The Extract-Transform-Rewrite Cycle
A Step towards MetaCARE

Jürgen Ebert Bernt Kullbach Andreas Panse
Institute for Software Technology
University of Koblenz
56075 Koblenz, Germany
(ebert|kullbach|panse)@informatik.uni-koblenz.de

Abstract

A conceptual reengineering framework is presented that proposes extract, transform and rewrite as three characteristic steps to be performed within a reengineering cycle.

The cycle is illustrated by an application example and a prototype tool for the C programming language. This tool supports its user in consistently renaming identifiers, in moving functions and in editing function comments.

Then, the generalization of this approach into a metaCARE technology is sketched and ways of implementing a metaCARE tool are given.

1. Introduction

The evolution of software, e. g. with respect to the year 2000, Euro currency or individual changes in requirements, results in a permanent need for reengineering activities. So, much effort has been put and has to be put into program understanding and visualization, user interface migration, rehosting or retargeting, reverse engineering of specific programs, data and databases. All this has to be accompanied by studies on cost, effectiveness, maintainability and change impact.

[Software] reengineering, also known as *renovation* or *reclamation*, is widely accepted to denote the ”‘examination and alteration of a software system to reconstitute it in a new form”’ and ”‘the subsequent implementation of the new form”’ [5]. Within this definition, reengineering includes *reverse engineering* and subsequent *forward engineering*. Reverse engineering aims at identifying the components of a system and their relationships. It normally yields a system representation on a higher level of abstraction. This representation is used a the basis for a new forward engineering activity. Transformations of one representation into another on the same level of abstraction that do not change functionality or semantics are known as *restructuring*.

Other workers on terminology like McClure [13] present a stricter notion of reengineering and restructuring. McClure identifies reengineering as the process of examining a software system and modifying it using automizing tools. In this context she defines restructuring as the standardization of data, definitions and logical program structure in order to improve maintainability and productivity of software. Byrne [4] equals the term restructuring with *alteration* and places it between *abstract* and *refine* steps. He classifies alteration into *re-coding*, *re-design* and *re-specification* depending on the level of abstraction, and he introduces *rewrite, rework* and *replace* as different reengineering strategies. Yu [17] defines re-engineering as all software engineering activities designed to effect the transformation of existing systems. He uses the terms *restructuring* and *redesign* to denote the transition from existing systems to new ones within the same implementation language.

Reengineering activities have to be supported by appropriate tools. There is a wide range of different approaches ranging from fully automatic transformations e. g. elimination of gotos [18] or source-to-source translators [6, 16], to highly manual, user-guided manipulations e. g. individual renaming or module reorganization. Amman and Cameron [1] propose an intermodule renamer that is based on a metaprogramming system. They use lambda calculus as a metalanguage. Colbrook and Smythe [7] discuss the retrospective introduction of abstract data types into software systems in oder to aid software maintenance. They propose a tool using their approach to leave code structured with respect to data and control flow. Moore [14] presents a tool for restructuring the inheritance hierarchy of self programs by maximizing the sharing of features and by refactoring shared expressions into new methods. Burson, Kotik, and Markosian [3] describe an approach to reengineering that is based on the Refine toolset. They combine object-oriented databases, program specification and pattern matching capability with program transformations facilities. Battaglia and Savoia [2] refer to the ReverseNICE reengineering toolset which supports reverse engineering into a hierarchical object-oriented design to enable manipulation and subsequent forward engineering.

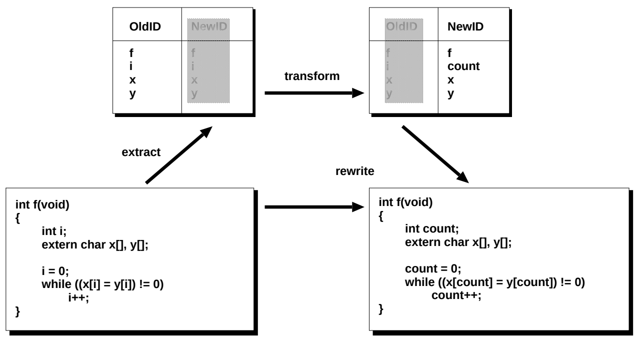
In this paper we try to establish a general view on reengineering which has many applications and which can be used as a guideline for building reengineering tools. Especially reengineering activities that require a high degree of user interaction can easily be put into this framework. Furthermore, it leads to a formalization of reengineering activities.

