Generating Robust Parsers using Island Grammars

Leon Moonen

CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
http://www.cwi.nl/~leon/
leon@cwi.nl

Abstract

Source model extraction—the automated extraction of information from system artifacts—is a common phase in reverse engineering tools. One of the major challenges of this phase is creating extractors that can deal with irregularities in the artifacts that are typical for the reverse engineering domain (for example, syntactic errors, incomplete source code, language dialects and embedded languages).

This paper proposes a solution in the form of island grammars, *a special kind of grammars that combine the detailed specification possibilities of grammars with the liberal behavior of lexical approaches. We show how island grammars can be used to generate robust parsers that combine the accuracy of syntactical analysis with the speed, flexibility and tolerance usually only found in lexical analysis. We conclude with a discussion of the development of* MANGROVE, *a generator for source model extractors based on island grammars and describe its application to a number of case studies.*

Keywords and phrases: *Island grammars, parser generation, source model extraction, partial parsing, fuzzy parsing, reverse engineering, program analysis.*

1. Introduction

Software engineers spend a large amount of their time on understanding the system that is being maintained (estimates of up to 50% are not uncommon). Consequently, much research is being invested in the development of tools that assist with such program understanding and program maintenance activities. The majority of these tools consist of three phases: (1) extraction of information (often referred to as *source models*) from the system's artifacts, (2) manipulation, querying and abstraction of source models, and (3) presentation of the results. This paper focuses on the first phase: *extracting source models from system artifacts*.

One of the challenges reverse engineering tools have to cope with is parsing the artifacts during the extraction phase. These artifacts typically contain *irregularities* that make it hard (or even impossible) to parse the code using common parser based approaches. Our goal is to obtain *robust* parsers that can handle artifacts with such irregularities. Examples of the kind of irregularities we want to deal with include:

Syntax errors: In a program maintenance environment, we want to be able to deal with systems containing syntax errors (e.g., browse or query code to fix those errors). Most parser based techniques will fail when encountering syntactic errors.

Completeness: The source code of a system may be incomplete. A typical situation is that some of the header files (or copybooks) of a system are lost or mutilated over the years, making a full reconstruction impossible.

Dialects: A legacy language like COBOL (but also a language like C) has a large number of, slightly different, vendor-specific dialects. Ideally, we can support them all. However, a parser for one dialect may not accept code written in another.

Embedded languages: Several programming languages have been upgraded with embedded languages for database access, transaction handling, screen definition, etc. COBOL examples include SQL, CICS, and IDMS. Whether we choose to analyze or to ignore such extensions, the extraction should not be hampered by them. However, a standard parser will.

Grammar availability: When supporting legacy systems, we will come across languages for which there is no grammar available. These can be proprietary languages, for which a grammar was never disclosed, or languages for which there never was a grammar since the parser (or processor) was hand-written. Reviving such grammars from scratch is expensive, and may not pay back at all.

Customer-specific idioms. Systems can use specific idioms (e.g., assigning values to "special" variables) in combination with libraries to interface with other systems, or to bypass limitations in a compiler or runtime system. Standard parsers will not recognize such customer-specific idioms and are generally not flexible enough to be made aware of them. An example regarding COBOL CALL analysis is shown in Section 2.1.

Preprocessing: Conceptual problems can arise with analysis of code that uses a preprocessor: Parsers usually read preprocessed code so the resulting models are based on preprocessed code. However, a maintainer's mental model is based on unpreprocessed code. It can be very hard to map these models

1095-1350/01 /$10.00 © 2001 IEEE

onto another, especially when conditional compilation is used.

People have tried to bypass these problems by reusing an existing parser via a common exchange format (e.g., GXL [18]), or via interface generation (for example, GENII [14]). Although these are good solutions from an engineering perspective (you may not have to write a parser yourself) they do not *solve* the problems described above.

Others have proposed to use *lexical analysis* techniques to remedy these problems [28, 11]. Lexical analysis provides a flexible and robust solution that can handle incomplete and syntactically incorrect code at the cost of losing some accuracy and completeness.

An additional advantage of lexical analysis is that it often takes less time to develop a solution based on lexical analysis than on syntactical analysis. It is tedious and expensive to write a parser for a language or to write a grammar that can be used to generate such a parser. For example, van den Brand *et al.* report a period of four months for the development of a fairly complete COBOL grammar [8].

This paper proposes another solution to remedy these problems: we describe the use of *island grammars* to generate robust parsers that are used to build source model extractors. Island grammars are grammars that contain detailed productions (rules) describing the language constructs of interest, and generic productions that capture the remainder. Island grammars have been briefly sketched before in [12, 13]. In this paper, we present a more detailed account.

By generating parsers from island grammars, we combine the accuracy of syntactical analysis with the speed, flexibility and robustness of lexical analysis. The remainder of this paper presents island grammars and their use in MANGROVE, a generator for source model extractors based on island grammars. We propose a reusable framework for defining island grammars and describe how the mapping from parse results to source models can be specified using patterns in a term rewriting language and in JAVA. We conclude with the application of MANGROVE in a number of case studies and a discussion of related work.

2. Island Grammars

Parsers for reverse engineering tools have a number of requirements: The parser should recognize certain *constructs of interest* in a given language. Additionally, the parser should be *robust*: it should not be obstructed by irregularities in the input. In this paper, we study how such parsers can be generated from (context-free) grammar definitions.

