Pattern-Supported Architecture Recovery*

Martin Pinzger and Harald Gall
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1, A-1040 Vienna, Austria, Europe
{pinzger, gall}@infosys.tuwien.ac.at

Abstract

Architectural patterns and styles represent important design decisions and thus are valuable abstractions for architecture recovery. Recognizing them is a challenge because styles and patterns basically span several architectural elements and can be implemented in various ways depending on the problem domain and the implementation variants. Our approach uses source code structures as patterns and introduces an iterative and interactive architecture recovery approach built upon such lower-level patterns extracted from source code. Associations between extracted pattern instances and architectural elements such as modules arise which result in new and higher-level views of the software system. These pattern views provide information for a consecutive refinement of pattern definitions to aggregate and abstract higher-level patterns which finally enable the description of a software system's architecture.

1. Introduction

Developing complex software systems requires a description of the structure or structures, which comprise software components, the externally visible properties of those components, and the relationships among them [1]. Such a description, called *software architecture*, also is basic for further engineering activities concerning reuse, maintenance, and evolution of existing software components and systems.

Changes are in the nature of software systems and also have impacts on the architecture of a system. In most cases they cause a drift between the *as-designed* and *as-built* architecture because not seldom they are only realized in the implementation but not in the design of a software system. In the field of product lines the impact of such changes increases even more because there is not only one single system but a family of systems. Therefore, changes are one primary reason for analyzing and recovering the architecture of existing systems.

Architecture recovery refers to all techniques and processes used to abstract a higher-level representation (i.e., software architecture) from available information such as existing artifacts (e.g., source code, profiling information, design documentation) and expert knowledge (e.g., software architects, maintainers). Basically this means the extraction of those building blocks which constitute architectural properties and finally the software architecture. From point of this view we think of *architectural styles* and *patterns* which are inherent in almost any design and thus are primary objectives for architecture recovery. But recognizing such styles and patterns is a challenge because they comprise several architectural elements (subsystems, modules, classes, functions, variables) and are implemented in various ways (depending on the problem at hand and on the programming language).

In this paper we extend our architecture recovery framework described by Jazayeri et al. [8] and introduce an iterative and interactive architecture recovery approach which is based on patterns. In this context we refer to patterns as *solutions to recurring problems on different levels of abstraction (e.g., code patterns, design patterns, and architectural patterns)* [13, 5]. Each pattern has typical properties and elements which indicate it. These so called *hot-spots* are the point to start architecture recovery, because they enable an abstraction of higher-level patterns. In our approach we primarily base on existing knowledge of experts and design documents and on a fast and effective recognition of hot-spots. Therefore, we start architecture recovery with gathering knowledge about the software system and specify pattern definitions, which pertain to hot-spots in terms of source code structures. We use an extended string pattern matching technique to match the pattern definitions with source code. Associations between extracted pattern

*This work is funded by the European Commission under EUREKA 2023/ITEA-ip00004 'from Concept to Application in system-Family Engineering (CAFÉ)'.

instances and architectural elements such as modules arise which result in new views of the software system. These *pattern views* contain the key information of higher-level pattern and enable the refinement of pattern definitions until finally the software architecture is reconstructed.

Following this introduction, Section 2 provides related work concerning architecture recovery using patterns. Section 3 describes the pattern-supported architecture recovery approach in detail. Evaluation of the method with a case study is presented in Section 4. Finally, Section 5 summarizes this work and indicates future work.

2. Related work

Architecture recovery has received considerable attention recently and various frameworks, techniques and tools have been developed. Basically, existing knowledge, obtained from experts and design documents, and various tools are mandatory to solve the problem. Hence, a common idea is to integrate several tools in architecture workbenches such as *Dali* [9]. In this a variety of lexical-based, parser-based and profiling-based tools are used to examine a system and extract static and dynamic views to be stored in a repository. Analyses of these views are supported by visualization and specific analysis tools. They enable an interaction with experts to control the recovery process until the software architecture is reconstructed.

