Extracting Structured Data from Natural Language Documents with Island Parsing

Alberto Bacchelli
Faculty of Informatics
Univ. of Lugano, Switzerland

Anthony Cleve
Faculty of Informatics
University of Namur, Belgium

Michele Lanza
Faculty of Informatics
Univ. of Lugano, Switzerland

Andrea Mocci
DEI
Politecnico di Milano, Italy

***Abstract*—The design and evolution of a software system leave traces in various kinds of artifacts. In software, produced by humans for humans, many artifacts are written in natural language by people involved in the project. Such entities contain structured information which constitute a valuable source of knowledge for analyzing and comprehending a system's design and evolution. However, the ambiguous and informal nature of narrative is a serious challenge in gathering such information, which is scattered throughout natural language text.**

We present an approach–based on island parsing–to recognize and enable the parsing of structured information that occur in natural language artifacts. We evaluate our approach by applying it to mailing lists pertaining to three software systems. We show that this approach allows us to extract structured data from emails with high precision and recall.

I. Introduction

Every system has a history: A history of human decisions during the design process, of changes during the development, and of successes and failures during the whole evolution.

Research in mining software repositories explores how a system's history can help to (1) support program comprehension, (2) predict future evolutions, and (3) plan various aspects of software projects. Most researchers investigated *structured data* repositories (*e.g.,* source code, versioning systems, issue databases) to answer questions about a system's history, devising approaches to find which code entities change the most, which ones present design flaws, and where defective components might be in the future.

The history of a software system, however, is not to be found only in structured data repositories. Other repositories contain the output of a process involving people, who need to share their *knowledge* about the system in order to collaborate for designing and evolving it. When knowledge is not spread through face-to-face meetings, it is put down in textual artifacts that are mostly written in informal natural language (NL) text and have different contents, writing styles, and target audiences. The majority of these documents encloses information that is structured. For example, the documents produced since the first phases of the implementation of a system (*e.g.,* development emails, IRC messages) often report and discuss particular source code fragments; issue reports often contain stack traces, or parts of defective classes and methods; and development emails are frequently used to exchange code patches and discuss implementations.

Being able to extract and parse the structured data enclosed in NL artifacts is the first step for recovering and exploring new facets of a system. For example, the fragments included in emails can be compared with the history of the implementation, to see how and when they start to diverge.

We propose an approach to *extract* the structured information to be found in documents written in NL that concern software development, to allow subsequent data parsing and modeling. The main contributions of this paper are: (1) A novel, robust and flexible technique that uses *island parsing* [3] for mining and extracting structured information of Java systems from textual artifacts (*e.g.,* stack traces, source code, file names); (2) An evaluation of the proposed technique for the extraction of the structured information, based on a significant sample of real-world complex textual documents (*i.e.,* development emails).

II. Source Fragment Extraction

Our purpose is to extract fragments of code and structured data within NL artifacts. Since these fragments are scattered within other information expressed in natural language, we need a robust parsing technique to deal with such information.

Our extraction approach relies on *island grammars*: "Detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water)" [3]. In our case, the structured fragments are the islands, everything else is water.

Our implementation is based on the ASF+SDF Meta-Environment [7]. We use it to define the context-free grammar necessary for our approach, by means of the syntax definition formalism (SDF) [2]. With SDF we can define context-free grammars in a modular way, thus facilitating the derivation of an island grammar from any existing programming language grammar; moreover, we can use a *Scanner-less Generalized LR parser* (SGLR), which does not impose any restrictions on the grammar. This property is essential when writing island grammars, which must deal with ambiguities. For *ambiguity management*, we make use of the disambiguation constructs, such as priorities, restrictions, or preference attributes to favor one particular production when several alternatives exist. Finally, we use *traversal functions* [6] that enable the analysis and rewriting of complex parse trees by focusing only on particular nodes of interest (*e.g.,* code fragments).

