Reverse Engineering to the Architectural Level[*]

David R. Harris[◇], Howard B. Reubenstein[†], Alexander S. Yeh[◇]
[◇]The MITRE Corporation, 202 Burlington Road, Bedford, MA 01730, USA
[†]GTE Laboratories, 40 Sylvan Road, Waltham, MA 02254, USA
drh@mitre.org, hbr@gte.com

Abstract

Recovery of higher level "design" information and the ability to create dynamic, task adaptable software documentation is crucial to supporting a number of program understanding activities. This paper presents research that demonstrates that reverse engineering technology can be used to recover software architecture representations of source code.

We have developed a framework that integrates reverse engineering technology and architectural style representations. Using the framework, analysts can recover custom, dynamic documentation to fit a variety of software analysis requirements. Our goal is to establish coherent abstractions appropriate for helping analysts to understand large software systems. We discuss a code coverage metric useful for assessing the degree of program understanding achieved.

1 Introduction

This paper presents research that demonstrates that reverse engineering technology can be used to recover representations of software systems at a higher level than the typical detailed design documentation level (e.g., set-use or structure chart reports). To achieve this, we combine software architectural representations and reverse engineering technology to explore whether architectural representations can be recovered from the source code.

Recovery of higher level "design" information and the ability to create dynamic, task adaptable software documentation is crucial to supporting a number of program understanding activities. The problem with conventional paper documentation is that analysts cannot tailor it for each of the wide range of tasks that a software maintainer or developer might wish to perform, e.g., general maintenance, operating system port, language port, feature addition, program upgrade, or program consolidation. A dynamic form of documentation that can be flexibly generated from existing source code and that can evolve to support emergent tasks is needed. This need can be filled by recovering architectural representations of the source code.

Commercially available reverse engineering tools [1] provide a set of limited views of the source under analysis. They share the problem with paper documentation that they present static abstractions of the code; however, these views are an improvement over detailed paper designs in that they provide accurate information derived directly from the source code. The commercial tools also provide source code navigational capabilities ameliorating source code access problems. However, the views these tools present still have a fixed information content across systems they are capable of analyzing, i.e., typically systems written in a particular source language.

We have developed a framework that integrates reverse engineering technology and architectural style [2] representations. Using the framework, analysts can recover custom, dynamic documentation to fit a variety of software analysis requirements. The representation of styles provides knowledge of software design beyond that defined by the syntax of a particular language.

Our goal is to establish coherent abstractions appropriate for helping analysts to understand large software systems. Specifically, we would like to answer the following questions:

- When are specific architectural commitments actually present?

[*]This work has been funded by The MITRE Corporation under its internal research program.

- What percent of the code is used to achieve an architectural commitment?
- Where does any particular code fragment fall in an overall architecture?

We argue that it is practical and effective to automatically (sometimes semi-automatically) discover architecture notions embedded in legacy systems.

The paper describes our overall architecture recovery framework, the difficulties in bridging the gaps to software understanding, a preliminary notion of code coverage metrics, and the results of applying our recovery technique to a moderately sized (30,000 lines of code) system.

2 Architecture Recovery Framework

Our motivation for building this framework stems from our efforts to understand legacy software systems. While it is clear that every piece of software conforms to some design, it is often the case that existing documentation provides little clue to that design. For example, while the system block diagram portrays an "idealized" software architecture description, it typically does not even hint at the source level building blocks required to construct the system. In contrast, the primary purpose of our framework is to recover the "as-built" architecture - a description of the architectural structures that actually exist in the code.

As-built architectures differ from idealized architectures. Even when idealized architectures do commit to particular connections (e.g., pipes or application progamming interfaces), actual systems contain source code artifacts (e.g., procedure parameter passing or use of Unix pipes) to achieve the connections. Other differences are due to architectural violations. Reasons for such violations are varied. Some are due to a developer's failure to honor commitments or failure to understand the entailments of an architectural commitment. Other violations are due to the inability of an existing or required framework (e.g., language, host platform, development tools, or commercial enabling software) to adequately support the idealized view and thus may occur with the earliest engineering decisions.

