Building Documentation Generators

Arie van Deursen and Tobias Kuipers
http://www.cwi.nl/~{arie,kuipers}/
{arie,kuipers}@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
In order to maintain the consistency between sources and documentation, while at the same time providing documentation at the design level, it is necessary to generate documentation from sources in such a way that it can be integrated with hand-written documentation. In order to simplify the construction of documentation generators, we introduce *island grammars*, which only define those syntactic structures needed for (re)documentation purposes. We explain how they can be used to obtain various forms of documentation, such as data dependency diagrams for mainframe batch jobs. Moreover, we discuss how the derived information can be made available via a hypertext structure. We conclude with an industrial case study in which a 600,000 LOC COBOL legacy system is redocumented using the techniques presented in the paper.

1991 ACM Computing Classification System: D.2.2, D.2.5, D.2.7, D.3.4
Keywords and Phrases: Redocumentation, legacy systems, documentation generation, source code analysis, island grammars
Note: Work carried out under projects SEN1.1, Software Renovation, and SEN1.5, Domain Specific Languages. This report will appear in the Proceedings of the International Conference on Software Maintenance 1999 (ICSM'99).

1. Introduction

The documentation of a system is needed to understand that system at a certain level of abstraction, in a limited amount of time. It is needed, for instance, if a system is migrated or re-engineered. It can be used to map functional modification requests as expressed by end users onto technical modification requests, and to estimate the cost of such modifications. Finally, documentation will help in the process of outsourcing maintenance or when engineers that are new to the system need to learn about the system.

The source code of a system can be viewed as its most detailed level of documentation: All information is there, but usually we do not have enough time to comprehend all the details. Luckily, we do not usually need to know *all* the details. Instead, we would like to have enough information so that we can build a *mental model* of the system, and *zoom in* to the specific details we are interested in. The level of detail (or abstraction) we are interested in depends very much on what we intend to do with the system.

This flexibility should be reflected in the documentation, which, therefore, should adhere to four criteria:

1. Documentation should be available on different levels of abstraction.
2. Documentation users must be able to move smoothly from one level of abstraction to another, without loosing their position in the documentation (zooming in or zooming out).
3. The different levels of abstraction must be meaningful for the intended documentation users.
4. The documentation needs to be consistent with the source code at all times.

Unfortunately, these criteria are not without problems. Criterion 4 implies that documentation is generated from the source code. In practice this is seldomly done. Consequently, it is violated by many legacy systems, which are modified continuously without updating the accompanying technical documentation.

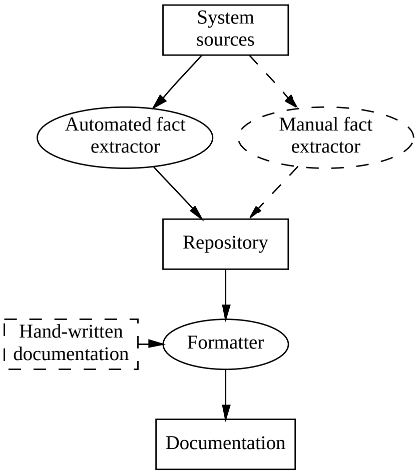


Figure 1: Deriving documentation from legacy sources. Solid lines indicate automatic processing, augmented with manually derived information indicated by dashed lines.

Criterion 3 makes documentation generation hard. Meaningful abstractions can benefit immensely from design information which is usually not present in the source code itself. Such information needs to be added manually to the documentation.

For new systems, mechanisms like literate programming [14] provide systematic ways of putting design information in the source code. For legacy systems this would involve a significant manual updating of program comments. Besides, design information is more often than not lost for legacy systems.

In this paper, we study ways in which we can update the documentation of legacy systems such that all four criteria are met. We propose a combination of manual and automatic (re)documentation. Whatever documentation can be generated from the sources is derived automatically. This then is combined with information provided by hand. Depending on the state of the system, and the knowledge about the system, either one of those activities can play the predominant role in the final documentation that is delivered. Figure 1 shows the architecture of the documentation generators that are built this way.

The remainder of this paper is organized as follows. In the next section, we introduce *island grammars*, the technology we use for extracting facts from a system's source code. In Section 3 we discuss what information should be contained in documentation, and how we can derive it from the legacy sources. In Section 4 we explain how the information extracted can be presented at what level of abstraction, using graph visualization and hypertext as primary tools. In Section 5 we describe a real-world Cobol legacy system, what its documentation problems were, and how we applied the techniques described in this paper to build a documentation generator for that system. We end the paper with related work, a summary of the main contributions, and suggestions for future work.

2. Source Code Analysis

In order to generate documentation from a system, we need to analyze the source code of that system. We have tried several analysis approaches. In this section we will discuss these approaches in detail. In later sections we discuss how we have used these approaches.

2.1 Lexical analysis

When generating documentation for a system, only a few constructs in the source code are of interest. After all, the documentation should be a useful abstraction of the system. The constructs of a language that are of interest very much depend on the type of documentation that should be generated. If these constructs have an easily recognizable lexical form, lexical analysis is an efficient way to find them. If, for instance, we are looking for files that are opened for reading in a Cobol source, we simply look for the string “open input” and take the word directly following that string as the file handle that has been opened.

