Using Interconnection Style Rules to Infer Software Architecture Relations

Brian S. Mitchell, Spiros Mancoridis, and Martin Traverso

Department of Computer Science
Drexel University, Philadelphia PA 19104, USA,
{bmitchell, spiros, umtraver}@drexel.edu,
http://www.mcs.drexel.edu/~{bmitchel, smancori}

Abstract. Software design techniques emphasize the use of abstractions to help developers deal with the complexity of constructing large and complex systems. These abstractions can also be used to guide programmers through a variety of maintenance, reengineering and enhancement activities. Unfortunately, recovering design abstractions directly from a system's implementation is a difficult task because the source code does not contain them. In this paper we describe an automatic process to infer architectural-level abstractions from the source code. The first step uses software clustering to aggregate the system's modules into abstract containers called subsystems. The second step takes the output of the clustering process, and infers architectural-level relations based on formal style rules that are specified visually. This two step process has been implemented using a set of integrated tools that employ search techniques to locate good solutions to both the clustering and the relationship inferencing problem quickly. The paper concludes with a case study to demonstrate the effectiveness of our process and tools.

1 Introduction

Programmers are routinely given limited time and resources to perform maintenance on software systems. Without a good understanding of the software architecture maintenance tends to be performed haphazardly.

Software clustering techniques [11,13] have been used successfully to create abstract views of a system's structure. These views consolidate the module-level structure into architectural-level subsystems. The subsystems can be further clustered into more abstract subsystems, resulting in a subsystem hierarchy. While helpful, the subsystem structure exposes the architectural-level components (*i.e.*, subsystems), but not the architectural-level relations.

Like subsystems, architectural-level relations are not specified in the source code. To determine these relations we developed a technique that searches for them guided by a formalism that specifies constraints on allowable subsystem-level relations. This formalism, called ISF [10], is a visual language that can be used to specify a set of rules that collectively represent an interconnection style. The goal of an interconnection style is to define the structural and semantic

K. Deb et al. (Eds.): GECCO 2004, LNCS 3103, pp. 1375–1387, 2004.
© Springer-Verlag Berlin Heidelberg 2004

properties of architectural relations. Like our approach to software clustering [12, 13], computing the architectural relations from the subsystem hierarchy for a given interconnection style is computationally intractable. We overcome this problem by using search techniques.

After providing an overview of our integrated software clustering and style-specific relation inference processes and tools, we present a case study to show how our tools can assist with the maintenance of large systems.

2 Related Work

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 [17,4,15], concept analysis [9,7,1], or information available from the system implementation such as module, directory, and/or package names [2]. Our approach to clustering differs because we use metaheuristic search techniques [5] to determine the clusters [12, 13,14].

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 [6,16] connections are mappings from services required by one component to services provided by another component. In ADLs [18] connections define the protocols for integrating sets of components. Our approach uses ISF [10], which is a visual formalism for specifying interconnection styles. We also developed a tool that can generate a Java program to check the well-formedness of ISF rules.

3 The Style-Specific Architecture Recovery Process

This section provides an overview of the style-specific architecture recovery process. Figure 1 outlines the process, which consists of using a series of integrated tools. To simplify the explanation of the process, we use a common reference system throughout this section to highlight important concepts.

JUnit is an open-source unit testing tool for Java, and can be obtained online from `http://www.junit.org`. JUnit contains four main packages: the framework itself, the user interface, a test execution engine, and various extensions for integrating with databases and J2EE. For the purpose of this example, we limit our focus to the framework package. This package contains 7 classes, and 9 inter-class relations.

