Separating Architectural Concerns to Ease Program Understanding

Vladimir Jakobac
Computer Science Dept.
Univ. of Southern California
Los Angeles, CA 90089, USA
jakobac@usc.edu

Nenad Medvidovic
Computer Science Dept.
Univ. of Southern California
Los Angeles, CA 90089, USA
neno@usc.edu

Alexander Egyed
Teknowledge Corporation
Marina Del Rey,
CA 90292, USA
aegyed@teknowledge.com

ABSTRACT
This paper presents an iterative, user-guided approach to program understanding based on a framework for analyzing and visualizing software systems. The framework is built around a pluggable and extensible set of clues about a given problem domain, execution environment, and/or programming language. The approach leverages two orthogonal architectural views of a system and describes how a proper identification of boundaries for separate concerns helps in reasoning about the system.

1. INTRODUCTION
Adding new functionality to a large software system for which the documentation does not exist or is outdated becomes a difficult process that requires understanding of the system's underlying architecture. Many low-level details in the source code obstruct the process of creating a system's high-level, architectural abstraction that aids in reasoning about the system. A number of software "clustering" techniques have been developed to cope with this problem [5-7,10] but these techniques fail to provide much rationale behind the architecture. This becomes particularly important if we consider that the source code may actually contain accidental or emergent functionality and relationships which are not intended by the system's developers. For this reason, we posit that architectural recovery, and software clustering in particular, need to be accompanied by a system understanding activity, which includes the use of semantic information before any syntactic dependencies are considered. Various representations can be used to describe successive levels of system's abstractions. Incited by Perry and Wolf's observation [8] that the key architectural elements of a software system are (1) processing, (2) data, and (3) connecting, we have developed ARTISAn [3], a tool-supported, pluggable framework intended to aid program understanding and, ultimately, architectural recovery.[1]

Existing software architecture recovery and program understanding approaches do not necessarily and explicitly identify all the three types of architectural elements, but rather focus on the computation (i.e., processing) part alone. Program code elements (e.g., classes) that provide implementation of communication services often remain scattered across computational components in the recovered architecture, instead of being identified as separate entities. For example, client- and server-side elements in a system that leverages the TCP/IP communication often end up being parts of the components that are actually using them. Furthermore, not having connecting elements explicitly identified poses an additional challenge to engineers, who must reason about which parts of the system may interact with which other ones, and about the protocol of possible communication. Similarly, since data elements only contain the information that is used or transformed by processing elements, by identifying and then abstracting away data elements, the reasoning about the system is considerably improved (e.g., in applications built using data-intensive architectural styles, such as pipe-and-filter).

While in [3] we described the framework in more details and provided an illustrative example, in this paper we give a summarization of the approach with an emphasis on identification of program concerns at the architectural level. We use the term "concern" twofold: firstly, we distinguish among the parts of a system that account for the computation, interaction, and storage of the actual application-level information that is used and transformed (i.e., data) in trying to discover where their exact boundaries are and how they are realized in a system; secondly, the program elements are analyzed based on the usage scenarios in which they can participate. By separating program concerns at the architectural level, the evolution process becomes easier (e.g., changing the type of communication in a system, or porting to a new platform when the functionality remains the same), the overall complexity is reduced, and reusability improved.

2. OVERVIEW OF THE ARTISAn APPROACH
There are two main dimensions of concerns [9] in our approach: (1) *purpose*, i.e., functionality concerns

[1] ARTISAn stands for *A*rchitectural *R*ecovery via *T*ailorable, *I*nteractive *S*ource-code *An*alysis.

represented in the form of Processing, Connecting (also referred to as Communication), and Data elements (P, C, D); and (2) *usage* concerns, describing what is shared among parts of a system, and what is exclusively used by individual parts. For example, a class is an element that models a particular kind of object. This object can provide some functionality (P), or data (D), or communication (C). Another concern of interest is how this object is used in a system: based on its relationships with other objects it can belong to a region of objects exclusively used by some other regions, or to a shared region. These two dimensions are orthogonal, and complement each other.

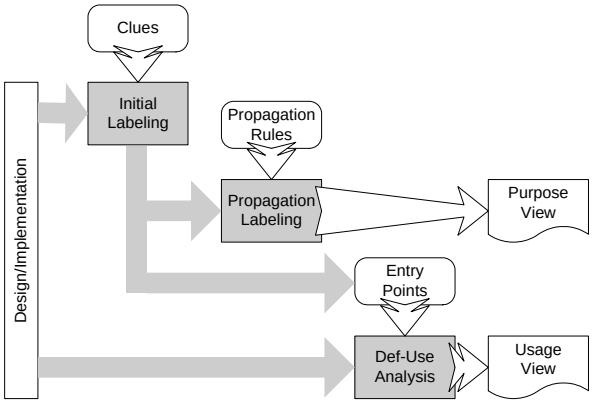

Figure 1: The ARTISAn framework.

