A Repository for a CARE Environment

Toni A. Bünter

Technical Report CUCS-018-93
COLUMBIA UNIVERSITY

Abstract

Repositories in CASE hold information about the development process and the structure of developing software. The migration or reuse of CASE repositories for CARE (Computer Aided Re-Engineering) is not adequate for the reengineering process. The main reasons for its inadequacy are the emptyness of such repositories, and the nature of the process itself. In the following report we will define a CARE architecture, from the reengineering point of view, and derive a structure of a repository appropriate to the reengineering process.

Contents

1 Introduction

Repositories in CASE environment store meta information about the engineering phases of a software project. This information is not only relevant during the production of a software system, but can also be used for documentation of the components of the software system. The availability of production information enables more effective reuse of source code or design documents, especially for further reengineering activities. Furthermore, CASE tools and environments are often seen as equally useful tools for reengineering and reuse of released software systems. CASE tools can also provide support for reengineering software systems produced with CASE. But as the CASE decade has just started, a tremendous amount of source code without formal design documentation and without machine loadable repository data can not enter any CASE environment. For the most part, reengineering has to be done with source code files which are in a poorly structered language, and without the presence of meta data. This lack of meta data is a major shortcoming of using CASE systems. Additionally, the difference between the engineering and reengineering processes necessitates a different overall achitecture of CARE systems than CASE systems.

In this paper we will give an architecture for a CARE environment charged with the task of reengineering a source code given in a poorly structered language, e.g. COBOL. According to the reengineering process, we draw a CARE system architecture with the repository as the central feature. The repository is defined in terms of the relational database technology. It turned out that using the entity relationship approach for representing source code worked comfortably, and rendered the repository easy to extend.

For the purpuse of simplicity and independence we will define an intermediate language (COBAPA) comprising elements from COBOL, Basic and Pascal. COBAPA is strong enough to cover the main paradigms of these languages, with the exception of pointer-like structures. (Pointer-like structures cause no conceptual obstacles and will be dealt with in further discussions.) Appendix A contains a shortened definition of COBAPA. Appendix B shows a program example in COBAPA, with its entries in the stated repository.

The main difference between CARE systems, including those currently in development (as [10]), and our approach is the complete mapping of the source code to the repository's relational structure. As far as we know there is no other comparable publication with the same paradigms and assumptions as to the repository structure.

2 Reengineering

Reengineering is a growing discipline within software engineering. It is commonly defined as two successive activities: *reverse engineering* and *forward engineering*. Reverse engineering is the recreation of the design and specifications of given source code. It comprises analysis, restructuring, decomposition and modularisation. Its products are flow diagrams, a module hierarchy, module descriptions, specifications, glossaries and data dictionaries, depending on the methods used. Forward engineering executes the desirable changes and enhancements on recovered specifications. Reusing the generated design and the renewed specifications yield a new design. Finally, new source code is produced with the reuse and/or enhancement of the given components.

3 A CARE Architecture

When reengineering source code, CASE tools can only be applied if the given source code was produced by CASE tools, and if the relevant production documents are available. However, these conditions are not usually present. Therefore thinking about the CARE process for reengineering source code given in a poorly structured language leads to a different architecture than the one used for a CASE environment.

Our proposed CARE architecture consists of three units: the *source code configuration management, the repository*, and different *program views.* Figure 9 illustrates the components of the architecture and the data flows.

The configuration management is responsible for the management and storage of different versions of source code and meta information in the form of files. These files will be read as the repository is being filled. After a working session the generator produces source code and meta information files, which are then stored and managed by the configuration management.

The central part of this CARE architecture is the *repository.* It contains all information about the source code components, the design, the specifications and the reengineering process. The central part of the repository is the *mimimal repository*, a representation of the source code in which semantics are emphasized. The minimal repository is comprehensive enough to be functionally equivalent to the original source code. More exactly spoken: *We proved that a computationally equivalent source code can be generated using the repository entries. The derived source code differs only in syntactically equivalent structures (e.g. repeat-for, if-case) and in the sequence of the instructions of the sequencial program blocks.*

The *program views* represent the user interfaces built for the reengineering programmer. They provide tailored views of the software system, in order to fullfill a given task in a predefined way. They vary over different representations (e.g.: textual, graphic) and different abstraction levels (e.g.: source code, data and control flow, specification languages). The basic structure of a program view consists of three parts (Figure 9):

1. *Activity Menu.* The commands enabling the user to navigate, analyze, edit, or change parts of the software system.

