addition to the person carrying out the actual reconstruction, at least one person is needed who is familiar with the system being reconstructed (e.g., an architect or software engineer).

The reconstructor extracts the information from the system and, either manually or using tools, generates views of the architecture. The reconstructor begins by generating 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 results. 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 the system.

2.1 Recommendations for Reconstruction Projects

The following are general recommendations for reconstruction projects:

- Have a goal and a set of objectives or questions in mind before undertaking an architecture reconstruction project. For example, reusing part of the system in a new application may be a goal. Without these goals and objectives, a lot of effort could be spent on extracting information and generating architectural views that may not be helpful or serve any purpose.

- Obtain a high-level architectural view of the system before beginning the detailed reconstruction process. This view guides the

  - extraction process by helping to identify the information that needs to be extracted from the system
  - reconstruction process by helping to determine what to look for in the architecture and what views to generate

- Use the existing documentation to generate only high-level views of the system. In many cases, the existing documentation for a system may not accurately reflect the system as it is implemented, but it should still give an indication of the high-level concepts.

- Involve the people who are familiar with the system early in the project to get a better understanding of the system being reconstructed. Tools can support the reconstruction effort and shorten the reconstruction process, but they cannot perform an entire reconstruction automatically. Architecture reconstruction requires the involvement of people (e.g., architects, maintainers, and developers) who are familiar with the system.

- Assign someone to work on the architecture reconstruction project full-time. Architecture reconstruction involves an extensive, detailed analysis of a system and requires significant effort.

The following sections describe the architecture reconstruction process in more detail and present guidelines that can be used to carry out each phase. Most of these guidelines are not specific to ARMIN, can be used with other tools, and are useful even if the architecture reconstruction is carried out manually.

3 Information Extraction Phase

The Information Extraction phase involves analyzing the existing design and implementation artifacts of a system to construct a model based upon multiple source views. From the source artifacts (e.g., code, header files, build files) and other artifacts (e.g., execution traces) of the system, the elements of interest and the relations between them can be identified and captured to produce several fundamental views of the system. Table 1 shows a list of typical elements and several relations between 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	C preprocessor #include of one file by another
File	contains	Function	definition of a function in a file
File	defines_var	Variable	definition of a variable in a file
Function	calls	Function	static function call
Function	access_read	Variable	Read access on a variable
Function	access_write	Variable	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, showing how the various functions in the system interact. The "includes" relation between files shows the dependence view between files in the system. The "access_read" and "access_write" relations 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 relation information is used to determine how data is passed between various parts of the system. For example, it can determine whether 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 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" can help to identify components later in the reconstruction process.

The set of elements and relations extracted will depend on the type of system being analyzed and the extraction support tools 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 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 fusing is discussed in Section 5.) 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. The runtime configuration should be captured and used when carrying out the reconstruction.

A source view can be extracted by applying 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 the following:

- parsers (e.g., Understand for C/C++/Java, Imagix, SNiFF+, C++ Information Abstractor [CIA], rigiparse)
- abstract syntax tree (AST)-based analyzers (e.g., Gen++, Refine)
- lexical analyzers (e.g., Lightweight Source Model Extractor [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. Analysis tools can be built 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 lexical patterns to be matched and the elements to be output. An example of a lexical pattern would be one 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 name of the file within the angle brackets (< >). Finding and extracting instances of this lexical 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 lexical 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 lexical 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. Using them does not usually require the addition of 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 make it output some specific information (e.g., what processes connect with each other at runtime) while the system is executing [McCabe 02]. All of these tools and techniques 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 can be extracted *statically* from source code, compile-time artifacts, and design artifacts. However, this may not produce enough information 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 might 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 reside in their source artifacts and hence cannot be reverse engineered using static extraction tools.

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

3.1 Guidelines

The following guidelines apply to the Information Extraction phase:

- Use the "least effort" extraction. Consider the kind of information that needs to be extracted 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 reconstruction goals are simple.

Table 2: Guiding Principles for Choosing Types of Extraction

Guiding Principles	Type of Extraction Required
The information that is to be extracted is lexical in nature. A set of lexical patterns can be written that allows the information to be extracted.	Lexical analysis. (Simple lexical analysis utilities such as Perl and Grep may be of use.)
The information that needs to be extracted cannot be identified lexically. Elements and relations can be identified through the use of a grammar for a language.	Parsing
More contextual information (semantic information) must be available to clearly identify certain elements and relations.	AST-based analyzers. (These allow an AST to be built and updated after parsing with semantic information.)

