Search Based Reverse Engineering

Brian S. Mitchell, Spiros Mancoridis and Martin Traverso
Department of Mathematics & Computer Science
Drexel University
Philadelphia, PA, USA
{bmitchel, smancori, umtraver}@mcs.drexel.edu

ABSTRACT

In this paper we describe a two step process for reverse engineering the software architecture of a system directly from its source code. The first step involves clustering the modules from the source code into abstract structures called subsystems. The second step involves reverse engineering the subsystem-level relations using a formal (and visual) architectural constraint language. We use search techniques to accomplish both of these steps, and have implemented a suite of integrated tools to support the reverse engineering process. Through a case study, we demonstrate how our tools can be used to extract the software architecture of an open-source software package from its source code without having any a priori knowledge about its design.

1. INTRODUCTION & BACKGROUND

Modern software systems tend to be large and complex, thus appropriate abstractions their structure must be determined to simplify program maintenance and improve program understanding. Ideally, these abstractions are documented, however, such documentation is often out-of-date or nonexistent.

Since source code is often the only accurate documentation available to software developers and maintainers, the reverse engineering community has been active in developing tools and techniques to recover high-level structural information directly from source code. These techniques tend to focus on a specific area of design recovery (*e.g.*, software clustering, program slicing, source code analysis, etc.), but not on the overall software architecture.

According to Shaw and Garlan [20], the software architecture of a system consists of a description of the system elements, interactions between the system elements, patterns that guide the construction of the system elements, and constraints on the relationships between the system elements. For smaller systems the elements and relations may be modeled using source code entities such as procedures, classes, method invocation, inheritance, and so on. However, for larger systems, the desired entities may be abstract (high-level), and modeled using architectural components such as subsystems and subsystem relations.

Subsystems provide developers with structural information about the numerous software components, their interfaces, and their interconnections. Subsystems generally consist of a collection of collaborating source code resources that implement a feature or provide a service to the rest of the system. Typical resources found in subsystems include modules, classes, and possibly other subsystems. Subsystems facilitate program understanding by treating sets of related source code resources as high-level software abstractions. Subsystems can also be organized hierarchically, allowing developers to study the organization of a system at various levels of detail by navigating through the hierarchy.

The entities and relations needed to represent software architectures are not found in the source code. Thus, without external documentation, we seek other techniques to recover a reasonable approximation of the software architecture using only source code artifacts.

Our research attempts to achieve the above goal by treating the software architecture recovery problem as a two step process, supported by a suite of integrated tools. The first step uses our clustering tool, named Bunch, to generate the subsystem hierarchy automatically. Using the reverse engineered subsystem hierarchy as input, we then use a second tool, called ARIS (Architecture Relation Inference System), that enables software developers to specify the rules and relations that govern how modules and subsystems can relate to each other. These formal descriptions are called *interconnection styles*, and are created using a visual architectural constraint language called ISF [12]. ARIS automatically infers relations from the subsystem decomposition that satisfy the restrictions imposed by the *interconnection style*. ARIS is also able to validate whether a design that already includes the architectural relations satisfies a set of constraints imposed by the style (*i.e.*, stylistic constraints).

Other researchers in the reverse engineering community have applied a variety of approaches to the software clustering problem. These techniques determine clusters (subsystems) using source code component similarity [17, 5, 14], concept analysis [11, 8, 1], or information available from the system

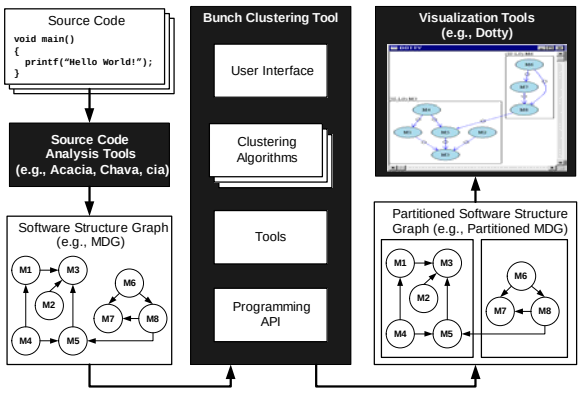

Figure 1: The Design Extraction Process

implementation such as module, directory, and/or package names [2].