Recall from compiler class that, given a language L_0, we can give a description of L_0 by defining a context-free grammar G such that the language $L(G)$ generated by G satisfies $L(G) = L_0$.[1] In order to satisfy the requirements stated above, we need to describe L_0 using a grammar that on the one hand generates more sentences than available in the actual language L_0 (namely also sentences with irregularities) but on the other hand should give an exact specification of the interesting parts of that language. This is what an island grammar amounts to:

Definition 2.1 *An* island grammar *is a grammar that consists of detailed productions describing certain constructs of interest (the* islands*) and liberal productions that catch the remainder (the* water*).*

or expressed in terms of language properties:

Definition 2.2 *Given a language L_0, a context free grammar $G = (V, \Sigma, P, S)$ such that $L(G) = L_0$ and a set of constructs of interest $I \subset \Sigma^*$ such that $\forall i \in I : \exists s_1, s_2 \in \Sigma^* : s_1\, i\, s_2 \in L(G)$. An island grammar $G_I = (V_I, \Sigma_I, P_I, S_I)$ for L_0 has the following properties:*

1. $L(G) \subset L(G_I)$ *G_I generates an extension of $L(G)$.*
2. $\forall i \in I : \exists v \in V_I : v \overset{*}{\rightarrow} i$
   $\exists s_3, s_4 \in \Sigma^* : s_3\, i\, s_4 \notin L(G) \wedge s_3\, i\, s_4 \in L(G_I)$
   G_I can recognize constructs of interest from I in at least one sentence that is not recognized by G.
3. $K(G) > K(G_I)$ *G has higher complexity than G_I.*[2]

Note that island grammars do not require the use of a particular grammar specification formalism or parsing technique. However, the limitations of the chosen formalism and technique may influence the island grammar. In this paper, we express island grammars in SDF, a syntax definition formalism that is supported by *generalized LR* parsing [17, 38]. We benefit from the expressive power of this combo which makes development of island grammars easier. Other formalisms and parsing techniques can, and have been used. For example, JAVACC (the Java parser generator by MetaMata/Sun Microsystems) has been used for an island grammar developed together with our industrial partner, the Software Improvement Group, as part of their documentation generator DOCGEN [12, 13]. The requirements originating from the LL parsing technique used in JAVACC made development and extension of this grammar unwieldy. The tooling described in the next section enables us to reimplement this grammar based on SDF and generalized LR parsing.

2.1. Island Grammar Example

Figures 1 and 2 show an example island grammar that describes COBOL CALL statements. The specification uses the

[1] In short: if $G = (V, \Sigma, P, S)$ is a context-free grammar with sets of *nonterminals* V, *terminals* Σ and *productions* $P \subseteq (V \cup \Sigma)^* \times V$, a *start symbol* $S \in V$, and $V \cap \Sigma = \emptyset$, then a string $s \in \Sigma^*$ is a *sentence* of G, iff $S \overset{*}{\rightarrow} s$ (s can be derived from S by repeatedly applying productions from P). The *language* generated by G contains all sentences $L(G) = \{s \mid s \in \Sigma^* \wedge S \overset{*}{\rightarrow} s\}$. We refer to [35, pp. 43–64] for more information.

[2] The complexity of a context free language $K(G)$ can be computed by analyzing the productions of G. See [16] for a detailed discussion.

```
module Layout                                          (1)
lexical syntax                                         (2)
  [\␣\t\n]                 → LAYOUT                    (3)
module Water                                           (4)
imports Layout                                         (5)
context free syntax                                    (6)
  Chunk*                   → Input                     (7)
  Water                    → Chunk                     (8)
lexical syntax                                         (9)
  ~[\␣\t\n]+               → Water {avoid}            (10)
```

Figure 1. Base for island grammars.

modular syntax definition formalism SDF. Note that productions in SDF are reversed with respect to BNF: on the right-hand side of the arrow is the non-terminal that can be produced by the symbols on the left-hand side. Section 3.1 gives a short introduction to SDF.

The grammar contains three modules: The module Layout specifies the lexical non-terminal symbol LAYOUT containing whitespace characters. This symbol has special meaning in our parsers since it can be recognized between any two symbols in a context-free production.

The module Water uses the definitions from module Layout (line 5) and adds two context-free non-terminals: the symbol Input that can be produced from a list of zero or more Chunks (line 7) and the symbol Chunk that can be produced from Water (line 8). Later, we will add more productions for Chunk, thus providing alternatives that can be recognized instead of Water. The lexical non-terminal Water consists of a list of one or more characters that are not whitespace (line 10). The attribute "{avoid}" prevents the parser from using this production if others are applicable. This allows us to specify *default behavior* that can be overridden by other productions (without generating ambiguities).

The grammar specified by module Water is extremely robust: it describes almost all programming languages. It is, however, not very useful by itself since the terminal symbols in a parsed sentence are indistinguishable. We can turn this into a useful grammar by adding *islands* that specify constructs of interest: The module Call adds such an island by specifying that a Chunk can also be produced by the literal CALL followed by an identifier (line 4). Identifiers are characters followed by zero or more characters or digits (line 7).

This very simple grammar allows us to generate a parser that searches for program calls in COBOL code. Although this may not be a spectacular example (something similar could be done, for example, using a tool like `grep`), we will show below how easy it is to extend this grammar to do a much more complicated analysis. Furthermore, the modularity of SDF allows us to reuse the base grammar developed here for other island grammars.

```
module Call                                            (1)
imports Water                                          (2)
context free syntax                                    (3)
  "CALL" Id                → Chunk {cons(Call)}        (4)
lexical syntax                                         (6)
  [A-Z][A-Z0-9]*           → Id                        (7)
```

Figure 2. COBOL program calls.

```
module CallHandler                                     (1)
imports Call                                           (2)
context free syntax                                    (3)
  "MOVE" Id "TO" "CALLEE" → Chunk {cons(Call)}         (4)
  "CALL" "HANDLER"        → Chunk {reject}             (5)
```

Figure 3. Dealing with a call-handler.