- Validate the source information that is extracted. 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. Also make sure 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 with 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 a lot of runtime or late binding occurs 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, use it to extract runtime information. If the system runs on a platform where no profiler is available, it might be necessary to instrument the code to obtain the runtime information. When dynamic information cannot be extracted, only static information will be available for generating architectural representations.

4 Database Construction Phase

The set of extracted views are converted into Rigi Standard Format (RSF) or the Graphical eXchange Language (GXL) format and loaded into ARMIN during the Database Construction phase. This conversion is done using Perl scripts that read the data and convert it into a file in RSF. The extracted views may be in many different formats depending on the tools used to extract them. For example, an extraction tool, such as Understand for C/C++/Java or Imagix-4D, can be used to load the source code of a system into its internal representation, and this information can then be dumped to a set of flat files indexed by file or function. These files have a uniform structure, and scripts can be developed in Perl to read these files and output information about elements and relations. Figure 1 depicts this process.

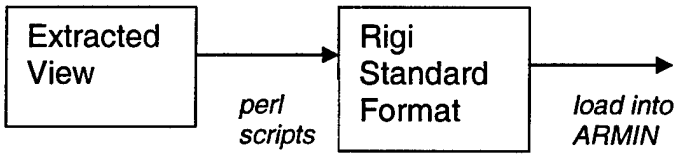


Figure 1: Conversion of the Extracted View to RSF

Once the elements and relations file (the extracted view) is converted to RSF or GXL, it can be loaded into ARMIN. Figure 2 shows an excerpt from a sample RSF file. The entire file could be loaded into a new database in ARMIN.

```
contains String.BaseNodePtr.Map.cc StringBaseNodePtrMap::clear
contains String.BaseNodePtr.SplayMap.cc StringBaseNodePtrSplayMap::leftmost
calls StringBaseNodePtrSplayMap::OK StringBaseNodePtrSplayMap::leftmost
calls StringBaseNodePtrSplayMap::OK StringBaseNodePtrSplayMap::succ
calls StringBaseNodePtrSplayMap::OK StringBaseNodePtrSplayMap::succ
calls StringBaseNodePtrSplayMap::OK StringBaseNodePtrMap::error
defines_var Application::ImportMapping Application::ImportMapping//pmMapping
defines_var Application::DrawTree Application::DrawTree//prend
includes String.BaseNodePtr.Map.cc String.BaseNodePtr.Map.h
```

Figure 2: Excerpt from a Sample RSF File

In addition to the tools and techniques used with ARMIN, new ones can be created to convert the format or formats 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 RSF and loaded into ARMIN. The conversion tool used can become part of the reconstruction workbench.

In the current version of ARMIN, the Architecture Reconstruction Language (ARL) provides commands for generating and manipulating the architectural views. (Examples are shown in Section 4.)

4.1 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 so the query will not need to be run again. If the results of that query are required for building further queries, they can be accessed through the table easily.
- As with any database construction, consider carefully the database design before getting started. What will the primary (and possibly secondary) key be? Will any database joins be particularly expensive because they span multiple tables?
- Use Perl, Awk, and similar lexical tools to change the format of data extracted using various tools into the RSF so that ARMIN can use the data. These tools are less expensive in terms of development time and resource utilization than writing more complex tools using other programming languages.

5 View Fusion Phase

In the View Fusion phase, the extracted views are manipulated to create fused views. For example, a static call view might be fused with a dynamic call view. As noted earlier, a static view might not provide all of the architecturally relevant information. In the case of late binding in the system, some function calls might not be identifiable until runtime, so a dynamic call view needs to be generated. 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 examples in Sections 5.1 and 5.2. The first shows the improvement of a static view of an object-oriented system with the addition of dynamic information. The other shows the fusion of several views to identify function calls in a system.

5.1 Improving a View

Consider the two code views shown in Figure 3, which 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 3: Static and Dynamic Data Views

The differences between these views are shaded in Figure 4.

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: The Differences Between Static and Dynamic Views

The dynamic view shows that `List::getnth` is called. However, this method is not included in the static analysis view because it was not identified by the static extraction tool. That shows that the static extraction tool is not perfect, making it necessary to validate the results of the information extraction. 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 architectural view.

