Lightweight Impact Analysis using Island Grammars

Leon Moonen

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

Abstract

Impact analysis is needed for the planning and estimation of software maintenance projects. Traditional impact analysis techniques tend to be too expensive for this phase, so there is need for more lightweight approaches.

In this paper, we present a technique for the generation of lightweight impact analyzers from island grammars. We demonstrate this technique using a real-world case study in which we describe how island grammars can be used to find account numbers in the software portfolio of a large bank. We show how we have implemented this analysis and achieved lightweightness using a reusable generative framework for impact analyzers.

Keywords and phrases: *Island grammars, parser generation, impact analysis, program understanding, software exploration.*

1. Introduction

Estimates indicate that 70% of software budgets are spent on software maintenance [1]. The two most expensive activities in software maintenance are *understanding* the software system that has to be maintained and determining the *impact* of proposed change requests [3]. Consequently, research that addresses techniques to assist maintainers in performing these tasks can make an important contribution.

A significant part of the program understanding research focuses on generic tools such as program browsers and documentation generators. These tools generally try to provide various means of querying or navigating through a software system that can be used by maintainers to answer their questions. There are obvious advantages to such a generic approach: it offers wide applicability, and it is easy to see cost-performance benefits of such tools.

However, we think that there is also a need for program understanding tools that are more tailored towards the questions to be answered. These tools should generate detailed reports or browsers that allow a maintainer to understand code with respect to such a specific question.

A typical task that would benefit from such problem-directed tooling is, for example, assessing the costs of mass change project such as Euro-conversion or database migration. This typically boils down to estimating questions like: How many systems are affected? How much code needs to be changed? Where do we need to make changes? Finding the answers to such questions is the domain of *(software change) impact analysis* [3].

One would expect that making such estimates is relatively cheap; It is hard to justify that the costs of estimating a mass change project are similar to performing the project. However, that is exactly what would happen if those estimates were based on a full blown impact analysis. Performing such an analysis would take almost the same amount of time and resources as the actual project. Consequently, a more *lightweight form of impact analysis* is needed.

A common approach for achieving lightweightness is based on the use of lexical analysis [9]. This has several advantages: lexical analysis is a flexible and robust solution that can handle incomplete and syntactically incorrect code. Additionally, it often takes little time to develop solutions based on lexical tooling. Unfortunately, there are also some serious drawbacks: Lexical analysis tends to be sensitive to the layout of the code that is being analyzed, for example, a simple newline may prevent recognition of a language feature. Furthermore, it is hard to write lexical analysers that take the structure of a language into account. Consequently, lexical analysis results typically have lower accuracy and completeness than those of syntactical analysis.

We think that an approach based on island grammars [19] is better suited for creating lightweight impact analyzers. In this paper, we investigate this hypothesis. We will do this using a case study in which we revisit a project that was performed earlier by our spin-off, the Software Improvement Group, for one of their customers.

This project involved estimating in which parts of the software portfolio of a large Dutch bank changes have to be made when converting their 9-digit account numbers into 10-digit account numbers. The customer was interested in a quick-scan of their complete software portfolio for planning

Proceedings of the 10 th International Workshop on Program Comprehension (IWPC'02)
1092-8138/02 $17.00 © 2002 **IEEE**

and estimation purposes. This portfolio consists of 200 systems containing a total of 50 000 000 lines of COBOL code.

The paper is organized as follows: Section 2 gives an overview of the problem and Section 3 sketches the impact analysis that is needed to solve this problem. Island grammars are described in Section 4, followed by a discussion of how the impact analysis is translated into island grammars in Section 5. The implementation of our analyzer is described in Section 6. Section 7 generalizes our approach to other applications . Finally, Sections 8 and 9 summarize related work, discuss future extensions and draw conclusions.

2. Problem Description

Currently, most Dutch banks use client account numbers that consist of 9 digits. Collective agreements ensure that each number is used uniquely and each bank typically uses certain sub-ranges of the spectrum. The pool of unassigned account numbers is managed by a subsidiary where banks can apply for free numbers. Since this supply of unused numbers is running out, the banks have decided that they will convert their systems to account numbers that consist of 10 digits (by prefixing existing numbers with 0 and using the prefixes 1...9 for fresh numbers).

This poses several questions for managers that are responsible for the software portfolio of a bank: How much of our portfolio is affected by this decision? For a given system, where are the parts that need to be changed? How many of these changes can be done automatically?

In our case study, we investigate if it is possible to answer such questions using an impact analysis technique based on island grammars. It is important to keep in mind that we are looking for a *lightweight* technique that can be used to analyze the impact on the complete software portfolio. The goal is to enable correct *estimation and planning* of the next steps in this mass change project. Consequently, our focus is more on short development time and enabling quick feedback for the complete portfolio, rather than on the detailed and complete impact analysis that would be needed to actually remedy the situation.