The advantage of this approach is that we do not need to know the full syntax of the language we want to analyze. Another advantage is that lexical analysis is very efficient. This allows us to analyze large numbers of files in a short time, and also allows us to experiment with different lexical patterns: If a pattern does not yield the correct answer, the analysis can be easily changed and rerun.

The main disadvantage of lexical analysis is that it is (in general) not very precise, and that some language constructs are much harder to recognize lexically than others. For example, for the case study later discussed in this paper we need to find the files that were executed from a DCL program, the DEC job control language for VAX VMS. In DCL, we can look for the string "run", which is the DCL keyword for execution. If, on the other hand, we would want to know which files are executed from a Bourne shell script, we would need to specify all built-in functions of the Bourne shell language. There is no special keyword for execution in the shell, rather, it attempts to execute all words that are not built-in functions.

Strings such as "open input" and "run" obviously can occur in different contexts, and may mean completely different things in each context. These strings could occur in comment, for example, or inside a quoted string. Because we need to recognize different contexts in most cases, much of the original simplicity of the lexical pattern is gone. Furthermore, as long as we do not specify the full syntax of a language, there is the risk that we may have overlooked particular contexts in which a pattern can or cannot occur.

Most commonly used for lexical analysis are Unix tools such as `grep`, `awk`, and `perl`. Murphy and Notkin [17], describe LSME, a system which allows for the lexical specification of contexts of patterns, as well as the patterns themselves. For the analysis of Cobol, we have developed `recover` [8], which keeps track of the global structure of Cobol, and allows the user to specify patterns typically required in a program understanding context.

2.2 Syntactic Analysis

More precise analysis of source code can be achieved by taking the syntactic structure of the code into account, analyzing the abstract syntax tree instead of the individual source code lines. This makes the context in which a particular construct occurs explicitly available. Moreover, it abstracts from irrelevant details, such as layout and indentation.

Unfortunately, most legacy systems are written in languages for which parsers are not readily available. Developing a grammar from which to generate such a parser requires a significant investment. As an example, Van den Brand *et al.* [2] report a period of four months needed to develop a fairly complete Cobol grammar.

For program understanding and documentation purposes, however, only a handful of language constructs are needed, so it seems too much work to have to specify the full grammar of a legacy language. Therefore, we propose the use of "island grammars", in which certain constructs are parsed in full detail, whereas others are essentially ignored.

2.3 Island Grammars

An island grammar consists of (1) detailed productions for the language constructs we are specifically interested in (2) liberal productions catching all remaining constructs; and (3) a minimal set of general definitions covering the overall structure of a program.

As an example, suppose we have a simple language *L*. Programs in *L* consist of a list of one or more statements. For documentation generation purposes we are only interested in one statement, the "SELECT" statement. The definition of the island grammar is in Figure 2. We use the grammar definition language SDF2 [20] for our definition.[1] We can distinguish the following groups of productions:

- The definition of the statement of interest is on line (3), defining a a statement to be produced by the keywords "SELECT", a FileHandle, "ASSIGN", "TO", a FileName, a possibly empty list of Options, terminated with a "." character. Productions (4–9) define the details of the other non-terminals.

[1] Please note that productions in SDF2 are reversed with respect to languages like BNF. On the left-hand side of the arrow is the non-terminal symbol that is produced by the symbols on the right-hand side of the arrow.

syntax

$$\begin{array}{rcll}
\text{Stat}+ & \rightarrow & \text{Program} & (2.1)\\
\sim[\backslash.]+\ \text{”}\backslash.\text{”} & \rightarrow & \text{Stat} & (2.2)\\
\text{”}\texttt{SELECT}\text{” FileHandle ”}\texttt{ASSIGN}\text{”} & & & \\
\text{”}\texttt{TO}\text{” FileName Option}*\ \text{”.”} & \rightarrow & \text{Stat} & (2.3)\\
\text{Name}+\ (\text{”}\texttt{IS}\text{”})?\ \text{Value} & \rightarrow & \text{Option} & (2.4)\\
\text{Id} & \rightarrow & \text{FileName} & (2.5)\\
\text{Id} & \rightarrow & \text{Name} & (2.6)\\
\text{Id} & \rightarrow & \text{Value} & (2.7)\\
\text{Id} & \rightarrow & \text{FileHandle} & (2.8)\\
[\texttt{A}-\texttt{Z}][\texttt{A}-\texttt{Z0}-\texttt{9}\backslash\_]* & \rightarrow & \text{Id} & (2.9)
\end{array}$$

priorities

$$\begin{array}{rcll}
\text{”}\texttt{SELECT}\text{” FileHandle ”}\texttt{ASSIGN}\text{”} & & & \\
\text{”}\texttt{TO}\text{” FileName Option}*\ \text{”.”} & \rightarrow & \text{Stat} & >\\
\sim[\backslash.]+\ \text{”}\backslash.\text{”} & \rightarrow & \text{Stat} &
\end{array}$$

Figure 2: An example island grammar

- The liberal production catching all remaining constructs is on line (2), defined as any character that is not a "." (the tilde negates the character class containing the period), followed by a period.