We present this view as a conceptual framework called the *extract–transform–rewrite cycle* (ETR cycle for short). This suits a large variety of reengineering tasks and is a substantial step towards a general metaCARE approach (CARE = Computer Aided Reengineering). In chapter 2 we give a general overview of this framework, and chapter 3 presents a small case study based upon it. Chapter 4 presents implementation issues and discusses the generalizability of the approach into a metaCARE technology.

2. The Conceptual Framework

We propose three steps to be performed within a reengineering cycle. Firstly, some kind of data *extraction* has to be performed on the sources in order to reduce complexity. Secondly, some kind of *transformation* has to be done on the data that previously have been selected. Finally, the sources have to be *rewritten* i. e. the modified data have to be reintegrated with the original sources.

We start with a reengineering-in-the-small example to motivate our conceptual framework.



OldID	NewID
f	f
i	i
x	x
y	y

OldID	NewID
f	f
i	count
x	x
y	y

```
int f(void)
{
      int i;
      extern char x[], y[];

      i = 0;
      while ((x[i] = y[i]) != 0)
            i++;
}
```

```
int f(void)
{
      int count;
      extern char x[], y[];

      count = 0;
      while ((x[count] = y[count]) != 0)
            count++;
}
```

Figure 1. A reengineering-in-the-small example

Look at the C example program given in figure 1 that copies a global character string into another. Assume that the reengineering task to be performed is to rename identifiers consistently e. g. in order to choose more expressive names or to incorporate naming conventions.

Since conventional editing is error-prone and time-consuming there is the need for some tool support like the following:

- First, the relevant set of identifiers is presented in a tabular form.
- Then, the identifiers can be manipulated individually by simple textual editing.
- After the edit step the tool checks if the modified identifiers are legal and cause no conflicts.
- If some kind of conflict has occurred, the user will be informed. Otherwise the modified identifiers are directly rewritten into the original source.

While figure 1 presents the visual documents incorporated in a reengineering process, one technically deals with *internal representations* within formalization and implementation. The representations depend on the kind of repository chosen inside the tool. They are generated by parsing the visual documents (program source and tabular form in the example) and the documents can be derived from them by unparsing. Augmenting figure 1 with the internal representations leads into the conceptual framework presented in figure 2.

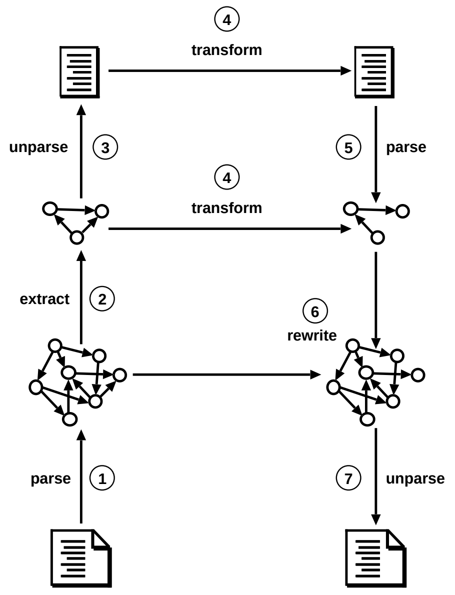


Figure 2. The Extract-Transform-Rewrite cycle

This example is an instance of a generally applicable reengineering procedure, which we call the *extract-*

transform-rewrite cycle (ETR cycle for short). An ETR cycle starts with a source program as a visual representation.