2. *Visualisation.* The visual representation of the focus initiated by the action menu.

3. *Change Constraints.* Controlling the activities and preserving change policies.

It is the responsibility of the management to provide tailored program views for the reengineering programmer. The change constraints allow the management to restrict the activities provided by the activity menu in order to conserve the software's quality.

The program views interact permanently with the repository. Their relation to the repository is one of client/server. The program views send two kinds of information to the repository server:

- *Information requests*

- *Update information*

The repository provides the information which a program view is requesting. During a working session with a program view, two types of information will be produced by the reengineering activities:

- repository entry information (adding comments, manual modularisation, extracting and defining design items).
- change information (design changes, structuring, replacement of modules, change of source code instruction).

The repository has to be given this information immediately, and retain it permanently. It has to manage update problems which occur due to the activation of more than one program view.

4 A CARE Repository

In this section we will look more closely at our definition of a CARE repository. We will compare it to common definitions, and argue for our extention of it.

4.1 The Repository

Systems that handle meta data about objects and their relations have been in use for more than two decades. With the advent of *data dictionaries*, which provided information about the logical and/or physical structure of data, the term repository came into use. A repository not only holds information about the data of an application, it also provides information about the application itself and information about the software production process. IBM defines a repository as an overall information storage that concerns management and the control of an enterprise:

> ...a repository is a place for storing information about items and activities of importance to your enterprise; it is an organized, shared collection of information that supports business and data processing activities. (page7, in [2])

Our approach adds a condition of completness to the former definition:
The Repository accumulates enough information about *the structure of the modules, the data and flow dependences and the expressions that the whole functionality can be derived from the repository entries.*
The important consequence of this definition is the independence of the source code itself; it can be generated from the repository entries. This changes the engineering and reengineering processes. Instead of analysing and changing the ASCII files of the source code, the reengineering programmer executes well defined, supervised queries and transactions on the repository. This makes it possible to control and determine changes without causing unexpected side effects.

4.2 Definition of the Repository

The structures we support in the repository definition are choosen from well known structures, each representing one aspect of source code. Below we give a list of useful references and literature:

- *Syntaxtree*
- *Control and Dataflow*
- *Cross Reference List* ([3])
- *Program Dependence Graph* ([5], [6])
- *Module Interconnection Language* ([7])

We divide the repository database into six levels, and define the primary entities for each level. The number of entities is extendible, as we will show in section 5. All attributes without an asterix (*) belong to the *minimal repository*. In [1] we proved that the minimal repository satisfies the additional condition of our repository definiton. We built the repository concerning our wide range language COBAPA (cf. Appendix A). Appendix B shows a program example in COBAPA, and all repository entries.

1. **Module Interconnection Level**

   The module interconnection level consists of programs, proceduree s, functions and their value and variable parameters. We define a relation with six attributes.

module_id	sort	p_list	p_dependence	ref_list*	call_list*

   The three possible values of the attribute sort are `main`, `procedure` and `function`. p_list is a list of the input/output parameters. p_dependence is a term that shows the data dependences between the parameters. ref_list is the list of all variables that are used in this module. The call_list consists of modules (resp. their identification numbers) which call the module (module_id). For the identification key module_id it is recommended to use the name provided by the source code.

   The p_dependence, the ref_list and the call_list provide resource information about the modules and their interconnections. This information is used for the overall data and control flow of the software system.

2. **The Control Flow Level**

   Every program, procedure and function consists of *sequential program blocks* connected by logical predicates which control the flow of the program blocks. We define a relation with four attributes.

module_id	program block_id	program block_Id'	case

   The relation represents the control flow from program block_id to the program block_id' when the condition value case holds. There is no additional information about the source code. The entries on this level provide the structure of the well known control flow charts.