  Obviously, this grammar is ambiguous, because a "SELECT" statement can be produced by both productions (2) and (3). To resolve this, Figure 2 defines **priorities** preferring production (2) to (3).

- Line (1) defines the overall structure of a program, which is defined as a list of statements.

The reason for using this grammar development technique, is that we significantly reduce the grammar development time. Another advantage is that the parse tree that is returned by the parser only contains the relevant information. We do not have to weed through dozens of complicated structures to get to the information we look for.

By far the biggest advantage is the flexibility of the technique. Although some legacy languages have a proper language definition, we have yet to see a legacy system that does not use compiler specific extensions, or locally developed constructs. Furthermore, most parsers for legacy systems are quite liberal in checking their input, so although a program is not syntactically correct according to the language definition, it does parse, compile, and run. Using our grammar development technique, we can either ignore these specifics (by writing a *catch-all* production such as (2) above), or add a production particular to a certain extension of the legacy system at hand.

In principle, island grammars can be used in combination with any parser generator, the best known ones being Yacc and Bison. We benefited from the use of SDF2, which has a number of attractive characteristics.

First, SDF2 is based on *scannerless* parsing, in which the distinction between lexical scanning and parsing has disappeared. Hence, the user of SDF2 is not restricted to regular expressions for defining lexical tokens. Moreover, explicit lexical disambiguation is permitted in the formalism.

Second, parsers for SDF2 are implemented using *generalized* LR parsing [19], which accepts arbitrary context-free grammars, not just LALR grammars accepted by Yacc and Bison. This avoids the notorious shift reduce conflicts inherent to the use of LALR grammars. A priority mechanism can be used to deal with ambiguities that may arise due to the use of arbitrary context-free grammars.

Last but not least, because SDF2 is a *modular* syntax definition language, we can specify an island grammar

Level	Documentation
system	overall purpose, list of subsystems
subsystem	purpose, list of modules, batch jobs, databases, screens, ...
batch job	programs started, databases accessed, frequency, ...
program	behavior, programs called, databases read or written, invoked by, parameters, ...
section	functionality, external calls, sections performed, conditions tested, variables used, ...

Figure 3: Cobol system hierarchy, with associated documentation requirements

in different modules. This way, for each analysis we can have a different grammar that is an extension of a common core language. This helps to keep the grammars as small and concise as possible. Consider the island grammar developed above. Here, productions (1), (2), and (5–9) can be viewed as being part of the core of language *L*. These can be put in a separate module. Then, the only productions needed for our "SELECT" analysis are productions (3–4), and the priority rule, which should be defined in a different module.

2.4 Parse Tree Analysis

The parser generated from the grammar in the previous section will return parse trees that can be easily analysed. The parse trees are encoded in *aterm* format [1]. This parse tree can be read in by a Java framework we wrote, thus giving access to the parse tree as a Java object. The framework implements the *visitor* design pattern [11], via a visitor class that can be specialized to perform a particular analysis on the tree. This is simplified by the fact that the Java framework has full knowledge of the island grammar that has been specified, and contains methods for matching patterns of productions in the grammar to corresponding nodes in the tree.

The analysis results that are of interest can be written to a repository, and from there they can be combined, queried and used in the rest of the documentation generation process. All extractions described in Section 3 were performed using this Java parse tree analysis framework. The data extracted were put in a repository. The presentations described in Section 4 were then generated from that repository.

This way of analyzing source code is similar in concept to a number of other systems, e.g. CIAO [5], in the sense that there is a chain of analysis, filter, and presentation events. In our approach, however, we start filtering the data during the first (analysis) phase, because we only deal with those language constructs defined in the island grammar.

3. Extracting Documentation

In this section, we will discuss the sort of information that should be contained in software documentation, and how this information can be identified in the legacy sources.

3.1 Manual versus Automated Extraction

Given the choice between manual or automatic extraction of information from source code automatic extraction (for example using island grammars) is the preferred option: it is consistent with the actual source code, and can be easily maintained by automatic regeneration.

If generation is not feasible, the facts needed to construct the documentation can be provided by hand. This may take the form of a list of programs and a one or two line description of their functionality. Whenever documentation is generated, data from this list is included as well. Moreover, automatic checks as to whether *all* programs are indeed contained in the lists can be made whenever documentation is regenerated, encouraging programmers to keep the descriptions up to date. The integration of manual and automated extraction is illustrated in Figure 1, which also shows how additional forms of externally available documentation can be included in the resulting documentation.

3.2 System Decomposition

We can decompose a large software system into several layers of abstraction, ranging from individual procedures up to the overall system. At each level, we need documentation, helping us to answer questions about the purpose (why?) of a component, the subcomponents it consists of (part-of relationships), the components it needs to perform its tasks (uses relationships), the way in which it performs its tasks (how?), the way in which the component can be activated (usage conditions), the system requirements the component corresponds to, etc.