Furthermore, since the need for this conversion has been known for some time, some of the newer (or updated) parts of the software portfolio are already prepared for 10-digit account numbers. An important aspect of this study is that we need to handle code that contains a mixture of "good" and "bad" account numbers, and that we need to distinguish between them in order to provide correct estimates.

3. Impact Analysis Approach

This section describes how we want to perform the impact analysis that was described in the previous section. In the next section, we will describe how we have implemented it.

3.1. Patterns

The case study started with talking to (representatives of) the maintainers of the software to see how they would normally perform this kind of impact analysis. From these discussions, it turned out that it was possible to search the system's artifacts for variables that might represent bank account numbers. This search is partially based on pattern matching on the names of the variables, so together with the maintainers, a list of patterns was compiled that would signal bank account numbers in the software. The starting point for the compilation of such a list is the organization's data dictionary. Typical examples of patterns in this list are `ACCOUNTNR`, `ACCNR`, `ACC-NO` and `GNR` (that last one is used for giro number).[1]

3.2. Classification

Besides the variable name, also the type of the variable plays an important role in the analysis. We would like to distinguish between variables with a *9-digit* type that need to be changed, and variables with a *10-digit* type that are already correct. Unfortunately, COBOL does not have a real type system. Instead, with each variable declaration, a description of the memory layout that this variable uses is given (so called *pictures* or *picture clauses*). Pictures give a character-by-character definition of the format of variable. The characters have special meanings. For example, '`X`' is used to denote a memory position that can hold an alphanumeric character; '`9`' is used for a numeric characters, and many others exist. Typical bank account numbers may be described in COBOL as follows:

```
01  ACCNR        PIC 9999999999.
01  FMT-ACCNR    PIC 99.99.99.999.
01  ACCOUNTNR    PIC 9(10).
```

The first line describes a variable with name `ACCNR` that consists of 9 digits (the picture `9` indicates 1 digit, `99` indicates two digits, and so on). The second line declares a variable `FMT-ACCNR` consisting of 9 digits but formatted using dots. The last example, shows a variable with name `ACCOUNTNR` consisting of 10 digits (the number between brackets indicates repetition).

We will classify the variables with matching names based on their picture clauses. We distinguish four classes:

A. *9-digit* variables with numeric pictures such as `999999999`, `9(9)`, `99.99.99.999`, and alphanumeric pictures such as `X(9)`, and `XX.XX.XX.XXX`.

B. *10-digit* variables with numeric pictures such as `9999999999`, `9(10)`, `999.99.99.999`, and alphanumeric pictures such as `X(10)`, and `XXX.XX.XX.XXX`.

[1] The examples in this paper were taken from a Dutch software system. Although we have translated variable names into English, some names or abbreviations may look strange or uncommon since there is no good translation. Most notably is "giro", which is a bank transfer service in Europe.

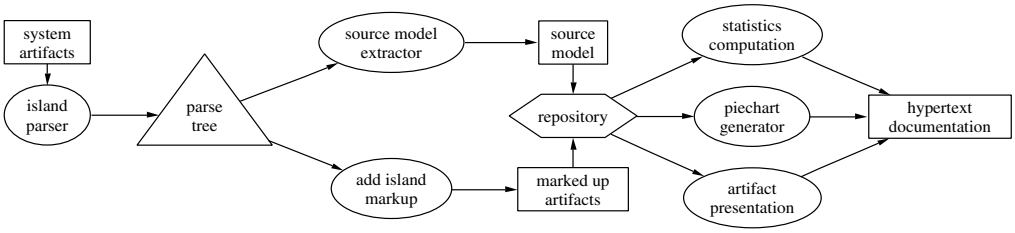

Figure 1. ISCAN **architecture.**

C. *Record* variables do not have their own picture description but consist of a number of sub-fields. These sub-fields can be used, for example, to address parts of an account number (some banks use the first 4 digits of account numbers to identify the branch where this account was opened).

D. *Other* variables whose names match with the patterns but whose pictures do not fall in the above classes.

3.3. Anti-patterns

As described in Section 3.1, we start with a number of patterns that might indicate that a variable is used as an account number. However, there are a number of variables that have names that match with these patterns but we know (for example, from code inspection) that they are not used for account numbers. We call such variables *false positives*. We have taken the following steps to reduce false positives:

- During the project, a number of *anti-patterns* have been identified that match with field identifiers that are certainly not used for account numbers;
- Whenever a variable name matches both a pattern and an anti-pattern, that variable is rejected. When a variable name matches a pattern and none of the anti-patterns, it is accepted.