Our extraction approach takes text files (*e.g.,* Figure 1) and returns the structured fragments contained (blue and bold parts in Figure 1).

```
1  Because of problems with "argoHome" location, I imported
2  java.net.URLDecoder and added the line
3  URLDecoder.decode(argoHome); just below this one :
4  public void loadModulesFromDir(String dir) in ModuleLoader.java.

5  Another problem in ModuleLoader :
6  since the cookbook explains how to make a PluggableDiagram (that's
7  exactly what I am doing, so I extend this class), I have not found
8  where the JMenuItem returned by method getDiagramMenuItem() in
9  PluggableDiagram is attached in Argo menus. It seems this is not yet
10 implemented, even though PluggableDiagrams implements Diagram!

11 So I have added those lines :
12 void append(PluggableDiagram aModule) {
13   ProjectBrowser.TheInstance
14       .appendPluggableDiagram((PluggableDiagram)aModule); }

15 Of course, such modifications must be reflected in ProjectBrowser.
```

Figure 1. Example document enclosing structured information

For each fragment, our approach keeps track of the exact location (*i.e.,* the area) within the container file. An extracted structured fragment consists of a list with three elements (*e.g.,* Figure 2, examples *a* and *b*): (1) the corresponding nonterminal to which the fragment has been reduced, *i.e.,* the fragment type (lines a.1 and b.1, Figure 2), (2) the file name and the fragment's coordinates (lines a.2 and b.2), and (3) the fragment content (line a.3 and lines from b.3 to b.5).

```
a.1  ( CLASS_NAME :
a.2    area-in-file("exampleDocument", area(2, 0, 2, 19, 57, 19)) :
a.3    java.net.URLDecoder )

b.1  ( METHOD_DECLARATION :
b.2    area-in-file("exampleDocument", area(12, 0, 14, 62, 608, 131)) :
b.3    void append(PluggableDiagram aModule) {
b.4      ProjectBrowser.TheInstance
b.5        .appendPluggableDiagram((PluggableDiagram)aModule); } )
```

Figure 2. Examples of extracted structured information

A. Island Definition

Common programming language grammars describe different constructs at different levels of abstraction (from identifiers and keywords to compilation units). We could define an island as a piece of code that can be reduced to any possible nonterminal in the grammar. For example, an interesting fragment could be a piece of code that can be parsed and reduced to a nonterminal *MethodDeclaration* (*e.g.,* lines 12 to 14, Figure 1, are reduced to the example *b* in Figure 2). However, by considering every possible nonterminal as an island, we might introduce *ambiguities*, which have to be resolved and could affect the performance and the effectiveness of the fragment extractor.

For example, we consider the Java syntax productions:

$$Modifier^{*}\ MethodRes\ MethodDeclarator\ Throws? \rightarrow MethodHeader$$
$$Type \rightarrow MethodRes$$

The last production declares that any $Type$ is also a valid $MethodRes$, which is, in this context, a valid return type for a method. If the aim of the approach is to extract both $Type$ and $MethodRes$ nonterminals emerging from water, a parser for the island grammar should be able to resolve the ambiguity between these two nonterminals, *e.g.,* we should define the preferred among every possible equivalent alternatives. This ambiguity management introduces performance degradation and should be avoided, except for a reduced number of chosen nonterminals. We limited ambiguities from productions to a few inevitable cases. The most important case is the extraction of some classes of valid identifiers, such as class and package names, in Section II-B, we show how we resolve these ambiguities by using naming conventions.

The choice of the productions is related to the kind of abstraction that we want to derive from the fragmented information in the artifact. Since we aim to extract useful information about the target software system, many possible fragments corresponding to some rules are irrelevant. For example, isolated expressions or generic statements are not directly carrying relevant information in terms of methods or classes of the system. Instead, they carry structural information if, and only if, they contain specific sub-operands (as in the case of expressions) or are specific statements. For example, in the valid Java expression "*getInteger*()/*n*", the most interesting information, for our purpose, is the method invocation. Thus, instead of choosing the nonterminal $Expression$ as a valid source fragment, we choose only those subexpressions that carry information about the system structure, such as method and constructor invocations.

The case of class bodies is different: by choosing to parse complete class declarations, we include more information.

1) Incomplete Productions: Incomplete productions are a source of differentiation from a traditional programming language grammar. As an example, we consider line 4 in Figure 1: It explicitly references a method signature, but the body is missing. The standard Java grammar includes only method definitions that are followed by either a semicolon or the whole method body. By considering only the fragments that can be reduced to a nonterminal in the standard grammar, we might lose relevant information, such as this "incomplete" method declaration. For this reason, we also extract incomplete information corresponding to a subset of a production that does not reduce to any nonterminal in the standard programming language grammar.

Considering every possible incomplete production results in another source of ambiguity which affects the performance of the fragment extractor. Incomplete productions must be selected according to the kind of data to be extracted from the artifact. Since they add relevant structural information, our island grammars include incomplete productions of the nonterminals representing declarations of methods, constructors, and classes. Such incomplete productions do not require a final semicolon or a block with the body of the construct.