Remember the customer specific idioms described in Section 1? We found a good example of that situation when analyzing a COBOL system were program calls were not made using the CALL statement but by setting a global variable and then calling a generic *call-handler.* This call-handler enabled the run-time system to dynamically load and execute the desired program (instead of static linking supported by the compiler). A standard call-graph extractor will not be able to generate useful graphs for such a system.

We can add support for that situation using the grammar in Figure 3. Suppose the name of the call-handler is HANDLER and the name of the global variable is CALLEE. We specify an assignment to CALLEE as if it is a program call (line 4). Furthermore, we prevent the parser from recognizing calls to HANDLER using the "{reject}" attribute (line 5).

The "{cons(Call)}" attributes in Figures 2 and 3 are used to explicitly specify the constructor function that has to be used to create an abstract syntax tree. Using this attribute we can map different concrete syntax productions to the same abstract syntax. This will make processing easier.

Note the source for potential errors here: (1) when there are two subsequent assignments to CALLEE before the call-handler is called, both will be recognized as calls; (2) when the value in CALLEE is computed instead of assigned, it will not be recognized. These problems can be remediated in a back-end that does a more detailed (data flow) analysis. In practice, however, we found that such call-handlers were used in a disciplined manner following strict coding conventions, so these situations did not occur.

2.2. Island Grammar Applications

The employment of island grammars is especially suitable for reverse engineering (as opposed to, for example, compiler construction) since it takes maximum advantage of the fact that such applications generally do not need the complete parse tree. Particularly analyzers that try to arrive at higher levels of abstraction (for example, architecture extraction) can profit from this early elimination of detail in the parsing phase.

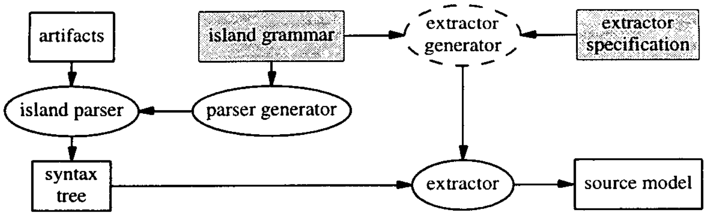

Figure 4. MANGROVE **architecture.**

By varying the amount and details in productions for the construct of interest, we can trade off accuracy, completeness and development speed. For example, it is possible to approach island grammars from a completely different side by starting with a complete grammar for a given language and extending that grammar with a number of liberal (*water*) productions. We will call such a grammar a *lake grammar.* This approach is typically useful to allow for arbitrary embedded code in the programs that can be processed by given tool. Furthermore, we can mix productions for water and islands to allow variations such as: *islands with lakes* to specify "nested" constructs such as conditional or iteration statements, and *lakes with islands* to combine extraction for a language with extraction for an embedded extension.

In our opinion, the main application area for island grammars is robust parser generation for source model extraction and simple analysis. Island grammars can be used for both local and non-local analysis. Obviously, grammars that only allow local analysis (for example, the CALL statements of Figure 2) will be simpler than those that allow non-local analysis. Additional work has to be done in the back end of a non-local analyzer to find and combine islands that "belong together".

The main advantage that island grammars have over lexical approaches is that it is much easier to use structure while specifying patterns (which requires state manipulation in a lexical approach). Moreover, solutions can easily be combined and are completely declarative making them easier to understand.

In theory, island grammars can be used for program transformations. Since the use is evidently restricted to the parts that are contained by the islands, applications are probably limited to local transformations. Examples one can think of include simple structure modifications, normalization of conditions, enforcement of some coding standards. In general, however, we believe that program transformations require more in depth knowledge of the source language than what is usually expressed in an island grammar.

2.3. Processing

There are a number of ways to process the parse trees obtained after parsing an input sentence. Initial observations indicate that in most island grammars, the Water symbols always occur in a sequence of symbols. Consequently, removing those subtrees from a parse tree does not invalidate the tree. Based on this observation, we have created a simple filter that removes all subtrees that have been parsed as *Water* from a parse tree. After applying this filter, processing the resulting term becomes both easier and faster (less input to consider). Simple analysis of the term can even be done using lexical techniques. Note that it is always possible to create grammars for which Water does not occur in a list context. Use of the filter will invalidate parse tree with respect to such grammars. This may or may not be a real problem depending on the processing that remains to be done on the tree.

Another way is to process the parse trees using hand-written C code. Currently, such processing is cumbersome but this might improve when supportive tooling becomes available that generates access functions on an AST level

In order to be able to create more involved source model extractors that are not hand-written in C, we have created MANGROVE, a generator for source model extractors. MANGROVE is described in the next section.

3. MANGROVE

MANGROVE is a generator for source model extractors based on island grammars. The design requirements were similar to those described by Murphy and Notkin for their lexical source model extractor [28]. The approach has to be:

- *Lightweight:* specification of new extractor should be small and relatively easy to write.
- *Flexible:* few constraints on structure of the artifact that is analyzed (possible to create analyzers for both source and structured data).
- *Tolerant:* few constrains on the condition of the artifact that is analyzed (possible to analyze code that cannot compile).