The actual splitting in layers of abstraction, and the corresponding documentation requirements, will differ from system to system. The hierarchy with associated documentation requirements we use for Cobol systems is shown in Figure 3.

3.3 Aggregation and Use Relations

The parts-of and uses relationships discussed in the previous section can be easily derived from the source code. In general, it is relatively straightforward to extract facts about calls, database usage, screens used, etc.

A factor complicating this extraction is that many legacy systems use non-standard conventions for, e.g., calling or database access. We have seen calling conventions in which all calls were redirected via an assembly utility, and database access conventions hiding all SQL operations via a set of Cobol modules. The flexibility of island parsing makes it particularly easy to tailor the extractors to such conventions.

3.4 System and Subsystem Partitioning

At the system level, the directory structure of program files or the naming conventions used usually provide a candidate partitioning into subsystems. If these are absent, or perceived as inadequate, we use automatic subsystem classification techniques to arrive at a better partitioning [15, 9]. Such alternatives can then be added to the documentation, helping the user to see component relations that do not immediately follow from the actual partitioning.

In addition to the decomposition of the overall system, short descriptions of the individual subsystems as well as of the overall behavior are needed in the documentation. In many cases, such top level documentation may already be available, in which case it can be included in the documentation generation process. If it is not, a description of the various subsystems should be added by hand.

3.5 Program Descriptions

In many systems, coding standards are such that each program or batch job starts with a *comment prologue*, explaining the purpose of this component, and its interaction with other components. If available, such a comment prologue is a very useful documentation ingredient which can be automatically extracted from the source. Observe that it is generally *not* a good idea to extract *all* comment lines from a program's source into its documentation: many comment lines are tightly connected to specific statements, and meaningless in isolation. Moreover, in many cases obsolete pieces of code have been "commented out", which clearly should not appear in system documentation.

3.6 Section Descriptions

For the sections (local procedures) of a Cobol program, it is usually not as easy to extract a description as it is for Cobol programs starting with a comment prologue. On the positive side, however, section names are generally descriptive and meaningful, explaining the purpose of the section. This is unlike Cobol *program* names, which generally have a letter/number combination as name indicating which subsystem it is part of, not what its purpose is.

Since we encountered an actual need for the documentation of sections that consisted of more than just the name, but at the same time was more abstract than simply the complete source code, we decided to search for ways in which to select the essential statements from a section. In terms of the theory of program comprehension as proposed by Brooks [3], we try to select those statements that act as *beacons* for certain understanding tasks, such as finding out under what condition a certain piece of code is being executed.

Statements we include in such section descriptions are the *conditions* checked in control statements, calls to external programs, database access operations, calls to other sections, statements containing arithmetic compu-

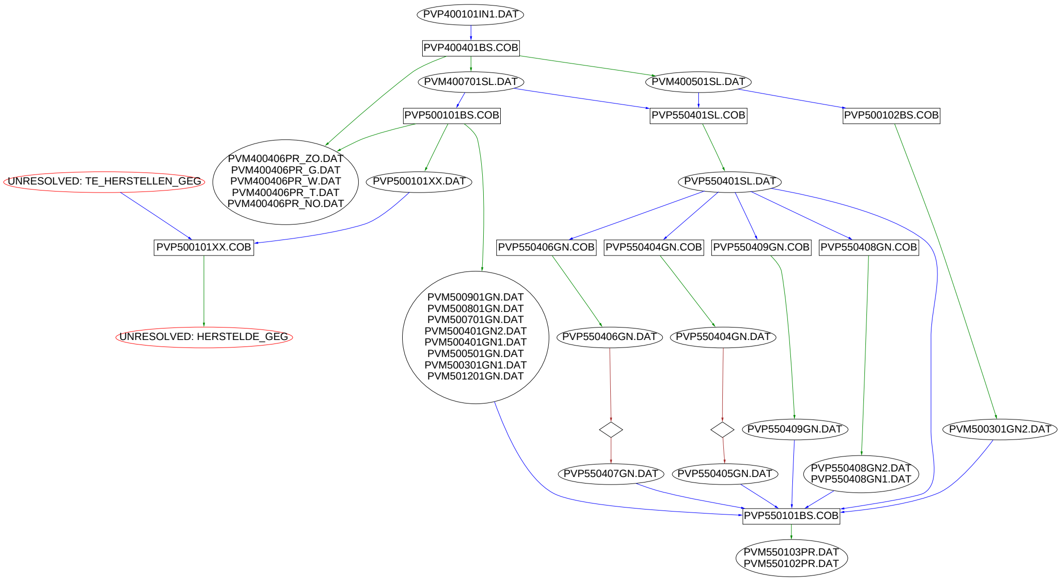


Figure 4: An example of the visualization of data dependencies in a DCL job

tations, and comment lines. This explicitly excludes the frequently occurring MOVE statement, which in Cobol is an assignment from one variable to another. As it does nothing “interesting” (no computation) leaving it out of the documentation directs the reader’s attention to those statements that do perform some task of interest.