Research into Architectural Description Languages (ADLs), and their earlier manifestations as Module Interconnection Languages (MILs), provide support for specifying software systems in terms of their components and interconnections. Different languages define interconnections in a variety of ways. For example, in MILs [7, 16] connections are mappings from services required by one component to services provided by another component. In ADLs [19, 6] connections define the protocols for integrating sets of components.

A unique aspect of our reverse engineering techniques is that they use heuristic-search algorithms [13, 12]. Reverse engineering the subsystem hierarchy from source code, and the architecture-level relations from the subsystem hierarchy is a hard problem, as the search space of all possible solutions to both of the above problems grow exponentially with respect to the size of the software system. Since locating the "optimal" solution to the above problems is computationally intractable for all but the smallest systems, we use heuristic searches to locate acceptable solutions quickly. Another feature of our reverse engineering techniques is that we implemented them in a suite of integrated tools, which can be downloaded over the Internet [18].

The remainder of this paper describes our search techniques, and presents a case study using Apache's regular expression class library [3], to demonstrate the software architecture recovery process.

2. REVERSE ENGINEERING THE SUBSYSTEM HIERARCHY

Our software clustering environment is depicted in Figure 1. The first step in the software clustering process involves parsing the source code and storing the resultant information about the structure of the system in a database. Readily available source code analysis tools – supporting a variety of programming languages – can be used for this step [4, 10]. After the resources and relations have been stored in a database, the database is queried and a *Module Dependency Graph* (MDG) is created. The *MDG* is a directed graph that represents the software modules (*e.g.*, classes, files, packages) as nodes, and the relations (*e.g.*, function invocation, variable usage, class inheritance) between modules as directed edges. Once the *MDG* is created, Bunch's clustering algorithms can be used to create the partitioned *MDG*. The clusters in the partitioned *MDG* represent subsystems that contain one or more modules, relations, and possibly other subsystems. The final result can be visualized and browsed using a graph visualization tool such as dotty [15].

Figure 1 shows that one of the primary services provided by Bunch is a family of clustering algorithms. These algorithms use search techniques to determine clusters from the *MDG*. Bunch currently supports the following algorithms:

1. **Hill-Climbing Search Algorithm.** Bunch's hill-climbing clustering algorithms [13] start by generating a random partition of the *MDG*. Modules from this partition are then rearranged systematically in an attempt to find an "improved" partition. If a better partition is found, the process iterates, using the improved partition as the basis for finding even better partitions. The hill-climbing search algorithm eventually converges when no improved partitions of the *MDG* can be found.

2. **Genetic Algorithm (GA).** The Bunch GA [9] uses operators such as selection, crossover, and mutation to determine a "good" partition of the *MDG*.

3. **Exhaustive Search Algorithm.** The exhaustive search algorithm examines all partitions of the *MDG* and selects the "best" partition as the solution. This algorithm is impractical for most systems (*i.e.*, systems having more than 15 modules) because the number of ways the *MDG* can be partitioned grows exponentially with respect to the number of its nodes (modules) [13].

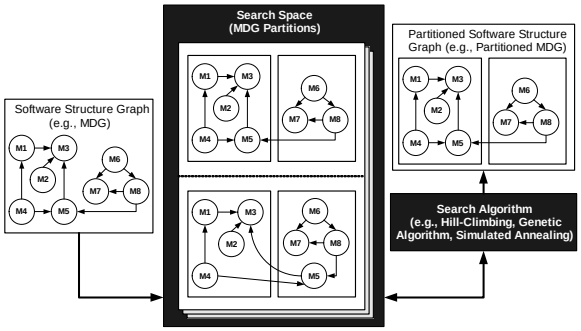

Figure 2: Bunch's Search Algorithms

Figure 2 illustrates the generic search approach used by all of Bunch's software clustering algorithms. Although each of Bunch's search algorithms works differently, they all examine partitions from the very large search space[1] of *MDG* partitions. Thus, Bunch's search algorithms require a way to determine if one *MDG* partition is "better" than another. To address this need we define an objective function, which we call *Modularization Quality* (*MQ*), to evaluate the relative "quality" of *MDG* partitions.

[1]The number of *MDG* partitions is roughly $O(N!)$, where N is the number of modules in the *MDG*.