Our recovery framework (see Figure 1) is made up of three components:

- an architectural representation that supports both idealized and as-built architectural representations with a supporting library of architectural styles and style components
- a source code recognition engine and a supporting library of recognition queries
- a "Bird's Eye" program overview capability

The framework supports architectural recovery in both a bottom-up and top-down fashion. In bottom-up recovery, analysts use the bird's eye view to display the overall file structure and file components of the system. The features we display (see Figure 2) include file type (diamond shapes for executable; rectangles for non-executable), name, pathname of directory, number of top level forms, and file size (indicated by the size of the diamond or rectangle). Since file structure is a very weak form of architectural commitment, only shallow analysis is possible; however, the files are a place where information (including results of progress toward recognition of other styles) is registered. The one analysis option we provide is the detection of cluster dominance. A cluster X dominates a cluster Y when some procedure in X invokes a procedure from Y, while no procedure in Y invokes a procedure in X. This analysis supports an analyst in reorganizing the information into more meaningful clusters based on the dominates relation.

In top-down recovery, analysts use architectural styles to guide a mixed-initiative recovery process. Architectural styles initially represent an idealized view of the system. There is consensus in the literature that an architectural representation consists of components, connectors, and constraints [3, 2, 4]. In contrast to program synthesis approaches (e.g., DSSA [5]), our use of styles is to support architectural analysis, i.e., recovery. From our point of view, the semantics of an architectural style place an expectation on what will be found in the software system. The style establishes a set of recognition commitments which define components to be found in the software. Recognition queries are used to satisfy these recognition commitments. Once the style is recognized, the mapping from the style to its realization in the source code forms the as-built architecture of the system. Note that in this process, we preserve the links from the architectural description to the source code that implements the design.

We expect that multiple styles will be found in a target software system. Each recognized style provides a view of the system that partially explains the software. The collection of these views partially recovers the overall design of the system. In addition, we use these multiple views to define a code coverage metric (described in Section 4) that informs an analyst about the percentage of the code that implements architectural commitments.