3.1 Preparing the Software System

The first step in our process, shown on the left side of Figure 1, involves parsing the source code, transforming the source-level entities and relations into a

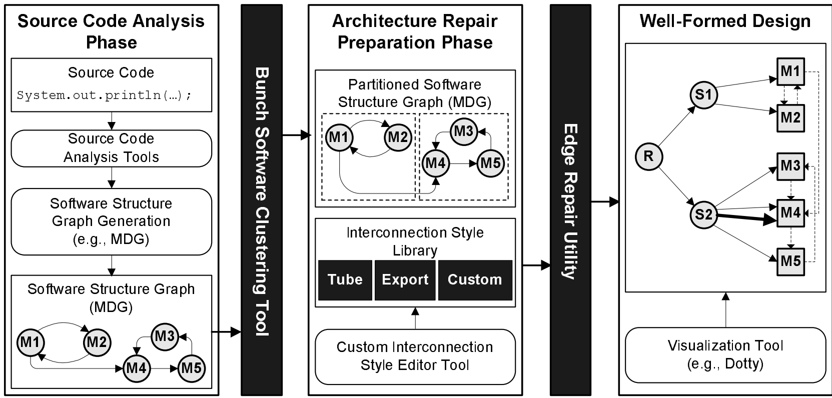

Fig. 1. The Style-Specific Architecture Recovery Environment

directed graph called the Module Dependency Graph (MDG). Readily available source code analysis tools – supporting a variety of programming languages – can be used for this step [3,8]. The MDG for JUnit, which was generated automatically, is illustrated on the left side of Figure 2.

3.2 Clustering the Software System

The software clustering process accepts the MDG as input, and produces a partitioned MDG as output. The Bunch tool can perform this activity automatically. Bunch uses search techniques to propose solutions based on maximizing an objective function [11,12,13,14]. The partitioned MDG is used as input into the architecture repair process, which is depicted in the top-center of Figure 1.

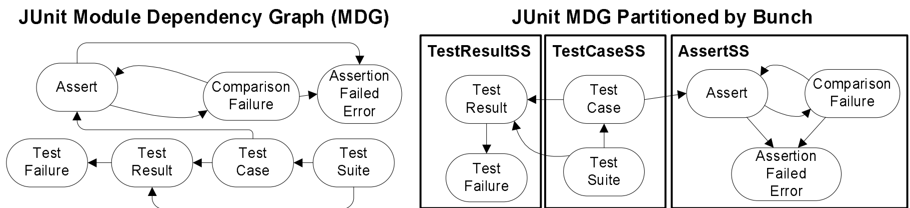

Fig. 2. The JUnit System Before and After Clustering

The right side of Figure 2 illustrates the JUnit system clustered into subsystems by Bunch. Even though this system is very small, there are still 877 unique ways to partition the MDG. Bunch produced the result shown in Figure 2 (which is the optimal solution) in 0.08 seconds, examining 47 partitions. The heuristic used to guide the search is based on an objective function that maximizes cohesion and minimizes inter-cluster coupling.

3.3 Inferring the Style-Specific Architectural Relations

The next step in the process (bottom-center of Figure 1) is to select an *interconnection style* to specify the allowable subsystem-level relations. For the purpose of this example, let's examine a useful interconnection style that we call the *Export Style*. The goal of the export style is to identify the *subsystem interfaces*. We define the subsystem interface to be the subset of the modules within a subsystem that provide services to modules in other subsystems. This information is helpful for determining the impact of a change to a subsystem.