The MQ function works by calculating a value which we call the *Cluster Factor* (CF) for each cluster in the MDG. Given an MDG partitioned into k clusters, MQ is calculated by summing CF for each cluster of the partitioned MDG. CF_i for cluster i ($1 \leq i \leq k$) is defined as a normalized ratio between the total weight of the internal edges (edges within the cluster) and half of the total weight of external edges (edges that exit or enter the cluster). The weight of the external edges is split in half in order to apply an equal penalty to both clusters that are are connected by an external edge. We refer to the internal edges of a cluster as intra-edges (μ_i), and the edges between two distinct clusters i and j as inter-edges ($\varepsilon_{i,j}$ and $\varepsilon_{j,i}$ respectively). If edge weights are not provided by the MDG, we assume that each edge has a weight of 1. MQ is evaluated as follows:

$$MQ = \sum_{i=1}^{k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{\mu_i}{\mu_i + \frac{1}{2} \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & otherwise \end{cases}$$

The MQ measurement design is based on the assumption that good software systems are designed with highly-cohesive subsystems (clusters in the MDG) that are loosely coupled together.

2.1 RegExp Case Study (Part 1)

This section presents a case study to highlight the capabilities of our software clustering algorithms. We extend on this case study in a later section (Section 3.4) to illustrate the process of recovering subsystem-level relations.

This study examines the open-source Apache Regular Expression[2] (Regexp) class library [3]. The MDG for the Regexp class library was recovered automatically using the Chava [10] source code analysis tool, and is shown in Figure 3. The edge labels in Figure 3 indicate the number of relationships (*e.g.*, method invocation, inheritance, etc.) between the classes in the Regexp package.

In Section 2 we described that the first step in Bunch's software clustering algorithms is to generate a random partition. For example, the random partition generated by Bunch for this case study is illustrated in Figure 4. By inspecting this figure, it should be clear that this is not a good solution. In fact, the only subsystem that contains an intra-edge is SS2 – the edge between `ReTest` and `ReCompiler`. All of the other edges are inter-edges, which lower the MQ value.

There are 190,899,322 partitions of RegExp's MDG, and Bunch is equally likely to generate any one of these partitions as a random starting point for the software clustering algorithm. Figure 4 has a MQ value of 0.093, which is low, and consistent with our intuition that this is a poor solution.

[2]We chose the RegExp package for this case study because its relatively small size enabled us to analyze the results in detail. We have also applied our search techniques to larger systems [18] including Linux, Tomcat, and the Java Swing class library.

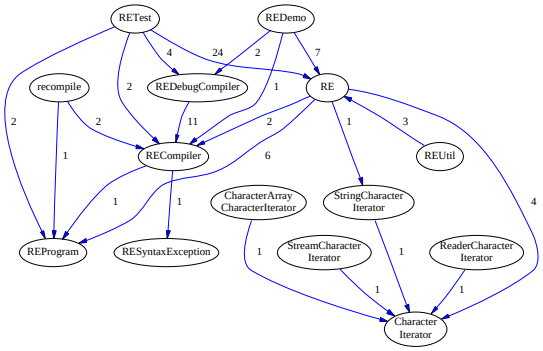

Figure 3: Apache's RegExp Library

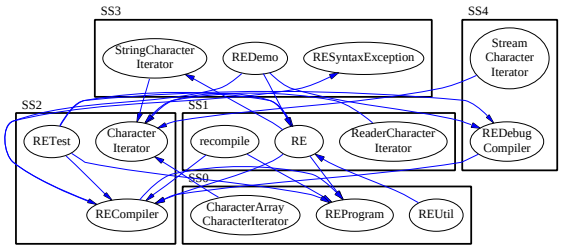

Figure 4: Random Partition of Apache's RegExp Library

It would be possible, but very time consuming, for Bunch to examine and measure the MQ for all of the 190.9 million partitions that exist for the Regexp class library (an exhaustive search). However, in 0.15 seconds, Bunch's hill-climbing algorithm started with the partition shown in Figure 4, iteratively improved it by generating 304 new partitions, and produced the result shown in Figure 5. This result has an MQ value of 1.81, which is almost 20-times better than the MQ of the random starting partition (Figure 4).