Figure 1 describes the approach in more details. It comprises three steps that are initially performed sequentially but may then be revisited in any order by the user. The first step, termed *initial labeling*, results in a classification of individual elements (e.g., classes) into *processing* (P), *data* (D), and *communication* (C) [8] based on ARTISAn's *clues*. The result obtained during the initial labeling phase and a pluggable set of propagation *rules* provide input to the *propagation labeling* step. During this phase, some non-labeled elements become labeled (i.e,. classified as P, D, or C), based on the recognition of structural patterns and relationships with other, already labeled elements. Furthermore, this step also identifies possible structural inconsistencies among labeled elements and alerts the user about them. Initial labeling and propagation labeling result in an interpretation of the system that suggests the *purpose* of each of the system's individual elements.

Finally, during the *def-use analysis* phase, regions of related elements are identified based on invocation and inheritance relationships. The obtained regions distinguish between elements that are shared across regions and those that are exclusive to a region. The result of this phase is a system's *usage* view representation, which provides information on parts of the system that could be grouped together based on their usage scenarios.

Individually, the purpose and usage views provide the user with a classification of elements and their grouping based on usage analysis, respectively. These two views also complement each other. For example, if some unlabeled elements from the purpose view end up belonging to the same region with labeled elements of a single type, then one can surmise the purpose of the unlabeled elements. In total, our approach gives the user a better understanding of the system, and an opportunity to faster locate its parts that are of particular interest (e.g., for maintenance purposes).

3. ARTISAn CLUES AND INITIAL LABELING

At the most general level, software systems integrate *processing* elements that exchange *data* via *communication (connecting)* elements [8]. By determining the type of a system element, one can distinguish elements with application-specific functionality from those with application-independent functionality. Typically, processing elements provide application-specific functionality as they implement the system's requirements. On the other hand, communication elements typically provide application-independent interaction facilities. In Java, for example, classes interact by invoking each other's methods and/or sharing data through public variables, regardless of the classes' functionality. In addition, a number of off-the-shelf communication elements (e.g., middleware) are available. A useful starting point in understanding the source code of a system is thus in the reusable, application-independent nature of its communication elements. Similarly, data elements only contain the information that is used or transformed by processing elements. Therefore, identifying and then abstracting away data elements can further improve the reasoning about the system.

Software systems are generally described by their design or implementation models (e.g., class diagrams). Often, the models are too detailed, so that their understanding becomes obscured. In ARTISAn, constituent elements of these models (e.g., classes) are at first classified into the three aforementioned categories, providing a user the opportunity to quickly gain an overview on the purpose of individual elements and the structure of their composition. The process of classifying system elements into one of the three categories is termed *labeling*. The labeling is based on various design and implementation snippets, termed *clues*. Clues carry syntactic, semantic, and possibly domain-specific information, which is searched for in a system's model. For example, in Java-based systems, if there is an attribute in a class that declares a use of the standard network socket library *java.net.**, one can use it as a clue to the existence of a communication channel, which is directly used by this class. Similarly, all classes that implement the static method *main*, or inherit from the library class *java.lang.Thread*, or implement the *java.lang.Runnable* interface are likely to be processing elements. Furthermore, classes with no methods other than constructor(s) are very likely to be data elements.

The clues described above comprise a subset of all the clues that belong to ARTISAn's extensible and pluggable set of clues, which is not intended to be complete, but rather a starting point. We expect each programming paradigm and language, domain, and/or application to have their own set of

clues. ARTISAn distinguishes between the following clue categories:

- Domain-independent clues, such as the *Socket* class being classified as C, or a class with no methods being recognized as D. This is the most general set of clues.
- Domain-specific clues, e.g., in case a system is built on top of a known middleware platform. For example, an element of the *Siena* middleware is classified as C and the classes having access to *Siena* are appropriately marked.
- Application-specific clues, such as a class of name "*jigsaw.Resource*" in the Jigsaw Web server being recognized as D.

To visualize these concerns in a given system the ARTISAn tool uses different colors to represent different classes' labels on the class diagram, or combinations of these colors if a class has more than one label. Unlabeled classes remain transparent.

