Revealer: A Lexical Pattern Matcher for Architecture Recovery *

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

Abstract

Program comprehension is crucial for software maintenance activities and is supported by reverse engineering techniques. Many of them analyze source code and use parsers to create higher-level representations of software systems that are more meaningful to engineers. But the application of parsers is for some reasons not always desirable. In this paper, we introduce Revealer a lightweight source model extraction tool that combines advantages of lexical analysis with syntactical analysis. It uses an easy-to-use pattern language that supports engineers in defining pattern definitions of diverse granularity depending on the problem at hand. In this way our tool enables fast extraction of simple and complex code patterns that allow engineers a quick insight into particular architectural aspects that are expressed via source code patterns.

Keywords: program comprehension, reverse engineering, lexical analysis, architecture recovery, patterns

1. Introduction

An important key principle of program understanding is that *higher-level representations facilitate understandability*. A basic step of abstracting higher-level representations is based on the extraction of source models, consisting of lower-level elements and their relations from source code. The majority of program understanding tools, such as for example Rigi [23] or Software Bookshelf [7], rely on traditional parsing techniques or use lexical tools in the style of grep, awk [1], lex [15], or Perl [22] to generate source models.

Parsing techniques focus on the extraction of syntactic constructs that are specified by *context free grammars*. A scanner such as lex tokenize source files and a parser reads this token stream and generates an Abstract Syntax Tree (AST). By using grammar specifications parsers are able to recognize complex syntactic constructs and generate more abstract and thereby language independent representations of source code (i.e., source models). On the other hand parsers place stringent constraints on the artifacts from which source models are to be extracted, such as for example all header files must be available and syntactically correct.

These idiosyncrasies and the existence of unparsable programs often drive engineers back towards lexical analysis tools, such as the ones mentioned above. They base on *regular expressions* which can be matched with every kind of textual information imposing few structural constraints. In general, lexical analysis tools are fast, robust, and easy to use. Whereas the first two advantages result from the transformation of regular expressions to deterministic transition automatons (DTA), latter results from the facility of intuitive pattern definitions. But accurate pattern definitions of purely lexical analyzers turn out to be a major drawback because of its limitations in handling whitespaces and syntactic structures.

Considering these advantages and drawbacks of both approaches we introduce Revealer, a lightweight source model extraction tool that combines lexical with syntactic analysis capabilities. Following the criteria of lexical analysis our tool is based on regular expressions constituting the basic pattern elements. These elements also handle the problem of whitespaces and other textual elements such as strings or comments that often complicate definitions. Following the criteria of structural analysis various pattern elements are combined in a tree-like structure to specify typical syntactic constructs as found in programming languages or similar textual representations. But in contrast to parsers the user specifies only his parts of interests with the level of accuracy he needs or can provide without considering all details of context free grammar specifications. In this way and by the use of XML, whose characteristics further facilitate tree-

*This work is partially funded by the Austrian Ministry for Infrastructure, Innovation and Technology (BMWIT) and the European Commission under EUREKA 2023/ITEA-ip00004 'from Concept to Application in system-Family Engineering (CAFÉ)'.

Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)
1095-1350/02 $17.00 © 2002 **IEEE**

like composition of elements, our approach is *lightweight*, *flexible*, and *tolerant* [17] with respect to source model extraction.

Following this introduction, Section 2 reviews related work concerning lexical-based analysis focusing on lightweight source model extraction. Section 3 describes the basic context of our work and the application of Revealer in the field of architecture recovery. The specification language is described in Section 4 and demonstrated by a case study in Section 5. Section 6 discusses our approach and addresses future work.

2. Related work

The necessity of program comprehension in software engineering activities has led to the development of several source code analysis tools. Basically, they can be divided into two groups: *lexical analyzers* and *syntactical analyzers*. Concerning lexical analysis there are several tools available, such as for example grep and its variants, that use regular expressions to match strings in text files. But their use for source model extraction is limited in describing syntactic constructs with pure regular expressions and no or only few capabilities to access matches.

More advanced lexical analyzers such as the scanner generator *lex* [15] or *awk* [1] include access to matches by providing support to define actions that are performed when a match occurs. Both tools define patterns as rules consisting of a regular expression and an action. Matches are stored in specific variables and actions can access and assign them to user-defined variables (i.e., tokens). Often these tokens are used as input to other applications such as parsers for further processing. A slight difference between lex and awk is that lex provides no support for unification, whereas awk does so for a subclass of regular expressions. Nevertheless, these tools, stand-alone, are not suitable for source model extraction, basically by a lack of expressibility to describe syntactic constructs as used by most programming languages.