An overview of the MANGROVE architecture is given in Figure 4. Tools are drawn as ellipses, artifacts as boxes. The generation of a extractor is based on two types of input (the grey boxes in Figure 4): The first defines an island grammar describing the syntax of constructs that need to be recognized. It is used to generate an island parser; The second specifies the mapping of those constructs to the desired source model. It is used with the grammar to generate an extractor that reads the output of the island parser and converts it to the source model.

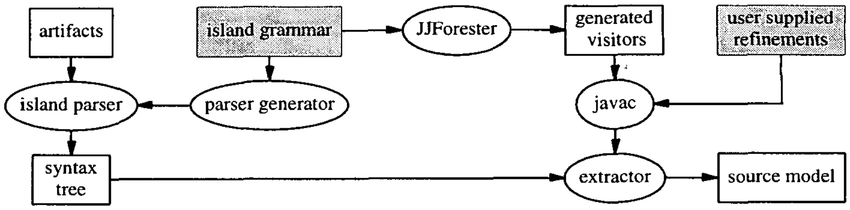

Figure 5. MANGROVE **instantiation that allows processing in** JAVA.

In contrast to most lexical approaches, our approach separates parsing and analysis instead of attaching semantic actions to the constructs to be recognized. This has the advantage that the resulting analyzers are easier to adapt and that it is easier to combine two existing analyzers into a new one. Most lexical analyzers are hard to adapt since the analysis logic is entangled with the constructs that have to be recognized. Combining two of these analyzers into a single new one is even more tricky.

The two inputs are generally small and easy to write down; therefore, we feel that our approach satisfies the lightweight requirement. The flexibility and robustness requirements are satisfied by using island grammars to generate the parser.

The *extractor generator* in Figure 4 is drawn with a dotted line to indicate that there are several possible instantiations. These allow the user to choose the language in which he describes the mapping of constructs on the source model. We have made two instantiations of this tool that are described below. One allows the user to write the mapping using traversals over the AST in Java, the other using concrete syntax patterns in a simple functional specification.

3.1. Syntax Definition in SDF

MANGROVE reads island grammars that are written in the syntax definition formalism SDF [17, 38]. These definitions combine the definition of lexical and context-free syntax in the same formalism. The definitions are purely declarative (as opposed to, for example, definitions in YACC that can use semantic actions to influence parsing) and describe both concrete and abstract syntax.

SDF definitions can be modular: productions for the same non-terminal can be distributed over different modules and a given module can reuse productions by *importing* the modules that define them. This allows for the definition of a base or kernel grammar that is extended by definitions in other modules. An example of this is module Water defined in Figure 1 that is extended by module Call in Figure 2.

SDF provides a number of operators to define optional symbols ($S?$), alternatives ($S_1|S_2$), iteration of symbols ($S+$ and $S*$), and more. These operators can be arbitrarily nested to describe more complex symbols. Furthermore, SDF provides a number of disambiguation constructs such as relative priorities between productions, preference attributes to indicate that a production should be preferred of avoided when alternatives exist, and associativity attributes for binary productions (for example, $S\ op\ S \rightarrow S$ {left}).

SDF is supported by a parser generator that generates *generalized LR* (GLR) parsers. Generalized parsing allows definition of the complete class of context-free grammars instead of restricting it to a non-ambiguous subclass of the context-free grammars, such as the LL(k), LR(k) or LALR(1) class restrictions common to most other parser generators [36, 30]. This allows for a more natural definition of the intended syntax because a grammar developer no longer needs to encode it in a restricted subclass. Moreover, since the full class of context-free grammars is closed under composition (the combination of two CF grammars is again a CF grammar), generalized parsing allows for better modularity and syntax reuse. For more information on SDF, we refer to [17, 38].

3.2. MANGROVE/JAVA

MANGROVE/JAVA allows the extractor builder to process the results of the island parser using the object-oriented programming language JAVA. An overview of the tool is given in Figure 5. Apart from the obvious advantage of being able to process using a mainstream object-oriented programming language, this also allows the tool builder to reuse the large amount of tools, libraries and interoperability techniques that are available for JAVA.

From an island grammar in SDF, we generate JAVA code for the construction, representation, and manipulation of syntax trees in an object-oriented style. The generated classes relate to the abstract syntax of the grammar using the following scheme: (i) for every non-terminal, an abstract class is generated and (ii) for every production, a concrete class is generated that refines the abstract class corresponding to the result of the production. Factory methods are generated to convert a parsed input string into an abstract syntax tree (object structure). Furthermore, several variants on the Visitor pattern are generated that provide tree traversals over these ASTs. We have reused JJFORESTER for the generation of this JAVA code [22].

The generated code can be extended by a tool builder to perform the actual mapping between the AST and the desired source model. This is done by refining the generated visitors and feeding them to the generated accept method of a given AST node. These accept methods perform the actual traver-

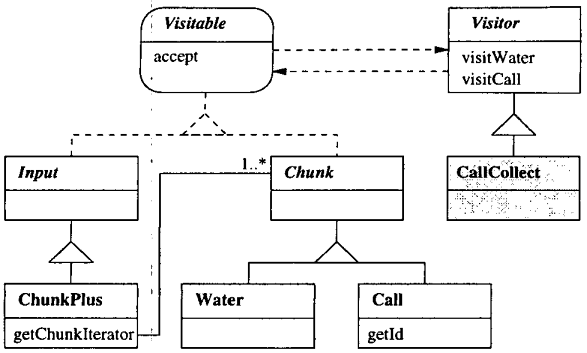

Figure 6. UML class diagram for Call collector.

sal over the AST and call visit methods defined in the visitor. This approach has the advantage that the user does not have to reconstruct the traversal behavior when refining visitors. Consequently, it is easier and less error-prone to write extensions and refinements of the generated code.