In addition, the static extraction shows that the `PrimitiveOp` class has a method called `Compute`. The dynamic extraction results show no such class, but do show classes such as `ArithmeticOp`, `AttachOp`, and `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. To get an accurate view of the architecture, the static and dynamic views of `PrimitiveOp` must be reconciled. To do this, a fusion is performed over the extracted static "calls," dynamic "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 5 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).

Added to Fused View

```
InputValue::InputValue
InputValue::~InputValue
List::List
List::~List
List::getnth
```

Not Added

```
ArithmeticOp::Compute
AttachOp::Compute
 . . .
StringOp::Compute
```

Figure 5: Items That Were Added to and Omitted from the Overall View

5.2 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` to which there might be calls. 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 the names of procedures, methods, and other named elements unique, allowing them to be referred to unambiguously in the architecture reconstruction process. Without the view fusion, name clashes would persist, and the reconstruction results would be ambiguous.

5.3 Guidelines

The following guidelines apply to the View Fusion phase:

- Fuse views when no single view provides the information needed for architecture reconstruction. For example, the calls view needs to show the functional decomposition of the system. If a static calls view and a dynamic calls view are present, they are fused to produce a single calls view that shows the decomposition.
- Fuse views when there is ambiguity within a view and a single view does not provide clear information.
- Consider using different extraction techniques to extract different view information. For example, both dynamic and static extraction techniques are available. Different instances of the same kind of technique can be used if a single instance might provide erroneous or incomplete information. For example, use different parsers for the same language if each provides different information.

6 Architectural View Composition Phase

The Architectural View Composition phase consists of two primary activity areas:

- visualization and interaction
- command script definition and interpretation

The visualization and interaction area provides a mechanism that allows the user to visualize, explore, and manipulate views interactively. The Aggregator component of ARMIN is used to present views to the user as a hierarchically decomposed graph [Wong 94]. An example presentation of an architectural view is shown in Figure 6. Using the Aggregator, the user can see views in a variety of layout styles including hierarchical, spiral, and orthogonal.