Murphy and Notkin explored these problems of lexical analyzers for source model extraction and introduced the *Lexical Source Model Extraction* (LSME) tool [17]. The approach is based on regular expressions and uses two classes of tokens (single-character and identifier) to specify hierarchical related expressions. Based on this definition LSME generates scanners to extract the specified tokens from text files, and analyzers to further process or reject matches.

Cox and Clarke developed a similar approach called *MultiLex* [6]. MultiLex is a parser-like tool and focuses on the extraction of low-level models from source code by applying analyzers that take the token stream emitted by a scanner (lex) as input. It differs from LSME in that it requires the user to specify each base level token instead of having only a single-character or an identifier. Also the implementation of hierarchical matching in MultiLex is not expressed by pattern definitions but by linking analyzers together.

Our approach differs from LSME and MultiLex in the way tokens are handled and organized. We extend tokens to *objects* that can be combined to search-graphs. In contrast to tokens, objects facilitate flexible and intuitive combination of pattern elements such as the specification of hierarchical, inner or conditional patterns and provide users with more options to control the level of accuracy for source model extraction. Objects also enhance the handling and tracing of extracted information by storing and analyzing it directly in pattern elements instead of shifting it to additional analyzers.

Concerning syntactical analyzers Moonen proposes a solution for lightweight source model extraction in the form of *island grammars* [16]. Island grammars combine the behavior of parsing with that of lexical approaches by only analyzing the interesting parts of source code (i.e., islands) and ignoring uninteresting ones (i.e., water). Nevertheless, users have to handle grammar specifications and have to generate separate parsers for each problem. Our approach is 100% lexical-based and provides pattern elements that allow fast and intuitive specifications in XML, and avoids regeneration of extractors when changing pattern definitions. Moreover, the object-oriented implementation in Perl [22] enhances extensibility and portability, that often is missing in other approaches.

3. Using source code patterns for architecture recovery

A major aspect of reverse engineering is the generation of meaningful higher-level representations from available information, called *architecture recovery*. As described in [13] and shown in Figure 1 source code pattern-based architecture recovery is an iterative and interactive process consisting of three major phases:

1. *Extraction* of architectural elements from software artifacts available using predefined patterns and domain/application knowledge.

2. *Abstraction* of higher-level views by refinement of pattern definitions and aggregation of coherent architectural elements.

3. *Visualization* of extracted and computed architectural views.

The most important problem of architecture recovery is the abstraction of information, in which the non-trivial

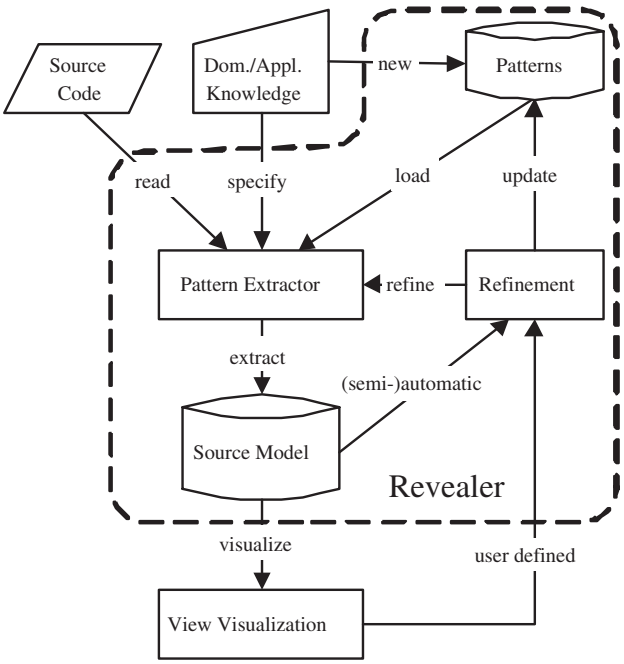

Figure 1. Source code pattern-based architecture recovery framework

```
package hot.spot.example;
import java.net.ServerSocket;

public class Server extends Thread {
  ...
  public void run()
  {
    ServerSocket serverSocket = null;
    ...
    serverSocket = new ServerSocket(2448);
    ...
    while (true) {
      ...
      s = serverSocket.accept();
      ...
    }
    ...
  }
  ...
}
```