In examining the result produced by Bunch, several observations can be made:

- The initial level of clustering produced 3 clusters (drawn as the dark inner boxes). Upon inspecting the source code, the classes in the SSCompiler subsystem implement the functionality for processing regular expression strings, the classes in the SSIterator subsystem provide a collection of interfaces for processing input data against a regular expression string, and the classes in the SSRegExp subsystem include the main interface to the regular expression class library (class `RE.java`), and several test and utility classes.
- Upon performing further analysis of the source code we determined that class `REDemo` is a built-in demonstration program that provides examples showing how to use the regular expression class library, and class `RETest` is a built-in test harness for unit testing of the regular expression class library. It is appropriate to include these classes in the same subsystem as class `RE`, which is the main interface to to the Regexp package, because testing programs should rely on the main interface of the class library (the large edge weights in the MDG between `RETest` and `REDemo` to class `RE` supports this assumption).

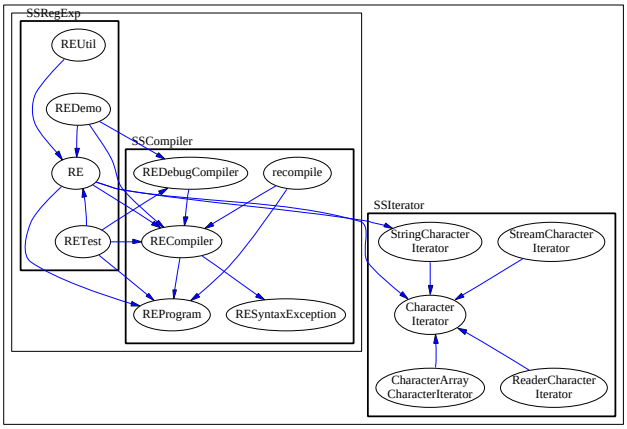

Figure 5: Clustered View of Apache's RegExp Library

- Bunch uses an agglomerative clustering approach [13]. After creating the clusters that contain the modules from the *MDG*, additional clustering is performed on the subsystems themselves to create a higher-level partitioned *MDG*. Thus, the final result is a tree of clusters (shown as nested boxes in Figure 5). In looking at Figure 5, we observe that at the second clustering level, SSRegExp is combined with SSCompiler, and SSIterator remained by itself. This is appropriate because the SSCompiler and SSIterator subsystems appear to be dependant upon each other. The final clustering level, shown by the outermost box, contains all of the modules and clusters, and is the root of the subsystem hierarchy. We have found agglomerative clustering useful for studying large systems because it enables the structure of a system to be examined at different levels of detail, by navigating up and down the subsystem hierarchy.

3. REVERSE ENGINEERING SUBSYSTEM RELATIONS

In this section we describe another search process that is designed to compute the interfaces and relations between the subsystems that were created during the software clustering process. Unlike a typical procedural interface – which is composed of a set of variables – or a typical module or class interface – which is composed of a set of procedures – a subsystem interface is composed of a set of modules, classes, or other subsystems. Hence, just as exported (visible) procedures comprise the interface of modules and classes, the interface of subsystems typically consist of exported modules, classes, and other subsystems.

To achieve this goal we developed the ARIS [21] (Architecture Relation Inference System) tool to enable software developers to specify *interconnection styles* that categorize allowable subsystem relations formally. Interconnection styles allow designers to control the interactions between components by use of rules and subsystem-level relations. When the relations are not present in the recovered subsystem decomposition, ARIS automatically infers the relations that are missing in order for the design to satisfy the constraints imposed by the *interconnection style*.

We define an *interconnection style* to be a description of:

1. the types of components in the design (*e.g.*, module, subsystem)
2. the types of relations in the design (*e.g.*, import, export)
3. the set of all well-formed (syntactically legal) configurations of components and relations
4. the semantics of each well-formed configuration (*e.g.*, exported components are visible to external client components)

The *Export* style is an example interconnection style that formalizes the specification of subsystem interfaces. Subsystem interfaces are defined using the *export* relation between two components. For example, if a subsystem S exports a module M, the module is considered part of the interface of S. Note that the *export* relations would not be produced by existing modularization (clustering) techniques. This is not surprising, since the *export* relation is not found in the source code.