This process can be applied iteratively: whenever inspection of the results shows false positives, anti-patterns are added and the analysis is repeated. Thus, the precision of the analysis can be increased by investing in these iterations.

3.4. Presentation

We will report our findings using hypertext documentation consisting of: (i) pages displaying statistics and pie-charts summarizing the analysis results, and (ii) hyperlinked and pretty-printed artifacts that will lead the maintainer to all affected sites (i.e., all occurrences of account numbers). We add this second type of reports since they allow the maintainer to inspect analysis results and check hypotheses about impact. Furthermore, they are useful for identifying false positives and adding anti-patterns. The account numbers in the hyperlinked code are colored to show their classification: e.g., red for 9-digits, green for 10 digits, etc.

3.5. Tool Support

We have created ISCAN, a lightweight impact analysis tool to derive the described information. The basic structure of this tool is depicted in Figure 1. It follows the extract-query-view approach quite common to reverse engineering tools.

We start with parsing the system artifacts using an island parser. The parse results are processed in two ways:

Source Model Extraction: we extract source models describing for each artifact, the account numbers that were found, their classification and details about their origins (file and position information of the actual code). This data will be used later on for statistics and pie charts. The origins can be used for hyperlinking the results.

Island Markup: to enable problem-directed pretty-printing, we add markup to the artifacts, tagging all account numbers for later reference.

The results are stored in a repository which is used by a number of tools: a statistics tool queries the source models to generate statistical overviews, summarizing the account numbers per program, per system and in the complete portfolio. Another tool generates pie-charts that give a different view of the impact and affected code. A third tool presents the artifacts as a hyperlinked website. This tool uses the markup to pretty-print the code, visualizing the classification with different colors. Furthermore, it generates cross-references such as tables of content and indexes.

4. Island Grammars

One of the challenges of building a lightweight impact analysis tool is parsing a system's artifacts to extract the information we need. There are a number of reasons why it is hard (or even impossible) to parse the artifacts using common parser based approaches:

Grammar availability: We might want to analyze legacy systems written in a language for which there is no grammar available. Writing such grammars from scratch is tedious and expensive, and may not pay back at all. For example, van den Brand *et al.* report a period of four months for the development of a fairly complete COBOL grammar [6].

Completeness: The source code of a system may be incomplete. For example, some of the header files (or copybooks) may be lost making a full reconstruction impossible, or collecting all files may be too time consuming, making it unfeasible for cost estimation purposes only.

Dialects: Legacy languages such as COBOL (but also languages like C) have a number of, slightly different, vendor-specific dialects. 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. We might want to consider both languages in our analysis. Most parser based approaches have difficulties with that.

Customer-specific idioms: Some systems 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. A standard parser will not recognize such constructions.

Preprocessing: The use of preprocessor directives can hinder parsing but analyzing already preprocessed code might give results that are not expected by the maintainer (since his mental views are based on unprocessed code).

It has been proposed to use *lexical analysis* techniques to remedy these problems [20, 9]. Lexical analysis provides a flexible and robust solution that can handle incomplete and syntactically incorrect code. Additionally, it often takes less time to develop a lexical analyzer than a syntactical one.

However, there are also disadvantages to a lexical approach: the analysis results typically have lower accuracy and completeness than those of syntactical analysis. Lower accuracy means an increase of false positives (analysis finds properties for code which it does not have in reality). Lower completeness means an increase of false negatives (analysis misses properties that are present in reality).

In this paper, we set out to use syntactical analysis based on *island grammars* [19] to remedy these problems.

Definition 4.1 *An* island grammar *is a grammar that consists of two parts: (i) detailed productions that describe the language constructs that we are particularly interested in (so called* ***islands****), and (ii) liberal productions that catch the remainder of the input (so called* ***water****).*

In a way, island grammars mix the behavior of parsing with that of lexical approaches by analyzing the interesting parts of a grammar and brushing aside the non-interesting parts. By doing that, they combine the accuracy of syntactical analysis with the speed, flexibility and robustness of lexical analysis. Table 1 gives an overview of the approaches.

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 [16, 23]. We benefit from the expressive power of this combination 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 spin-off, the Software Improvement Group, as part of the documentation generator DOCGEN [10]. The requirements originating from the LL parsing technique used in JAVACC made development and extension of this grammar unwieldy. The tooling described in [19] enables re-implementation based on SDF and generalized LR parsing.

4.1. Syntax Definition in SDF

Before we continue with an island grammar example, we give a short overview of the syntax definition formalism SDF. Syntax definitions in SDF combine the definition of lexical and context-free syntax in the same formalism. The definitions are purely declarative (e.g., as opposed to 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 in Figure 2 that is extended by module DataFields in Figure 3.