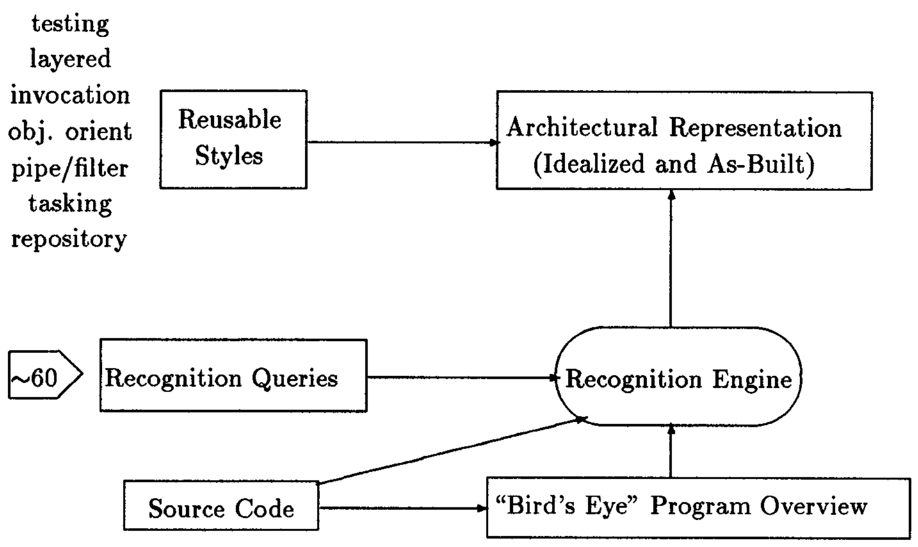

Figure 1: Architectural Recovery Engine

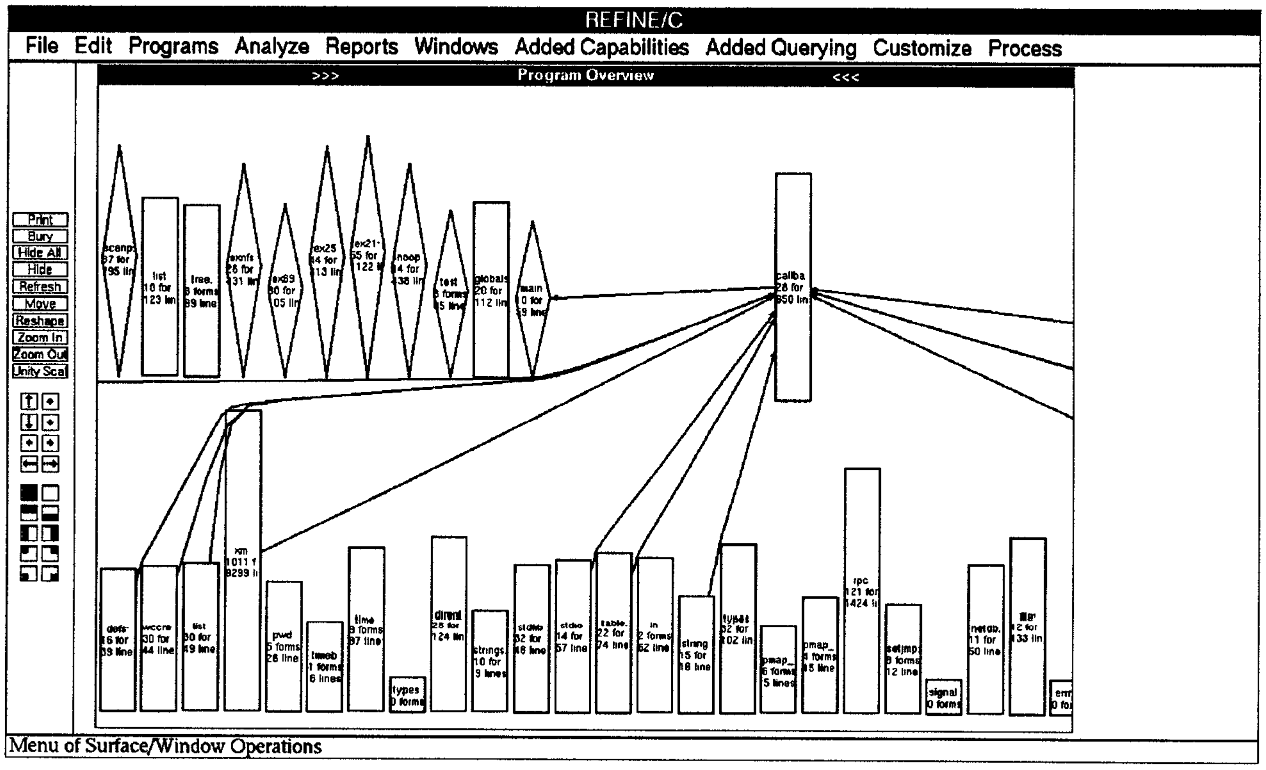

Figure 2: Bird's Eye Overview

2.1 Architectural Styles

The research community has provided detailed examples [6, 7, 8, 2] of idealized architectures - abstractions that can guide architectural discovery. We have attempted to recognize instances of these abstractions in source code.

2.1.1 Entity and Relation Representation

To support recognition, we have developed a dictionary of entity and relation terminology within the software architecture domain. Entities include: clusters, layers, processing elements, repositories, objects, and task-modules. Entities are typically implemented as *components* - "large" segments of the source code (e.g., a layer may be implemented as a set of procedures). Relations such as contains, initiates, spawns, and is-connected-to describe how the components are linked. Object participation in a relation follows from the existence of a *connector* - a very specific code fragment (e.g., special system invocations) or the infrastructure that handles these fragments. This infrastructure may or may not be part of the body of software under analysis. For example, it may be found in a shared library or it may be part of the implementation language itself.

Figure 3 details the representations for task-module entity and the spawns relation. Task-modules are a kind of processing element that might be implemented by objects of type file (more generally by collections of procedures linked via limited entry points) and are recognized by a query named "executables". Spawns relates task-modules to task-modules (i.e., parent and child tasks respectively). Spawns might be implemented by objects of type system-call (e.g., in C, a "system" call that starts a new process via a shell command) and recognized by a query named "find-executable-links"

We use entity and relation taxonomies to provide analysts with architectural discovery guidance. For example, when analysts find few examples of an architectural entity within legacy code, they will have the option of expanding a search by following generalization and specialization links and searching for related information. This capability complements the query indexing scheme based on code level vocabulary (described in Section 2.2).