User extensions are compiled together with the generated code using a standard JAVA compiler to create an extractor (i.e., byte code that can be executed using the JAVA virtual machine). This extractor interfaces with the generated island parser using a utility that implodes the parse tree into an abstract syntax tree.

Example: Figure 6 presents an UML class diagram showing the classes that are generated for the island grammar presented in the COBOL program call example (Section 2.1). The grey class (CallCollect) was not generated but is an example of an analysis that can be added by a user. This class refines the standard visitor so that it collects the identifiers of all called programs. The JAVA code that implements this class is shown in Figure 7.

3.3. MANGROVE/ASF

MANGROVE/ASF allows the extractor writer to process parse results in a functional fashion using the term rewriting language ASF [4].

Programming in ASF is done by creating specifications that consist of a number of rewrite rules. These rules are defined using pattern matching on concrete syntax defined in an SDF grammar. The use of concrete syntax has the advantage that the extractor writer does not have to learn a new language for processing terms. The use of term rewriting allows for a natural expression of the translation of one language into another.

```
public class CallCollect extends Visitor {
   public Set set = new HashSet();
   public void visitCall(Call c) {
      set.add(c.getId());
   }
}
```

Figure 7. JAVA **visitor for collecting program calls.**

```
module CallCollect                                      (1)
imports CallHandler Set                                 (2)
context free syntax                                     (3)
  collect( Input )        → Set                         (4)
  collect( Input , Set )  → Set    {traverse}           (5)
variables                                               (8)
  "in"                    → Input                       (9)
  "set"                   → Set                         (10)
```

Figure 8. Grammar for collecting program calls.

The combination of syntax definition formalism SDF and term rewriting language ASF is supported by the ASF+SDF Meta-Environment [20, 6]. This environment generates parsers and syntax directed editors from SDF definitions and provides an interpreter and compiler for ASF specifications.

In MANGROVE/ASF, we instantiate the extractor generator using the ASF+SDF Meta-Environment. For an architectural overview, we refer to the MANGROVE overview in Figure 4.

The ASF+SDF Meta-Environment contains support for the generation of term traversal functions [7]. When a user attaches a "{traverse}" attribute to a production in SDF, additional functionality is inferred that can perform a traversal of the first argument of the production. Conceptually, adding such an attribute is shorthand for adding a set of productions and rewrite rules (which can be calculated from the grammar). The default behavior of the generated rewrite rules is to do nothing. A user can override that behavior by adding a concrete rewrite rule for a particular (sub)term.

Example: Figures 8 and 9 show an example of the use of generated traversals for the program call example described in Section 2.1. Again, we will build a tool to collect the identifiers of all called programs. The grammar (Figure 8) defines two functions: one that we will use to start the traversal (line 4) and the actual traversal function in line 5. This traversal function has two arguments, the first contains the term to traverse, the second is the accumulator in which traversal results are gathered. The ASF equations in Figure 9 define the rewrite rules. We see that rule $[c_1]$ starts the traversal using a copy of the input and an empty accumulator. The other two rules contain patterns for which we want specific behavior: Rule $[c_2]$ specifies that whenever a CALL statement is matched with arbitrary identifier, we add that identifier to the accumulated set. Call-handlers are supported using rule $[c_3]$ that collects all identifiers that are assigned to the CALLEE variable.

equations

$[c_1]$ collect(*in*) = collect(*in* , {})

$[c_2]$ collect(CALL *id* , *set*) = {*id*} ∪ *set*

$[c_3]$ collect(MOVE *id* TO CALLEE , *set*) = {*id*} ∪ *set*

Figure 9. Equations for collecting program calls.

4. Case Studies

We have done a number of case studies to validate our hypothesis that island grammars can be used to create robust parsers that allow for construction of lightweight, flexible and tolerant source model extractors.

The first case uses island grammars to build an analyzer that computes the cyclomatic complexity of COBOL programs. The second case was done in cooperation with the Software Improvement Group and involves the creation of a source model extractor for UNIFACE systems.

4.1. COBOL Cyclomatic Complexity

McCabe's *cyclomatic complexity* measure [27] is one of the better known software metrics that can be computed from source code. In this case study we build a simple analyzer that computes this complexity measure for COBOL programs using island grammars.

The cyclomatic complexity metric is based on the control graph of the program. It computes the number of linearly independent control flow graphs using the number of nodes (n) and edges (e) in a control flow graph. For a graph with n nodes and e edges, McCabe defines the cyclomatic complexity as $G(v) = e - n + 2$.

However, there is a simpler definition that does not require us to construct a control flow graph in advance. In the NIST *report on structured testing*, McCabe defines the cyclomatic complexity by counting the number of decision predicates in the code [40]. We will use this latter approach in this case. Our analyzer basically traverses a parse tree and counts occurrences of decision predicates. We show how we use MANGROVE/JAVA to build the analyzer in four steps.

First, we create an island grammar for COBOL that describes the constructs that can influence the cyclomatic complexity. In the case of COBOL, these are standard constructs like `IF-THEN`, `REPEAT-UNTIL`, and `EVALUATE-WHEN` (COBOL's case statement) but also constructs like `GO-DEPENDING` that jumps to one of a list of locations based on the value of a variable. Other constructs of interest are predicates that surround code that has to be executed in case of errors, such as `ON-ERROR` and `ON-OVERFLOW` for computational statements, and `INVALID-KEY` and `AT-END` for access to flat-file databases.