Figure 2. Hot-spots of server socket pattern

question "Which architectural elements should be combined to a more abstract and self-contained element (e.g., component)?" has to be answered. In [18] we described an approach that views *patterns* [9, 19] as such self-contained architectural elements which occur on all levels of abstraction. Each pattern represents specific architectural characteristics – so-called *architectural hot-spots* – that indicate it and are reflected in source code. Such clues can be for example identifiers, single or sequences of method-calls, type or attribute definitions, control structures, or also comments. Figure 2 shows an example of hot-spots of a code pattern for implementing a socket server in Java. In this example the most important clues are given by the server socket creation statement that indicates this module as socket server listening to port number 2448, and the while loop containing an accept-statement handling client requests. Further, the surroundings containing these two hot-spots yields architectural information, meaning for example the method calling it, or the class and package implementing it.

Based on the idea of hot-spots we developed *Revealer* to extract these code clues and the surrounding text blocks containing them. The extraction of surroundings is important because they often contain the information that is needed to further classify and abstract higher-level patterns. Revealer introduces a simple specification language that focuses on the extraction of hot-spots and patterns, and assists users in specifying pattern definitions. We will describe this specification language in the next section.

4. Pattern specification

Programmers need a specification language that facilitates specification of pattern definitions ranging from simple regular expressions to complex parser-like expressions including hierarchical and recursive patterns. Additionally, the specification language should be simple, intuitive, and extensible. Based on these requirements and on the objective to extract text blocks containing architectural hot-spots we developed a pattern specification language introducing *specific pattern elements* that are based on *regular expressions* and easily can be combined to simple and complex pattern definitions.

Concerning simple pattern definitions for matching text-blocks and contiguous sequences of characters and strings that constitute hot-spots, such as for example, programming statements we introduce *primitive pattern elements* including:

- *RegExp* to handle regular expressions as supported by Perl [22].
- *StringExp* to handle strings and whitespaces.
- *Block* to handle text-blocks enclosed by start and end delimiter.

The *StringExp* element extends *RegExp* and facilitates definitions of simple text patterns by handling the problem of whitespaces that often complicates pure regular expressions. Also the *Block* element addresses a problem of pure regular expressions that is in defining arbitrary nested blocks. Although, these primitive pattern elements can be used stand-alone, the strength of the specification language is in composing more complex pattern definitions.

Revealer pattern compositions form a directed graph in which each pattern element is a node and each relation between two elements is an edge of the graph. To describe such graphs (i.e., pattern definitions) we use XML documents. XML [24] is an adaptable standardized markup language that also includes facilities for data management and reuse of existing documents. The syntax of XML documents and hence also for our pattern definitions is given by a DTD (Document Type Definition). Figure 3 shows the DTD used by Revealer specifying the general graph structure for composing pattern elements.

```
<!ELEMENT RevealerPattern ((pe | rel)*)>
<!ELEMENT pe (attr*)>
<!ATTLIST pe
  id    ID              #REQUIRED
  type  NMTOKEN         #REQUIRED>
<!ELEMENT rel (attr*)>
<!ATTLIST rel
  id    ID              #IMPLIED
  from  IDREF           #REQUIRED
  to    IDREF           #REQUIRED
  type  NMTOKEN         #REQUIRED>
<!ELEMENT attr EMPTY>
<!ATTLIST attr
  name  NMTOKEN         #REQUIRED
  value CDATA           #REQUIRED>
```

Figure 3. DTD of Revealer pattern definitions

Basically, the DTD contains the mandatory elements to describe graphs including *pe* as nodes and *rel* as edges. Both graph elements can have zero or more attribute elements (*attr*) each having a *name* and a *value*. Each node (*pe*) has a unique *id* and a *type* parameter specifying the type of the pattern element (e.g., RegExp, StringExp, Block). Two nodes are connected by an edge (*rel*) by specifying a *from*- and a *to*-reference given by node identifiers. As well as nodes, edges have a *type* parameter specifying the kind of relation between two nodes. In the following paragraphs we will describe basic types of nodes and edges currently supported by Revealer and give some examples of pattern definitions.

