Feature Location and Connector Recovery: New Approaches for Software Understanding*

Daniel Simon Thomas Eisenbarth
Universität Stuttgart
Breitwiesenstr. 20–22
70565 Stuttgart, Germany
{eisenbarth,simon}@bauhaus-stuttgart.de

Abstract

The most challenging and time consuming task software engineers have to master is to understand a software system's behavior and its implementation. About 50% of the time used for maintenance is devoted to program understanding [13]. This task has to be performed for different objectives such as correcting or optimizing a system, making a system match up to its environment's evolution, and reusing some of a system's functionality.

1 Feature Location

For the maintenance and evolution of a large software system the comprehension of the program as a whole is in many cases neither feasible nor advisable. For the implementation of specific change requests it is often sufficient to have partial knowledge about the system. This part of the paper describes an approach of localizing the system's relevant parts by starting from a user's point of view utilizing dynamic and static analyses.

1.1 Introduction

Demands for changes to software systems are often expressed in domain specific terms, e.g., *"Add a new object type to the predefined set of objects of the drawing tool"*.

In order to implement the requested changes the programmer only has to find out where the various features are implemented. This search for the features' implementations can be realized by either static or dynamic analyses and is improved by the combination of both (see also Figure 1).

*This work is supported by T-Nova Entwicklungszentrum Darmstadt.

In general, it is not obvious which components implement a given feature. The precise mapping of the features to the program's source or architectural components is essential for minimizing the time needed for program understanding. In previous work [6, 7, 8] we have shown that the analysis of feature interdependencies further leads to the understanding of the software's architecture.

1.2 Starting Points

Besides our own work there are the following other attempts to locate features in code.

Wilde and Scully [15] take a "dynamic" approach by analyzing traces of program's execution. Their *Software Reconnaissance* is based on test-data and finds features that may or may not be executed due to the test-data (e.g., features that may be turned on or off by command-line switches). The program is run twice, and by subtracting the parts of the program present in both runs, one yields the area of code where the feature is implemented.

In contrast, Rajlich et al. [3] propose a semi-automatic method that involves a human-guided search on the program's *Dependence Graph*. The method takes into account documentation as well as domain knowledge and therefore can be quite expensive. Since this approach is human-guided it seems a bit more flexible than Wilde's work.

In [14], Wilde and Rajlich together perform a case study on legacy Fortran code and propose to use their methods depending on the long term goals of the maintenance task. For large and infrequently changed programs, Software Reconnaissance is superior, while for smaller and frequently modified systems the Dependence Graph methods seems preferable.

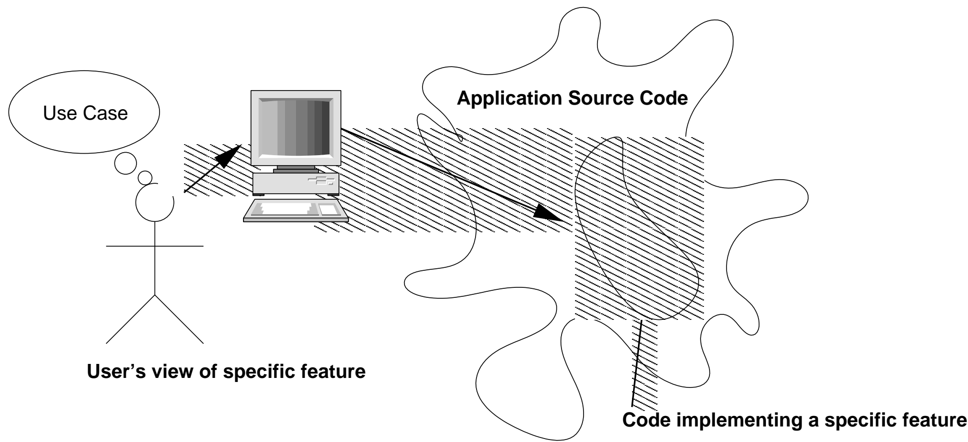

Figure 1: Feature location in a "slice" of the code.

1.3 Research Directions

The Bauhaus project at the Universität Stuttgart [5] provides a strong platform for software analysis. Advanced compiler technologies for data flow and control flow analysis are used to support reverse engineering activities and code validation. Since the focus is currently on the programming language C, the primary objectives for feature location are software systems written in that language.

So far, we have successfully used sophisticated combinations of call profile analyses and mathematical founded concept formation [8] to gain quick and "deep" insight into medium sized C programs (up to 100KLOC). The information gained about legacy software is not only useful for maintenance tasks, but also gives valuable hints for rearchitecting code assets towards product line architectures [6]. Based upon that experience, the Bauhaus approach addresses several research questions for feature location.

In most cases, the features located by Wilde or Rajlich can easily be isolated from other features. How can more abstract or intermingled features be located and separated from each other? How can program usage scenarios give relevant insights?

Both Wilde and Rajlich study individual features. By looking at sets of features, how can commonalities, differences, and feature interdependencies be explored?