ARIS defines relations that are part of the style being followed by a design. We call such relations *style-specific relations*. The user specifies the stylistic constraints visually and ARIS uses this description to induce the missing relations automatically. ARIS is also capable of validating a design that already contains high-level structural relations against an interconnection style. The syntax for style specifications is based on the Interconnection Style Formalism (ISF) [12].

ISF allows for the definition of two kinds of rules:

1. *Permission* rules, which define the set of well-formed configurations of software designs that follow a specific style.
2. *Definition* rules, which are used to define new relations based on patterns of components and relations.

We next present a small example illustrating the Export style to demonstrate the expressiveness of ISF.

3.1 ISF Example: The Export Style

ISF is a visual notation that enables designers to specify constraints on configurations of components and relations, and the semantics of such configurations. Circles represent system components (*e.g.*, modules, subsystems) and arrows represent relations between components (*e.g.*, import, export, use).

This notation depicts rules as directed labeled graphs, allowing all relations, including the containment relation to be represented uniformly as directed edges. The nodes in the graph represent modules and subsystems. Figure 6 depicts an example of a set of style constraints defined using the ISF notation.

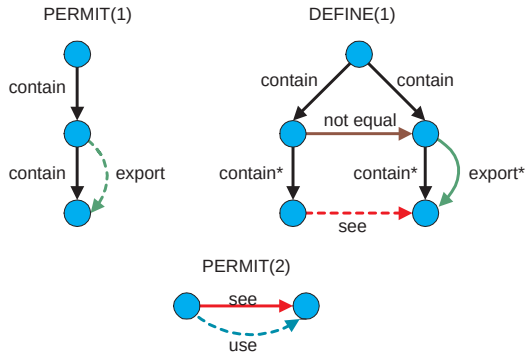

Figure 6: ISF Specification of the Export Style

In the Export style, *use* relations between modules are controlled by encapsulating modules into subsystems with well-defined interfaces. An encapsulated module may be used by other modules outside of its parent subsystem (container) if the encapsulated module is exported by its parent. These constraints are described by the graphs in Figure 6. Each rule is represented by a distinct graph, and each graph contains a single dashed arrow. The meaning of the dashed arrow depends on the kind of rule.

The meaning of each of the three rules that comprise the ISF specification of the Export style (Figure 6) is given below.

- **PERMIT(1):** A subsystem may be exported by the subsystem that contains it. The dashed arrow represents the permitted *export* relation. Only subsystems contained in other subsystems are allowed to export their children (*i.e.*, subsystems may not be exported outside the application, represented by the root subsystem).
- **PERMIT(2):** A subsystem or module SS1 can *use* another subsystem or module SS2 if SS1 *sees* SS2.
- **DEFINE(1):** A module SS1 can *see* another module SS2 if SS2 is transitively exported to a common level with an ancestor of SS1. Unlike permission rules, which specify when design relations are permitted to occur, definition rules actually define new relations (*e.g.*, relation *see*).

3.2 The ARIS Tool

ARIS has two main components, namely, the *Style Editor* and the *Edge Repair Utility*. The Style Editor allows the user to define style specifications visually using simple graphic elements, such as circles and arrows, as shown in Figure 6. The Edge Repair Utility performs the well-formedness validation of a design and the automatic induction of missing relations with respect to a style.

The ARIS tool is bundled with support for the Export and Tube styles [12], however the user may specify any valid ISF style using the ARIS graphical editor.

ARIS takes a clustered *MDG* as input and attempts to find the missing style relations. If a clustered *MDG* is not available, ARIS uses Bunch to generate the partitioned *MDG* automatically.

The goal is to induce a set of style relations that will make all of the *use* relations well-formed. A relation is well-formed if it does not violate any permission rule described by the style. A solution to this problem, which we call the *edge repair* problem, not only has to be well-formed, but it must limit the exposure (visibility) of encapsulated modules and subsystems as much as possible, which is a desirable property of good designs. Visibility is a measure of the degree of subsystem exposure or, more concretely, of the number of *see* relations that can be induced in a design.