To match contiguous text blocks, pattern elements are linked to *pattern sequences*. Each element of the sequence must match, and only whitespaces or other ignorable characters (e.g., comments or strings) are allowed between matches. To link pattern elements to a sequence the relation of type *next* is used. Revealer also provides a separate pattern element of type *Pattern* to handle pattern sequences. Figure 4 shows an example of a simple pattern sequence matching *"new ServerSocket(...)"* statements. The Pattern element *SS* denotes a pattern sequence consisting of a StringExp *new* and a Block *pB*. The SringExp contains an attribute with name *expr* specifying to match a string *"new ServerSocket"*. Following a matched StringExp there must be a text-block enclosed by *"("* and *")"* specified by two attributes *startDel* and *endDel* of the Block element. The first relation of type *next* links the StringExp and the Block element to a sequence that next is connected to the Pattern element, handling the pattern sequence, by a *contain*-relation.

```
<RevealerPattern>
  <pe id="SS" type="Pattern"/>
  <pe id="new" type="StringExp">
    <attr name="expr"
          value="new SocketServer"/>
  </pe>
  <pe id="param" type="Block">
    <attr name="startDel" value="("/>
    <attr name="endDel" value=")"/>
  </pe>
  <rel from="new" to="param" type="next"/>
  <rel from="SS" to="new" type="contain"/>
</RevealerPattern>
```

Figure 4. Simple pattern sequence to match Java server socket creation statements

The relation of type *contain* only is used by some pattern elements, such as for example by Pattern as shown in Figure 4. Basically, such a relation references a list of pattern elements that is processed by Revealer depending on the type of pattern element that contains the list. For example, the *ORPattern* element references a list of patterns by its *contain*-relation and returns the first pattern that matches. In contrary the *ANDPattern* references a list of patterns where all pattern elements must match to get a valid *ANDPattern*.

Structural text such as source code is organized hierarchically containing nested text blocks. Revealer supports the description of such text structures by *nested pattern definitions*. To nest patterns Revealer provides a relation of type *constraint* that references an inner pattern definition. The consequences of inner pattern definitions are that higher-level patterns match only if all of their inner pattern definitions match.

An example of use of nested pattern definitions is shown

in Figure 5 where we reuse the previous pattern definition of Figure 4 and extend it to extract all Java classes containing a server socket creation statement. To do this we add a pattern sequence *CL* consisting of pattern elements matching the signature and the implementation block of Java classes (e.g., *"class Server ..."*). The inner pattern definition is specified by the pattern sequence *SS* and linked to the element *block* by a *constraint*-relation. Consequently, the block element only matches a class implementation block if it contains a server socket creation statement.

```
<RevealerPattern>
  <pe id="CL" type="Pattern"/>
  <pe id="class" type="StringExp">
    <attr name="expr" value="class"/>
  </pe>
  <pe id="clId" type="Var"/>
  <pe id="block" type="Block">
    <attr name="startDel" value="{"/>
    <attr name="endDel" value="}"/>
  </pe>
  <rel from="class" to="clID"
       type="next"/>
  <rel from="clID" to="block"
       type="next"/>
  <rel from="CL" to="class"
       type="contain"/>
  <rel from="block" to="SS"
       type="constraint"/>
</RevealerPattern>
```

Figure 5. Hierarchical pattern definition to match Java classes implementing a socket server

Hierarchical patterns also are used to drive the matching process towards a stepwise extraction of code patterns. First, coarse grained text blocks are extracted that in the next steps are investigated in more and more detail. Users can specify several levels of inner patterns, hence resulting in several levels of detail. This capability of hierarchical patterns make them useful in controlling the performance and accuracy of source model extraction.

Grammar specifications of programming languages facilitate several variations that have to be considered when extracting source models. Basically, such variations are due to *optional elements* in statements and due to *arbitrary sequences* of statements. Source code analyzers such as parsers manage these variations by adding branches to their search trees. Also Revealer includes these capabilities in its specification language providing specific pattern elements such as *ORPattern* or *IFPattern*. The ORPattern is used to combine pattern elements by a logical *or*. The first pattern that matches is returned. An IFPattern provides users with the facility to specify *conditions* in pattern definitions in the form of if-then-else statements equal to that used in programming languages. The only difference to programming languages is that the if-condition and the two branches are pattern definitions.

Summarized, Revealer provides a set of pattern elements and a set of relations that facilitate easy specification of simple and complex pattern definitions. Figure 6 shows all elements and their relations currently implemented by Revealer. The figure also shows that all pattern elements, except *SendTo* and *Definition* which are used for handling pattern definitions and matches, are derived from one base class *PatternElement*. Hence, adding new pattern elements is easy and basically requires only an implementation of the matching algorithm of the new pattern element.