① The source document is *parsed* into an internal representation according to some conceptual model.

② An *extract* operation on this representation yields some kind of internal data structure.

③ This is *unparsed* into some visual document that enables manipulation.

④ This document is *transformed*, e. g. by textual editing.

⑤ In another *parse* step, the document modified in such a way is mapped back onto an internal data structure.

⑥ This structure is integrated with the original source in a *rewrite* step.

⑦ A final *unparse* yields a program document that reflects the change(s) performed.

This framework is applicable to a large variety of reengineering tasks. Besides the renaming of identifiers presented within the example above one may similarly

- move definitions within a document,
- distribute definitions onto different documents,
- manipulate function parameters in number and order,
- combine variables into a structure,
- initialize variables,
- transform variables into constants,
- change the storage class of identifiers,
- introduce standardized comments,
- and so on.

Some of the reengineering operations have to be done user-guided, others may be performed automatically. E. g. one may individually change the sequential order of functions or let them be sorted by some criteria, or one may rename identifiers individually or automatically provide them with canonical affixes. In the case of user interaction the transformation reduces to an editing step. Sometimes automatic transformation and editing are intertwined.

3. A Case Study

The ETR cycle approach to reengineering described in the last section has been implemented in a prototype tool for the C programming language. This language can be stated to be sufficiently complex to have the workability of the ETR approach concluded.

The prototype tool, called APPREC (Application for Reengineering C), has been coded in C++ and is based on the RogueWave template library as well as on the StarView GUI routines.

APPREC uses abstract syntax *graphs* as representations of source texts and *bags* (i. e. nested lists) containing graph vertices for the extracted data as internal representations. It allows transformation by manual editing of the extracted information in tabular forms.

The tool includes a graph generating *parse*r ① and an *unparse*r ⑦ both covering the C programming language in its full extent. (Preprocessor directives are to be resolved beforehand.) The parser and the unparser use graphs as an internal representation. They have originally been developed within the GUPRO project on program understanding [10] and suit to a fine-grained model of the C programming language.

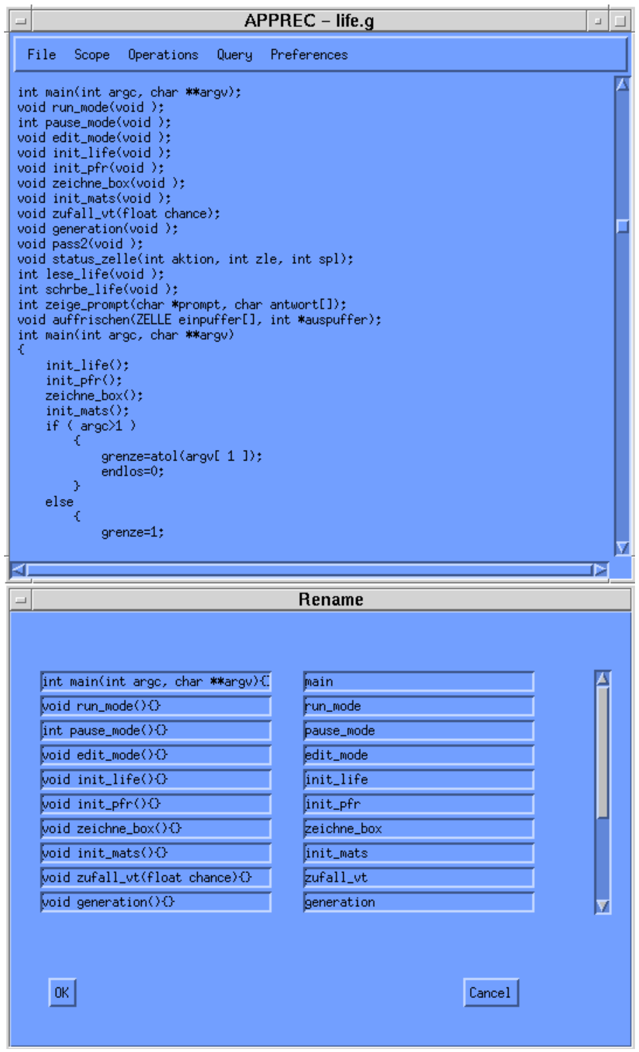
*Extract*ing ② of the concepts to be modified is performed by queries on the graph using the GReQL query language [12] which delivers bags as results.