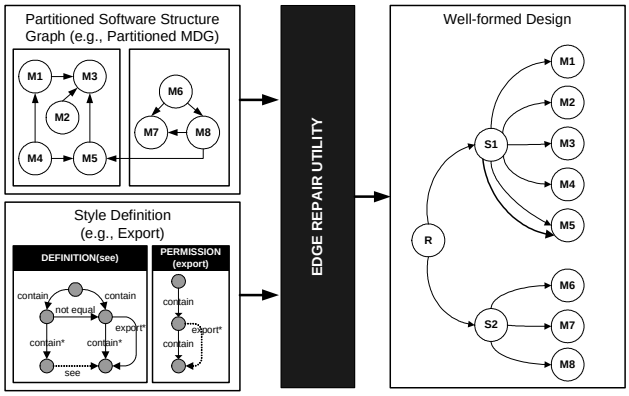

Figure 7: The ARIS Environment

Figure 7 shows how our Style Editor and Edge Repair tools are integrated with the Bunch clustering tool. The Edge Repair Utility accepts two inputs, namely, the *MDG* and *Cluster Tree* (the partitioned *MDG* with the subsystems arranged as a tree) specifications as well as the style definition. The Edge Repair Utility algorithm adds style relations to the Cluster Tree and finishes either when every relation is well-formed or when no more relations can be made well-formed. The first case implies that a solution was found; the second case, that there is no way to make every *use* relation well-formed.

Figure 7 shows the result of finding the missing export relations for the partitioned *MDG* shown in the upper right of the same figure. For clarity, *use* relations are not shown in the diagram. Also, unlike Bunch, ARIS illustrates subsystem containment as a tree instead of nested boxes. The *containment* relations are drawn as thin arrows, while thick arrows represent the induced *export* relations. For example, the result on the right side of Figure 7 shows that one style edge was added to indicate that module `M5` is exported from subsystem `S1`. Adding the *export* relation to M5 indicates that its resources are used by another subsystem. This is validated by inspecting the *MDG*, which shows that module `M8` in `S2` uses the services of `M5` in `S1`.

3.3 Edge Repair as an Search Problem

An exhaustive approach to solving the edge repair problem is to try all possible configurations of relations permitted by the style and keep track of the one that satisfies all of the constraints of the style and minimizes overall exposure of modules. While this approach might work for small systems, using it on larger systems is not feasible since the number of possible configurations grows exponentially with respect to the size of the system.

Since a brute-force solution to solving the edge repair problem is not possible for all but the smallest systems, we treat the edge repair process as a search problem. Like most search problems, the goal is not to find the optimal solution, which would require examining all of the configurations in the search space, but a solution that is "good enough". Search problems tend to work well when the ratio between good and bad solutions in the search space is small, and the relative quality of a proposed solution can be mathematically evaluated by the use of an *objective function*.

The objective function that we designed into the ARIS system measures the well-formedness of a configuration in terms of the number of well-formed and ill-formed relations it contains. Given these premises, we can define our new goal as finding a configuration that makes every *use* edge well-formed while keeping the number of ill-formed relations and visibility low.

Thus, the quality measurement should exhibit the following properties:

- configurations with a large number of well-formed *use* relations should get a high quality score
- configurations with a large number of ill-formed style relations should get a low quality score
- configurations with large visibility (*i.e.*, many *see* relations) should get a low quality score

Using the above design criteria for our objective function, we define the quality of a configuration as follows:

$$\mathsf{quality}(C) = \begin{cases} \frac{\text{wfs}}{\text{ifs}+\text{ifu}} & \text{ifs} \neq 0 \ \textit{or} \ \text{ifu} \neq 0 \\ \text{MaxS} + \frac{1}{\text{wfs}} & \text{ifs} = 0, \text{ifu} = 0, \text{wfs} \neq 0 \\ \text{MaxS} + 2 & \text{ifs} = 0, \text{ifu} = 0, \text{wfs} = 0 \end{cases}$$

Table 1: Definitions

ifu	number of ill-formed *use* relations
ifs	number of ill-formed style relations (*e.g.*, *export*)
wfu	number of well-formed *use* relations
wfs	number of well-formed style relations
MaxS	maximum number of style relations that can exist in a configuration

The ARIS quality measurement distinguishes between the three types of solutions (ill-formed, well-formed and perfect) and ranks them in order of desirability. It is easy to infer from the formula that the terms have a quality value in the range $(0, MaxS]$, $(MaxS, MaxS + 1]$ and $[MaxS + 2, MaxS + 2]$, respectively. $MaxS$ refers to the maximum number of style relations that can exist in a configuration.