While Wilde deals only with dynamic information, Rajlich just takes the static dependencies into account. How can the integration of dynamic and static program knowledge be realized? How does this combination lead to better results?

After the features are located, can the information be used to extract them from the system? Can the feature's interfaces be located? Can the software's architecture be recovered or validated?

Finally, the developed methods and tools have to be evaluated in case studies and controlled experiments. The result have to be compared with alternative methods. The integration of the feature location and extraction into the existing Bauhaus infrastructure will ensure its usability.

2 The Recovery of Aggregatable Connectors

This part of the paper describes basic ideas of a recovering technique capable of finding connectors in a system's source code. The analysis serves for program understanding and reverse engineering purposes. The connectors focused by the presented technique connect dynamic instances of atomic components.

2.1 Introduction

The proposed connector recovery method yields multiple results useful for program understanding and reverse engineering:

- A connector's interface is described in an abstract way by trace graphs and communicating sequential processes (CSP) useful when a connector's implementation has to be replaced or altered.
- After changes are applied the new implementation can be easily checked for consistency by

comparing the trace graphs before and after the change (syntactic check).

- All code fragments contributing to a connector instance are marked as well as overlap with other connector instances. So changing a connector's implementation is facilitated, side effects on other connections can be detected.
- Discussion of global changes to the system's architecture are simplified since the overall system structure is revealed.
- Boundaries for reuse of components are made explicit facilitating effort estimation.

2.2 Components and Connectors

The terms *component* and *connector* are used in different meanings in literature, so it is necessary to define them for this paper's scope as follows:

Components are locations of computation and state. All application relevant state information is stored in the system's components and all activity is invoked from within components. In our scope components are instances of atomic components as defined by [10]. Components possess ports where connectors can be attached.

Connectors govern control and data flow between components, thus their communication. Connectors are either atomic connecting facilities as routine call and shared variables or they consists of groupings of those atomic connectors. Connectors possess roles attachable to components' ports.

Configurations describe how components' ports and connectors' roles are attached.

Shaw [12] argues that there is a need for explicit connector support in an architectural description language (ADL). A connector is described by its protocol, its connector type, its roles, and its implementation.

The term **aggregated** connector denotes a connector consisting of some other connectors and components. An connector is **aggregatable** if it may be used as part of an aggregated connector.

Connectors vary a lot in possible implementation strategies even when the intended semantics—considered from a high level of abstraction a system architect normally takes—are similar or even the same.

Unlike components, connectors' implementations generally do not preserve the prominent role connectors play in a system's architecture [4] because connector types tend to get intertwined into code. So connectors expose themselves not directly, although there are attempts made to make connectors explicit. In legacy code we typically find connectors implicitly coded.

2.3 Aggregated Connectors

Systems implemented in conventional programming languages use at least routine call, parameter passing, and variable access as means of connection, which form an assembly language of connection [12].

Aggregating routine calls, shared variables, already aggregated connectors, and components yields so called **aggregated connectors** resembling superposed connectors introduced by [9]. The term *aggregated connector* is better suited for our notion of connectors because the superposition mechanism used is always aggregation.

Allen [2] researches the possibilities of describing connectors. He uses a CSP-like description for the interaction of connectors and components which is useful here, too.

2.4 Recovery and Description of Aggregated Connectors

A simple example of an aggregated connector is a combination of multiple function call connectors and shared variables.

Call and shared variable connectors are directly visible in our RFG representation [11]. For shared variable connectors one has to prove that a pair of set/use relationships really expresses data flow.

Nevertheless, these relationships are not helpful as tool to a reengineer which possesses little knowledge of a system as the RFG tends to be cluttered with call, set, and use relationships. Therefore we use the RFG only as hint where to look for connectors.

An abstract data object encapsulates state information. The source code in Figure 2 illustrates a situation were an ADO component of *List* with interface `Init`, `Attach`, and `GetFirst` is used as connector between component A (`rep_of_comp_a`) and B (`rep_of_comp_b`). *List* is aggregated with an encryption connector (`Encode` and `Decode`). This is revealed by data flow analysis: Tracking the parameters of functions in the interfaces of *List* and *Encrypt* reveals that the result of `Encode` is passed to `Attach`, the result of `GetFirst` is passed to `Decode`. The connectors get aggregated by overlaying

```
void rep_of_comp_a (void) {
  Init ();
  item sth = ComputeSth();
  Attach(Encode(sth));
}

void rep_of_comp_b (void) {
  item sth = Decode(GetFirst());
  use(sth);
}

int main (void) {
  /* call rep_of_comp_a and rep_of_comp_b()
     in some arbitrary order */
}
```

Figure 2: Example code for aggregation of List and Encoding connectors.