Concerning architecture reconstruction much work has been on techniques which combine bottom-up and top-down approaches. Bottom-up they use reverse engineering tools to extract source models (e.g., Abstract Syntax Tree) and top-down they apply queries to extract expected patterns. Fiutem et al. [3] describe such an approach. They use a hierarchical architectural model that drives the application of a set of recognizers. Each recognizer works on the Abstract Syntax Tree (AST) and is related to a specific level of the architectural model. They produce different abstract views of the source code which describe some architectural aspects of the system and are represented by hierarchical architectural graphs.

Harris et al. [7] outline a framework that integrates reverse engineering technology and architectural style representations. In bottom-up recovery the bird's eye view is used to display the file structure and file components of the system, and to reorganize information into more meaningful clusters. Top-down style definitions place an expectation on what will be found in the software system. These expectations are specified by recognition queries which are then applied to an extrated AST. Each recognized style provides a view of the system and the collection of these views partially recovers the overall design of the software system.

Guo et al. [6] outline an iterative and semi-automatic architecture recovery method called *ARM*. Existing knowledge gained from design documentation is used to define queries for potential pattern instances which are then applied automatically to extracted and fused source model views. Human evaluation is required to determine which of the detected pattern instances are intended, which are false positive and false negative. ARM supports patterns at various abstraction levels and uses lower-level patterns to build higher-level patterns and also composite patterns. In this way the approach aims particularly at systems that have been developed using design patterns whose implementations have not eroded over time.

Another approach which uses source models and queries as basic inputs for architecture recovery is introduced by Sartipi et al. [12] and called *Alborz*. The problem is viewed as approximate graph matching problem whereas the extracted source models and defined queries are represented as attributed relational graphs. Based on existing knowledge obtained from experts and design documents abstract patterns are defined using an Architectural Query Language (AQL). Each query is expanded into a graph which next is approximately matched with the source model graph using the branch and bound search algorithm. In each iteration the user may refine his queries and generate a more accurate model.

These related approaches [3, 7, 6, 12] and our approach have in common that they all take into account patterns to reconstruct the architecture of a software system. But there are two basic differences: first in the view of patterns and second in their extraction. We regard patterns as the key elements of software systems residing in *all* levels of abstraction. Thereby we start pattern recognition from the lowest level (i.e., source level) and use *hot-spots* to stepwise abstract higher-level patterns. Hot-spots indicate patterns and are represented by meaningful source code structures (e.g., variables, functions, data structures, program structures). To detect such hot-spots in source code we apply extended string pattern matching which facilitates fast and effective queries. In contrast the related approaches mentioned above regard patterns as *associated architectural elements* (e.g., a specific sequence of function calls) residing in *higher* levels of abstraction (e.g., code-strucure level). Basically, these approaches transform the software system into a source model representation such as an AST and apply queries to recognize expected patterns. The transformation of existing artifacts (e.g., source code) implies the use of reverse engineering tools such as parsers and profilers which extract source models containing the architectural elements. But these reverse engineering tools typically are time and memory consuming and in a first step of architecture recovery too costly. In this context our string pattern matching approach represents a more effective solution which also allows a later involvement of other pattern matching techniques, such as those described before.

3. Pattern views

Source code typically is structured and contains semantically rich programming constructs such as variables, functions, data structures, and program structures which indicate patterns and therefore are valuable inputs for architecture recovery. The extraction of these basic patterns provides the user with additional views of the software system which we call *pattern views*. In this paper we primarily focus on the generation of such views and introduce an approach which consists of the following steps:

1. Analysis and pattern candidate identification:
   Based on design documentation and expert knowledge expected pattern candidates are identified.

2. Pattern definition:
   Based on the expected pattern candidates appropriate pattern definitions are taken from a pattern repository or otherwise generated using a specific pattern language.

3. Pattern recognition:
   Pattern definitions are matched with source code and information about recognized patterns is stored in a repository.

4. Pattern view computation:
   Associations between recognized pattern instances and other architectural elements are computed. They result in various new views of the software architecture.

5. Analysis of patterns and views:
   Resulting views and recognized patterns are analyzed to abstract architectural patterns. Already applied patterns are refined, new pattern definitions are validated and stored in the repository.