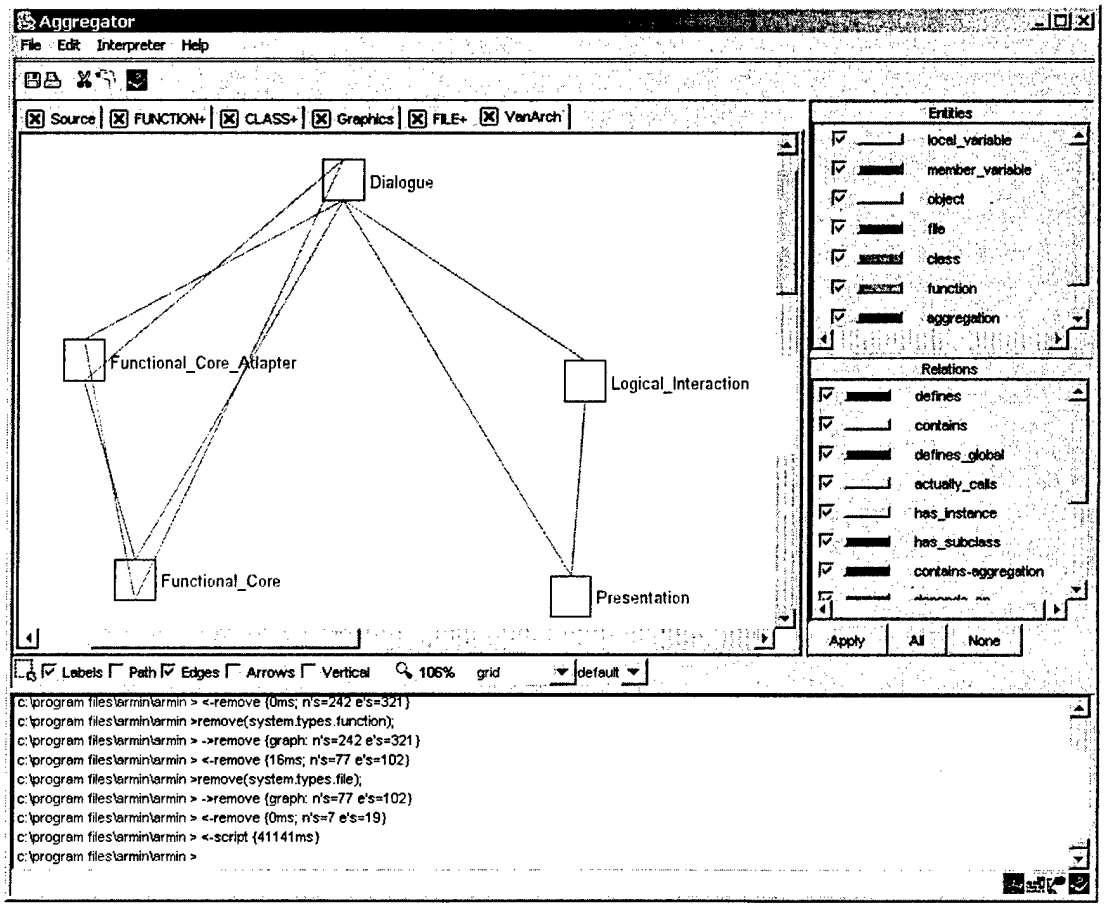


Figure 6: An Architectural View of a System Presented in ARMIN

The command script definition and interpretation activity area provides facilities for abstracting the low-level information to generate architectural views. The command scripting facilities allow a user to write scripts to construct more abstract views from more detailed ones by identifying aggregations of elements. ARMIN scripts are written using the ARL and any editor, and are then loaded into ARMIN.

Architecture reconstruction is not a straightforward process. Architectural constructs are not represented explicitly in the source code, making reconstruction especially difficult. 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 architectural elements are "reconstructed," the inverse of the mappings needs to be applied.

Architecture reconstruction is an interpretive, interactive, and iterative process, 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 command scripts using ARMIN. These scripts 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 scripts 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 the users, so the goals of the reconstruction can be achieved.

Consider the subset of elements and relations shown in Table 3.

Table 3: Subset of the Elements and Relations

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

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 7.

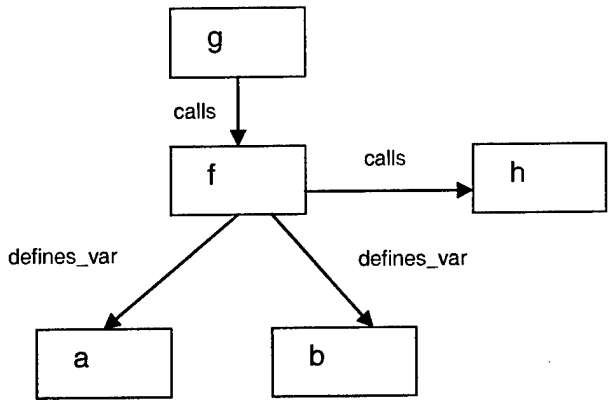


Figure 7: Graphical Representation of Elements and Relations

The local variables are not important during an architecture reconstruction because they provide little insight into the architecture of the system. Therefore, instances of local variables can be aggregated to the functions in which they occur. A script such as the one shown in Figure 8 can be written for this purpose.

```
#collapse the function's local_variables
$c = desc(system.types.function);
$c.merge(/ext="+");
collapse($c,/graph="FUNCTION+",/type=system.types.function);
show();
```

Figure 8: Script for Aggregating Local Variables to the Function in Which They Are Defined

The first line is a comment as it begins with a pound sign (#). The second line gets the descendants of a function (using the `desc` command)—in this case, local variables. The `desc` command returns a three-dimensional array of the functions and their local variables. The third line merges the function name with the local variables for each function and adds a plus sign (+) to the end of the name. Using the `collapse` command, the fourth line removes from the graph all the local variable names and leaves only the function name with the +. This new graph is named FUNCTION+. The last line displays the graph in the Aggregator window using the `show` command.

The result of applying the script is represented graphically in Figure 9. Most command scripts in ARMIN are developed in a similar manner.