3. **The Progam Block Level**

The program block level focuses on the instructions (assignments and jump predicates), which are part of a sequentional block. We define a relation with five attributes.

module_id	program block_id	instruction_id	instruction_id'	variable_id

An entry in this relation is an edge in a labeled *ref-def graph*. Let a and b be two instructions in the same sequential block, and let v be a variable. If a defines a new value for v with no new definition of v between a and b, then there exists an edge (a, b, v) in the ref-def graph. For every variable there exists two additional entries marking the beginning and end of the variable usage:

- (in, b, v): The prior definiton of v is in a preceding program block.
- (a, out, v): The next usage of v is in a following program block.

This representation is syntactically independent. If two program blocks consist of the same set of instructions and are computationally equivalent, then they have the same entries on the program block level.

4. **The Instruction Level**

The instruction level comprises assignments, conditional jumps, and their logical predicates and their variables and expressions, respectively. We define a relation with six attributes.

module_id	instruction_id	sort	def_list*	ref_list*	expression_id list

The attribute sort specifies the type of the instruction. There are three possibles values:

- `assign`: a value assignment to one or more variables by one or more expressions, including function calls.
- `call`: a procedure call.
- `jump`: a logical expression for branching the control flow.

The def_list consists of the variable id's which will be defined by new values. The ref_list includes the referenced variables in the assignment. The expression_id_list is an ordered list of the expression id's in the assignment.
The def_list and the ref_list enable straightforward data flow analysis and program slicing (as defined in [8]).

5. **The Expression Level**

   The expression level manages the representation of all defined expressions. We define a relation with four attributes.

module_id	expression_id	type*	syntax tree

   The attribute type describes the appropriate type of the expression. Its entries are: `integer`, `float`, `string`, `boolean` or structured types. For the representation of the expression we use the syntax tree which provides easier analysis of the structure. The type attribute is very useful for human understanding of the meaning of instructions.

6. **Variable and Data Structure Level**

   The variable and data structure level administrates the variables and data types. We define a relation with four attributes.

module_id	variable/type_id	type	lex_name*

   The type attribute is similar to the type attribute of the expression level. The lex_name is an important extension of the minimal repository. It stores the lexical name of a defined type which contains, if well choosen, information about its intended use.

5 Extensions For Reverse Engineering

The reverse engineering process includes analysis of the given software system. For representing the extracted information, it is possible to enhance the repository relations without loosing the semantic connection to the given repository entries. We distinguish between additional relations for meta data and extensions concerning the minimal repository.

5.1 Extensions For Meta Data

The repository defined in the last section makes possible the inclusion of meta data. In fact it is possible to add meta data to every item defined in the repository. The generic form of a meta data extension is conceptually simple. For every item we can add the relation:

item_id*	document_id*	document*	document structure*

Adding this relation does not harm the former definiton of the repository. It gives a direct connection to the items. The attribute document represents possible data about the item. There is no range limit in the way the data is formulated. The document structure signifies possible structures among different entries of the meta data itself.

5.2 Minimal Repository Extensions

When using sophisticated tools for program analysis and for source code manipulation it is sometimes necessary to change and enhance the minimal repository. The repository representation of source code makes possible the expression of new ideas which could not be expressed with source code in ASCII file representation. In [1] we showed a method for restructuring. The result of the restructuring, using doubled sequential blocks, is a control flow graph similar to the extended Nassi-Shneiderman diagram. Applying this method directly on source code yields copy parts of the source code. After restructuring the source code, the location of the copied program blocks is hidden, and determining their whereabouts becomes an NP-complete string matching problem. Instead of copying source code, we add a relation to the control flow level called the *copy relation*:

module_id	program block_id	copy_of_ block

If it is necessary to dublicate a program block $a1$ to $a2$ and integrate the new block with respect to the restructuring method, changes in the control flow relation entries occur. Instead of doubling all the levels below the control flow level, we add the entry:

module_id	$a2$	$a1$

Whenever there is a need for information about the program block $a2$, the copy relation at the control flow level refers to the program block $a1$.
Adding a *copy relation* to the control flow level simplifies the understanding and maintenance of the changes. Additionally, from the storage point of view, it is better to manage a new entry in a data base than to manage an inserted program fragment in the source code file.