In the following sections we describe each of the steps in more detail.

3.1. Pattern identification

The primary focus of architecture recovery is on finding *key information* (i.e., patterns) which enables the description of architectural properties of an existing software system [8]. The information base containing this patterns consists of all artifacts comprising the software system (e.g., source code, documents, running system). This leads to a huge amount of data so that knowledge from experts and existing software documents is necessary to extract the essential information. Therefore the primary activity in this step is to investigate existing design documents and contact experts who are familiar with the design of the software system to gain knowledge about the software system and its primary architectural properties and their implementation by patterns. Clues about these expected patterns are crucial to initialize and control the recovery process (e.g., communication between components is implemented in C using sockets).

3.2. Pattern definition

Based on the information gathered in the first step appropriate pattern definitions are either user-defined or taken from a repository. Because our approach takes into account significant text and structural information of source code we use a pattern definition language which facilitates *regular expressions* and *source code structures*. Currently there are several tools for string-based source code analyses (e.g., grep, perl, LSME, SCRUPLE) available but they all either do not support structures, need a huge amount of memory or disk space or require a parser for each target programming language. We extended Knor's et al. [10] ESPaRT (Enhanced String Pattern Recognition Tool) to allow pattern specification in XML. ESPaRT overcomes the mentioned shortcomings by implementing a lexical tool which is based on regular expressions and considers structural information. It provides a definition language which enables the specification of patterns with preconditions and follow-up examinations (a pattern match has to fulfill the precondition and can be further investigated through a sub match definition). Figure 1 shows the primary structure of an ESPaRT pattern definition in XML-format. The term *pattern expression* stands for an ESPaRT definition which can be a simple regular expression or a more complex one containing a combination of ESPaRT specific commands and regular expressions.

```
<pattern id="patternid">
  <precondition match="true">
    <!-- pattern expression -->
  </precondition>
  <match>
    <!-- pattern expression -->
  </match>
  <submatch match="true">
    <!-- pattern expression -->
  </submatch>
</pattern>
```

Figure 1. ESPaRT pattern definition

The organization of pattern definitions in different sections is crucial for addressing the problem that patterns are implemented in various ways. The clue is to give a more general pattern definition (e.g., a text block containing one or more hot-spots) which limits the search space but nevertheless takes into account all relevant matches which are used as inputs for subsequent detail matching processes. In this way ESPaRT enhances accuracy and performance of the pattern recognition process.

3.3. Pattern recognition

The pattern recognition process of ESPaRT takes the definitions specified in the former step as input and matches them with the information base (e.g., source files). In a first matching process the huge amount of information is sliced and text blocks are extracted based on the specified pattern definition. A further optional condition for matched text blocks can be defined in the precondition section. This filters possible wrong matches and minimizes the input for the following matching process where the extracted pattern instances are investigated in more detail by the application of sub match pattern definitions. The result of this stepwise pattern recognition process contains all detected primary and sub pattern instances described by quadruples (*pid*, *fid*, *start*, *end*) where *pid* indicates the pattern definition and *fid*, *start*, and *end* the location (source file name, start and end line number) of the matched pattern. All generated quadruples are stored in a central repository for further analysis and computation of pattern views.

3.4. Pattern view computation

Based on the idea of *views* from Kazman et al. [9] related pattern instances consider the architecture of a software system from an important point of view - patterns. Like other extracted views – such as static and dynamic call views – pattern views overlap and complement one another. The primary objective of this step is to associate matched *pattern instances* with themself and other architectural elements and to visualize them. The results are views which provide engineers with information for refining pattern definitions and guiding the recovery process in the right direction. For the visualization of views we use existing tools such as *Rigi* from Wong et al. [14]. Regarding the associations we focus on three different pattern views: pattern composition view, pattern-element view, and pattern-module view.

The architecture recovery process is initiated by pattern definitions which primarily focus on the key elements of patterns. To continue the process and abstract higher-level patterns it is necessary to refine these pattern definition statements. One possible way is to analyze the composition of patterns. Taking the location property of the extracted and stored pattern instances as input a simple algorithm (e.g., SQL-statement) computes a directed graph showing the composition of patterns (Figure 2).