We used our pattern specification language to define particular patterns to represent architectural aspects such as:

- *communication*
  sockets, remote procedure calls, messaging.
- *synchronization*
  locking, synchronization interfaces.
- *control*
  scheduling, dispatching, event handling.

A more detailed catalogue of architectural aspects can be found in [13]. Given the architectural characteristics of a particular software system the reverse engineer can select some expected patterns from a central *pattern repository* to be searched for in the source code. In this paper, we concentrate on the architectural aspects of *communication* that are implemented in our case study. On-going work include the specification and addition of pattern definitions concerning other architectural aspects such as those mentioned above.

The architectural characteristics are derived from information about the application and its domain as usual in architecture recovery activities [20].

5. Case study in recovering architectural aspects with Revealer

The case study demonstrates the application of Revealer for extracting conceptual architectural views, such as for example described by Hofmeister et al. [11], of an existing distributed software system. Conceptual architectural views are close to the application domain and hence assist users in program comprehension. Architectural views also provide specific information related to architectural properties such as *communication* or *synchronization*. Depending

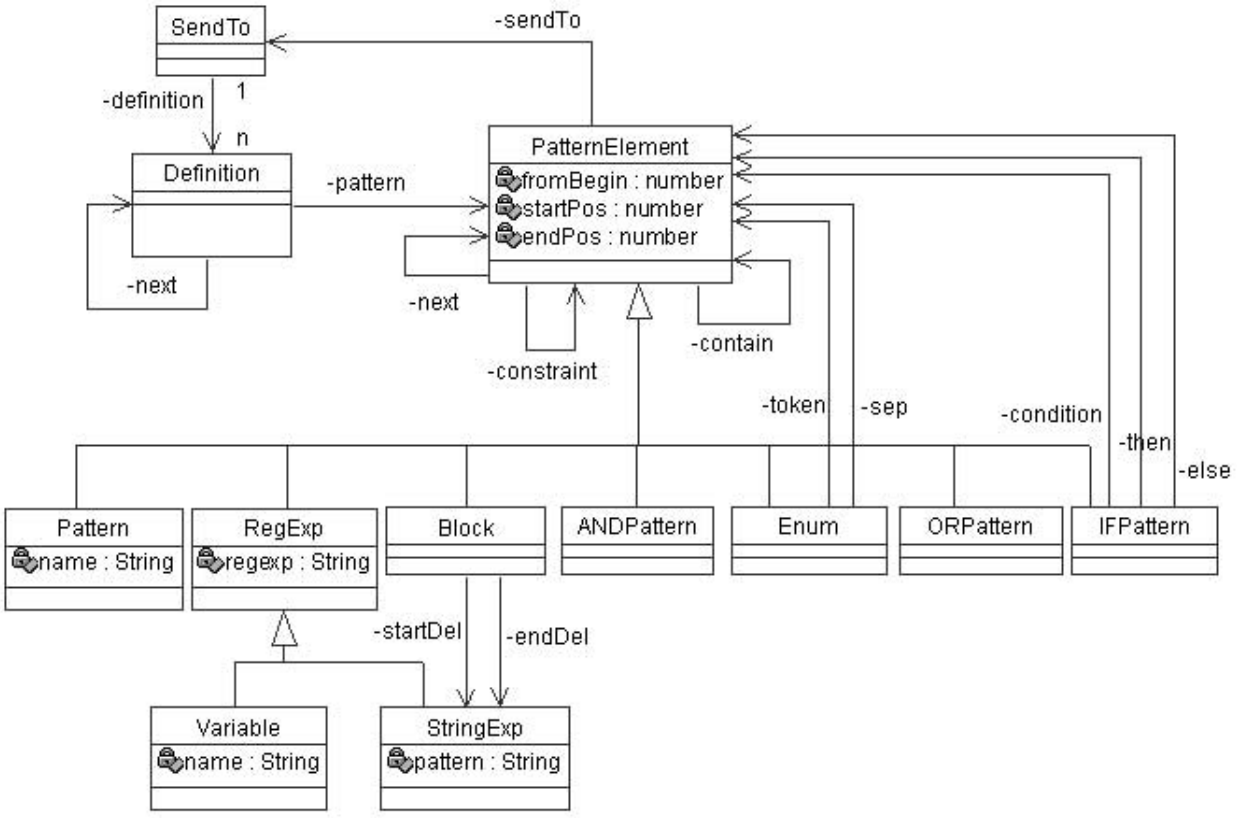

Figure 6. Class diagram of pattern elements currently supported by Revealer

on the implementation language there are various code patterns indicating potential higher-level patterns that are typically used to realize these architectural properties. Therefore, the objective using Revealer is to extract these patterns and their associated architectural information.