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 or avoided when alternatives exist, and associativity attributes for binary productions (for example, $S\ op\ S \rightarrow S$ {left}). Productions can be labeled with identifiers using the {cons} attribute. These labels appear in the parse tree so we can see which production was used to construct a given (sub)term.

SDF is supported by a parser generator that produces *generalized LR* (GLR) parsers. Generalized parsing allows

	lexical analysis	syntactical analysis	
		full grammar	island grammar
accurate	–	+	+
complete	–	+	+
flexible	+	–	+
robust	+	–	+

Table 1. Lexical vs. syntactical analysis.

```
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 2. Base for island grammars.

definition of the complete class of context-free grammars instead of restricting it to a non-ambiguous subclass such as LL(k), LR(k) or LALR(1), which is common to most other parser generators [22, 21]. 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 (unlike restricted subclasses), generalized parsing allows for better modularity and syntax reuse. For more information on SDF, we refer to [16, 23].

4.2. Island Grammar Example

Figures 2 and 3 show an example island grammar that describes COBOL data fields. 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.

The grammar contains three modules: The module Layout specifies the lexical non-terminal symbol LAYOUT containing white-space 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 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

```
module DataFields                               (1)
imports Water DataParts                         (2)
context free syntax                             (3)
  Level DataName        → Chunk  {cons(Data)}   (4)

module DataParts                                (5)
lexical syntax                                  (6)
  [A-Z][A-Z0-9\-]*      → DataName              (7)
  [0][1-9]              → Level                 (8)
  [1-4][0-9]            → Level                 (9)
```

Figure 3. COBOL data fields.

```
module DataFieldsWithContext                    (1)
imports Water DataParts                         (2)
context free syntax                             (3)
  "DATA DIVISION" DdChunk*                      (4)
  "PROCEDURE DIVISION" → Chunk                  (5)
  Level DataName       → DdChunk {cons(Data)}   (6)
  Water                → DdChunk                (7)
```

Figure 4. COBOL data fields in context.

instead of Water. The lexical non-terminal Water consists of a list of one or more characters that are not white-space (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 DataFields in Figure 3 adds such an island by specifying that a Chunk can also be produced by a Level number followed by a DataName (line 4). DataNames are characters followed by zero or more characters or digits (line 7). Level numbers lie between 01 and 49 (lines 8 and 9)

This very simple grammar allows us to generate a parser that searches for data fields 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 we can extend this grammar to do a more complicated analysis. Furthermore, the modularity of SDF allows us to reuse the base grammar developed here for other island grammars.

The grammar in Figure 3 is not very discerning. Consider an input program that contains the following line:

```
IF C > 10 AND C < 20
```

The grammar will recognize this line as water containing the text `IF C >`, followed by a data field with level `10` and name `AND`, followed by water containing `C < 20`. Something similar will happen for all other number–name sequences in the code. Obviously, this is not correct, so we need to improve our grammar.

The solution is to restrict the grammar so it will only look for data fields in the data division of the program. This can be done by refining the grammar into the one that is shown in Figure 4. The production on line 4 and 5 specifies that non-terminals of type DdChunk are only to be recognized in a context that starts with the text `DATA DIVISION` and and ends with the text `PROCEDURE DIVISION`. Line 6 defines these DdChuncks to be our data fields that can be produced by a Level and a DataName (similar as in our original grammar). Furthermore, our grammar needs to be made robust

```
module DataNames                                          (1)
imports Patterns AntiPatterns                             (2)
lexical syntax                                            (3)
  DNPart Pattern DNPart       → DataName                  (4)
  DNPart AntiPattern DNPart   → DataName  {reject}        (5)
  [A-Z0-9\-]*                 → DNPart                    (6)

module Patterns                                           (7)
lexical syntax                                            (8)
  "ACCNO" | "GNR" | ...       → Pattern                   (9)