The internal representation of the result is *unparse*d ③ to be visualized for editing in a corresponding dialog box. Objects can then be textually *edit*ed ④, i. e. they can be renamed, they can be rearranged in their order via drag and drop, and they can be annotated with comments, depending on the task to be performed. These changes are finally *unparse*d ⑤ again into bags. In APPREC, the parse/unparse steps ③ and ⑤ depicted in figure 2 are only virtually existent because the table editor directly operates on the bag representations, i. e. all editing is technically performed on the bags resulting from extraction.

Within the *rewrite* step ⑥ renamed identifiers, modified order and inserted comments are integrated with the original graph to yield a new graph reflecting the change. In rewriting, complex consistency checking is done: When renaming a variable it must not get into conflicts with naming conventions in C. The C programming language provides name spaces with complicated rules, e. g. the scope of a label identifier is always a whole function body while the scope of enumeration constants is always the least upper program block. Furthermore, extern declarations in inner blocks may provide identifiers with non contiguous scopes. Checking these consistency conditions is again implemented by GReQL queries on the repository.

Unparsing ⑦ the internal graph representation resulting from rewrite is performed by a traversal of the graph that reproduces source code by pretty printing.

Figure 3 shows an example screen shot of the tool. The main window on the left contains the main menu that offers file operations, browsing, reengineering operations, and queries. Within the text area of the main window the source text in focus is displayed. Within the window on the right renaming of functions is performed by editing. It presents



```
int main(int argc, char **argv);
void run_mode(void );
int pause_mode(void );
void edit_mode(void );
void init_life(void );
void init_pfr(void );
void zeichne_box(void );
void init_mats(void );
void zufall_vt(float chance);
void generation(void );
void pass2(void );
void status_zelle(int aktion, int zle, int spl);
int lese_life(void );
int schrbe_life(void );
int zeige_prompt(char *prompt, char antwort[]);
void auffrischen(ZELLE einpuffer[], int *auspuffer);
int main(int argc, char **argv)
{
    init_life();
    init_pfr();
    zeichne_box();
    init_mats();
    if ( argc>1 )
        {
            grenze=atol(argv[ 1 ]);
            endlos=0;
        }
    else
        {
            grenze=1;
```

int main(int argc, char **argv){	main
void run_mode(){}	run_mode
int pause_mode(){}	pause_mode
void edit_mode(){}	edit_mode
void init_life(){}	init_life
void init_pfr(){}	init_pfr
void zeichne_box(){}	zeichne_box
void init_mats(){}	init_mats
void zufall_vt(float chance){}	zufall_vt
void generation(){}	generation

Figure 3. A Screen Dump

the function names to be edited together with their type information.

One special feature of the tool is that extraction can be performed by querying and browsing. Using a query, the expert user can directly advance to the intended focus while through browsing even a beginner can handle the tool without knowing the underlying model at all. From an implementation point of view browsing and querying do not differ much since internally browsing corresponds to a sequence of predefined queries.

Beginning with the translation unit one can descend through functions to blocks and further to blocks being subordinated to these. Navigation is enabled via visualizing the source code fragment corresponding to the scope in focus in a read-only window. Depending one the current scope a predefined set of object types can be modified.

Within the global level, functions, variables, types, struct and union tags, and enumeration constants can be processed, and within the scope of a certain function, labels and function parameters are added for manipulation. For each of the concepts at least renaming can be selected for execution. Functions can also be moved and annotated with comments.