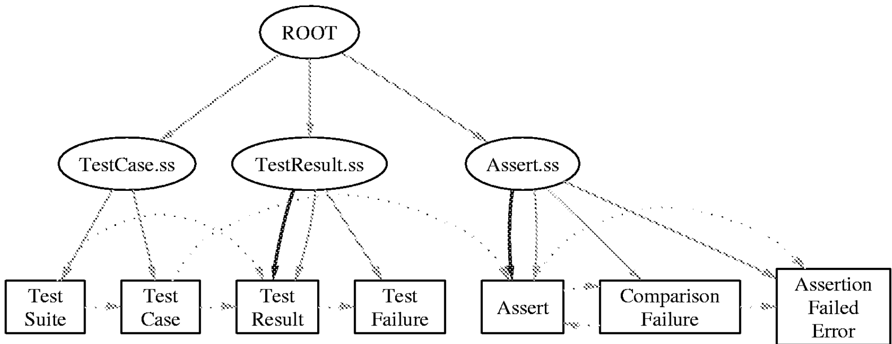

Fig. 3. Example Showing the **Export Style** Edges for JUnit

Figure 3 illustrates the result produced by our tool when the Export style is applied to the subsystem hierarchy shown in Figure 2. The leaf nodes in this figure are the modules in JUnit, and the dotted lines represent module-level dependencies (*i.e.*, the *use* relation). The intermediate elliptical nodes, connected by solid lines (*i.e.*, the *contain* relation) show the subsystem hierarchy. The results of recovering the inferred style-specific relations are denoted by the bold lines (*i.e.*, the *export* relation). Since the relation is of type Export, the bold lines are to be interpreted as: The `TestResult.ss` subsystem exports the `TestResult` module, and the `Assert.ss` subsystem exports the `Assert` module. We examine the usefulness of identifying the style-specific relations in further detail in Section 5.

4 Style-Specific Software Architecture Relations

This section describes a search process that is designed to compute architectural relations between the subsystems. To achieve this goal we developed a tool that enables software developers to specify *interconnection styles* formally. Interconnection styles allow designers to control the interactions between components by the use of rules and subsystem-level relations. Since the relations are not present in the recovered subsystem decomposition, our tool automatically infers

the subsystem-level relations that are missing in order to satisfy the constraints imposed by the interconnection style. The syntax for style specifications is based on the Interconnection Style Formalism (ISF) [10]. ISF allows for the definition of two kinds of rules:

1. *Permission* rules, which define the set of well-formed configurations of subsystems, modules, usage and architectural relations that adhere to a specific style.
2. *Definition* rules, which are used to define new relations based on patterns of components and relations.

4.1 Defining Styles

ISF enables designers to specify constraints on configurations of components and relations, and the semantics of such configurations. In the ISF notation, circles (nodes) represent system components (*e.g.*, modules, subsystems) and arrows (edges) represent relations between components (*e.g.*, import, export, use). The directed edges either depict a direct relation, or a transitive relation (represented using a double-headed arrow). Figure 4 shows the set of rules for the Export style using the ISF notation.

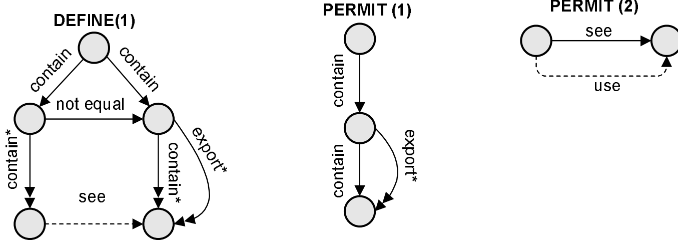

Fig. 4. Specification of the **Export Style**

In the Export style, *use* relations are extracted from the source code directly, as they represent the module-level dependencies. The clustering activity encapsulates modules into subsystems which is used to define the *contain* relations. The inferred *export* relations represent the subsystem interfaces. With this particular style a module may be used by other modules outside of its parent subsystem (container) if and only if the module is exported by its parent subsystem. Each module decorated with an inferred *export* relation is part of its parent subsystem's interface.

The Export style belongs to a family of styles that only allows relations between ancestors and decedents in the containment hierarchy. Another important family of styles are those that define relations between different subtrees in the subsystem containment hierarchy. An example of the latter family of styles is the *T*ube style, which is illustrated in Figure 5.

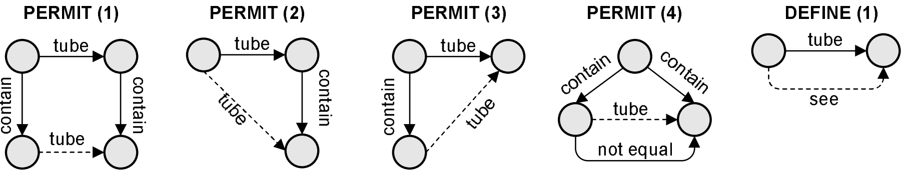

Fig. 5. Specification of the **Tube Style**

The Tube style definition specifies the tube relation, which enables a particular subsystem to "see" another subsystem. Specifically, two subsystems can be connected by a Tube if they are proper siblings, or if their parents are connected by a tubes according to the rules specified in Figure 5.