The software system under study is a distributed intrusion detection system called *SPARTA* [14] that 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.

Because of its distribution over a number of hosts communication is a crucial architectural property of SPARTA. In general, engineers apply several patterns to implement *communication* such as for example sockets, messaging, or remote procedure calls. Each of these patterns is indicated by specific attributes, commands, and classes. For example using sockets implies the creation of a socket and connecting it to an address. Consequently, such statements are architectural hot-spots and at the same time anchors for extracting architectural information.

Concerning socket communication we extracted hot-spots and their surrounding text blocks comprising methods, classes, and modules by combining the following three pattern definitions:

- socket creation statement
  (e.g., *socket* = new Socket(...))
- method signature
  (e.g., *id* (...) [throws ...] { ... })
- class signature
  (e.g., class *id* ... { ... })

The overall structure of the pattern definition is shown in Figure 7 whereas the *socket creation statement* is the hot-spot. The first definition *class* extracts all *classes* that contain a socket creation statement. The implementation block of each matched class is sent to the second pattern definition *method* that matches all *method-implementations* containing the hot-spot. To extract *all* socket creation statements contained in a matched method we sent the matched block to the third pattern definition *socket creation*. The XML specifications of all three pattern definitions are similar to the ones given in Figure 4 and 5.

The result of this pattern comprises all modules, classes, methods and hot-spots (i.e., socket creation statements). Ta-

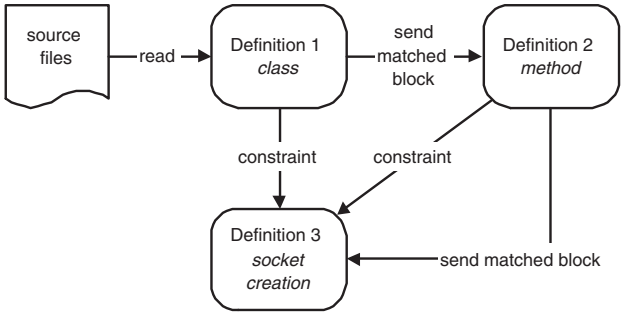

Figure 7. Nested pattern definition to extract modules, classes, methods, and hot-spots of socket communication pattern in Java

ble 1 shows an example of information extracted by Revealer concerning matched pattern definitions and their locations in the SPARTA module *SnortPlugin.java*.

SnortPlugin.java	byte-location
SnortPlugin	23555 - 31633
clearRules ()	2697 - 3569
s = new Socket(InetAddress ...)	155 - 202
addRules ()	3603 - 5946
s = new Socket(InetAddress ...)	1279 - 1326
addRule ()	5946 - 7044
s = new Socket(InetAddress ...)	191 - 238
removeRule ()	7065 - 7623
s = new Socket(InetAddress ...)	61 - 108

Table 1. Extracted information of client socket pattern

To obtain more information about the communication property in SPARTA we extended the pattern definitions adding possible variations of socket implementations in Java including server, multicast and datagram sockets. We further created pattern definitions regarding other communication mechanisms in Java such as Remote Method Invocation (RMI).

The extracted information results in views, such as represented in Figure 8, showing the architectural aspect *communication* of SPARTA with regard to sockets. For every architectural aspect one can draw a picture like this. As a consequence the engineer gets a comprehensive picture of those parts of the source code that are concerned with communication aspects. Visualization of these pictures could be performed by interfacing known reverse engineering tools such as Imagix-4D or Rigi to exploit their visualization capabilities, but this functionality although currently being implemented is beyond the scope of this paper.