6 Discussion and Future Work

Building systems that comply with the necessary conditions of reengineering is one of the harder challenges in software engineering research. One way is to develop CASE methods that include a later reengineering process. Another way is building CARE systems for the large amount of non-CASE developed software. As long as there remains so much source code which is not CASE-developed, the need for CARE environments will persist. With our stated architecture, we hope to open new possibilities in solving the maintenance problem.

In contrast to related work, our CARE approach is driven by the belief that semantically appropriate representations of the source code are necessary, and that they relieve the pain of reading source code in syntactically 'inhuman' programming languages. A further shortcoming of many known CASE and CARE Systems is the lack of references between documents from different phases of the engineering process. In our approach to repositories we enable tailoring references between all atomic units, independently of the process phase.

To date we have written a prototye in Prolog that consists of the parsing of a COBAPA program and the representation of the database relation as Prolog clauses. Future work includes the extension of the prototype repository to more abstracted entities, and the development of the query mechanism for the program views. A further important requirement for CARE (as well as for CASE) is the concurrent access to the software system by different reengineering programmers. With this concern in mind, we intend to set our system on the rule-based development environment MARVEL [4].

7 Appendix A

The following definition defines the syntax of COBAPA. The definitions of logical_expression, expression, list_of_variables, list_of_expressions, newline, line_number and variable_name are considered to be as they are generally known.
We show the **syntax** of the control structure using the EBNF-grammar formalism [9]:

program	=	{ line_number instruction_line newline }
instruction_line	=	begin_loop
	|	end_loop
	|	conditional_jump
	|	jump
	|	instruction
	|	program_end
	|	procedure_call
	|	begin_proc
	|	end_proc
	|	begin_func
	|	end_func
begin_loop	=	**for** assignment **do**
	|	**while** logical_expression
	|	**loop**
	|	**repeat**
end_loop	=	**next**
	|	**endwhile**
	|	**endloop**
	|	**exit** [line_number]
	|	**until** logical_expression
conditional_jump	=	**if** logical_expression **then** [**exit**] [line_number]
	|	**else**
	|	**endif**
	|	**case** expression case_list
case_list	=	{ '('constant, line_number')' }
		'(' **default**, line_number')'
jump	=	**goto** line_number
instruction	=	assignment
	=	**read** [file_name] list_of_variables
	=	**write** [file_name] list_of_expressions
	=	**end**
assignment	=	variable := expression
program_end	=	**end**
procedure_call	=	**call** proc_name [list_of_expressions]
begin_proc	=	**proc** proc_name [parameter list]
end_proc	=	**endproc** proc_name
begin_func	=	**fun** funcion name [parameter list]
end_func	=	**endfun** function name
	|	**return** expression
parameter list	=	{ [**var**] variable_name : typ }

The syntax of the data structure and its reference:

declaration	=	{ line_number variable_name : typ newline }
typ	=	**integer**
	|	**real**
	|	**boolean**
	|	**char**
	|	**string**
	|	**array** index_range **of** typ
	|	**record** { variable_name : typ } **endrecord**
	|	**file of** typ

The dereferencing is done, as it is in Pascal, using a period for **record** and bracets for **array**.