The ISF visual rules can be translated into formal specifications in a straightforward way. Given this capability we developed a tool that allows architectural styles to be defined using the ISF visual syntax, and then generates a Java program on the fly that is integrated dynamically into the search strategy to infer the desired style-specific relations. Figure 6 shows the GUI for the style editor, which depicts the ISF rules for the Import relation. The style editor can be used to create a reusable library of useful styles. More details on ISF Styles are available in an IJSEKE paper [10].

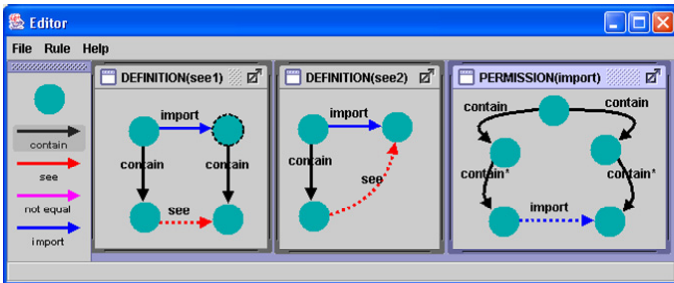

Fig. 6. The Custom Style Editor GUI

4.2 Style-Specific Edge Recovery

The goal of the style-specific edge recovery process is to take the subsystem containment hierarchy produced by the clustering process, along with a set of style rules specified in ISF, and locate the set of style relations that satisfies the ISF constraints. An exhaustive approach to recovering the style relations is to try all possible configurations permitted by the style and keep track of the one that has the fewest inferred relations (minimum visibility) that also satisfies all of the constraints of the style.

Unfortunately, exhaustive analysis is not possible. To understand why, let us consider a subsystem containment graph with N nodes. The maximum number of edges that can exist in the graph for each permitted relation type (e.g., export, import) is $E = N^2$ (i.e., one edge coming out of each node and into every other node, as well as the source node itself). If the style permits R different relation types, the graph can contain a total of $M = RE = RN^2$ style relations. What we need to know, however, is how many possible configurations exist for a specific style (based on the number of relations that the style defines).

Given that a style can contain a maximum of M total style relations, a particular configuration will introduce e style relations where $0 \leq e \leq M$. To determine how many total configurations exist for each style, we must consider how many configurations exist for when $e = 1, 2, \ldots M$. It turns out that the number of configurations is quite large:

$$\sum_{e=0}^{M} \binom{M}{e} = 2^M = 2^{RN^2}$$

Because considering all possible configurations is not possible, we use search techniques to find good (if not optimal) solutions to the style-specific relation recovery problem. The remainder of this section will focus on an approach[1] that uses a hill-climbing algorithm.

The hill-climbing approach for recovering the style-specific relations is straightforward. The first step generates a random configuration. Incremental improvement is achieved by evaluating the quality of neighboring configurations using an objective function. The set of neighboring configurations for a given configuration are determined by adding or removing a single style edge. The search iterates until no new neighboring configurations can be found with a higher objective function value. Figure 7 illustrates how neighboring configurations can be generated for a given configuration (left), by adding and removing export style relations.

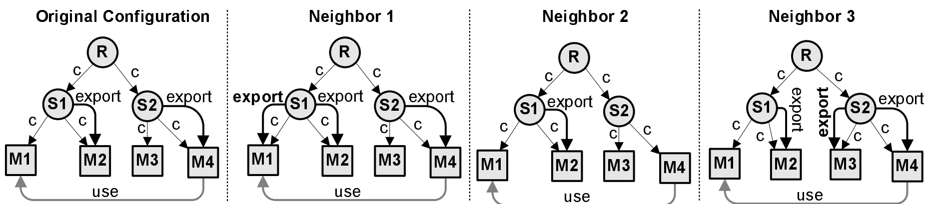

Fig. 7. Example Showing the Generation of Neighboring Configurations

Before the objective function is introduced, we need to provide several important definitions. Every configuration considered during the search will have a set