Following this strategy, the length of a summarized section is about one third of the length of its full source code. To make the summary as comprehensible as possible, we reuse indentation of the original source code, giving the reader a sense of the nesting level.

3.7 Batch Job Dependencies

For a Cobol mainframe application, the batch jobs determine which programs are started every day or night, and how datafiles used for communication purposes are sorted, renamed, etc. In many cases, the batch jobs are the least understood components of a Cobol system, in spite of their importance for the daily data processing. Documentation should help in understanding such batch jobs. We have experimented with visualizing the data dependencies in such batch jobs.

Finding the data dependencies for a batch job is a three step process. First, we identify the Cobol programs executed in a batch job. Second, we analyze these Cobol programs, determining which data files are read and which ones are written. Third, we return to the batch files, to see whether these data files occur in them, for example for sorting or renaming.

Recognizing these dependencies involves two island grammars: one for the job control language, finding the execution, sort and renaming statements, and one for Cobol, identifying the data file manipulation statements.

Once the data dependencies are found, they can be visualized. The visualization of an example batch job is shown in Figure 4. The resulting graph only shows the functional dependencies: Dynamic dependencies, such as the order of execution, are not explicitly visible. Also observe that in some cases, it will be impossible to determine the name of a data file, because it is determined at run time. Special nodes in the graph are used to mark such file names.

4. Presenting Documentation

Once we have decided which information to put into the documentation, we can decide how to *present* that information to the user. Hypertext has been proposed as a natural way of presenting software documentation [4, 18] as the hyperlinks can be used to represent, for example, part of and uses relationships between between the documented components.

The most natural way of organizing all the information derived is to follow the system hierarchy, producing essentially one page per component. For Cobol this would result in pages corresponding to the full system, subsystems, programs, batch jobs, and sections, following the decomposition of Figure 3.

If a user knows what programs he wants to read about, finding an initial node to start browsing is simple. In many cases, however, there may not be such a straightforward starting point. Therefore, we provide various indexes with entry points to the hypertext nodes, such as:

- Alphabetic index of program names;
- Keyword search on documentation contents;
- Graphs representing use relationships. In particular, navigating through a call graph may help to find execution starting points or modules frequently used. We have used the graph drawing package `dot` [12] to integrate clickable image maps for various call graphs and data-dependency graphs into generated documentation. In order to prevent visual cluttering of graphs, we have applied *node concentration* on them, as can be seen in Figure 4.
- Hand-written index files, establishing links between requirements and source code elements.

Many presentation issues are not specific to software documentation. By using a standard format such as HTML, the generated documentation can benefit from various future developments of the Web, such as search engines, page clustering based on lexical affinity, link generation from textual documentation files, the use of XML to establish a better separation content from presentation, etc.

5. Business Case

We have used all the techniques and ideas discussed in this paper in a commercial project aiming at redocumenting a Cobol legacy system. In this section, we describe our findings.

5.1 Background

PensionFund is a system for keeping track of pension rights of a specific group of people in the Netherlands. It consists of approximately 500 Cobol programs, 500 copybooks, and 150 DEC DCL batch jobs, totaling over 600,000 lines of code. The main tasks of the system are processing pension contributions and pension claims.

Several years after the initial delivery, the organization responsible for *PensionFund* decided to outsource all maintenance activities to a division of Dutch software house Roccade, specializing in software management and maintenance. In order to make a realistic estimate of the anticipated maintenance costs involved before accepting maintenance commitments, Roccade performed a *system scan* in which a number of key factors affecting maintainability are estimated.

One of the outcomes of the scan was that the documentation for *PensionFund* was not adequate. In fact, documentation was not kept up to date: for example, although in 1998 a number of major *PensionFund* modifications were implemented, the documentation was never updated accordingly. Very little documentation maintenance had been performed, although the need for documentation grew as more and more programmers who had participated in the original design of *PensionFund* moved to other projects.

The lack of proper documentation resulted in:

- A growing backlog of major and urgent modification requests, which by early 1999 had risen to 12.
- Difficulty in carrying out adequate year 2000 tests, since the documentation did not help to identify the sources of errors encountered during testing.
- Difficulty in mapping modification requests, phrased in terms of desired *functionality* modifications, onto changes to be made in actual programs.
- Difficulty in splitting the large number of daily batch jobs into clusters that could be run independently and in parallel, which was becoming necessary as the increasing number and size of the batch jobs caused the required daily compute time to grow towards the upper limit of 24 hours.

5.2 Documentation Wishes

To remedy these *PensionFund* problems, a redocumentation project was planned. The plan was to compose a number of MS-Word documents, one per program, containing:

- A short description
- Calls made (from other Cobol programs or batch jobs) to this program, and calls made from this program;
- Database entities as well as flat files read and written;
- Dataflow diagram;
- Description of functionality in pseudo-code.

Apart from the per program documentation, per batch file one dataflow chart was planned for. Management was willing to make a significant investment to realize this documentation.