8 Appendix B

We show here a sample COBAPA program, the computation of the faculty, with full input output computation. The source code is followed by the complete repository entries.

```
10 input : integer
20 answer : char

100 write 'Computation of the faculty'
110 write 'Issue a number: '
120 read input
130 write 'The faculty of '; input; ' is'; faculty(input)
140 write 'Do you wanna another computation?'
150 read answer
160 if answer = 'y' then 110

200 fun faculty n: integer
210 fac, i: integer
220 fac := 1
230 i := n
240 while i > 1
250 fac := fac * i
260 i := i - 1
270 endwhile
280 return fac
290 endfun faculty
```

The Repository Entries

Module Interconnection Level

module_id	sort	p_list	p_dependence*	ref_list*	call_list*
1	main	-	-	input answer i-o-file	faculty

module_id	sort	p_list	p_dependence*	ref_list*	call_list*
faculty	function	0.faculty 1.**val** n: **integer**	**ref**(n) **def**(faculty) **r-d**(n,faculty)	-	-

Control Flow Level

module_id	program block_id	program block_Id'	case
1	start	block_1	-
1	block_1	block_2	-
1	block_2	block_2	**true**
1	block_2	ende	**false**

module_id	program block_id	program block_Id'	case
faculty	start	block_1	-
faculty	block_1	block_2	-
faculty	block_2	block_3	**true**
faculty	block_3	block_2	-
faculty	block_2	block_4	**false**
faculty	block_4	ende	-

Program Block Level

module_id	program block_id	instruction_id	instruction_id'	variable_id
1	block_1	in	inst_1	i-o-file
1	block_1	inst_1	out	i-o-file
1	block_2	in	inst_2	i-o-file
1	block_2	inst_2	inst_3	i-o-file
1	block_2	inst_3	inst_4	i-o-file
1	block_2	inst_4	inst_5	i-o-file
1	block_2	inst_3	inst_5	input
1	block_2	inst_5	inst_6	i-o-file
1	block_2	inst_3	inst_7	input
1	block_2	inst_3	out	input
1	block_2	inst_6	inst_7	i-o-file
1	block_2	inst_7	inst_8	i-o-file
1	block_2	inst_8	inst_9	i-o-file
1	block_2	inst_9	inst_10	answer
1	block_2	inst_9	out	i-o-file
1	block_2	inst_9	out	answer

module_id	program block_id	instruction_id	instruction_id'	variable_id
faculty	block_1	inst_2	in	n
faculty	block_1	inst_2	out	i
faculty	block_1	inst_1	out	fac
faculty	block_2	in	inst_3	i
faculty	block_3	in	inst_4	fac
faculty	block_3	in	inst_4	i
faculty	block_3	inst_4	out	fac
faculty	block_3	inst_5	out	i
faculty	block_3	in	inst_5	i
faculty	block_4	in	inst_6	fac
faculty	block_4	in	inst_6	fac
faculty	block_4	inst_6	out	fac

Instruction Level

module_id	instruction_id	sort	def_list*	ref_list*	expression_id list
1	inst_1	assignment	i-o-file	i-o-file	expr_1
1	inst_2	assignment	i-o-file	i-o-file	expr_2
1	inst_3	assignment	input	i-o-file	expr_3
1	inst_4	assignment	i-o-file	i-o-file	expr_4
1	inst_5	assignment	i-o-file	i-o-file	expr_5
1	inst_6	assignment	i-o-file	i-o-file	expr_6
1	inst_7	assignment	faculty	input	expr_7
1	inst_7'	assignment	i-o-file	i-o-file faculty	expr_7
1	inst_8	assignment	i-o-file	i-o-file	expr_8
1	inst_9	assignment	answer	i-o-file	expr_9
1	inst_10	branch		answer	expr_10

module_id	instruction_id	sort	def_list*	ref_list*	expression_id list
faculty	inst_1	assignment	fac		expr_1
faculty	inst_2	assignment	i	n	expr_2
faculty	inst_3	branch		i	expr_3
faculty	inst_4	assignment	fac	fac i	expr_4
faculty	inst_5	assignment	i	i	expr_5
faculty	inst_6	assignment	faculty	fac	expr_6