Our approach for incomplete productions also extracts the entity declaration even in the case of a incomplete body or that contains water. For example, for a class declaration with a partial body, our method extracts a single fact for the declaration, as an incomplete class declaration, and parses the partial body as it was a sentence of the island grammar.

Another kind of incomplete fragments, which contain interesting information, are *class relationship* fragments. These express inheritance and implementation relations between classes and interfaces. For example, in line 10 of Figure 1, we find two potential class names separated by the keyword *implements*. From this, we can derive that "Diagram" is an interface, and there is an implementation relation between the two entities.

2) Additional Structured Information: Programming platforms like JAVA are composed not only of the language definition, but also of additional components, such as the virtual machine, for which new entities are defined in terms of a precise syntax. This kind of entities range from stack traces to simple file name requirements. These entities, usually language-specific, carry interesting information about the system. We chose to enrich the island grammar to support the extraction of JAVA stack traces and file names that follow the language naming conventions. For example, we structure a JAVA class name reference as in the following production:

$$(LexPackagePath\ LexPathSep)?\ LexJavaClassFileName\ LexJavaExtension \rightarrow JavaFileName$$

It defines a filename where the extension can be *.java* or *.class* and the path is optional (*e.g.,* in Figure 1, the last part of line 4 is recognized and reduced to this production).

Table I
STRUCTURED FRAGMENTS IN THE ISLAND GRAMMAR

Nonterminal	**Description**
CompilationUnit	Class decl. with package imports
ClassDeclaration	(In)complete class decl.
MethodDeclaration	(In)complete method decl.
ConstructorDeclaration	(In)complete constructor decl.
FieldDeclaration	Class field declaration
MethodInvocation	Method invocation
ConstructorInvocation	Constructor invocation
JavaClassName	Optional JAVA class names
JavaFileName	JAVA file names
JavaStackTraceLine	Stack Trace lines
IfThenStatement IfThenElseStatement	Conditional blocks
TryStatement	Try/Catch blocks
WhileStatement ForStatement DoStatement	Loops
ClassRelationshipFragment	Implements/Extends relations
Block	Single blocks

Table I summarizes the nonterminals of the JAVA programming language that we chose to extract as fragments.

B. Ambiguity Resolution

Island grammars are inherently ambiguous: Water is defined as *anything* that is not an interesting fragment, *i.e.,* an island. Language definition systems supporting ambiguous grammars, like SDF, provide constructs to resolve ambiguities. To choose between alternative derivations, we use two disambiguation keywords offered by SDF:

- *avoid*: the parser removes alternative derivations that have avoid at the top node, only if there are no other alternative derivations with avoid at the top node.
- *prefer*: the parser removes all other derivations that do not have prefer at the top node.

By using the *avoid* keyword for any reduction to water, we favor any other production against it, *i.e.,* we prefer islands.

Consider the complete method declaration in Figure 1 (lines 12-14). As shown in Table I, at the same level of *MethodDeclaration*, we have *Block* and *IncompleteDeclaration*. The construct is ambiguous: It can be parsed as a *MethodDeclaration* or as a sequence of *IncompleteDeclaration* and *Block*. To disambiguate, we favor what reduces to a *single* nonterminal, as it keeps bindings among parts.

We also support more complicated cases. Consider the fragment *"by method getDiagramMenuItem()"* (Figure 1, line 8). The previously described island grammar would select *method getDiagramMenuItem()* as a valid *IncompleteMethodDeclaration* fragment: *method* is a valid identifier, and thus a lexically valid return type for the JAVA grammar. However, JAVA naming conventions prescribe that class names must be capitalized, thus, *method* violates them. We exploit this to exclude those reductions to incomplete method declarations where the supposed return type is likely to be invalid.

For every extracted fragment, we define an ASF function, called *isValidSource*, which takes a source fragment as input and returns true if the fragment is valid. The base case for that function is true for any fragment. For incomplete method declarations, *isValidSource* returns true if the return type is *void* or an identifier that respects the naming conventions.

We implement similar definitions for potential constructor declarations, which must respect valid naming conventions.

Going back to the previous example, the *method getDiagramMenuItem()* is not a valid method declaration, but it can be restructured as a method invocation fragment *getDiagramMenuItem()*. We define an ASF transformation rule to translate what was parsed as a method declaration to a method invocation. The rule takes a fragment parsed as an incomplete method declaration violating the naming conventions, extracts the identifier corresponding to the method name, and produces a source fragment of type *MethodInvocation*. The location of the transformed fragment is the same as *MethodDeclaration*, which is the nonterminal corresponding to the method name and its formal parameter declaration, that in this case is empty. This transformation has to be performed only with methods without parameters.