It should be noted that the clues are designed in such a way that applying them may identify one or more categories that an element belongs to (e.g., it is a processing and communicating element), but also one or more categories to which the element does not belong (e.g., it cannot be a processing or communication element). We refer to the former as an *inclusion* list, and to the latter as an *exclusion* list. This information is of particular importance during the propagation labeling phase.

4. ARTISAn RULES AND PROPAGATION LABELING

It is very likely that not all elements in a system can be labeled based on ARTISAn clues. However, the existing knowledge about a system could be used to reason additionally about the system. Information obtained from clues can be propagated from labeled elements to their neighboring elements (both labeled and unlabeled) when certain conditions are satisfied.

We refer to this kind of reasoning as *clue propagation*. Clue propagation serves as a basis for the ARTISAn *propagation rules*. The pluggable set of propagation rules and the result obtained during the initial labeling phase provide input to the *propagation labeling* step (Figure 1). During this phase, some non-labeled elements become labeled, based on the application of the propagation rules.

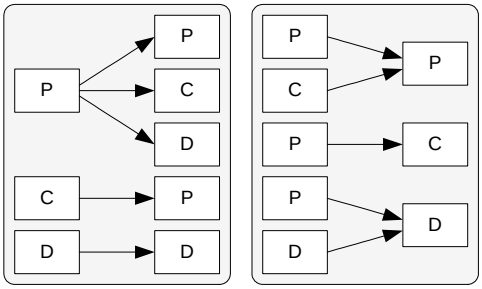

Figure 2: Propagation scenarios.

Propagation rules are derived from structural and interaction patterns involving different types of elements. Figure 2 illustrates these patterns.

The left-hand side of Figure 2 shows that a processing (P) element could call other processing, communication (C), and data (D) elements. In other words, there are no restrictions on what type of elements might be called by a processing element. On the other hand, our experience has shown that off-the-shelf communication elements usually do not invoke any other element (e.g., in the case of socket-based communication), or if they do, the invoked elements could only be processing elements (e.g., in the case of COM-based communication element). We should note here that some technologies that are used to bridge across different computing platforms (e.g., the Java-to-COM bridge) may involve communication elements calling other communication elements. However, in those cases we would be dealing with specialized solutions that would allow us to recognize such situations on a case-by-case basis. Furthermore, these cases would be amenable to capture by specialized domain- or application-specific propagation clues and rules, which would result in an appropriate identification and labeling of all such elements. Finally, data elements are expected to be passive entities that may perform some rudimentary internal processing, but are otherwise not interacting with processing or communication elements. To describe the propagation rules in a more formal way, we will use the right-hand side of Figure 2, which is a transpose of its left-hand side (e.g., only P or C can call P).

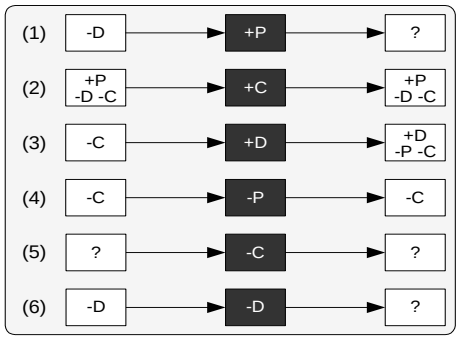

Figure 3: Propagation rules based on scenarios.

Based on the caller-callee relationships in Figure 2, we can deduce six propagation rules, which are depicted in Figure 3. For example, rule 1 in Figure 3 states that if an element is known to be a processing element (denoted by +P in the middle box), then all elements that call it (its callers) cannot be data elements (denoted by -D). The rationale for this is as follows: from the right-hand side of Figure 2 we know that either a P or a C can call another P. This implies that the caller cannot be D. Since we do not know whether the actual caller is P or C, we only write that it is not D. In this way we avoid having to make an early (but possibly incorrect) decision. The question mark in the right-hand column of rule 1 indicates that we cannot say anything about the elements being called by that element (its callees). Similarly, if an element is known not to be a processing

element (-P), as in rule 4, then neither the caller nor the elements being called can be communication elements. This rule is again derivable from Figure 2. If an element is not P then it is either C or D. We know that C can be called only by P, and that D can be called by P or D. It follows that C or D can be called by at most the union of their callers, which is P or D. Since we do not know whether it is P or D, we simply write that it is not C. All other rules can be derived in a similar way.

The algorithm for applying propagation rules is based on the changes in the inclusion and the exclusion lists for each of the system's elements. As long as there are changes in any of the two lists (e.g., an unlabeled element becomes labeled, or a processing element becomes classified as non-connecting element), an appropriate propagation rule is run.