2.1.2 The Style Library

Our styles are expectations about the components and connectors that will be found together in software that actually exhibits the style. The styles we have codified are: pipe-and-filter, interacting objects, abstract-data-type, implicit-invocation, transparent-layered, opaque-layered, API-users, optimizing-algorithm, task-spawning, and repository. Four of these styles indicate the extent of coverage.

```
defentity TASK-MODULE
 :specialization-of processing-element
 :possible-implementation file
 :recognized-by executables

defrel SPAWNS
 :specialization-of initiates
 :possible-implementation system-call
 :recognized-by find-executable-links
 :domain task-module
 :range task-module
```

Figure 3: Example Style Components

Task Spawning: In a task spawning architecture, task-modules (i.e., executable processing elements) are linked when one task-module initiates a second task-module. Task spawning is a style that is recognized by the presence of its connectors (i.e., the process invocations). Its components are task-modules, repositories, and tasks. Its connectors are spawns (relating task-modules to task-modules), spawned-by (the inverse of spawns), uses (relating task-modules to any task-modules with direct interprocess communications and to any repositories used for interprocess communications), and conducts (relating task-modules to functional descriptions of the tasks performed).

Layered: In a layered architecture the components (layers) form a partitioning of a subset, possibly the entire system, of the program's procedures and data structures. As mentioned in [6], layering is a hierarchical style: the connectors are the specific references that occur in components in an upper layer and reference components that are defined in a lower layer. One way to think of a layering is that each layer provides a service to the layer(s) above it. A layering can either be opaque: components in one layer cannot reference components more than one layer away, or transparent: components in one layer *can* reference components more than one layer away.

Data Abstractions and Objects: Two related ways to partially organize a system are to identify its abstract data types and its groups of interacting objects [9, 6]. A data abstraction is one or more related data representations whose internal structure is hidden to all but a small group of procedures, i.e., the procedures that implement that data abstraction. An object is an entity which has some persistent state (only directly accessible to that entity) and a behavior that is governed by that state and by the inputs the object receives. These two organization methods are often used together. Often, the instances of an abstract data type are objects, or conversely, objects are instances of classes that are described as types of abstract data.

2.2 Query Library

At the style level, it is relatively straightforward to express invariants on entities and relations. For example, spawns is a specialization of the initiates relation, i.e., spawns(x,y) iff initiates(x,y) and process(y). Ideally, this definition could be used to either recognize implementations of the concept or to check to see if code fragments are related in a form dictated by the style. However, rather than directly recognizing style level invariants in the code (a difficult task), we link styles to appropriate queries or families of queries expressed in a code level vocabulary. These links capture a default set of queries that return implementation objects satisfying the invariants and also provide user support by focusing on a family of queries any of which will return objects that may implement the style concept.

Our query library contains approximately 60 queries. We developed each query to satisfy some requirement to retrieve implementations of style level entities and relations (e.g., calls to particular functions implementing the spawning of tasks). In addition to serving as default recognizers of architectural style components, queries support a free-form discovery mode to indirectly help analysts discover what architectural structures are present.

2.2.1 Source Code Query Language

The style recognition routines are written in a query language for writing routines to analyze a program which has been parsed into a graph called an abstract syntax tree. The language is similar to Refine [10] but has some features not found in Refine. These queries can either return a set of abstract syntax tree nodes representing the code fragments of interest, or a set of sequences where each sequence contains one of those nodes and some additional information on that node. Figure 4 shows an example query. This query finds all the procedure `calls` that invoke a system service. Then for each call, the query finds the corresponding node that represents the invoked service (`target`) and the procedure the call is in (`layer`). The query then returns a set of sequences where each sequence contains a call and its corresponding invoked service and enclosing procedure.

```
defquery FIND-PORT-CONNECTIONS
   :effects know-network-call, know-service
   :desc "LINKS between program layers
          and local or network services"
   :method "
 let (invocations = {})
  (for-every call in
   invocations-of-type('service-invocations)
    do
    let (target =
         service-at(get-port-nbr(call)))
     let (layer = enclosing-procedure(call))
      if ~(target =  undefined) then
       invocations <-
        prepend(invocations,
                [call, layer, target]));
  invocations"
```