module AntiPatterns                                       (10)
lexical syntax                                            (11)
  "DAGNR" | ...               → AntiPattern               (12)
```

Figure 5. Matching patterns in data names.

against other data that can occur in this context (i.e., parts of the input that do not match with the data field definition). We do this by also allowing water to be recognized between the markers (line 7).

This example shows some of the advantages that island grammars have over lexical approaches. Most importantly, it is much easier to use structure while specifying patterns to be analyzed. For example, it would be really hard to limit `grep` so it would only match in the data division. In other lexical approaches one would use state manipulation to achieve such results. However, when parts of the analysis logic are hard-coded, adapting the analyzer or combining two analyzers into a single new one becomes a tricky job. In contrast, solutions based on island grammars can easily be combined and are declarative, making them easier to understand.

5. Performing Impact Analysis using Island Grammars

To perform impact analysis, we create an island grammar in which the islands are based on the patterns, anti-patterns and classification described in Section 3. When we then parse the artifacts using a parser generated from this grammar, the parse trees will contain the analysis results.

Unfortunately, the complete grammar that was used for our case study is too large to show in this paper. It consists of 148 productions which is mainly due to the large amount of patterns and anti-patterns (125 to be precise). Below, we will describe and show the most interesting parts.

When developing the grammar, we start with an "empty" island grammar that "parses" the complete input as water. This is the grammar shown in Figure 2.

Next, we extend this grammar with island productions for data fields with generic field identifiers. The resulting grammar can be used to extract all data fields from COBOL sources. However, we are interested in more specific information: we are looking for data fields whose names match with the account number patterns. Therefore, we refine the identifier syntax so it only matches with the account number patterns. The consequence of this refinement is that all data fields whose names do not match are now parsed as water.

The following step is filtering all data fields whose names match with anti-patterns. In SDF, we can use the `reject` attribute to prevent that names that match with an anti-pattern can be parsed as valid data names. The resulting grammar will only parse data fields whose names match with one of the patterns and with none of the anti-patterns.

The grammar parts responsible for recognizing data names by matching patterns and rejecting anti-patterns are shown in Figure 5. The actual data names are described in module DataNames: line 4 defines that a DataName contains at least one of the patterns, line 5 defines that a DataName containing one of the anti-patterns is not valid (i.e, should be rejected), and line 6 defines the possible pre- and postfixes for the patterns.

The patterns are defined in modules Patterns and AntiPatterns. For brevity, we show only a few patterns of the actual list. The anti-pattern `DAGNR` in line 13 is a Dutch abbreviation for day number which would normally give a false match with `GNR`.

Classification of the account number variables is done by building up all potential picture clauses from a number of patterns for each class. This is shown in Figure 6 on the next page. We start with a production describing the 9-digit pictures (line 1), followed by the 10-digit pictures in line 2. Furthermore, we want a production for all remaining pictures of variables that match a pattern. We let the parser construct that class by specifying that it consists of an arbitrary PictureString (line 3 and 7) but prevent that the 9-digit and 10-digit pictures are parsed by this production using the reject in line 4. In a way, this reject allows us to "subtract" a set of pictures from the large set described by the production in line 7.

The grammar that combines data names with picture clauses into data descriptions is shown in Figure 7. We can use these data descriptions to refine the grammar from Figure 4 by replacing the definition of DdChunk in line 6 with a production: DataDesc → DdChunk. The resulting island

```
module DataDesc                                           (1)
imports DataNames Pictures                                (2)
context free syntax                                       (3)
  Level DataName "."          → DataDesc {cons(Rec)}      (4)
  Level DataName Water*                                   (5)
  "PIC" Picture               → DataDesc {cons(Field)}    (6)
lexical syntax                                            (7)
  [0][1-9]                    → Level                     (8)
  [1-4][0-9]                  → Level                     (9)
```

Figure 7. Recognizing valid data descriptions.