Initially, the idea was to write this documentation by hand. This has the advantage that documentation writers can take advantage of their domain or system knowledge in order to provide the most meaningful documentation. Unfortunately, hand-written documentation is very costly and error prone. Because it is not a job many people like to do, it is difficult to find skilled documentation writers.

Therefore, it was decided to try to *generate* the documentation automatically. This has the advantages that it is cheap (the tools do the job), accurate, complete, and repeatable. If necessary, it was argued, it could be extended with manually derived additional information.

5.3 Derived Documentation

The contents requirements of the *PensionFund* documentation corresponds to the wishes discussed the previous section. The specific information derived per program is shown in Figure 5. Arriving at this list and determining the most desirable way of presentation was an interactive process, in which a group of five *PensionFund* maintenance programmers was involved.

The fact extraction phase mainly involved finding the structure of PERFORM, CALL, and database access statements, and was implemented using island parsing. For those extraction steps for which a line by line scan was sufficient (for example, Cobol comment extraction), or for the ones which required the original layout and indentation (summarizing sections) lexical analysis was implemented using Perl.

The result of the fact extraction was a set of relations, which were combined into the required relations per program using Unix utilities such as join and AWK. The final production of HTML code from the resulting relation files was written using Perl.

All the documentation per program could be generated automatically. Even the the two-line description per program could be generated, as this was an easily recognizable part of the prologue comment. Had this not been the case, this would have required a manual step. As top level indices we generated alphabetic lists, lists per subsystem, and clickable call graphs. Moreover, we composed one index manually, grouping the programs based on their functionality.

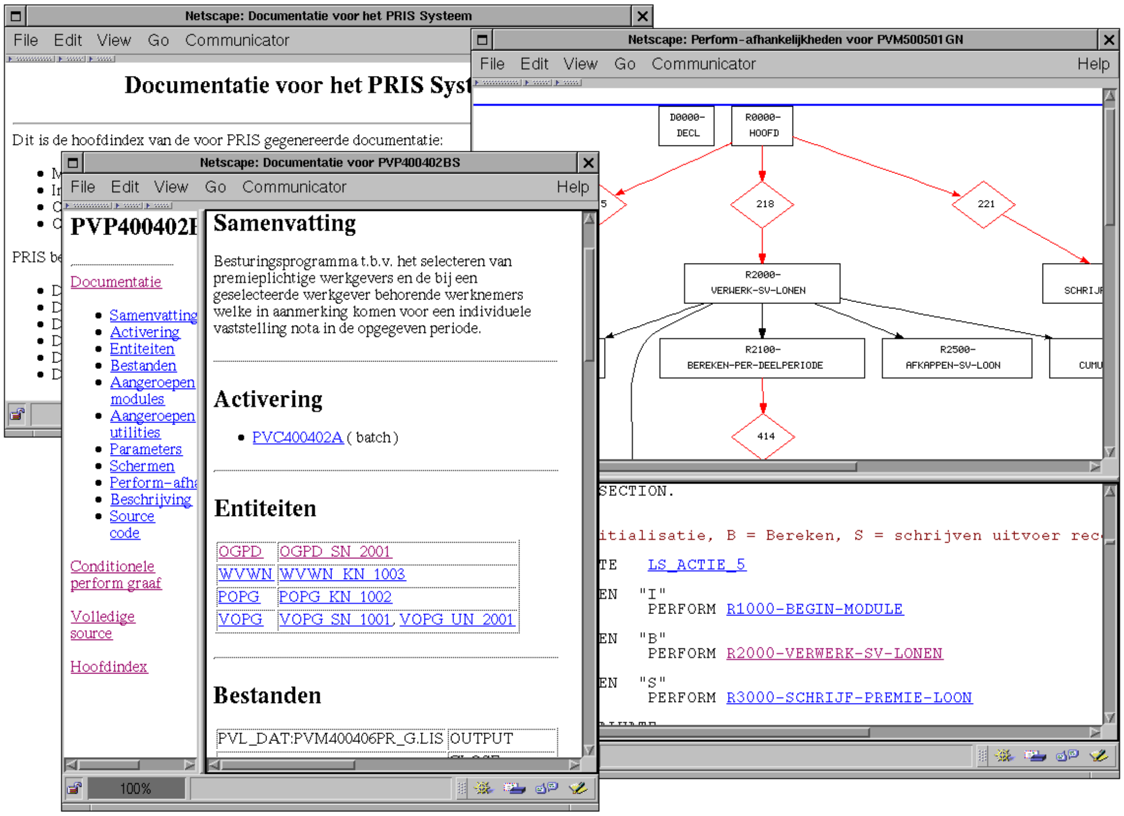
As a separate top level view, we used the data dependency visualization we derived from the batch files. For each DCL file, we used the techniques described in Section 2 to find all Cobol programs that are executed. We then analyzed these Cobol programs to find the data files they read and write to. Using static analysis it is impossible to find all the data file names, because, in this system, some file names were obtained dynamically. This occurs especially in error conditions, where the name of the file to write the error data to is somehow related to the kind of error. The files we could not find names for are only a small fraction of all data files. In order to visualize these unnamed files at a later stage, we introduced special filenames for these files. In Figure 4 these unresolved filenames can be seen on the left side, and are clearly marked: “unresolved”.