Note that we have to take special precautions to prevent occurrences of these constructs in strings or comments from being recognized as real occurrences (so called *false positives*). This can be done by adding specific productions to the island grammar that specify that strings should be recognized as water and that comments should be considered LAYOUT. An example of such productions can be found in Figure 10.

Second, a parser and JAVA classes are generated from this island grammar as described in Section 3.

Third, we refine the generated visitor so that computes the cyclomatic complexity during traversal of the parse tree. This is done by incrementing a counter every time the abstract syntax tree contains one of the complexity increasing constructs that were specified in the island grammar.

Finally, we compile the code to build an executable analyzer. The parts that we had to write to create such an analyzer are small and easy to write: construction, testing and refinement took 4–5 hours. The grammar consists of 17 productions, 10 for describing constructs of interest, 4 we reused from the base grammar of Figure 1, and 3 were added to prevent false positives. The JAVA code that refines the generated visitor contains one integer field (the complexity counter) and seven methods that each perform exactly one statement: increment the complexity.

We have applied our analyzer to a number of COBOL systems (each around 100.000 lines) that were written in different dialects and contained various extensions (SQL, CICS, IMS). These irregularities posed no problems for the analysis. Initial results show that the performance is good but should be measured in more detail. For example, the implosion prototype that converts parse trees to ASTs is slow for very large inputs. A reimplementation will solve these issues.

4.2. UNIFACE Component Coupling

In a case study performed in cooperation with the Software Improvement Group (SIG) we developed an island grammar and source model extractor to parse UNIFACE components and collect facts about the coupling between them.

UNIFACE is a 4GL application development environment that is marketed by Compuware [10]. It allows for the development of both conventional and web-based applications. The application development is model-driven and component-based. Developers create models of business processes. These models are used to generate components that inherit properties from the model. Whenever the model is changed, components are updated accordingly. To eliminate the need to build systems from scratch, developers can reuse components from other systems and standard libraries. Components contain operations that specify behavior. Components can interoperate with each other by activating operations in other components (similar to objects and methods in an object-oriented setting).

To get insight in UNIFACE systems, a SIG customer would like to get information about the components in a system and the coupling between them. To collect this information, we have build a source model extractor that analyses UNIFACE components and gathers facts about the activation of other components and of the activation parameters.

```
module StringsAsWater
lexical syntax
  [\"] ~[\"]* [\"]            → Water
```

Figure 10. Strings as water.

The extractor was generated using an island grammar that describes module activation and parameter passing in UNIFACE. This grammar extends the base grammar from Figure 1 and was developed without prior knowledge of UNIFACE (but with help of `activate` documentation). It took approximately one day to develop, test and refine the island grammar and about the same amount of time to develop the source model mapping in JAVA.

The complete island grammar contains 38 productions, including the base grammar and productions to prevent false positives. This relatively high number is influenced by the act that UNIFACE is case insensitive, thus our grammar contains a number of productions whose sole purpose is to specify case insensitive variants of keywords that have to be recognized.

The resulting source model extractor can process both UNIFACE source listings and XML dumps of modules. The extractor emits a source model that describes component coupling in textual or in GXL format [18].

5. Discussion

5.1. Expressive power

Island grammars do not depend on a particular grammar specification formalism or parsing technique. However, the expressive power of an island grammar is limited by the chosen syntax definition formalism and more important by the chosen parsing technique. In MANGROVE, we have chosen to express island grammars in SDF, a syntax definition formalism that is supported by *generalized LR* parsing techniques. Since we inherit the expressive power, we can express the complete class of context free languages using our island grammars.

The different MANGROVE instantiations allow an extractor writer to choose a processing language that fits his needs. The JAVA instantiation enables processing in a mainstream object-oriented programming language and allows reuse of the large amount of tools, libraries and interoperability techniques that are available for JAVA. The ASF instantiation allows processing using term rewriting with patterns over concrete syntax . This has the advantage that the extractor writer does not have to learn a new language and term rewriting allows for natural expression of translation between languages.

5.2. Accuracy

Island grammars do not give a restrictive description of the language that is analyzed. On the one hand, we consider this an advantage since this is, after all, the property that allows for irregularities, releases structural requirements on the artifacts and increases development speed. On the other hand, however, this lack of detail may result in erroneous results.

We distinguish two kinds of extraction errors: (i)*false positives* occur when the grammar allows constructs to be recognized in places were they should not have been recognized. (ii)*false negatives* occur when the grammar is too restrictive and does not allows constructs to be recognized in places were they should been recognized.

False positives can be solved by extending the part of the grammar that specifies Water. For example, false recognition of constructs inside of strings can be prevented by adding a production that specifies string syntax as Water. Figure 10 gives a simple example of such a specification. It specifies strings as starting with a double quote, a number of characters and ending with a double quote.

False negatives are not that straightforward to solve. One needs to reconsider the grammar and look for productions that are too restrictive. A common source of false negatives are "nested" constructs, for example statements such as `if-the` and `while-do` that contain statements themselves.

6. Related Work

Related work can be divided into methods that perform lexical analysis and syntactical analysis. Another division comes from application domain with research focus-sing on computer language processing or on natural language processing.