```
module Pictures
context free syntax
  ”999999999”  |  ”9(9)” | ”99.99.99.999” | ...   → Pict        {cons(Short)}   (1)
  ”9999999999” | ”9(10)” | ”999.99.99.999” | ...  → Pict        {cons(Long)}    (2)
  PictureString                                    → RestPict    {cons(Other)}   (3)
  Pict                                             → RestPict    {reject}        (4)
  Pict | RestPict                                  → Picture                     (5)
lexical syntax                                                                  (6)
  [0-9XxAa\(\)pZzVvSszBCRD\/\,\$\+\-\*\:]+         → PictureString               (7)
```

Figure 6. Recognizing and classifying picture clauses.

grammar can be used to extract account numbers from a COBOL source and classify them using their picture clauses.

6. Generation of Impact Analyzers

This section describes implementation details of how we have built the ISCAN impact analyzer. As discussed in the introduction, we expect that there is general interest for the kind of lightweight impact analyzers described in Section 3. Therefore, we set out to build tooling that can help the maintainer to create such tools. Our design goal is to minimize the amount of work needed for creation of a new analyzer.

We have composed a generative framework for the creation of impact analyzers using island grammars. A maintainer can instantiate the framework using simple specifications detailing the problem at hand. This results in generation of a new impact analyzer that performs an analysis dedicated to the given problem. An overview of the generator framework is shown in Figure 8 on the next page. The gray boxes depict maintainer inputs.

The minimal amount of work that needs to be done to create a new analyzer is very small: it consists of writing the island grammar that specifies affected sites in the artifacts. This grammar plays a central role in the generation of the remaining parts. It is used to generate an island parser and it is part of the inputs needed for the generation of source model extraction and artifact markup.

For the remaining steps in the process, we supply generic defaults. These include:

- A source model extractor that stores information regarding islands recognized by the island parser in a repository.
- A transformation that adds markup to the artifacts, tagging islands recognized by the island parser with their respective types.
- Tools for computing statistics and generating pie charts based on the types of islands available in the grammar.

These generic components can be refined by the maintainer to perform a more specific task (the dashed inputs in Figure 8).

Source Model Extraction The source model extractor is created using MANGROVE/JAVA, a generator for source model extractors based on island grammars [19]. The extractor processes the results of the island parser using JAVA. The default extractor specification that we provide is a simple JAVA class that stores information regarding all islands that were recognized by the parser in a repository. This class can easily be refined by a user to perform a more specific task, for example, storing only information about particular islands or computing extra information based on a combination of islands.

Island Markup The transformation that adds markup is generated using MANGROVE/ASF [19]. It processes parse results in a functional fashion using the term rewriting language ASF [2]. Specifications written in ASF can be executed using the ASF+SDF Meta-Environment [17, 4]. This environment contains support for the generation of term traversal functions [5]. We use these in our default specification to tag all islands that are recognized by the parser with their respective types. This specification can also be refined by the user to perform more specific tasks.

We have chosen to do this transformation using the ASF+SDF Meta-Environment since it allows us to keep the original layout intact while transforming the artifacts [7]. Preserving layout is an important feature in a maintenance tool since it helps a maintainer to orientate when visiting a system that he has seen before.

Presentation We use XML to mark up the artifacts. The marked up artifacts are used for pretty-printing and to generate indexes and tables that cross-reference the various classes and sources. Our current back end generates a series of HTML documents. The transformation of XML to HTML is done using XSL transformations (XSLT). These transformations can be done either on the server side, for example using the XALAN XSLT processor[2], or on the client side using a modern browser such as NETSCAPE 6 or INTERNET EXPLORER 5. The account numbers in the generated documentation are colored to show their classification: red for 9-digits, green for 10 digits, etc. The actual colors that are used can be changed easily by editing a style-sheet.

[2] `http://xml.apache.org/xalan-j/`

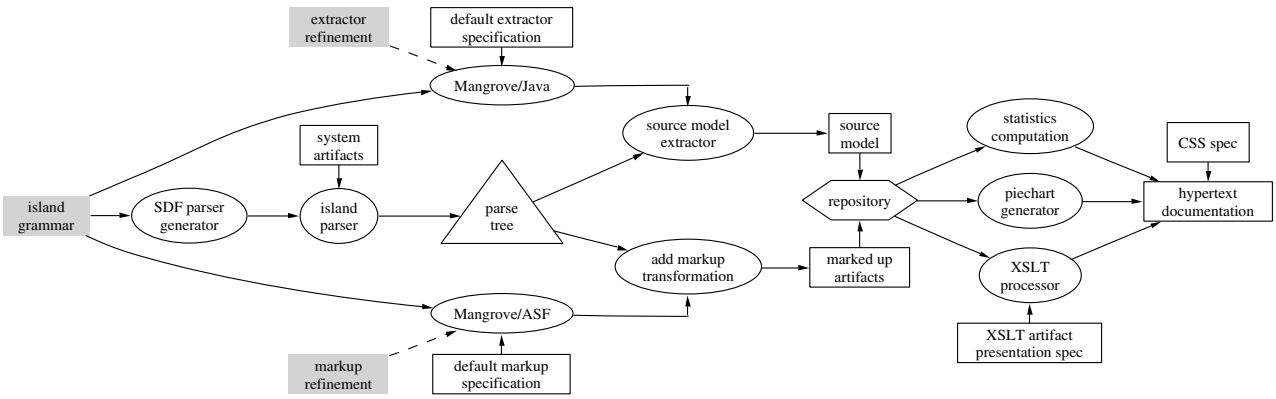

Figure 8. Implementing the ISCAN architecture.

7. Applications

In this paper, we have focused on solving a specific case: the impact of expanding 9-digit bank account number into 10-digit numbers. There are many more of such problems to which our technique can be applied:

Product codes: We have encountered a problem that was very similar to the bank account number analysis in another project that was done by our spin-off, the Software Improvement Group. The problem there was a large software system that used product codes that consisted of 2 digits. The goal of the project was a transformation of this system that expanded the product codes to consist of 3 digits (surprisingly with a maximum of 299 instead of 999).

Trading natural gas: Liberalization of the gas-market in Western Europe makes it possible for consumers to pick the gas supplier they like. To enable this, the various gas networks have been interconnected, making it easier for producers to sell their gas in more remote markets. Before liberalization, gas trading contracts were based on capacities in m^3 per hour. However, the caloric value of natural gas differs between gas reserves, so the actual energy value that is purchased/sold with 1 m^3 also differs per gas reserve. Since this is not a competitive price model when trading between various gas reserves, gas trading companies want to convert from capacities in m^3 per hour to capacities in kW per hour. Obviously, this conversion implies mass changes in their trading and accounting software.