The list of data files was then matched against the DCL files again, to see whether the data was manipulated there. In the *PensionFund* system, we looked at the `sort` statement, which takes one file and a number of sort parameters, and writes to a different file. They are visualized as diamonds in the figure.

An example browsing session trough the generated documentation is shown in Figure 6. A typical session would be a maintenance programmer trying to find out why a particular batch job did not work as expected.

Header	Content
Summary	Name, lines of code, two-line description
Activation	Batch jobs or Cobol programs called by
Parameters	List of formal parameters
Data	Databases and flat files read or written
Screens	List of screens sent or received
Calls	Modules and utilities called
Overview	Clickable conditional perform graph
Sections	Clickable outline for each section

Figure 5: Contents of the HTML document derived for each *PensionFund* program.


Netscape: Documentatie voor het PRIS Systeem
File Edit View Go Communicator
Netscape: Perform-afhankelijkheden voor PVM500501GN
File Edit View Go Communicator Help
D0000-DECL
R0000-HOOFD
5
218
221
R2000-VERWERK-SV-LONEN
SCHRIJF
R2100-BEREKEN-PER-DEELPERIODE
R2500-AFKAPPEN-SV-LOON
CUMU
414
Netscape: Documentatie voor PVP400402BS
File Edit View Go Communicator Help
100%

Documentatie voor het PRIS Syst

Dit is de hoofdindex van de voor PRIS gegenereerde documentatie:

PVP400402I

Documentatie

- Samenvatting
- Activering
- Entiteiten
- Bestanden
- Aangeroepen modules
- Aangeroepen utilities
- Parameters
- Schermen
- Perform-afh
- Beschrijving
- Source code

Conditionele perform graaf

Volledige source

Hoofdindex

Samenvatting

Besturingsprogramma t.b.v. het selecteren van premieplichtige werkgevers en de bij een geselecteerde werkgever behorende werknemers welke in aanmerking komen voor een individuele vaststelling nota in de opgegeven periode.

Activering

- PVC400402A (batch)

Entiteiten

OGPD	OGPD_SN_2001
WVWN	WVWN_KN_1003
POPG	POPG_KN_1002
VOPG	VOPG_SN_1001, VOPG_UN_2001

Bestanden

PVL_DAT:PVM400406PR_G.LIS	OUTPUT

```
SECTION.

itialisatie, B = Bereken, S = schrijven uitvoer rec
TE    LS_ACTIE_5
EN    "I"
       PERFORM R1000-BEGIN-MODULE
EN    "B"
       PERFORM R2000-VERWERK-SV-LONEN
EN    "S"
       PERFORM R3000-SCHRIJF-PREMIE-LOON
```

Figure 6: Browsing through the documentation generated for *PensionFund*.

He starts browsing the visualization of the data dependencies in the batch job, follows the links to a specific program, reads the purpose of that program, searches the perform graph for relevant sections, ending in the section responsible for the incorrect system behavior.

5.4 Evaluation

As we have demonstrated in the previous section, the documentation generator we have built for *PensionFund* exactly fulfills the wishes the *PensionFund* owners had. Furthermore, the documentation that is generated adheres to the four criteria mentioned in first section of this paper. Compared to the initial plan of manually deriving all documentation, significant cost savings were achieved by employing documentation generators, even if the time needed for configuring the documentation generators is taken into account.

A question of interest is whether this approach is applicable to other legacy systems as well. Our approach takes the *good* properties of a system in to account. For *PensionFund*, these are the systematic coding style which meant that certain properties (such as program descriptions) were automatically derivable from the source. Furthermore, the programs were relatively short, which made them a natural starting point for documentation generation. Another result is that the (conditional) perform graphs are not too big, making them easily comprehensible. Finally, the fact that the sections were relatively short made the section summaries feasible.

Although other systems may not share the desirable properties of *PensionFund*, they usually have some of these and possibly other strong points. Apart from the program description, all other documentation can be generated for any (Cobol) system. Program descriptions could then be added by hand once, such that subsequent generation steps have this information available. It is part of our future work to see how the generation of the documentation as described here is useful in other systems. We may decide for other systems, that certain levels of documentation are of no use, and new ones are more natural.

We believe the techniques described in this paper are flexible enough to enable us to build different types of documentation generators for different types of systems rather easily.

6. Concluding Remarks

Related Work Chikofski and Cross define *redocumentation* as the creation of a semantically equivalent representation of a software system within the same level of abstraction. Common tools include pretty printers, diagram generators, and cross-reference listing generators [6]. Landis *et al.* discuss various documentation methodologies, such as Nassi Schneiderman charts, flow charts and Jackson diagrams [16].