Lexical Analysis Several tools are available that perform lexical analysis of textual files. The most well-known tool is probably `grep` and its variants (`fgrep`, `egrep`, `agrep`, etc.) that allows one to search text for strings matching a regular expression. These tools generally give little to no support to process the matched strings, they just print matching lines.

Such support is available in more advanced text processing languages as AWK [2] and PERL [39] and in the LEX scanner generator [25] that allow a user to execute certain actions when a specific expression is matched. TLEX provides a pattern matching and parsing library for C++ that generates parse trees for the strings that match a regular expression [19].

Hierarchical Lexical Analysis Murphy and Notkin describe the Lexical Source Model Extractor (LSME) [28]. Their approach uses a set of hierarchically related regular expressions to describe language constructs that have to be mapped to the source model. By using hierarchical patterns they avoid some of the pitfalls of plain lexical patterns but maintain the flexibility and robustness of that approach.

The MULTILEX system of Cox and Clarke [11] uses a similar hierarchical approach. The main difference with LSME is that it focuses at extracting information at the abstract syntax tree level whereas LSME extracts higher level source models.

This hierarchical technique is related to work in computational linguistics that divides natural language into chunks that can be recognized using a finite-state cascade parser [1].

Syntactic Matching Parser based approaches are used to increase the accuracy and level of detail that can be expressed. Syntactic matchers create a syntax tree of the input and allow the user to traverse, query or match the tree to look for certain patterns. This relieves them from having to handle all aspects of a language and focus on interesting parts.

Systems in this category are A* [23] that provide traversals over parse trees with AWK-like pattern matching and processing, TAWK [15] that provides similar operations on abstract syntax trees with processing in C.

Other tools support querying of the abstract syntax trees such as GENOA [14] that uses its own traversal language, REFINE [26] that allows queries in first order logic and SCRUPLE [29] that allows queries using concrete syntax.

The disadvantage of these systems is that they are all based on a full parse of the complete language making it hard/impossible to deal with incomplete sources, dialects or syntax errors. However, with the proper amount of interfacing, it should be possible to connect them to the island parsers we generate which would remove such problems.

Fuzzy parsing The notion of *fuzzy parsing* comes in two flavors. The first flavor are parsers that recognize a sentence as belonging to a language with a certain degree of correctness (thus allowing for grammatical errors) [24]. This type of fuzzy parsers is mainly used in computational linguistics for natural language processing. Productions in a *fuzzy grammar* are annotated with correctness degrees that are used to assess the quality of the input sentence. This can be used to model grammatical errors by adding special productions with a correctness degree less than 1 to an ordinary grammar. For more information, we refer to [3].

The second flavor of fuzzy parsers are parsers that are able to discard tokens and recognize only certain parts of a programming language [21]. The SNIFF programming environment was the first to use this kind of fuzzy parsing [5]. Since then, it has been used in a number of other programming environments and program browsers such as: CSCOPE[3], SOURCE NAVIGATOR[4], SOURCE EXPLORER[5], and the CRTAGS[6] tool. These fuzzy parsers are hand crafted to perform a specific task. They focus mainly on fuzzy parsing C and C++ to support program browsing. Typically this involves extracting information regarding references to a symbol, global definitions, functions calls, file includes, etc.

Parser Reuse Some approaches address the problems associated with parser or grammar development by reusing existing parsers (for example, in GENOA/GENII [14]). Others reuse or retrieve grammars that are used in existing tools [33]. However, both approaches ignore the fact that the structure of a grammar used in a tool is often tightly coupled to the design of that tool. Another tool may need a completely different grammar. Such parser reuse problems were also signaled by Reubenstein *et al.* [32]. Furthermore, this does not solve the robustness issues (dealing with missing code, embedded extensions or syntactical errors).

Island Parsing The term island parsing is also used in computational linguistics (for example [9, 34]). However, this is different notion referring to island parsers that start at some point in a sentence (by recognizing an island) and parse the complete sentence by extending that island to the left and right (in contrast to left-to-right scanning done by LL and LR parsers). This technique is used for example for speech recognition. A similar approach has been applied by Rekers and Koorn for computer languages to provide error recovery and completion in syntax directed editors [31].

Island Grammars The term island grammars was coined in [12] which provides an informal definition and small example but does not present a detailed discussion, nor does it describe tool support. We try to fill those gaps by improving the definition, describing properties of island grammars and providing a number of detailed examples that result in a reusable framework for island grammar definitions. Furthermore, we present a generator for source model extractors based on island grammars that supports various programming languages and show how it can be used in a number of case studies. A case study for COBOL island grammars is described in [37].

7. Conclusions

Robust parsing is a prerequisite for most reverse engineering tools. This paper shows that island grammars can be used to generate such parsers. The generated parsers combine the accuracy of syntactical analysis with the speed, flexibility and tolerance usually only found in lexical analysis.

Contributions of this paper are the extension of previous work on island grammars [12, 13] with a detailed discussion and definition of island grammars. We present MANGROVE, a generator for source model extractors based on island grammars. We provide a reusable framework for the definition of island grammars in syntax definition formalism SDF and support various processing languages allowing a developer to pick the language that fits his needs. We have shown how MANGROVE supports JAVA and ASF programmers by providing generated traversals that ease the mapping from parse results to source models. We report on the application of MANGROVE to a number of case studies and provide a detailed discussion of related work.

The combination of island grammars with generated traversals combines two forms of attractive default behavior: (i) island grammars allow us to limit ourselves that part of the grammar necessary to describe the problem at hand, and (ii) generated traversals allow us to treat only those cases for which we need specific behavior. Consequently, extractor specifications are small and easy to write, modify and combine resulting in a *lightweight, flexible* and *tolerant* approach.