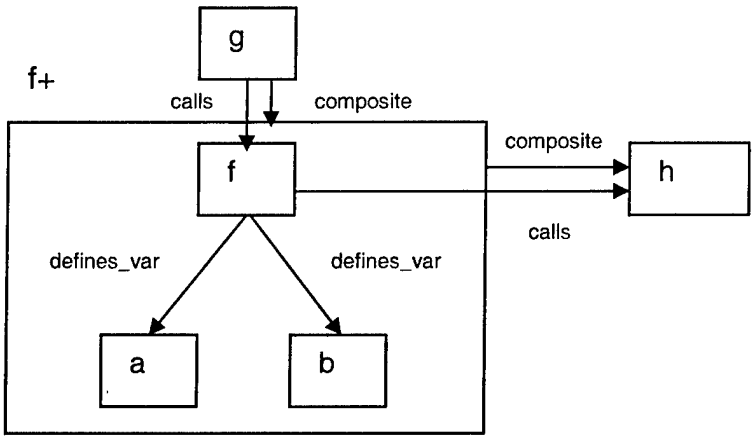


Figure 9: Result of Applying the Script to Aggregate Local Variables

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

- identify types
- aggregate local variables with functions
- aggregate members with classes
- compose architecture-level elements

An example of a script that identifies an architectural-level component, `Logical_Interaction`, is shown in Figure 10. This script says that if the class name is `Presentation`, `Bspline`, or `Color`, or if the class is a subclass of `Presentation`, it belongs in the `Logical_Interaction` component.

```
#create Logical_Interaction component
#all classes that are subclasses of Presentation
$log = {{{"Logical_Interaction"},
         {"Presentation++","BSpline++","Point++","Color++","interpolate+",
          desc(system.types.has_subclass,"Presentation++",/dim=1,/grade=1) }
       }};
$comps.append($log);
```

Figure 10: Query to Identify the Logical_Interaction Component

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

6.1 Guidelines

These guidelines apply to the Architectural View Composition phase:

- Be prepared to work with the architect closely and to iterate several times on the architectural abstractions that are created. This is particularly important in cases where the system has no explicit, documented architecture. In such cases, architectural abstractions can be created as hypotheses, and these hypotheses can be tested by creating the views and showing them to the architect and other stakeholders. Based on the false negatives and false positives found, the architect may decide to create new abstractions, resulting in new ARMIN scripts to apply (or perhaps even new extractions that need to be done).
- When the reconstructor is not familiar with the technology the system uses, he or she needs to work with someone who is.
- When developing scripts, try to build ones that are succinct and do not list every source element. The script shown in Figure 10 is an example of a good one; an example of a bad script is shown in Figure 11. In the bad one, the source elements that comprise the component are simply listed, which makes the script difficult to use and understand especially if a lot of element names are listed. It also makes the script difficult to reuse as the names of the elements in this component would not match those in any other components.
- Scripts can be based on naming conventions if the naming conventions are used consistently throughout the system. For example, a naming convention could specify that all functions, data, and files that belong to the Interface component be given names that begin with the letter i and an underscore (i_). Writing a script that abstracts elements with such names is straightforward.
- Scripts can be based on the directory structure where files and functions are located. Component aggregations can be based on these directories.
- Architecture reconstruction is the effort of redetermining architectural decisions, given only the results 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 reintroduce the architectural decisions. This process introduces bias from the reconstructor, thus reinforcing the need for involvement by an architecture expert.
- Reconstructing an entire system might be unnecessary if the goal of the reconstruction is to determine whether the system uses certain architectural styles or tactics [Bachmann 03]. According to Stoermer and associates, architecture reconstruction is linked closely to an analysis framework for various system quality attributes, and those attributes are achieved through the use of architectural styles or tactics [Stoermer 03].

```
#create Graphics component
#all of the following list of classes
$gfx = {{{"Graphics"},
         {" "BlackPixel+","DefaultColormap+","DefaultDepthOfScreen+",
            "DefaultScreen+","GLwMakeCurrent+","WhitePixel+",
            "auxSolidSphere+", "auxWireSphere+","auxSolidTetrahedron+",
            "aux WireTetrahedron+", "tkDrawStr+","fclose+","fflush+",
            "fopen+","fprintf+","printf+","return+","strcpy+","strlen+"+"}
       }};
$comps.append($gfx);
```

Figure 11: Example of a Bad Script

7 Other Architecture Reconstruction Approaches

This section explores other approaches for architecture analysis.

7.1 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 information (elements in Dali) 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. The resulting structure was refined by moving source files between subsystems.

Unlike the approach used in Dali, this one is primarily manual. The reconstructor carries out subsystem and component identification by manually selecting source file elements to belong to these views. Dali is more automated, so queries can be written to carry out these tasks. The first step in Bowman and associates' approach was to develop a conceptual architecture. This step is not part of the phases of using Dali outlined earlier, but developing a conceptual architectural view with the help of the developers, maintainers, or the architecture is certainly part of the overall approach when Dali is used. This conceptual architectural view 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, the lower level elements that comprise those components become visible, and by selecting a link between two components, the relations represented become visible. Bowman's approach does not appear to provide this level of interaction.