A directed association between two pattern instances PI_1 and PI_2 indicates that pattern PI_2 is *part of* pattern PI_1. Par-

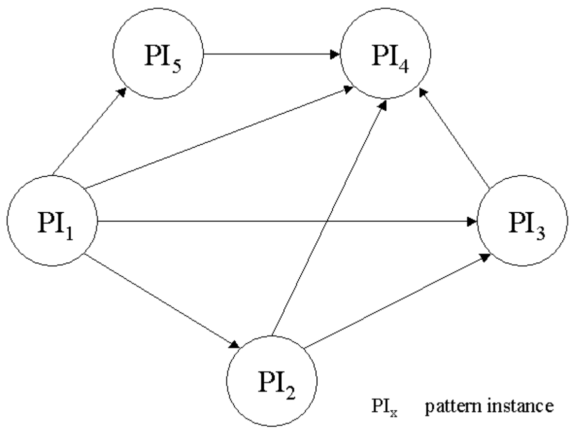

Figure 2. Pattern composition view

ticularly pattern instances with a high fan-in (PI_4) or fan-out (PI_1) are of interest because they depict pattern instances which on the one hand are aggregated and on the other hand are key elements of patterns.

Commonalities of architectural elements are an interesting aspect for further investigations because they provide information which patterns are appropriate for aggregation and abstraction. But additional views that show these commonalities in more detail are required. One step towards a more detailed view is the combination of pattern instances and source model elements, such as functions, variables or data structures. For example Figure 3 represents a combination of patterns and function calls. The source model elements are obtained from reverse engineering tools such as *Imagix4D* or *SniFF+* and stored in a central repository. A simple algorithm expressed, for example, in SQL is appropriate for relating detected pattern instances and source model elements. A directed association between a pattern and a source model element is established if the element is *part of* the pattern. The resulting directed graph shows all considered source model elements which constitute the pattern and the commonalities between patterns (e.g., PI_1 and PI_2 have f_1 and f_4 in common).

The pattern-element view considers associations between pattern instances at a function-level. Continuing the architecture recovery process increases the abstraction level because architectural elements are aggregated and abstracted. At higher levels aggregated and more abstracted elements such as modules are added to the input data of succeeding recovery iterations. Typically a module is implemented in one source file containing functions, variables, and data structures. Based on the file-relation of modules and matched pattern instances a pattern-module view is computed which shows associations between modules. This means that two modules are associated if a pattern in-

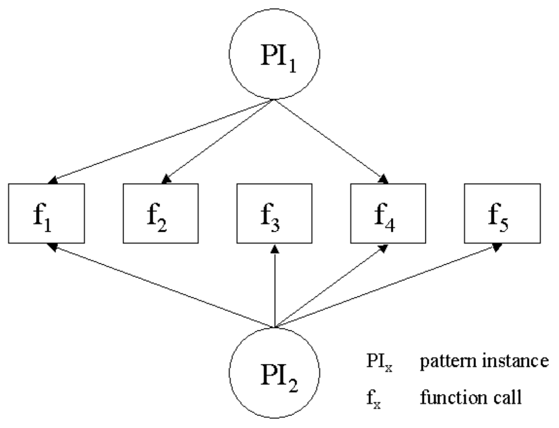

Figure 3. Pattern-element view

stance (of the same pattern definition) is identified in both modules. An example is shown in Figure 4 where the modules M_1, M_2, and M_3 are associated by pattern definition PD_1. Similar pattern definitions often indicate similar responsibility and assist the engineer in classifying architectural elements (e.g., modules). This is performed in the next step of our architecture recovery process.