Figure 4: Example Query

```
defcalls SERVICE-INVOCATIONS
 % arg-nbr gives location for input structure
 % to the connect or sendto call. Approach
 % is to find the assignments that set the
 % sin_port attribute of that structure.
 :call-desc "Service Invocations"
 :call-type service-invocation
 :call-ref-names "connect", "sendto", "bind"
 :port-arg-nbrs 2, 5, 2
 :port-attribute-name sin_port
```

Figure 5: Example Call Specification

One query indexing scheme implemented is *retrieval by effect*. The result of running a query may be that some part of source code has been annotated with markers that will support the analyst in additional discovery. Thus we think of the "effects" of running a query. That is, we "know" all the occurrences of some code level cliche since we have our hands on the objects or can review annotations on an abstract syntax tree. As an additional indexing capability we have found that it is very useful to simply retrieve all queries whose description contains a string matching a user-entered text string.

Our goal is to achieve language independence of the queries, but there are residual dependencies. Primarily, there is the dependence of a set of reusable query functions on specifics of the source language under analysis and the parsed program representation. While the queries themselves are language-independent, the names of special function calls and the decoding of argument lists are dependent on the constructs of the language. We encode these dependencies in language-specific call specifications such as the one in Figure 5. In C/Unix, **connect, sendto,** and **bind** calls are all used for connecting to a particular service via a port number. This port number is found as a specific attribute (sin_port) of a data-structure that the programmer references as an argument to the call. The argument number depends on the call type.

2.2.2 Examples

Layering and data abstraction queries indicate the scope of recognition requirements and the needs for interactive recognition processes.

Layering: A layering query finds a partition of the system under analysis. We assume that layering is a potentially loose relation between defined identifiers in a program. The actual layering will depend on ordering constraints imposed by the density of the reference relations between program modules. If procedure X is in a layer above procedure Y, then Y does not refer to X. If the layering is opaque, then X and Y cannot be more than one layer apart for X to be able to reference Y. We place cycles of procedure references (usually due to mutual recursion) in the same layer.

The recognizer only layers procedures, not variables or data-type declarations. When working within our framework, the analyst can modify the resulting layering to more accurately reflect system functional structure. The current algorithm assumes manual intervention to detect language dependent references, such as invocations of an executable via the Unix **system** procedure or access to a common data file via a Unix **read** procedure. Some automatic recognition of these language-specific references is available with other more focused queries and we may be able to use them in this setting as well.

Data Abstractions: The recognizer to recover abstract data types (ADTs) assumes that an ADT is implemented as one or a few structure (record) types whose internal fields are only referenced by the procedures that are part of the ADT. The recognizer constructs a graph with these procedures and structure types as nodes, and the references by the procedures to the internal fields of the structures as the edges. The connected sets of this graph form the set of ADTs. This approach is very useful for discovering families of procedures that manipulate data types (e.g., lists, tables). However, the recognizer often finds connected components that are too large for analyst to think of as ADTs. We have identified a number of causes for such anomolies and have implemented a collection of mixed-initiative supporting mechanisms to help analysts break up large clusters to discover more meaningful ADT views.

3 Bridging the Gaps to System Understanding

There are a number of assumptions and limitations behind the above explanation of the architectural recovery process.

First, how do analysts find applicable styles to match against the system under analysis in order to drive the top-down recovery process? There are a number of clues. Design documentation, while not an accurate index into the code, often gives a statement of design philosophy and technique that points to certain styles. Another clue is clusterings produced by the bird's eye view, layering, or ADT recovery presentations. In each case, clusters of structure may be associated with certain styles. Moreover, there is currently a considerable interest in migrating extant code to more modern paradigms such as object-oriented. In this setting, style selection will remain analyst driven. The analyst will try to determine the code suitable for migrating to the new style. Our work on code coverage is aimed at the predictiveness (i.e., how much of the code does it explain) of a style and should be supportive of such migration efforts.

A second question revolves around style language independence. Our current styles have been motivated by thinking about a C/Unix environment which does have its idiosyncratic programming idioms. However, our styles are phrased at general language independent level, e.g., regarding task spawning or service invocation. Therefore, the applicability of the styles really revolves around the definition of queries.