[1] We have investigated other techniques that are more efficient than hill-climbing, however they only work on certain styles.

of *use* and *style* relations. The *use* relations are extracted from the source-level dependencies and the *style* relations are defined by the ISF specification. We define $MaxS$ as the maximum number of style relations that can exist for a particular configuration. Each *style* relation can either be well-formed, or ill-formed. We define *wfs* as the total number of style relations that respect the rules of the provided ISF specification, and *ifs* as the total number of style relations that violate the rules of the ISF specification. As for the *use* relations, *wfu* is defined as the number of relations that are correctly modeled by the provided ISF style, and *ifu* is the number of *use* relations that violate the ISF style.

Given the above definitions the objective function is engineered to maximize *wfu*, minimize *ifs*, and minimize visibility. Minimizing visibility by keeping the set of inferred style edges as small as possible is important since not all *wfs* edges are necessary to cover the set of *use* edges for a particular system. It should also be noted that the objective function maximizes *wfu* by minimizing *ifu* since a given use edge must be classified as either well- or ill-formed. The objective function to measure the quality of a particular configuration is shown below:

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

5 Case Study

In this section we describe how we used our tools to gain insight into the design of Bunch, a clustering tool created by our research group.

Bunch is a research tool, and as such, is prone to frequent restructuring and extension due to changes in requirements. The original version implemented the core clustering algorithms and had no graphical user interface (GUI). Later, it was redesigned to make it extensible, and several features, including a GUI were added. Adding new clustering algorithms was easier in this version. Following that, an attempt was made to improve the performance of the algorithms by implementing a distributed version of the tool. Finally, the core algorithms were replaced with much faster incremental algorithms. An important fact about the development process is that each stage was carried out by different developers.

Without the help of reverse engineering tools, understanding the structure of the current version of Bunch and how its components relate to each other would be quite difficult. It is important that future maintainers be able to understand the system well enough to be able to make changes without breaking the application.

The first thing that we did in this case study was to obtain Bunch's MDG, and then use Bunch to cluster the MDG. We then based our analysis on a modified version of the Export style, which is depicted in Figure 8. This style permits two types of export relations: local export and broad export. Local export relations enable a module to be seen by subsystems in the immediate neighborhood (the parent subsystem's siblings). Broad export relations export modules

to more distant branches of the tree. Local export allows a subsystem to export only child nodes, whereas broad export dependencies allow subsystems to export only modules deeper than the grandchildren level, thereby allowing modules in entirely different branches of the tree to "see", and hence access, them.

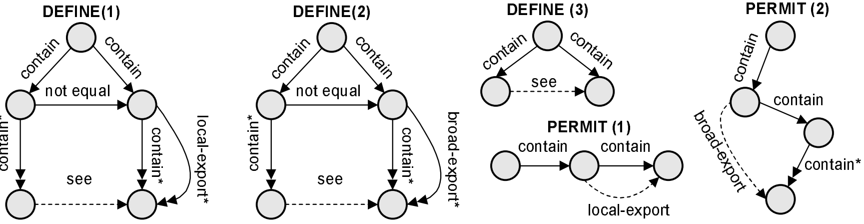

Fig. 8. Specification of the **Modified Export Style**

The results produced by our tool allow us to make several observations about the structure of the Bunch application. Subsystems that export a high percentage of their modules are particularly noteworthy, since they might reflect poor design choices. Sometimes, these subsystems might contain groups of modules that are used throughout the application and are considered to be libraries. Such situations are not bad, but raise the question whether those modules can be split into smaller modules that perform more specific functions and can be placed alongside modules that use them. Other times, those situations occur because modules have been classified into the wrong subsystems. For example, the `Graph` subsystem shown on the left side of Figure 9 contains the modules `TurboMQ` and `TurboMQIncr`, which are fitness function evaluation classes, unlike the `Graph` and `Cluster` classes which are important global data structures. This means that both of the `TurboMQ` classes have probably been misplaced in the `Graph` cluster, and would be better suited in the `ObjectiveFunctionCalculator` subsystem[2] instead.