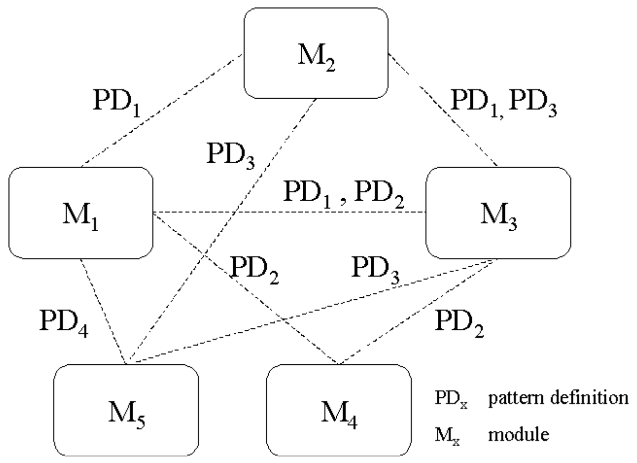

Figure 4. Pattern-module view

3.5. Analysis of patterns and views

Pattern views show significant pattern instances which could be key building blocks of a software system. Particular attention is paid to information which supports aggregating and abstracting patterns. A basic guiding principle is to investigate those pattern instances in more detail which are similar because they are potential candidates for aggregation and abstraction. Similarities arise from common *used* architectural elements such as functions, variables, data structures, lower-level patterns, already aggregated and abstracted patterns, and also components. Considering potential candidates additional views and also the calculation of metrics such as proposed by Sartipi et al. [12] assist in understanding the degree of similarity. They guide engineers towards the right decision which pattern definitions to refine and which pattern instances to aggregate and abstract in the next recovery iteration.

4. Case study

We applied our pattern-supported software architecture recovery to a distributed intrusion detection system called *SPARTA* [11] which consists of approximately 100 modules (100 KLOC) implemented in *C* and *Java*. The primary task of this software system is to detect distributed intrusion patterns (e.g., telnet chains, spreading worms). This is done by sniffing network traffic and applying certain rules to the input data. Matched packets are stored in a database and queried by mobile agents. For the purpose of demonstration and evaluation we basically focused on one architectural property called *data communication* [8] and considered to answer basic questions, such as:

- Which components contribute to communication?
- Who are the senders and receivers?
- Does the sender block the receiver?
- Is an architectural style such as Client/Server used?

First phase of recovery

We started the architecture recovery process with gathering knowledge about the problem domain of intrusion detection and the basic building blocks of related software systems. First, there is a tool interacting with the user to configure the sniffer by defining rules that specify the network packets which should be captured. Second, there is the sniffer-tool which, based on its configuration, observes the network traffic and generates an event whenever a packet occurs which conforms to a specified rule. Each event generated in this way and its properties are sent from the sniffer-tool to the logging-tool that writes the data to a repository. All three primary building blocks are connected through communication channels realized by *TCP/IP-sockets*.

Based on this information we specified initial pattern definitions to query socket-patterns of both implementation languages Java and C. The clue is to specify the hot-spots of patterns and to take into account as many implementation variations as possible. Relating to sockets such a hot-spot is the socket-creation statement (e.g., *mySocket = socket(...)* in C or *mySocket = new Socket(...)* in Java). In terms of C this results in a pattern definition as shown in Figure 5. The interpretation of this definition is: match text blocks starting with "{" and ending with "}" containing a string "= socket(...);" where "..." can be an arbitrary string. An analog pattern definition was also specified for sockets implemented in Java.

```
<pattern id="C-Socket">
  <match>
    <block start="{" end="}">
      <text>= socket(</text>
      <anytext />
      <text>);</text>
    </block>
  </match>
</pattern>
```

Figure 5. Initial pattern definition for matching potential socket implementations in C

For the recognition of pattern instances we fed all source files of each programming language and the corresponding pattern definition to ESPaRT [10]. The result of this first pattern recognition process is presented in Table 1 describing the location of each recognized pattern instance by source file name, start and end line number, and the identifier of the pattern definition (*C-S* for socket in C, *J-S* for socket in Java). Each match models a quadruple (fid, pid, start, end) which is stored in the repository.

file	location start - end	pattern-id
log.c	116 - 195	C-S
	810 - 826	C-S
snort.c	321 - 354	C-S
	2039 - 2095	C-S
SnortPlugin.java	889 - 943	J-S
	945 - 973	J-S
	976 - 1064	J-S
	1068 - 1105	J-S
	1108 - 1129	J-S

Table 1. Detected C and Java socket patterns

The advantage of this string pattern specification over regular expressions is the capability to manage text blocks. These blocks contain important information around detected hot-spots which is mandatory to recognize potential higher-level patterns.