This step also provides support for identifying any potential rule conflicts. For example, if a class is identified as a processing element through one propagation rule, but also as not a processing element using another rule, then we know that either the clue or one of propagation rules was erroneous. Conflicts are easily identifiable due to their simple implementation representation (+P and -P) and ARTISAn reports all inconsistencies to the user. At that point, the user has the choice to manually label the elements if they are of a known type, ignore the discovered conflict (e.g., in case when a helper class of known functionality has conflicting labels), or use that information to modify the set of clues, and rerun the propagation labeling step. In the last case, both the user and the tool are "learning" about new clues that could be used for other systems.

5. DEF-USE ANALYSIS IN ARTISAn

The next step in our approach is the identification of regions, i.e., groups of system elements that are closely related, or independent of other parts of the system. To this end, we adapt *def-use analysis*. Def-use analysis is an approach that has already been used in literature. [4,6] proposed the use of dominance analysis to identify regions of related modules. These regions indicate parts of a system that are exclusively used by its other part(s) and those that are shared. Each of the identified regions has an *entry point*, which is a module where processing starts (e.g., a class with the `main()` method implemented). Entry points in ARTISAn are directly obtained from the initial labeling step (Figure 1). Those are all elements that satisfy the "main" clue, but also include elements that are able to create a new processing thread. The rationale for this lies in the fact that systems often spawn their own subsystems by creating separate processing threads. We can identify spawning using clues which were discussed previously.

The information about regions enables a user to more easily recognize system elements that belong together. The *usage view* thus complements the *purpose view* by combining information about high-level functionality of individual elements with information about regions of related elements.

6. CONCLUSION

This paper discussed ARTISAn, an exploratory and tailorable framework that helps in program understanding tasks. The framework comprises replaceable components to accommodate the exact programming environment and supports developers in understanding large-scale, multi-lingual source code. The approach leverages a two-dimensional separation of concerns, which results in two orthogonal views of a system: (1) high-level functionality view (i.e., purpose) and (2) usage view of system elements. In tandem, these views provide the user with a better understanding of the system, and an opportunity to faster locate the parts that are of particular interest (e.g., for maintenance purposes). Furthermore, the approach illustrates how the same program element might be involved in more than one concern of interest.

There are numerous ways to improve our technique. Some of them include the use of reliability metrics that would depend on the reliability of each of the clues and rules applied, and may then be used to (automatically) resolve any of possible inconsistencies that result from the labeling process. The other direction of improvement is in providing a richer set of domain- and application-independent clues. For example, the fact that delegating classes act as facades or wrappers to other classes, might turn up to be useful in recognizing communication-processing relationships. Furthermore, the presented rule set can be extended by additional rules that include additional system concerns as subcategories of the three major element groups (P, C, and D), such as GUI (P) and interruptible communication (C) type elements. Such a richer propagation rule set would lead to a better understanding of the purpose of a system's elements and, ultimately, its architecture.

7. REFERENCES

1. M. Bauer and M. Trifu, "Architecture-Aware Adaptive Clustering of OO Systems," in *Proc. of the Eighth European Conference on Software Maintenance and Reengineering* (CSMR 2004), Tampere, Finland, March 24-26, 2004
2. D. R. Harris, A. S. Yeh, and H. B. Reubenstein, "Extracting Architectural Features from Source Code," In *Automated Software Engineering* 3, 1996, pp. 109-138.
3. V. Jakobac, A. Egyed, and N. Medvidovic, "Improving System Understanding via Interactive, Tailorable Source Code Analysis." To appear in *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, Edinburgh, UK, April 2005.
4. T. Lengauer and R. E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1, pp. 121-141, July 1979
5. N. Medvidovic and V. Jakobac, "Using Software Evolution to Focus Architectural Recovery," In *Journal of Automated Software Engineering*, To appear, 2004
6. N. Mendonca and J. Kramer, "An Approach for Recovering Distributed System Architectures," In

Journal of Automated Software Engineering, vol. 8, pp. 311-354, 2001

7. H. A. Müller, K. Wong, and S. R. Tilley "Understanding Software Systems Using Reverse Engineering Technology," In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994
8. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, vol 17 no 4 Oct 1992
9. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," In *Proc. Intl'l Conf. Software Eng.*, May 16 - 22, 1999 Los Angeles, USA, pp. 107-119
10. K. Wong, S. Tilley, H. A. Müller, and M. D. Storey, "Structural Redocumentation: A Case Study," *IEEE Software*, Jan. 1995, pp. 46-54.