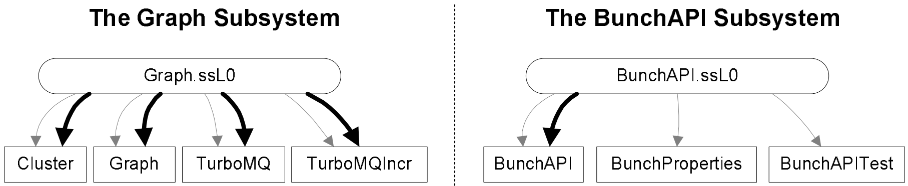

Fig. 9. Two of Bunch's Subsystems Decorated with Export Relations. The Light Edges are the Containment Relations and the Dark Edges are the Export Relations.

[2] This subsystem is not shown but it is present in the subsystem hierarchy generated by Bunch.

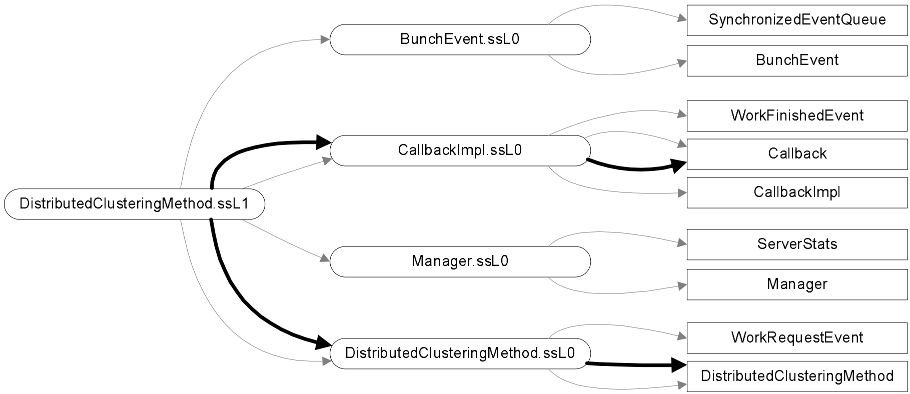

Fig. 10. The Bunch Subsystem that Supports Distributed Clustering

Since the clustering was automatically produced by Bunch, this error is probably related to the fact that the clustering algorithm does not take into account style-related properties. This is an example of how by considering the style relationships one can complement clustering tools. Regardless of whether the clustering was produced manually or by a clustering tool, a configuration where a subsystem exports a high percentage of its classes warrants further investigation.

The existence of export relations, or lack of them, gives us information on how a change to a module might affect the rest of the system. In particular, if a module is local to a subsystem (*i.e.*, it is not exported), it is clear that making modifications to it should not affect modules that are outside of the subsystem where it belongs. Maintenance efforts can be concentrated on a small subset of the system. The existence of an export dependency indicates that changes to the module may affect a wide range of modules in the system, and, thus, care must be taken when modifying the module. Even with a subsystem decomposition produced by a clustering algorithm, it is not obvious for all but the most trivial–sized systems, which modules encapsulated in a subsystem are only dependant on other modules within the subsystem. As an example, on the right side of Figure 9, we have confidence that an update to the `BunchProperties` class can at most affect the `BunchAPI` and the `BunchAPITest` classes.

Figure 10 shows the subsystem responsible for distributed clustering. Let us imagine that we want to add a new function to Bunch that depends on this subsystem. The first thing that we need to know is the interface to this subsystem. Since the documentation is often outdated or missing altogether, the most reliable way to find the information is by browsing the source code; a really tedious task, especially if the subsystem is complex. Export relations might ease the burden of such a task, since these modules are essentially the access points to the subsystem. In the case of the `DistributedClusteringSS` subsystem, we notice that the exported modules are `CallbackImpl` and `DistributedClustering-Method`. Thus, we only need to concentrate on those modules when trying to

learn how to use this subsystem, and may ignore the rest of the modules since they are not visible outside of the subsystem.