Selling natural gas: Another change that gas trading companies want to make has to do with consumer accounting. Historically, gas supply days run from 6am one day to 6am the next day. This poses several problems for service integration, for example, when one would like to combine gas and electricity billing. Therefore these supply days need to be changed into the standard 0am-12pm schedule.

Date format conversion: Changing the date representation in a system from the U.S.A. date format (MM/DD/YYYY), or the European date format (DD/MM/YYYY), into the international ISO date format (YYYY-MM-DD).

8. Related Work

Impact Analysis Bohner and Arnold give a tutorial style overview of research topics in the area of software change impact analysis [3]. The articles focus on the traditional full-blown impact analysis that one would use to process a change request, and not on the kind of lightweight impact analysis that we focus on. The techniques described in this book will be too expensive to be practical for the estimation and planning phase of a software change project. However, they will be needed in the next phases of the project.

Most of the traditional impact analysis approaches are based on program slicing [14] and program dependence graph analysis [18]. Han describes an impact analysis approach that is based on direct analysis of the system artifacts [15]. Similar to our approach, it analyzes the parse trees to determine impact and change propagation. Han argues that such a direct approach is better suited for providing impact analysis and change propagation as integral parts of a software engineering environment.

Fyson and Boldyreff describe the use of program understanding to support impact analysis [13]. They use information derived by a program understanding system to populate a so called ripple propagation graph. By tracing the edges of this graph, one can identify all systems that are affected by a change.

Lexical Approaches Several tools are available to perform lexical analysis. The most well-known tools are probably `grep` and `perl` that allows one to search text for strings using regular expressions.

Murphy and Notkin describe the Lexical Source Model Extractor (LSME) [20]. 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 [8] 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.

These tools offer no immediate support for impact analysis. They are directed at extracting the facts from system artifacts and not at querying, combining and presenting those facts to the maintainer to answer a question or to perform impact estimation.

Rapid System Understanding Van Deursen and Kuipers describe techniques for rapid system understanding that are based on lexical analysis [9]. They describe an open architecture for system understanding that can be easily adapted to perform a problem-directed analysis. This makes it easy to use their technique for performing impact analysis for estimation and planning.

Island Grammars Island grammars [19] are a technique for syntactical analysis that allows us to mix the behavior of parsing with that of lexical approaches by analyzing the interesting parts of a grammar and brushing aside the non-interesting parts. Thereby, island grammars combine the accuracy of syntactical analysis with the speed, flexibility and robustness of lexical analysis. In our earlier paper, we describe the definition of island grammars using the syntax definition formalism SDF and present MANGROVE, a generator for source model extractors based on island grammars that supports refinements in various programming languages and show how it can be used. For a more detailed discussion of related work regarding island grammars and source model extraction using syntactical and lexical analysis, we refer to [19].

9. Concluding Remarks

9.1. Evaluation

By using an island grammar to perform impact analysis, we limit ourselves to types of analysis that can actually be described using grammars. These are the kind of analyses that are based on determining the presence, absence, and classification of features in an artifact. They exclude, for example, analyses that are based on data flow information or dependency tracking. Note that when more detail is needed, an approach based on island grammars can be improved to a certain extent by doing more involved computations in the tools that extract source models and markup artifacts.

We argue that the type of analysis that can be described using island grammars is sufficient for our goal: lightweight impact analysis for estimation and planning. This is supported by the fact that others revert to lexical analysis techniques to achieve this goal (e.g., [9]). The use of island grammars has several advantages over lexical approaches. Most importantly, it is much easier to use structure in the specification of the patterns. Furthermore, solutions based on island grammars can easily be combined and are declarative, making them easier to understand.

test	no. of programs	total size (LOC)	analysis time (s)	speed (LOC/s)	memory usage
(a)	206	233 252	403	579	49 Mb
(b)	818	901 899	1 549	582	54 Mb

Table 2. Benchmark results

We identify two potential sources of problems with island grammars: (i) *false positives* that occur when the grammar allows constructs to be recognized in places where they should not have been recognized. (ii) *false negatives* that occur when the grammar is too restrictive and does not allows constructs to be recognized in places where they should been recognized. These errors can be solved by strengthening the grammar, we refer to [19] for a discussion of possible approaches.