III. Evaluation

For the evaluation, we consider open source systems (OSS) developed in Java, namely Freenet, Mina, and ArgoUML. Since we are interested in NL artifacts, to improve the generalizability of our validation, we chose development emails as documents, since they are a veritable acid test to validate our approach. In fact, email data is noisy, as it contains extra line breaks, extra spaces, and special character tokens; it can contain spaces and periods mistakenly removed, words misspelled, badly cased or non-cased [4].

A. Text normalization of emails

Due to the noisy nature of email content, we devised a pre-processing text normalization phase in which: (1) We remove the email metadata and the occurrences of the characters (*i.e.,* >) used to mark different quotation levels; (2) we normalize patches by removing the lines marked as deleted (to avoid recovering what is explicitly no longer valid) and the + signs at the beginning of the lines; (3) we normalize stack traces removing incorrect line breaks by using a regular expression that exploits the structure of stack traces.

B. Validation of the information extraction

Table II details the systems used in our evaluation. The second column reports the mailing list size, without automatically generated emails (*e.g.,* by issue tracking systems).

Table II
Systems used to validate the information extraction

System	emails			Results
	List	with code	Sample	Precision/Recall
ArgoUML	24,876	12%	50	99% / 95%
Freenet	22,095	9%	39	99% / 97%
Mina	12,869	29%	99	99% / 94%

We randomly chose a statistically significant sample of emails *with code* to inspect. Based on our previous work on classifying emails and lines containing source code [1], we know the proportion of messages with code in the chosen mailing lists (reported in the third column of Table II). This allows us to use this information for calculating the size of significant samples [5]. The fourth column of Table II reports the sample sizes. With these sample sizes, we are 95% confident that the emails represent the population with a 9% error. To evaluate our approach, we use the common information retrieval measures of *precision* and *recall*.

Before manually creating a benchmark with all the emails in our samples, we processed emails with our normalization phase (Section III-A). Then, we manually inspected each email and labeled all structured fragments with the correct grammar production. Conducting the normalization phase before the annotation had the beneficial side-effect that it allowed us to verify the normalization process.

The right side of Table II shows the results obtained with our approach, from the 183 emails containing source code that we labeled. We achieve near-perfect precision on the complete dataset for all three systems, which means that almost no NL words were classified as parts of structured fragment. The recall values are also considerably high, showing that the method recognized almost the totality of the fragments. We found no error in the grammar productions reported in the recognized fragments.

By manually inspecting the entire output of our technique, we gained a qualitative knowledge of the cause of the few errors generated. Most false negatives (which affect the recall) were caused by the noisy nature of emails. In particular, a few code fragments were truncated in the middle of an identifier, thus hindering a correct complete identification (although the surroundings were correctly recognized).

IV. Conclusions

We presented a technique, based on island parsing, to extract structured data from NL artifacts. We evaluated it on a statistically significant set of emails, and reached very high accuracy values, despite the noisy nature of email content.

The approach does not come without its limitations, especially with respect to performance, which we plan to address as a future work. Also, while the approach is tailored to fragments of Java systems, it can be applicable to other languages with limited effort.

Acknowledgements. Bacchelli is supported by the Swiss Science foundation (Project "SOSYA", No. 132175). Cleve carried out this work during the tenure of an ERCIM post-doctoral fellowship.

References

[1] A. Bacchelli, M. D'Ambros, and M. Lanza. Extracting source code from e-mails. In *Proc. of ICPC 2010 (18th Int'l Conf. on Program Comprehension)*, pages 24–33, 2010.

[2] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF—reference manual—. *SIGPLAN Notices*, 24(11):43–75, 1989.

[3] L. Moonen. Generating robust parsers using island grammars. In *Proc. of WCRE 2001 (8th Working Conf. on Reverse Engineering)*, pages 13–22. IEEE CS, 2001.

[4] J. Tang, H. Li, Y. Cao, and Z. Tang. Email data cleaning. In *Proc. of SIGKDD 2005 (11th Int'l Conf. on Knowledge Discovery in Data mining*, pages 489–498. ACM, 2005.

[5] M. Triola. *Elementary Statistics*. Addison-Wesley, 2006.

[6] M. van Den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM TOSEM*, 12(2):152–190, 2003.

[7] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proc. of CC 01 (Int'l Conf. on Compiler Construction)*, pages 365–370. Springer, 2001.