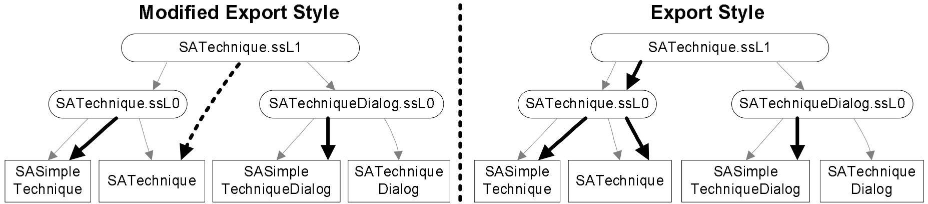

Fig. 11. The Simulated Annealing Subsystem with Local/Standard Export (bold arrow) and Broad Export (dashed arrow) Relations. The Left Side was Generated Using the Modified Export Style, and the Right Side was Generated Using the Standard Export Style.

The containment hierarchy of Bunch has a depth of four levels. This means that the application is composed of several major subsystems, which in turn, are composed of smaller subsystems. Some modules in these smaller subsystems are intended for local use, whereas, other modules are intended for use outside of the major subsystems (*i.e.*, they are part of the subsystem interface). The modified export style defined in Figure 8 can help us understand which modules play each of these roles. The `SATechniqueSS` subsystem depicted on the left side of Figure 11 implements a simulated annealing algorithm and was added quickly to try this idea. It exports module `SATechnique` which comprises the subsystem's interface. On the other hand, modules `SASimpleTechnique` and `SASimpleTechniqueDialog`, which are exported outside the smaller subsystems but not out of the `SATechnique` subsystem are intended for use only within `SATechnique`. Note that if we had analyzed this subsystem with the original Export style, we would have seen that `SASimpleTechnique` would have also been exported out of `SATechnique`, as shown in the right side of Figure 11. This outcome increases the visibility of the `SASimpleTechnique` module, giving the maintainer no choice but to assume that this module may be used from any of Bunch's subsystems.

The Bunch system contains 48K LOC packaged in 220 classes, with 764 inter-class relations. Our tools clustered Bunch in 0.57 seconds, and recovered the style-specific relations for the modified export style in 2.86 seconds. Although space prohibits a more detailed analysis, the few patterns that we examined in the clustered decomposition with the recovered style-specific relations gave us significant information on how Bunch is structured and how its modules relate to each other.

6 Conclusion

This paper highlighted techniques and tools that can be used to assist software maintainers tasked to fix, enhance or reengineer large and complex existing systems. These techniques focus on extracting abstract architectural-level artifacts directly from the detailed system implementation. This is a hard problem since these abstractions are not specified in the source code.

Ongoing research in software clustering has shown promising results for recovering useful architectural-entities directly from the source code. However, while software clustering techniques simplify understanding the relationships between modules grouped into the same subsystems, these techniques provide little help for understanding the complex inter-subsystem relationships. To address this need, this paper presented an integrated process that builds on existing software clustering technology by using formal style rules to infer architectural-level relations between subsystems.

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*, October 1999.
3. Y. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.
4. S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. In *IEEE Software*, pages 66–71, 1999.
5. J. Clark, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating Software Engineering as a Search Problem. *Journal of IEE Proceedings - Software*, 150(3):161–175, 2003.
6. 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.
7. A. van 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.
8. J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proc. Working Conference on Reverse Engineering*, October 1999.
9. 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.
10. S. Mancoridis. ISF: A Visual Formalism for Specifying Interconnection Styles for Software Design. *International Journal of Software Engineering and Knowledge Engineering*, 8(4):517–540, 1998.
11. S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. 6th Intl. Workshop on Program Comprehension*, June 1998.
12. B. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem.* PhD thesis, Drexel University, Philadelphia, PA, USA, 2002.

13. B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *Proceedings of Genetic and Evolutionary Computation Conference*, 2002.
14. B. S. Mitchell and S. Mancoridis. Modeling the search landscape of metaheuristic software clustering algorithms. In *Proceedings of Genetic and Evolutionary Computation Conference*, 2003.
15. 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.
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. 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.