Before the analysis process is started the stored data has to be preprocessed and represented in a form which supports the user in detecting eye-catching elements. We used Wong's Rigi-tool [14] for the visualization of views and some small Perl Scripts to transform the stored data into the Rigi Standard Format (RSF).

Regarding the first architectural question we built the pattern-module view shown in Figure 6. Each stored quadruple is read from the repository. The file and pattern identifiers constitute this view: each unique file identifier indicates a node; an edge between two nodes is generated if two quadruples contain the same pattern identifier but different file identifiers. One problem occurred because of the need to separate pattern definitions for each implementation language (Java and C). Both have different pattern identifiers but were used for the same purpose. The view does not differ between varying implementation languages and hence we assigned the same identifiers for the Java and C pattern definitions.

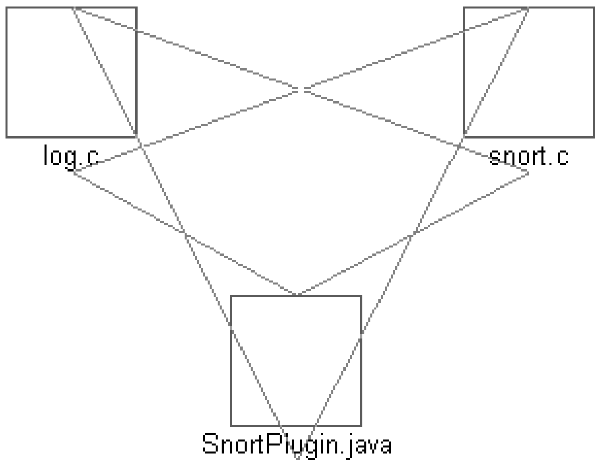

Figure 6. Pattern-module view of source files containing potential socket pattern instances

The resulting graph in Figure 6 shows that the modules *log.c*, *snort.c*, and *SnortPlugin.java* contain expected socket implementations. The mutual associations between the components arise from the fact that Rigi cannot display bidirectional associations. Unfortunately, by analyzing this graph it was not possible to conclude more detailed architectural information such as, for example, which module implements a server and which one a client. To get this information we had to reconsider the implementation of server and client sockets in each programming language and refine our pattern definitions.

Second phase of recovery

Before going deeper into investigations of the matched text blocks we looked up some implementation details about socket programming. In C client and server sockets have the same data type, but differ by the function calls following the *socket()*-statement. A client typically executes *connect(socket, address, ...)* to connect to a server. On the other hand a server first binds his socket to an address and next listens and waits for requests. The corresponding statements are *bind(socket, address, ...)*, *listen(socket, ...)*, and *accept(socket, ...)*. In Java the implementation differences between client and server sockets are similar to C and additionally varies in different data types (*Socket* for clients and *ServerSocket* for servers). Based on this information we refined our pattern definitions to detect implementations of client and server components. Figure 7 shows the refined pattern definition of a client socket implemented in C.

```
<pattern id="C-ClientSocket">
  <precondition match="true">
    <text>SOCKET</text>
    <variable id="mySocket" />
  </precondition>
  <match>
    <block start="{" end="}">
      <variable id="mySocket" />
      <text>= socket(</text>
      <anytext />
      <text>);</text>
    </block>
  </match>
  <submatch match="true">
    <text>connect(</text>
    <variable id="mySocket" />
    <anytext />
    <text>);</text>
  </submatch>
</pattern>
```

Figure 7. Refined pattern definition for matching client sockets implemented in C

Basically we extended the number of elements and added a precondition and a postcondition which perform additional checks. The precondition contains a check for a data type definition "SOCKET" in the extracted text block. A variable called "mySocket" was introduced to reuse a matched socket identifier in the postcondition. This leads to the following extended interpretation: match text blocks starting with "{" and ending with "}" containing a string "= socket(...);" where the string of the matched identifier is assigned to the variable "mySocket"; check each matched text block as valid if it contains a variable definition "SOCKET" for the matched identifier; further investigate each valid text block if it contains a string "connect(mySocket...);" where "mySocket..." stands for the matched variable identifier followed by an arbitrary string. While the match and precondition elements specify the creation of a socket, the sub match element specifies the statements which indicate a socket as client or server.