4. The Generalization into MetaCARE

The prototype CARE tool presented in the previous chapter only covers a small portion of the reengineering tasks possible. Furthermore, it is only suited for the C programming language. From a technical point of view it is just one instance of the ETR cycle. The tool is not generic in that steps ③, ⑤ and ⑥ (cf. figure 2) are hand-coded. But the implementation suggests to be generalized into a powerful metaCARE approach that offers adaptability to cover a large variety of reengineering tasks and programming languages.

The parsing/unparsing steps ① and ⑦ and the internal representation of syntax graphs for arbitrary languages can be generated from language and graph descriptions. This has been done in the GUPRO project which uses this approach in a program understanding tool [10] [9]. It is based on the *EER/GRAL approach* [11] which is a mature graph-based technology for tool building on the basis of information modeling: When modeling an application domain, e. g. a programming language, one has to specify the corresponding class of graphs. These are defined by extended entity-relationship (EER) diagrams which are annotated by additional restrictions in the $\mathcal{Z}$-like GRAL language. By combining the information of the language grammar with the corresponding EER diagram, parsers and unparsers can be generated almost automatically using the lex/yacc-based parser generator PDL [8].

Graphs representing sources can be queried using GReQL [12], which allows the extraction ② of arbitrary information from the syntax graphs into bags. GReQL is an SQL-like query language on graphs, which allows path expressions and nested queries.

The generation of editors ③ and ⑤ for the bag representations can be based on meta-editor technology like e. g. the synthesizer generator [15] or the transformation can be specified directly on the bags ④.

The bags are used in a graph manipulating routine that modifies the original source graph accordingly ⑥.

To formalize and to implement ETR following the EER/GRAL approach EER diagrams of the respective languages have to be established as concept models, which define the relevant graph class. From this, appropriate parsers are built that translate source code into graphs according to these concept models. The parsers can be generated using the parser description language PDL. Extraction from graphs into bags is performed by GReQL queries. These queries also have to refer to the underlying concept model. They have to produce a result that is suited to the transform operation. Transformation by editing is enabled by a simple dialog which is specifically tailored to a class of reengineering tasks, e. g. moving may be associated with some drag and drop activity while renaming can be enabled by simple textual editing. Rewriting has to integrate the modification into the original graph. In this context consistency checking is necessary to avoid that changes in functionality or semantics to occur. Rewriting is a process that normally has to consider the original graph as well as modified extract bag.

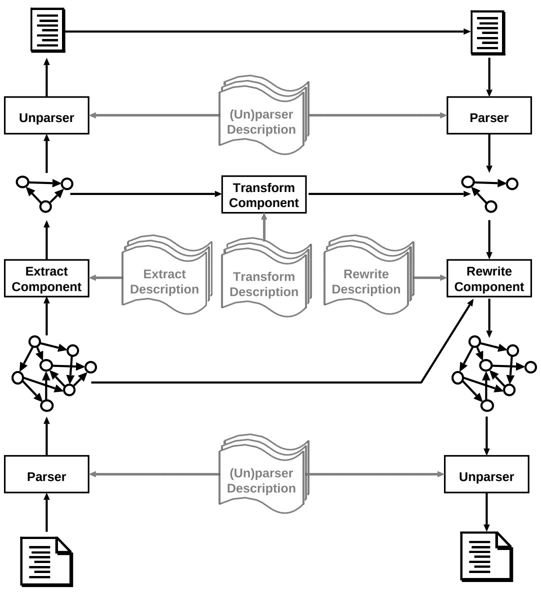


Figure 4. Architecture of a metaCARE tool

Generalizing these ETR steps leads to a *meta approach* to reengineering (cf. figure 4). The notion of meta tools is quite old, e. g. the *lex* and *yacc* tools permit to generate compilers from regular definitions and context free grammars. Today meta tools are widely accepted within the area of software development (CASE), where customizability in languages, methods, and development strategies is approached. With respect to computer aided software reengineering (CARE) the meta aspect is also near at hand. A metaCARE tool in this context has to provide relevant adaptability to special reengineering applications.