A final question concerns effective style recovery. We have observed the difficulty in bridging the gap between style level vocabulary and code level vocabulary. We are currently bridging this gap with parallel hierarchies. Style components, style connectors, and syntactically recoverable units are all organized and indexed in parallel hierarchies. Queries are cross referenced to the syntactically recoverable units. Thus, when an analyst determines the type of syntactic unit used by developers to implement a style component, the hierarchies help to focus on a relevant set of queries (i.e., queries that retrieve specializations) and/or broaden a search to consider related types (i.e., queries that retrieve generalizations).

4 Code Coverage

We need to calibrate architectural discovery both to determine the effectiveness of the style representations (e.g., what is the value-added of providing a new style) and to provide an indicator for analysts of how well they have done in understanding the system under analysis.

While it is rare to find systems of any complexity which strictly adhere to idealized commitments, all legacy systems exhibit some architectural aspects if only in the organization of the files and directories, modules, or subroutines of the code.

The measures we provide are potentially subject to some misinterpretation. It is difficult to determine how strongly a system exhibits a style and how predictive that style is of the entire body of code. As an extreme example, one could fit an entire system into one layer. This style mapping is perfectly legal and covers the whole system, but provides no abstraction of the system and no detailed explanation of the components.

In spite of these limits, there are experimental and programmatic advantages for defining code coverage metrics. The maintenance community can benefit from discussion on establishing reasonable measures of progress toward understanding large systems.

For each cluster (file or some other aggregation unit), we report several coverage statistics. *Connector-lines-of-code* estimates the number of lines of code (within a cluster) required for establishing arguments and invoking each connector. For each connector, we compute the size of the backward slice on the arguments of the connector. This allows us to count both the connector (probably a single line of code) and the code required to setup the use of the connector (potentially many lines) in our coverage metric. When only one style is under investigation, a ratio of connector-line-of-code to total lines of code close to 0% indicates the absence of that particular style, while a ratio close to 100% suggests that the style will not be particularly informative since virtually the entire code segment is involved with that single style.

Connector-procedures is the number of procedures that contain some connector. This measure gives an informal notion of the pervasiveness of architectural connections. A relatively large number suggests that developers made substantial commitments to the style. A small number may indicate that either the particular style is mismatched to the actual code or that the code procedures are organized around non-architectural features (e.g., numerical algorithms).

The third coverage measure is *component-procedures* counting the number of procedures that are associated with component-based styles. The approach we use counts all procedures located in any of the components. When a style positions everything into some component, this method can provide an overly confident view of what has been touched. At best, we really only know about restrictions on component interfaces and have information approximately at the level of a system structure chart. In contrast, when the as-built architecture reflects a more coherent view of the system building blocks (e.g., a view obtained when an analyst drops selected procedures/data structures into a semantically coherent buckets), this method can be very informative. A second approach that we will be exploring looks only at those procedures "near" the component interface. For example, in a layered architecture we would count only the callers and callees across layer boundaries ignoring the internal coherence of the layer.

A fourth statistic reported is the overall coverage at the procedural level. It is the percent of procedures counted in either connector-procedures or component-procedures.

5 An Example

During the past year, we employed our architecture discovery framework to discover style commitments in XSN a moderate sized (approximately 30,000 C source lines of code) MITRE-developed network-based program for Unix system management.

This program contains several common C/Unix building blocks and has the potential for matching aspects of multiple idealized styles. It is built on top of the X window system and hence contains multiple invocations of the X application program interface. It consists of multiple executables developed individually by different groups over time. These executables are linked in an executive module that uses system calls to spawn specific tasks in accordance with switches set when the user initiates the program. Each task is a test program consisting of a stimulus, a listener, and analysis procedures. Socket calls implement a client/server architecture for communications between host platforms on the network.

The following items summarize the style commitments we have been able to extract from the source code.

- ADT - interactive recognition of procedures that access structures and global variables.
- API Users - automatic recognition of those procedures that reference library APIs and special calls for connecting to services.
- Layered - partial recognition of layers by collapsing some connections found in a standard structure chart.
- File-based Repository - automatic recognition of procedures that access or modify repositories (specifically, data files).
- Task Spawning - automatic recognition of the executable modules and the specific system calls that are used to connect this modules.