Based on the information obtained from the first iteration we applied the specified pattern definitions (2 for C and 2 for Java) on the reduced search space. The outcome of the recognition process is shown in Table 2. Four pattern identifiers indicate the various socket implementations for Java and C (*C-CS* for client socket in C, *C-SS* for server socket in C, *J-CS* for client socket in Java, and *J-SS* for server socket in Java).

file	location start - end	pattern-id
log.c	116 - 195	C-CS
snort.c	321 - 354	C-SS
SnortPlugin.java	889 - 943	J-SS
	945 - 973	J-CS
	976 - 1064	J-CS
	1068 - 1105	J-CS
	1108 - 1129	J-CS

Table 2. Detected client and server socket patterns

Two previously recognized pattern instances in the source files *log.c* and *snort.c* were omitted because they did not match the precondition. The remaining quadruples represent client and server components. Whereas *log.c* implements a client and *snort.c* a server, *JavaPlugin.java* implements both. Regarding the next architectural questions mentioned above we first had to examine which client communicates with which server. Basically this is not always expressed in the source code and further expert knowledge is necessary, such as which kinds of sockets are used. Referring to our case study we knew that the sockets under study are based on TCP/IP and thus use an IP-address and a TCP-port number as connection parameters. A client who wants to communicate with a server opens a socket with the corresponding server-IP-address and the port to which the server is bound. One of the possible ways to get this information is by computing a view of the various client and server pattern instances and their called functions plus parameters. Taking the involved quadruples and an extracted source model we generated the pattern view shown in Figure 8.

This view represents a server socket in C (C-SS) and a client socket in Java (J-CS), their called functions and accessed variables. To extract the essential information we had to perform a more detailed analysis. The server accesses the variables *sd* and *addr* in its *bind()*-statement where *sd* is the socket identifier and *addr* contains the port which the socket is bound to. To identify the port number we further investigated the variable *sin_port* and retrieved a constant value *p*. The four recognized Java client socket implementations also use this constant port number in their *Socket()*-statement. The result showed that module *Snort-*

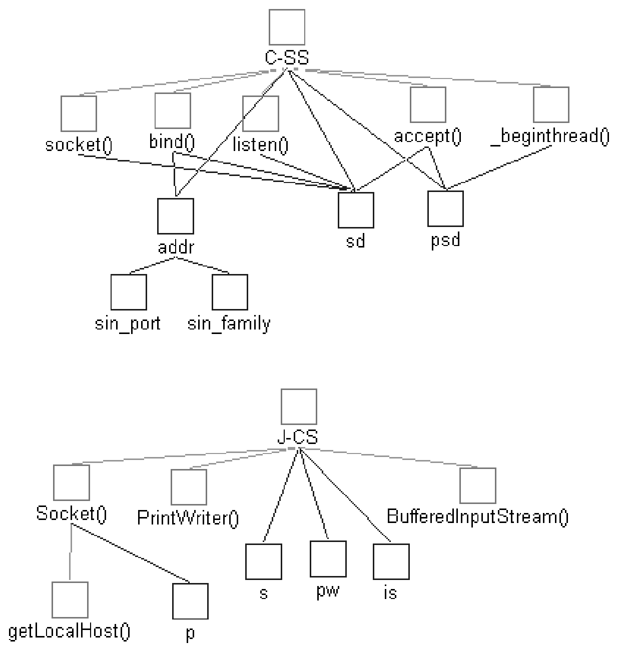

Figure 8. Pattern-element view of server and client pattern associated with source model elements

Plugin.java implements client sockets which connect to the server socket implemented in *snort.c*. Another such analysis showed that *JavaPlugin.java* also implements a server socket listening on the constant port number *q* which is accessed by a client socket implemented in *log.c*. Further analyses of this combined pattern and source model element view also indicated that both servers use threads to handle requests and are not blocked by clients.