The expressive power of an island grammar is limited by the chosen syntax definition formalism and more important by the chosen parsing technique. We express island grammars in SDF, a syntax definition formalism that is supported by *generalized LR* parsing techniques. Consequently, we inherit their expressive power, which allows us to express the complete class of context free languages.[3]

To get an indication of the speed and scalability of our approach, we have tested the generated impact analyzer on representative parts of the earlier described software portfolio (the complete portfolio could not be used due to disclosure restrictions). We have performed two tests: (a) one on a single system, and (b) one on a collection of four systems. Table 2 gives an overview of the test results. The analysis time is the user CPU time as reported by the GNU `time` command and the maximum memory usage was observed using `top`. The tests were performed on a computer with an AMD Athlon processor (1.2 Ghz) and 512 Mb main memory running linux 2.4.9-12.

Since the tests were done on representative systems (with similar average, largest and smallest program size), we think that these results can be extrapolated. Thus, impact analysis of the complete portfolio of 50 000 000 LOC will take approximately 1 day (24 hours), which is more than acceptable for estimation purposes on a project of this size.

9.2. Future Work

We are interested in investigating how we can extend our approach with dependency tracking to perform a more detailed analysis. We want to do this using the *type infer-*

[3] And some non-context free languages because the *reject* attribute allows us to compute the difference or intersection of two languages. For more details, see the discussion in [23, p. 52–56].

encing technique described in [11, 12]. The basic idea is as follows: we will use the data fields found by the lightweight impact analysis as *seeds*. Initially, these seeds get a unique type. These types will be propagated through the statements in the program and track all related (type equivalent) fields encountered during this propagation.

We need to make the following additions to our framework: First, the island grammar is refined so the generated parser will recognize assignments and expressions as islands. Then, we extend the source model extractor, so it emits primitive type relations for the seeds, type equivalencies for expressions and subtyping for assignments that are encountered. These relations can be used to find all fields that are type equivalent with the seeds following the algorithm described in [12, figure 5]. We can classify these sets using the classification that was found for the seeds since all type equivalent fields should be in the same category. Finally, we can use this new information to generate a more detailed overview of the impact on the code.

9.3. Contributions

Lightweight impact analysis is a prerequisite for estimating and planning large scale software maintenance projects. This paper shows that island grammars can be used to generate such lightweight impact analyzers.

We have given a detailed description of the process of translating an impact analysis problem into an island grammar. We have discussed the advantages that this approach has over other techniques for impact analysis. We have presented a generative framework that allows a maintainer to create lightweight and problem-directed impact analyzers. We have demonstrated our technique using a real-world case study where island grammars are used to find account numbers in the software portfolio of a large bank.

Acknowledgments The author would like to thank Arie van Deursen, Paul Klint and Joost Visser for providing valuable feedback on drafts of this paper.

References

[1] K.H. Bennet. An introduction to software maintenance. *Information and Software Technology*, 12(4):257–264, 1990.

[2] 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.

[3] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[4] M.G.J. van den Brand et al. The ASF+SDF Meta-Environment: a component-based language development environment. In *Proc. Compiler Construction*, LNCS, 2001.

[5] M.G.J. van den Brand, P. Klint, and J.J. Vinju. Term rewriting with traversal functions. Technical Report SEN-R0121, CWI, 2001.

[6] 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, 1997.

[7] M.G.J. van den Brand and J.J. Vinju. Rewriting with layout. In *Proc. First Int. Workshop on Rule-Based Programming (RULE'2000)*, September 2000.

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

[9] A. van Deursen and T. Kuipers. Rapid system understanding: Two Cobol case studies. In *Proc. 6th Int. Workshop on Program Comprehension*, pages 90–98, 1998.

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

[11] A. van Deursen and L. Moonen. Type inference for Cobol systems. In *Proc. 5th Working Conf. on Reverse Engineering*, pages 220–230, 1998.

[12] A. van Deursen and L. Moonen. An empirical study into Cobol type inferencing. *Science of Computer Programming*, 40(2–3):189–211, July 2001.

[13] M.J. Fyson and C. Boldyreff. Using application understanding to support impact analysis. *Software Maintenance: Research and Practice*, 10:93–110, 1998.

[14] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[15] J. Han. Supporting impact analysis and change propagation in software engineering environments. In *Proc. 8th Intl. Workshop on Software Technology and Engineering Practice (STEP'97)*, pages 172–182, July 1997.

[16] 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.

[17] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[18] J.P. Loyall and S.A. Mathisen. Using dependence analysis to support the software maintenance process. In *Proc. Int. Conf. on Software Maintenance*, pages 282–291, 1993.

[19] L. Moonen. Generating robust parsers using island grammars. In *Proc. 8th Working Conf. on Reverse Engineering*, pages 13–22, October 2001.

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

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

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

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