Now that the quality of a configuration can be evaluated, we describe two search algorithms that we have implemented whose goal is to maximize the objective function described above. Specifically:

- **Hill-Climbing.** The edge repair hill-climbing algorithm starts by generating a random configuration. Incremental improvement is achieved by evaluating the quality of neighboring configurations. A neighboring configuration (C_N) is one that can be obtained by a small modification to the current configuration (C). The search process iterates as long as a new C_N can be found such that $\mathsf{quality}(C_N) > \mathsf{quality}(C)$.

- **Edge Removal.** The edge removal algorithm is based on the assumption that as long as there exists at least one solution to the edge repair problem for a system with respect to a style specification, the configuration that contains every possible reparable relation will be one of the solutions. Using this assumption, the edge removal algorithm starts by generating the fully reparable configuration for a given style definition and system structure graph. It then removes relations, one at a time, until no more relations can be removed without making the configuration ill-formed.

3.4 RegExp Case Study (Part 2)

This section is a continuation of the case study from Section 2.1. Recall that Figure 5 presented the subsystem decomposition that was automatically recovered by Bunch.

Using the subsystem hierarchy from Figure 5 as input, the edge repair result generated by ARIS for the Export style of the RegExp class library, is shown in Figure 8. Several observations can be made:

- The results of our analysis of the Regexp class library raise some interesting questions about the system's structure, especially the classes in the SSIterator subsystem. Upon inspecting the source code we validated that all of the classes in this subsystem implement the `CharacterIterator` interface. Since the Regexp package is a class library, it is expected that users of the class library implement one of the iterator classes to encapsulate the input data to be matched against a provided regular expression string. This iterator class is passed polymorphically through the public interface of the `RE` class. This explains the relation between the `RE` and `CharacterIterator` classes. However, upon analyzing the clustering and edge repair results, it was puzzling why the SSIterator subsystem exports the `StringCharacterIterator` class. Further analysis of the source code revealed that a special interface was added to the `RE` class to support users who simply want to process a regular expression against a text string (this interface was probably added for convenience).

- Since the SSRegExp subsystem does not export any of its classes, it is safe to assume that changes to the classes in this subsystem will be localized. This seems appropriate, as class `RE` is the main interface into the Regexp class library, and the demo and test programs exercise the public interfaces in the `RE` class. Thus, the SSRegExp subsystem behaves like a driver, which can be confirmed by inspecting the partitioned *MDG* (no inter-edges enter the subsystem).

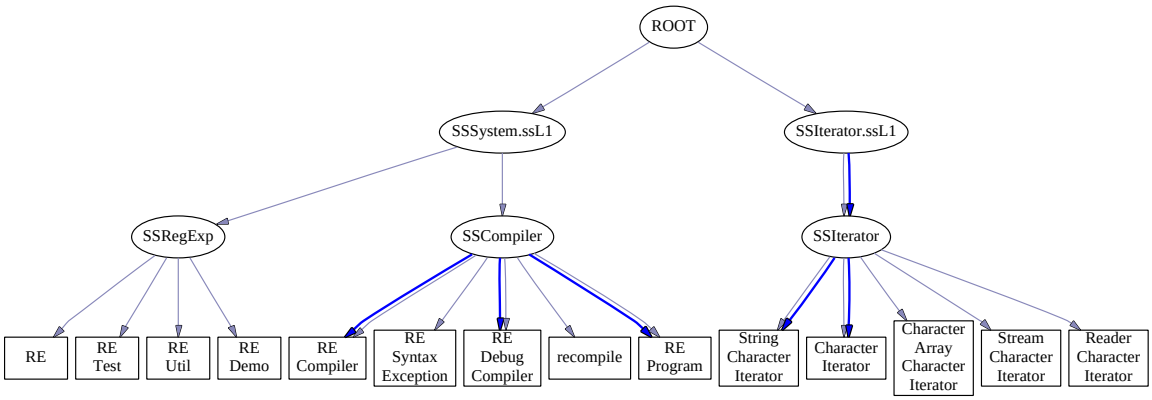

Figure 8: Recovered Export Style Relations for the RegExp Class Library

- The edge repair results also give us intuition that subsystem SSCompiler behaves like a library, since many of its classes are exported. This assumption can be validated further by examining the clustering results shown in Figure 5, where we see that all inter-edges incident to the SSCompiler subsystem are initiated by classes outside of the SSCompiler subsystem. Furthermore the classes not exported from the SSCompiler subsystem do not use any services outside of the SSCompiler subsystem (we already know that other subsystems do not use these classes because the edge repair utility did not export these modules).