Finally, we discussed our gained information with an expert and got corresponding results: There are two socket servers waiting for client requests which are either used for rule or event transfer. Rules are transferred from four clients implemented in *JavaPlugin.java* to the server in *snort.c*. Events are transferred from one client realized in *log.c* to the server implemented in *JavaPlugin.java*. Both servers execute a separate thread for each client request.

5. Conclusions

In this paper we presented an architecture recovery approach based on patterns defined on the level of source code structures. Using expert knowledge and design documents the key elements of expected patterns are specified. An extended string pattern matching technique allows a fast and effective recognition of these pattern definitions. The extracted pattern instances are associated with other architectural elements and form new views of the software architecture: a pattern-composition view, pattern-element view, and a pattern-module view. Analyzing these views pattern definitions are refined to aggregate and abstract higher-level patterns until the software architecture is reconstructed to the extend required by the engineer. The architecture recovery process thereby starts with specific properties of a system (e.g., “socket communication”) and iteratively completes the recovered architecture descriptions by refinement of patterns.

We demonstrated the applicability of our approach on a real-world case study where we recovered the data communication property of an intrusion detection system. The case study also showed the necessity of existing knowledge to analyze and interpret the generated pattern views. Our approach is straightforward in the respect that it starts with patterns built from source code structures but exhibits its full strengths in the generation of views revealing (inter-)relationships between architectural elements, patterns, and modules.

Ongoing work concentrates on the computation of additional pattern views and the formulation of guidelines to analyze these views and control the recovery process. More case studies will be performed to further demonstrate the applicability of the approach in different application domains, especially in the field of product families.

References

[1] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley, Reading, Mass. and London, 1998.

[2] J. Bosch. *Design and Use of Software Architectures: Adopting and evolving a product line approach*. Addison-Wesley, Reading, Mass. and London, 2000.

[3] R. Fiutem, A. Tonella, G. Antoniol, and E. Merlo. A cliché-based environment to support architectural reverse engineering. In *Proc. of the International Conference on Software Maintenance*, pages 319–328, Monterey, November 1996. IEEE Computer Society Press.

[4] H. Gall, R. Klösch, and R. Mittermeir. Pattern-driven reverse engineering. In *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, Las Palmas de Gran Canaria, Spain, February 1998. Springer-Verlag.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass. and London, 1995.

[6] G. Guo, J. Atlee, and R. Kazman. A software architecture reconstruction method. In *Proc. of the 1st Working IFIP Conference on Software Architecture*, pages 225–243, San Antonio, Texas, February 1999. Kluwer Academic Publishers.

[7] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Reverse engineering to the architectural level. In *Proc. of the 17th*

International Conference on Software Engineering, pages 186–195, Seattle, Washington, April 1995. ACM Press.

[8] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, Reading, Mass. and London, 2000.

[9] R. Kazman and S. J. Carriére. View extraction and view fusion in architectural understanding. In *Proc. of the 5th International Conference on Software Reuse*, pages 290–299, Victoria, BC, Canada, May 1998.

[10] R. Knor, G. Trausmuth, J. Weidl, and F. van der Linden. Reengineering c/c++ source code by transforming state machines. In *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, Las Palmas de Gran Canaria, Spain, February 1998. Springer-Verlag.

[11] C. Krügel and T. Toth. Sparta - a mobile agent based intrusion detection system. In *Proc. of the IFIP Conference on Network Security (I-NetSec)*, Belgium, November 2001. Kluwer Academic Publishers.

[12] K. Sartipi, K. Kontogiannis, and F. Mavaddat. A pattern matching framework for software architecture recovery and restructuring. In *Proc. of the 8th International Workshop on Program Comprehension*, pages 37–47, Limerick, Ireland, June 2000.

[13] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Vol. 2*. John Wiley & Sons, 2000.

[14] K. Wong, S. Tilley, H. Müller, and M. Storey. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, December 1999.