7.2 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: 1) the architectural representation, 2) the source code recognition engine and supporting library of recognition queries, and 3) 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 on a set of implementation language independent queries that are applied to an 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 also specific for each programming language (i.e., AST specific). Thus if a new language needs to be handled, 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. Using Dali, 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's approach does provide some metrics information about the amount of code covered by particular architectural styles in the system, which may be useful for maintenance and reengineering purposes. For example, if a particular architectural style in the system has to be changed or reimplemented, it is possible to get an idea of how big the problem will be. This type of information is not provided in the Dali workbench.

7.3 Guo and Associates

Guo and associates outline the semi-automatic architecture recovery method (ARM) that 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 architecture recovery work. An abstract pattern rule was then mapped into a concrete pattern rule and converted into an SQL query. This query was then applied to the database to extract instances of the pattern. The Guo method is intended for use on systems that have been developed using design patterns, limiting its applicability. It can only be used with systems that were developed using design patterns or in cases where the design pattern implementations have not eroded over time.

8 Summary

Four major phases of architecture reconstruction were outlined in this report:

- Information Extraction
- Database Construction
- View Fusion
- Architectural View Composition

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

In our work at the SEI, we have used ARMIN and Dali to support the reconstruction efforts on several systems in a wide variety of domains. One reason why ARMIN and Dali have been very useful is that both are language independent. They can be used to analyze information from many different languages, systems, and domains. ARMIN replaces the Dali workbench and will continue to be applied on new projects.

References

[Bachmann 03] Bachmann, F.; Bass, L.; & Klein, M. *Deriving Architectural Tactics: A Step Towards Methodical Architectural Design* (CMU/SEI-2003-TR-004, ADA413644). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. <http://www.sei.cmu.edu/publications/documents/03.reports/03tr004.html>.

[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.; & 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.

[Clements 02] Clements, P.; Kazman, R.; & Klein, M. *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA: Addison-Wesley, 2002.

[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, TX, 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, WA, 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). Pittsburgh, PA.: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports /00tr004.html>.

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

[McCabe 02] McCabe & Associates, Inc. McCabe IQ2 Suite. <http://www.mccabe.com> (2002).

[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.

[O'Brien 02] O'Brien, L. *Experiences in Architecture Reconstruction at Nokia,* (CMU/SEI-2002-TN-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <http://www.sei.cmu.edu/publications/documents/02.reports /02tn004.html>.

[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, CA: IEEE Computer Society Press, 1998.

[Stoermer 03] Stoermer, C.; O'Brien, L.; & Verhoef, C. "Moving Towards Quality Attribute Driven Software Architecture Reconstruction," 46-56. *Proceedings of the Working Conference on Reverse Engineering.* Victoria, British Colombia, Canada, November 13-16, 2003. Los Alamitos, CA: IEEE Computer Society Press, 2003.

[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

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (LEAVE BLANK)	2. REPORT DATE November 2003	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Architecture Reconstruction Guidelines, Third Edition		5. FUNDING NUMBERS 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-2002-TR-034
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-TR-2002-034
11. SUPPLEMENTARY NOTES		
12.A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS		12.B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

Architecture reconstruction is the process of obtaining the "as-built" architecture of an implemented system from the existing legacy system. For this process, tools are used to extract information about the system that will assist in building successive levels of abstraction. Although generating a useful representation is not always possible, a successful reconstruction results in an architectural representation that aids in reasoning about the system. This recovered representation is most often used as a basis for redocumenting the architecture of an existing system if the documentation is out of date or nonexistent, and can be used to check the "as-built" architecture against the "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.

This report describes the process of architecture reconstruction using the Architecture Reconstruction and Mining (ARMIN) tool developed by the Carnegie Mellon® Software Engineering Institute and the Robert Bosch Corporation. Guidelines are presented for reconstructing the architectural representations of existing systems. Most of these guidelines are not specific to ARMIN, can be used with other tools, and are useful even if the architecture reconstruction is carried out manually.

14. SUBJECT TERMS architecture representation, architecture reconstruction, architecture reengineering	NUMBER OF PAGES 42
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