Once the edge repair utility derives the style relations, this information can be used for future maintenance. ARIS can save the recovered style relationships, and then validate them against an updated *MDG*. As a system undergoes maintenance, ARIS can also be used to check that changes to the source code are not violating any of the recovered style relations. If ill-formed *use* relations are found, ARIS will report them. The user then has the choice to either re-run ARIS to determine the new style relations, or to change the implementation of the system to eliminate the ill-formed use relations (*e.g.*, add a new interface to an existing exported module).

4. CONCLUSIONS

Reverse engineering the software architecture from source code provides a valuable service to software practitioners. This approach is especially helpful when other forms of traditional design documentation are outdated or not available. The reverse engineered software architecture also can be used to validate the architecture against the implementation, or to measure the drift of the implementation from the system's intended architecture over a period of time.

In this paper we presented a two-step process, using a suite of integrated tools that we developed, to generate views of the software architecture directly from the system's source code. Our reverse engineering approach uses search techniques to propose reasonable solutions to this problem, as determining optimal solution is a computationally intractable problem. Also, our tools are available for use by other researchers and professional software engineers [18].

The approach outlined in this paper for generating the software architecture is a pipelined process, requiring the clustering results to be fixed prior to determining the architectural style relations. As future work we would like to investigate how we can provide a tighter integration of our tools such that the constraints defined by a given architectural style can be integrated into the clustering process.

5. ACKNOWLEDGEMENTS

This research is sponsored by grants CCR-9733569 and CISE-9986105 from the National Science Foundation (NSF). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

6. REFERENCES

[1] N. Anquetil. A comparison of graphis of concept for reverse engineering. In *Proc. Intl. Workshop on Program Comprehension*, June 2000.

[2] N. Anquetil and T. Lethbridge. Recovering software architecture from the names of source files. In *Proc. Working Conf. on Reverse Engineering*, Oct. 1999.

[3] The Apache Foundation – Regular Expression Package (Regexp). http://jakarta.apache.org/regexp.

[4] Y. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.

[5] S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. In *IEEE Software*, pages 66–71, 1999.

[6] C. Dellarocas. A Coordination Perspective on Software System Design. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering*, pages 318–325, June 1997.

[7] F. DeRemer and H. H. Kron. Programming in the Large Versus Programming in the Small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.

[8] A. v. Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *International Conference on Software Engineering, ICSM'99*, pages 246–255. IEEE Computer Society, May 1999.

[9] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of Software Technology and Engineering Practice*, Aug. 1999.

[10] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proc. Working Conference on Reverse Engineering*, Oct. 1999.

[11] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. International Conference on Software Engineering*, May 1997.

[12] S. Mancoridis. ISF: A Visual Formalism for Specifying Interconnection Styles for Software Design. *International Journal of Software Engineering and Knowledge Engineering, World Scientific Publishing Company*, 8(4):517–540, 1998.

[13] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. 6th Intl. Workshop on Program Comprehension*, June 1998.

[14] H. Müller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[15] S. North and E. Koutsofios. Applications of graph visualization. In *Proc. Graphics Interface*, 1994.

[16] R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *The Journal of Systems and Software*, 6:307–334, 1986.

[17] R. Schwanke and S. Hanson. Using Neural Networks to Modularize Software. *Machine Learning*, 15:137–168, 1998.

[18] The Drexel University Software Engineering Research Group (SERG). http://serg.mcs.drexel.edu.

[19] M. Shaw, R. DeLine, D. V. Klien, T. L. Ross, D. M. Young, and G. Zalesnik. Abstractions for Software Architectures and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21, April 1995.

[20] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[21] M. Traverso and S. Mancoridis. On the Automatic Recovery of Style-Specific Structural Dependencies in Software Systems. *Journal of Automated Software Engineering*, 9(3), 2002.