Concerning precision of Revealer Table 2 shows the number of matched socket patterns (hot-spots and classes containing them). From this point of view Revealer seems to be very accurate. But as with other lexical tools accuracy depends on the pattern definition meaning that inexact pattern definitions reduce the number of valid matches and increase the number of false positives and false negatives.

socket type	# of hot-spots	# of classes
ServerSocket	6/6	6/6
ClientSocket	17/17	7/7
DatagramSocket	0/0	0/0
MulticastSocket	9/9	5/5

Table 2. Matched Java socket patterns and classes

Therefore, an open issue of our approach is to provide a set of predefined pattern definitions considering different architectural aspects. Other open issues are inherently in the lexical analysis of source code: the lack of control and data flow information does not allow to follow certain paths through the source code to find all relationships of code pieces that are concerned with a particular architectural aspect. But the approach provides an effective filtering mechanism that enables a reverse engineer to focus on architectural hot-spots and continue his investigations using them as starting points. These hot-spots can then be analyzed with conventional source browsing or source navigation tools to find missing relationships and occurrences of an architectural concept. Since the size of a software system is a main distractor for architecture recovery such a lexical based approach can be beneficial in focusing recovery activities on architectural aspects in the first place.

6. Conclusions

Architecture recovery includes many sources of information ranging from domain and application information to all kinds of artifacts that have been produced during software development. In our recovery activities we have realized that every software system exhibits certain characteristics that are reflected in source code structures or elements. The approach presented in this paper exploits architectural aspects such as communication (via sockets, remote procedure call etc.) or control and integrates them actively in the recovery process.

Source code patterns that represent these architectural aspects such as communication are defined and fed into a

lexical pattern matching tool called Revealer that enables fast extraction of simple and complex code patterns. In contrast to other architecture recovery approaches, Revealer is a lightweight source model extractor based on source code patterns. It provides capabilities to generate search-trees that are equivalent to grammar specifications as used by parsers. But in contrast to parsers the user can control the level of detail and can focus the extraction on architectural hot-spots in source code, that are of primary interest.

The Revealer prototype has been applied to a medium-sized intrusion detection system of 100 KLOC C and Java to identify communication, control, and synchronization parts of the system. Experiences with the tool show that it is an effective filter for masking non-architectural source pieces and a focusing means to point to the right starting points for architectural analysis.

Future work will be concerned with the integration of Revealer with existing reverse engineering tools such as Imagix-4D or Rigi to exploit their visualization capabilities. The goal is to visualize different (integrated or combined) views of particular architectural aspects and by means of the reverse engineering tool allow conventional source code browsing and navigation once the right architectural hot-spots have been identified by Revealer.

References

[1] A. V. Aho, B. W. Kernighan, and P. Weinberger. Awk - a pattern scanning and processing language. *Software Practice and Experience*, 9(4):267–280, 1979.

[2] M. N. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Proc. of the 5th Working Conference on Reverse Engineering*, pages 30–39, Honolulu, USA, October 1998. IEEE Computer Society Press.

[3] M. N. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proc. of the 18th International Conference on Software Engineering*, pages 16–27, Berlin, Germany, March 1996. IEEE Computer Society Press.

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

[5] E. J. Chicofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[6] A. Cox and C. Clarke. A comparative evaluation of techniques for syntactic level source code analysis. In *Proc. of the 7th Asia-Pacific Software Engineering Conference*, pages 282–289, Singapore, December 2000. IEEE Computer Society Press.

[7] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[8] 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.

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

[10] W. G. Griswold, M. N. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proc. of the 4th International Workshop on Program Comprehension*, pages 144–153, Berlin, Germany, March 1996. IEEE Computer Society Press.

[11] C. Hofmeiser, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, Reading, Mass. and London, 2000.

[12] R. C. Holt, A. Winter, and A. Schürr. Gxl: Toward a standard exchange format. In *Proc. of the 7th Working Conference on Reverse Engineering*, pages 162–171, Brisbane, Australia, November 2000. IEEE Computer Society Press.

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

[14] 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.

[15] M. E. Lesk. Lex - a lexical analyzer generator. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, New Jersey, October 1975.

[16] L. Moonen. Generating robust parsers using island grammars. In *Proc. of the 8th Working Conference on Reverse Engineering*, pages 13–22, Stuttgart, Germany, October 2001. IEEE Computer Society Press.

[17] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.

[18] M. Pinzger and H. Gall. Pattern-supported architecture recovery. In *Proc. of the 10th International Workshop on Program Comprehension*, pages 53–61, Paris, France, June 2002. IEEE Computer Society Press.

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

[20] D. Soni, R. L. Nord, and C. Hofmeister. Software architecture in industrial applications. In *Proc. of the 17th International Conference on Software Engineering*, pages 196–207, Seattle, Washington, April 1995. IEEE Computer Society Press.