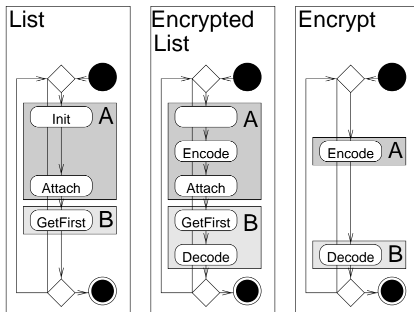

Figure 3: Traces for connectors in Figure 4.

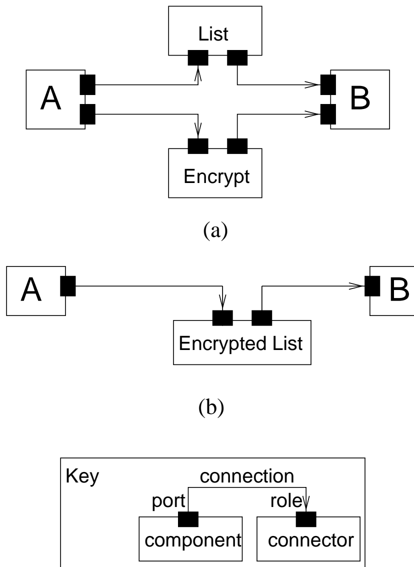

Figure 4: Non Aggregated vs. aggregated connector.

For describing connectors and component configurations one may use a notation as shown in Figure 4 similar to the one proposed in [1].

Figure 4 shows both a configuration for the non-aggregated case and the aggregated case. The traces recovered for our example are shown in Figure 3; they use UML-activity diagram notation. Shaded subsections denote which component was in control while the calls to the interface were made. The Encryption List trace shows the aggregated protocol. Ports and roles in Figure 4 contain the respective parts of the traces, e.g., in Figure 4(a) the lower port of A contains the "A" part of Encrypt's trace.

2.5 Conclusions

As we have seen in the introduction, architectural interconnection is a vital information to the maintenance engineer and reengineer. Today, there are no automated techniques providing information beyond components and high level connectors like pipes.

Work on connector recovery aims at facilitating the daily work of maintenance programmers and reengineering analysts—or anyone else who has to understand an implementation and/or apply modifications.

References

[1] Gregory Abowd, Robert Allen, and David Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–64, October 1995.

[2] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[3] Kunrong Chen and Václav Rajlich. Case Study of Feature Location Using Dependence Graph. In *Proceedings of the International Workshop on Program Comprehension*, pages 241–249, Limerick, Ireland, June 2000. IEEE Computer Society Press.

[4] Stéphane Ducasse and Tamar Richner. Executable Connectors: Towards Reusable Design Elements. In *Proceedings of the 6th European conference held jointly with the 5th ACM*

SIGSOFT symposium on Software engineering, pages 483–499, Zurich Switzerland, September 1997.

[5] Thomas Eisenbarth, Rainer Koschke, Erhard Plödereder, Jean-François Girard, and Martin Würthner. Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen. In *1. Workshop Software-Reengineering*, pages 17–26, Bad Honnef, Universität Koblenz-Landau, 1999. Fachberichte Informatik, Nr. 7-99.

[6] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Herleitung der Feature-Komponenten-Korrespondenz mittels Begriffsanalyse. In *Proceedings of the 1. Deutscher Software-Produktlinien Workshop*, pages 63–68, Kaiserslautern, Germany, November 2000.

[7] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Derivation of Feature-Component Maps by Means of Concept Analysis. In Pedro Susa and Jürgen Ebert, editors, *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 176–179, Lisbon, Portugal, March 2001.

[8] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Feature-Driven Program Understanding Using Concept Analysis of Execution Traces. In *Proceedings of the International Workshop on Program Comprehension*, pages 300–309, Toronto, Canada, May 2001. IEEE Computer Society Press.

[9] David Garlan. Higher-Order Connectors. In *Proceedings of the Workshop on Compositional Software Architectures*, Monterey, CA, USA, January 1998.

[10] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute for Computer Science, University of Stuttgart, Stuttgart, Germany, 2000.

[11] Rainer Koschke, Jean-François Girard, and Martin Würthner. An Intermediate Representation for Reverse Engineering Analyses. In *Proceedings of the Working Conference on Reverse Engineering*, Honolulu, HI, USA, October 1998.

[12] Mary Shaw. Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. *D.A. Lamb (ed) Studies of Software Design, Proceedings of a 1993 Workshop Lecture Notes in Computer Science No 1078*, pages 17–32, 1996.

[13] Anneliese von Mayrhauser and A. Marie Vans. Program Understanding – A Survey. Technical Report CS-94-120, Colorado State University, Fort Collins, CO, USA, August 1994.

[14] Norman Wilde, Michelle Buckellew, Henry Page, and Václav Rajlich. A Case Study of Feature Location in Unstructured Legacy Fortran Code. In Pedro Susa and Jürgen Ebert, editors, *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 68–75, Lisbon, Portugal, March 2001.

[15] Norman Wilde and Michael Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance: Research and Practice*, 7:49–62, January 1995.