Expression Level

module_id	expression_id	type*	syntax tree
1	expr_1	string	'Computation of the faculty'
1	expr_2	string	'Issue a number; '
1	expr_3	num_expr	-
1	expr_4	string	'The faculty of '
1	expr_5	string	input
1	expr_6	string	' is'
1	expr_7	num_expr	faculty(input)
1	expr_8	string	'Do you wanna another...'
1	expr_9	char	-
1	expr_10	log_expr	=(answer,'y')

module_id	expression_id	type*	syntax tree
faculty	expr_1	num_expr	1
faculty	expr_2	num_expr	n
faculty	expr_3	log_expr	¿(i,1)
faculty	expr_4	num_expr	*(fac,i)
faculty	expr_5	num_expr	-(i,1)
faculty	expr_6	num_expr	fac

Variable and Data Structure Level

module_id	variable/type_id	type	lex_name*
1	i-o-file	**file**	-
1	input	**integer**	input
1	answer	**char**	answer

module_id	variable/type_id	type	lex_name*
faculty	n	**integer**	n
faculty	fac	**integer**	fac
faculty	i	**integer**	i

References

[1] Toni A. Bünter. *Eine Architektur eines Software-Wartungssystems*. PhD thesis, Universität Zürich, 1992.

[2] IBM Corp. Repository Manager/MVS, General Information, September 1990.

[3] John R. Foster and Malcolm Munro. A documentation method based on cross-referencing. In *Conference on Software Maintenance*, pages 181–185. IEEE, September 1987.

[4] Gail E. Kaiser, Naser S. Barghouti, Peter H. Feiler, and Robert W. Schwanke. Database support for knowledge-based engineering environments. *IEEE Expert*, 3(2):18–32, Summer 1988.

[5] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *8th Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, New York, January 1981. ACM.

[6] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment*, pages 177–184. ACM, April 1984.

[7] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6:307–334, 1986.

[8] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4), 1984.

[9] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.

[10] Hongji Yang. The supporting environment for a reverse engineering system – the maintainer's assistant. In *IEEE Conference on Software Maintenance*, pages 13–22. IEEE, October 1991.

9 Figures

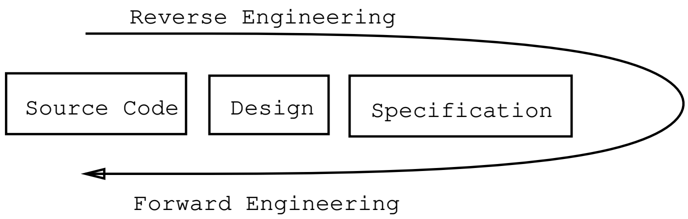

Figure 1: Reengineering

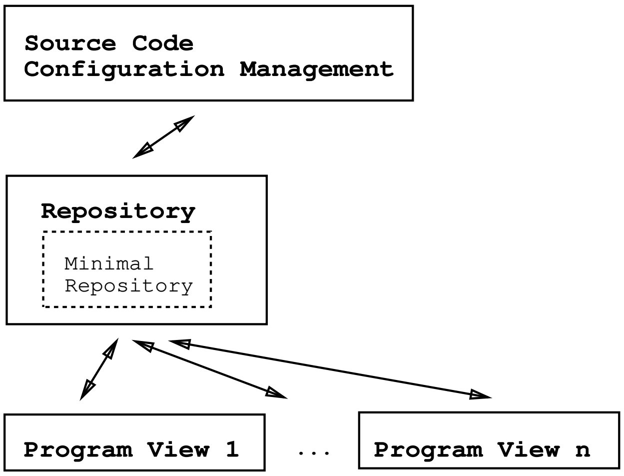

Figure 2: The architecture of the CARE system

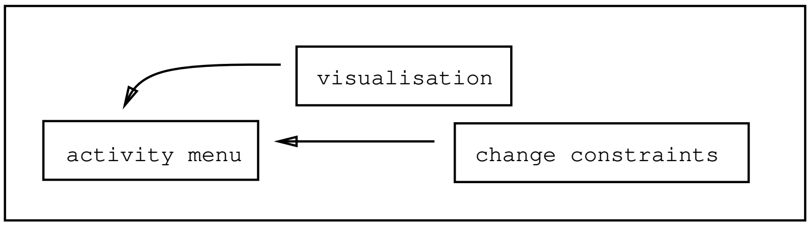

Figure 3: The components of a program view