Wong *et al.* emphasize *structural* redocumentation, which, as opposed to documentation *in-the-small*, deals with understanding architectural aspects of software. They use Rigi for the extraction, querying, and presentation, using a *graph editor* for manipulating program representations. Several *views* of the legacy system can be browsed using the editor. Our approach also focuses on the structural aspects of documentation. Rather than using a dedicated graph editor, we use standard HTML browsers for viewing the documentation. We determine the required views in advance, via discussion with the team of maintenance programmers.

The software bookshelf [10] is an IBM initiative building upon the Rigi experience. In the bookshelf metaphor, three roles are distinguished: the *builder* constructs (extraction) tools; the *librarian* populates repository with meaningful information using the building tools or other (manual) ways, and the *patron* is the end user of the bookshelf. For the building phase, the parsing is like our island approach, in that only constructs of interest are recognized. The parsers are written in Emacs macros, without using an explicit grammar. The parsing code directly emits the HTML code.

Several papers report on the use of hypertext for the purpose of documenting software [4, 18, 7]. Of these, [7] follows the *literate programming* [14] approach as also used in, for example, Javadoc, enabling the programmer to control the generation of HTML by manually adding dedicated comment tags.

The need for flexible source code extraction tools was also recognized by, for example, LSME [17], as discussed in Section 2.1. Another approach of interest is TAWK [13], which uses an AWK like language to match abstract syntax trees.

Contributions In this paper, we have described our contributions to the field of documentation generation. Specific to our approach are:

- The systematic integration of manual documentation writing with automated documentation generation in a redocumentation setting.
- The integration of different levels of documentation abstractness, and the smooth transition between the different levels of documentation.
- The island grammar approach to software fact extraction

- A method for building documentation generators for systems in the Cobol domain.
- The automatic visualization of data-dependencies in mainframe batch jobs.
- The application of the contributions listed above in a commercial environment.

We have shown how we can build documentation generators which adhere to at least the first three criteria from the introduction. The fourth criterion (documentation needs to be consistent with the source code) can only be achieved by eliminating the need for manual documentation. By combining automatically derived documentation and manually derived documentation, and by keeping the input for the two well separated, our documentation generators only need little human input to adhere to all four criteria.

Future Work At the time of writing, we are finalizing the *PensionFund* case study. Moreover, we are in the process of initiating other commercial redocumentation projects, which will help us to identify additional documentation needs and new ways of presenting the data extracted from the sources.

On the extraction side, we plan to elaborate the ideas underlying island grammars. In particular, we will take a close look at the best way of expressing the required analysis of the abstract syntax tree.

Acknowledgments We would like to thank Lars van Dinter from Roccade Finance for his role in the *PensionFund* case study, and for many pleasant discussions.

References

1. M. G. J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.
2. M. G. J. van den Brand, M. P. A. Sellink, and C. Verhoef. Obtaining a Cobol grammmar from legacy code for reengineering purposes. In *Proceedings of the 2nd international workshop on the theory and practice of algebraic specifications*, Electronic workshops in computing. Springer Verlag, 1997.
3. R. Brooks. Towards a theory of the comprehension of computer programs. *Int. Journal of Man-Machine Studies*, 18:543–554, 1983.
4. P. Brown. Integrated hypertext and program understanding tools. *IBM Systems Journal*, 30(3):363–392, 1991.
5. Y.-F. Chen, G. S. Fowler, E. Koutsofios, and R. S. Wallach. Ciao: A graphical navigator for software and document repositories. In G. Caldiera and K. Bennett, editors, *Int. Conf. on Software Maintenance; ICSM 95*, pages 66–75. IEEE Computer Society, 1995.
6. E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.
7. Ch. de Oliveira Braga, A. von Staa, and J. C. S. do Prado Leite. Documentu: A flexible architecture for documentation production based on a reverse-engineering strategy. *Journal of Software Maintenance*, 10:279–303, 1998.
8. A. van Deursen and T. Kuipers. Rapid system understanding: Two COBOL case studies. In S. Tilley and G. Visaggio, editors, *Sixth International Workshop on Program Comprehension; IWPC'98*, pages 90–98. IEEE Computer Society, 1998.
9. A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-99*, pages 246–255. ACM, 1999.
10. P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K.Kontogiannis, H. A. Müller, J. Mylopoulos, and S. G. Perelgut. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
12. E. R. Gansner, E. Koutsofios, S. North, and K-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

13. W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Fourth Workshop on Program Comprehension; IWPC'96*. IEEE Computer Society, 1996.

14. D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

15. A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, pages 211–231, March 1997.

16. L. D. Landis, P. M. Hyland, A. L. Gilbert, and A. J. Fine. Documentation in a software maintenance environment. In *Proc. Conference on Software Maintenance*, pages 66–73. IEEE Computer Society, 1988.

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

18. Vaclav Rajlich. Incremental redocumentation with hypertext. In *1st Euromicro Working Conference on Software Maintenance and Reengineering CSMR 97*. IEEE Computer Society Press, 1997.

19. E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, University of Amsterdam, Programming Research Group, 1997.

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