[21] A. Tonella, R. Fiutem, G. Antoniol, and E. Merlo. Augmenting pattern-based architectural recovery with flow analysis: Mosaic – a case study. In *Proc. of the 3rd Working Conference on Reverse Engineering*, pages 198–207, Monterey, USA, November 1996. IEEE Computer Society Press.

[22] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl, 2nd Edition*. O'Reilly & Associates, Inc., 1996.

[23] 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.

[24] Extensible markup language (xml) 1.0 (second edition). W3C Recommendation, http://www.w3.org/TR/2000/REC-xml-20001006, October 2000.

Appendix

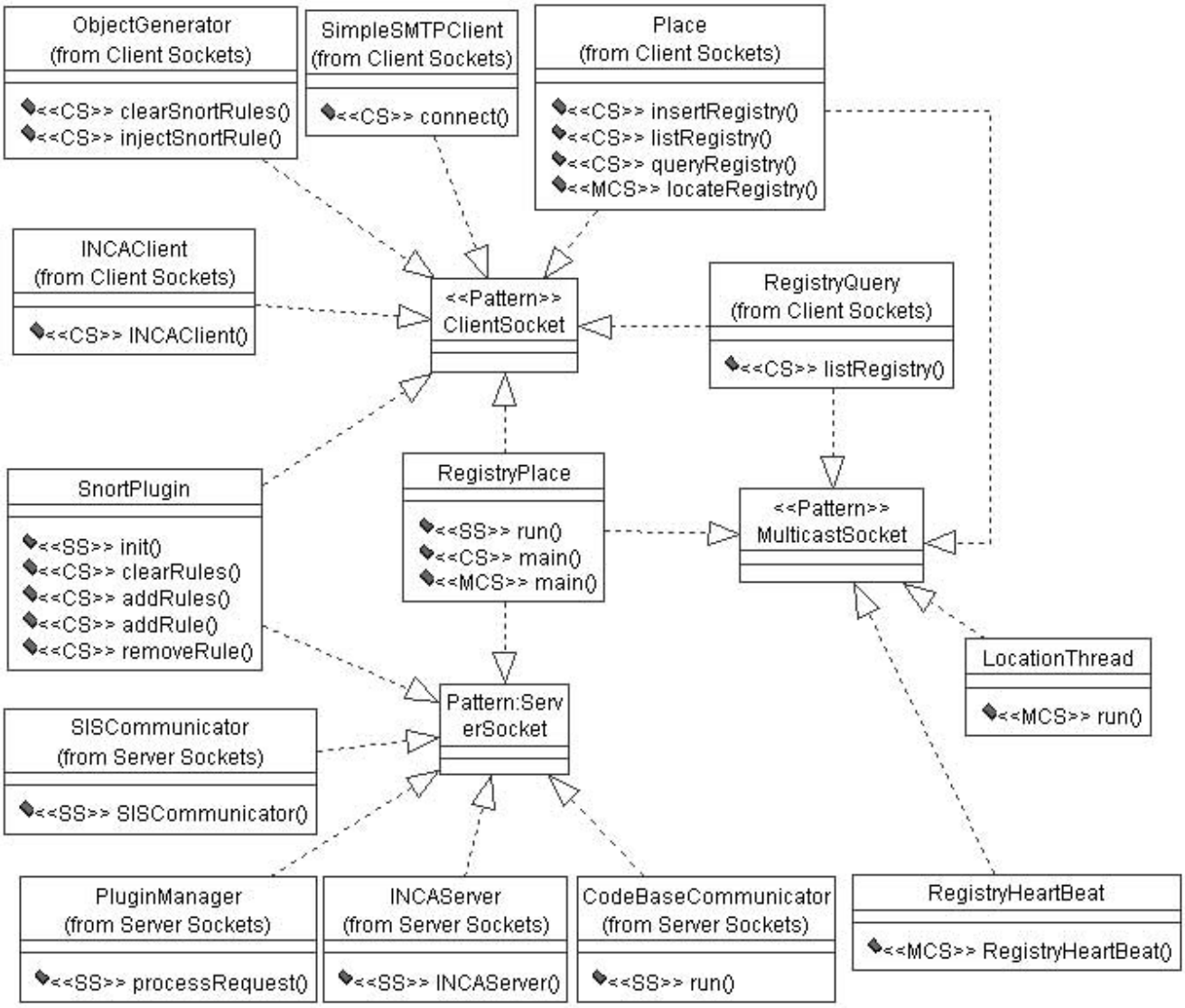

Figure 8. Extracted architectural view of SPARTA concerning socket communication