Figure 6 summarizes the amount of code in XSN covered by the various styles. The first row gives the percentage of the lines of code used in the connectors for that style. The second row gives the percentage of the procedures covered by that style:

Combining all the styles whose statistics are given results in a connector coverage of about 3% of the lines of code and over 47% of the procedures. (Procedure coverage total is less than the sum of its constituents in the above table because the same procedure may be covered by multiple styles.)

The method of acquiring and interpreting these numbers is still preliminary. For example, the numbers need to be normalized with respect to a priori expectations. Connector coverage is a limited way to span lines of code and will never include code such as type definitions and procedure headers. Thus, the 3% coverage reported is an underestimate compared to maximum expected results. Procedure percentage does not account for how much of the procedure is understood. Thus 47% is probably an overestimate.

6 Related Work

6.1 Architectural Representation

There has been much work on representations of software architecture, including [6, 3, 2, 5]. This work motivates the use of architectures, finding useful representations, and determining what a particular architectural style tells you about a system. This work does not describe how to recognize styles found in systems under analysis nor how to map source code pieces parts of a style.

Garlan and Shaw [6] describe a large set of architectural styles, including some used here (layering, objects, ADT). Abowd, Allen, and Garlan [3] formalize what an architectural style says about a system written in that style for two styles: pipe-filter and event system.

The Domain Specific Software Architecture program [5] is working to create a number of reusable architectures for specific military domains, i.e., helicopter avionics, command and control, vehicle control and management, and missile control and navigation. The goal of this effort is to support a program generation approach to software construction within the targeted domains.

6.2 Program Recognition Approaches

The reverse engineering and program understanding community has approached software understanding problems but generally with a bottom-up approach where a program is matched to a set of pre-defined plans/cliches from a library. This work is not motivated by the organizational principles of architecture essential for the construction of large systems. Current work on program concept recognition is exemplified by [11, 12] which continues the cliche-based tradition of [13]. Recognition is based on a precise data and control flow match which indicates that the recognized source component is precisely the same as the library template. Our approach is more of a top-down hypothesis driven recognition coupled with bottom-up recognition rules. Our recognition rules do not require algorithmic equivalence of the plan and the source being matched, rather they are based on source code level events [14] in the code. The existence of patterns of these events is sufficient to establish a match. Quilici [15] also explores a mixed top-down, bottom-up recognition approach using traditional plan definitions. The style of source code event-based recognition rules is also exemplified in [11, 12] which demonstrates a combination of precise control and data flow relation recognition and more abstract code event recognition.

Program structure can be analyzed independently to reveal system organization as discussed in [16, 17, 18]. General inquiry into the structure of software can be supported by software information systems such as LaSSIE [19]. Design recovery work, such as DESIRE [16], relies on: externally supplied cues regarding program structure, modularization heuristics, manual assistance and informal information. Informal information and heuristics can also be used

Style	ADT	API	Layered	Repository	Task Spawning
% connector LOC	0	0	0.3	2.2	0.7
% of procedures.	39.3	13.9	3.3	13.1	2.5

Figure 6: Code Coverage Measures for XSN

to reorganize and automatically refine recovered software designs/modularizations. In [17] they have developed a modularization tool using a clustering procedure based on similarity as measured by features shared and not shared. Clustering is not used to find components of any particular style.

Our work on ADT recovery is somewhat similar to work done at the University of Florida SERC [20, 21]. The SERC work detects clusters of procedures that either: (1) deal with the same data-types in their input and output, (2) share data-types in their input and output, or (3) modify items with the same data-types. The first two methods combine procedures that pass instances of an ADT along with procedures that manipulate the internal representation of an ADT. The third method separates procedures that modify the internal representation of instances of an ADT from the procedures that just examine the internal representation of instances of an ADT. Our own work has enabled us to qualify the effectiveness of this approach. We have identified limitations of the recognition procedure and which items in a program under analysis may cause problems (and thus should be excluded) and why. Both SERC's and our work also deal with object instance recovery.

7 Project Status

The tools we describe are implemented on top of Reasoning System's Refine/C reverse engineering workbench. The architectural recovery framework is layered on top of an in-house source code query mechanism that provides flexible access to the underlying Refine program representation. Our experiments with XSN continue as we define more styles and refine our recognition algorithms. One of our goals is to move our percentage of code covered as close to 100% as possible and, failing that, be able to characterize the code we could not recognize.