Acknowledgments The author would like to thank Mark van den Brand, Tobias Kuipers, and Joost Visser for fruitful discussions. Arie van Deursen, Jan Heering and Paul Klint provided valuable feedback on earlier versions of this paper.

[3] http://cscope.sourceforge.net/

[4] http://sources.redhat.com/sourcenav/

[5] http://www.intland.com/

[6] http://www.vital.com/crtags.html

References

[1] S. Abney. Partial parsing via finite-state cascades. In *Proc. ESSLLI '96 Robust Parsing Workshop*, 1996.

[2] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[3] P.R.J. Asveld. A bibliography on fuzzy automata, grammars and languages. *BEATCS*, 58:187–196, 1996.

[4] J. A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In *Algebraic Specification*, chapter 1, pages 1–66. ACM Press & Addison-Wesley, 1989.

[5] W. Bischofberger. Sniff—a pragmatic approach to a C++ programming environment. In *Proc. 1992 USENIX C++ Conference*, pages 67–82, Aug. 1992.

[6] M.G.J. van den Brand et al. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proc. Compiler Construction 2001*, LNCS, 2001.

[7] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. Technical report, CWI, 2001. To appear. Contact `markvdb@cwi.nl` for copies.

[8] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a Cobol grammmar from legacy code for reengineering purposes. In *Proc. 2nd Int. Workshop on the Theory and Practice of Algebraic Specifications*, EWIC. Springer Verlag, 1997.

[9] J. Carroll. An island parsing interpreter for the full augmented transition network formalism. In *Proc. 1st Conf. EACL*, pages 101–105, 1983.

[10] Compuware. *UNIFACE: An Environment for Building Complex, Business-Critical Applications*, Sep. 2000. White paper.

[11] A. Cox and C. Clarke. A comparitive evaluation of techniques for syntactic level source code analysis. In *Proc. 7th Asia-Pacific Softw. Eng. Conf.*, Dec. 2000.

[12] A. van Deursen and T. Kuipers. Building documentation generators. In *Proc. Int. Conf. on Software Maintenance*, pages 40–49, 1999.

[13] A. van Deursen, T. Kuipers, and L. Moonen. Arrangement and method for a documentation generation system. U.S. Patent. Applied Aug. 2000.

[14] P. Devanbu. Genoa - a customizable, front-end retargetable source code analysis framework. *ACM Trans. Softw. Eng. Meth.*, 8(2):177–212, Apr. 1999.

[15] W.G. Griswold, D.C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proc. 4th Workshop on Program Comprehension*, 1996.

[16] Jozef Gruska. Descriptional complexity of context-free languages. In *Proc. Mathematical Foundations of Computer Science*, pages 71–83, 1973.

[17] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF—Reference Manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[18] R. Holt, A. Winter, and A. Schürr. GXL: Towards a standard exchange format. In *Proc. 7th Working Conf. on Reverse Engineering*, pages 162–171, 2000.

[19] S. Kearns. Tlex. *Softw. Pract. Exp.*, 21(8):805–821, Aug. 1991.

[20] P. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Meth.*, 2:176–201, 1993.

[21] R. Koppler. A systematic approach to fuzzy parsing. *Softw. Pract. Exp.*, 27(6):637–649, 1997.

[22] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In *Proc. Workshop on Language Descriptions, Tools and Applications*, pages 28–52, 2001. ENTCS volume 44.

[23] D. Ladd and J. Ramming. A*: A language for implementing language processors. *IEEE Trans. Softw. Eng.*, 21(11):894–901, 1995.

[24] E.T. Lee and L.A. Zadeh. Note on fuzzy languages. *Information Sciences*, 1(4):421–434, 1969.

[25] M. Lesk and E. Schmidt. Lex—a lexical analyser generator. Computer Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, USA, Oct. 1975.

[26] L. Markosian, P. Newcomb, R. Brand, S. Burson, and T. Kitzmiller. Using an enabling technology to reengineer legacy systems. *Comm. ACM*, 37(5):58–70, 1994.

[27] T.J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.

[28] G.C. Murghy and D. Notkin. Lightweight lexical source model extraction. *ACM Trans. Softw. Eng. and Meth.*, 5(3):262–292, Jul. 1996.

[29] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Trans. Softw. Eng.*, 20(6):463–475, 1994.

[30] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[31] J. Rekers and W. Koorn. Substring parsing for arbitrary context-free grammars. *SIGPLAN Not.*, 26(5):59–66, 1991.

[32] H. Reubenstein, R. Piazza, and S. Roberts. Separating parsing and analysis in reverse engineering tools. In *Proc. 1st Working Conf. on Reverse Engineering*, pages 117– 125, 1993.

[33] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In *Proc. Int. Conf. on Software Maintenance*, 1999.

[34] O. Stock, R. Falcone, and P. Insinnamo. Island parsing and bidirectional charts. In *Proc. 12th Conf. on Computational Linguistics*, pages 636–641, 1988.

[35] T.A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley, 1988.

[36] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer, 1985.

[37] E. Verhoeven. COBOL island grammars in SDF. Master's thesis, Informatics Institute, University of Amsterdam, 2000.

[38] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[39] L. Wall and R.L. Schwarz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[40] A. Watson and T. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication, 500–235. U.S. National Institute of Standards and Technology, Washington, D.C., Sep. 1996.