There are three areas in which we intend to extend our work:

- COTS modeling: Systems that we wish to analyze do not always come with the entire body of source code, e.g., they may make use of COTS (commercial off-the-shelf) packages that are simply accessed through an API. From the analysis point of view, Unix is a COTS package. We have developed representations for COTS components that allow us to capture the interface and basic control and dataflow dependencies of the components. This modeling needs to be extended to represent architectural invariants required by the package.

- Define architecture independent of source code: Although the focus of this paper revolves around strongly binding idealized architecture descriptions to actual source code, there is a crucial need to be able to independently describe the software architecture of the system. One crucial use of this definition would be as a form of design specification against which implementations could be tested for conformance. If we can define the architecture, we can specify the software we want built in a way which will better guarantee required evolvability, portability, and interface requirements.

- User requirements modeling: To answer the important software maintenance question: "Where is this implemented?" we need a representation for the referrants of "this." For example, a user may want to ask where message decoding is implemented. Message and decoding are concepts at the user requirements level. We need to consider an additional level of modeling that will permit partial modeling of the conceptual entities in the user's model of the software. This capability will also help in answering the related question of "Why is this code here?"

Continuing work in architectural representation provides an important source of expectations and style definitions to drive our recovery framework.

References

[1] M. Olsem and C. Sittenauer. Reengineering technology report. Technical report, Software Technology Support Center, 1993.

[2] D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM Software Engineering Notes*, 17(4), 1992.

[3] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. *ACM Software Engineering Notes*, 18(5), 1993. Also in *Proc. of the 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering,* 1993.

[4] W. Tracz. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *ACM Software Engineering Notes*, 19(2), 1994.

[5] E. Mettala and M. Graham. The domain specific software architecture program. Technical Report CMU/SEI-92-SR-9, SEI, 1992.

[6] D. Garlan and M. Shaw. An introduction to software architecture. *Tutorial at 15th International Conference on Software Engineering*, 1993.

[7] M. Shaw. Larger scale systems require higher-level abstractions. In *Proceedings of the 5th International Workshop on Software Specification and Design*, 1989.

[8] M. Shaw. Heterogeneous design idioms for software architecture. In *Proceedings of the 6th International Workshop on Software Specification and Design*, 1991.

[9] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1984.

[10] Reasoning Systems, Inc., Palo Alto, CA. REFINE *User's Guide*, 1990. For REFINE^{TM} Version 3.0.

[11] W. Kozaczynski, J. Ning, and T. Sarver. Program concept recognition. In *7th Annual Knowledge-Based Software Engineering Conference*, 1992.

[12] A. Engberts, W. Kozaczynski, and J. Ning. Concept recognition-based program transformation. In *1991 IEEE Conference on Software Maintenance*, 1991.

[13] C. Rich and L. Wills. Recognizing a program's design: A graph parsing approach. *IEEE Software*, 7(1), 1990.

[14] M. Harandi and J. Ning. Knowledge-based program analysis. *IEEE Software*, 7(1), 1990.

[15] A. Quilici. A hybrid approach to recognizing program plans. In *Proceedings of the Working Conference on Reverse Engineering*, 1993.

[16] T. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, July 1989.

[17] R. Schwanke. An intelligent tool for re-engineering software modularity. In *13th International Conference on Software Engineering*, 1991.

[18] R. Richardson and N. Wilde. Applying extensible dependency analysis: A case study of a heterogeneous system. Technical Report SERC-TR-62-F, SERC, 1993.

[19] P. Devanbu, B. Ballard, R. Brachman, and P. Selfridge. LaSSIE: A knowledge-based software information system. In *Automating Software Design.* AAAI/MIT Press, 1991.

[20] S.-S. Liu and N. Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. Technical Report SERC-TR-39-F, Software Engineering Research Center, Computer and Information Sciences Department, University of Florida, January 1990.

[21] P. Livadas and T. Johnson. A new approach to finding objects in programs. Technical Report SERC-TR-63-F, Software Engineering Research Center, Computer and Information Sciences Department, University of Florida, June 1993.