The ETR cycle (as far as it is currently realized) includes generic components as well as hand-made functionality. Our intention is to generalize the individual components to result in a complete metaCARE approach. This should allow to separately exchange or modify all components participating in the ETR cycle:

- parser and unparser for program text ① and ⑦,
- extract interface ②,
- parser and unparser involved in manual editing ③ and ⑤,
- transformation ④,
- rewrite functionality ⑥.

Changing one of these tool component descriptions that serve as an input to an abstract metaCARE tool will yield reengineering tool instances with different functionality in:

- reengineering operations,
- transformation facilities, and
- the programming language, language extension or dialect supported.

Within our approach at least parsing/unparsing ① and ⑦ and extracting ② are generic in that the respective descriptions can easily be exchanged. The other steps are currently hand-coded. But there is evidence that they can also be generalized.

5. Conclusion and Future Work

We have presented a conceptual reengineering framework that proposes extract, transform and rewrite as three characteristic steps to be performed within a reengineering cycle. This ETR cycle has been illustrated by an application example, and a selection of reengineering operations that suit this approach has been outlined.

A prototype tool for the C programming language that incorporates this approach has been presented. This tool supports its user in consistently renaming identifiers, in moving functions and in editing function comments. The concepts to be changed are selected by querying and browsing through the program scopes where the source code fragment in focus in visualized.

Finally, the generalization of the ETR approach into a metaCARE approach has been sketched. It has been pointed out which parts of the ETR cycle are already generic and what is to be done to generalize the remaining tasks to yield a complete metaCARE approach.

Ongoing tool building activities develop in two directions: First, the existing system will be enhanced to cover more reengineering operations. Candidates are initialization of variables and combining variables into a data structure. Second, reengineering prototypes for other programming languages will be developed. Currently prototypes for PASCAL, C++ and Java are being established.

Furthermore, the general metaCARE approach will be elaborated. The extract step shall be changed to also deliver graphs with the intention to generalize the transform and rewrite steps by using general graph morphisms and transformations.

References

[1] M. M. Ammann and R. D. Cameron. Inter-Module Renaming and Reorganizing: Examples of Program Manipulation in-the-Large. In *International Conference on Software Maintenance 1994*, pages 354–361. IEEE Computer Society Press, Sept. 1994.

[2] M. Battaglia and G. Savoia. ReverseNICE: A re-engineering methodology and supporting tool. *Lecture Notes in Computer Science*, 1031:244–248, 1996.

[3] S. Burson, G. Sotik, and L. Markosian. A Program Transformation Approach to Automating Software Re-Engineering. In *Proceedings of the International Computer Software & Application Conference 1190, Chicago, lll*, 1990.

[4] E. J. Byrne. A conceptual foundation for software re-engineering. In *International Conference on Software Maintenance 1992*, pages 216–235. IEEE Computer Society Press, Nov. 1992.

[5] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[6] W. C. Chu. A re-engingeering approach to program translation. In *International Conference on Software Maintenance 1993*, pages 42–50. IEEE Computer Society Press, Sept. 1993.

[7] A. Colbrook and C. Smythe. The Retrospective Introduction of Abstraction into Software. In *International Conference on Software Maintenance 1989*, pages 166–173. IEEE Computer Society Press, 1989.

[8] P. Dahm. PDL: Eine Sprache zur Beschreibung grapherzeugender Parser. Diplomarbeit D 305, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, Oktober 1995.

[9] J. Ebert, R. Gimnich, and A. Winter. Wartungsunterstützung in heterogenen Sprachumgebungen, Ein Überblick zum Projekt GUPRO. In F. Lehner, editor, *Softwarewartung und Reengineering - Erfahrungen und Entwicklungen*, pages 263–275, Wiesbaden, 1996. Gabler.

[10] J. Ebert, M. Kamp, and A. Winter. A Generic System to Support Multi-Level Understanding of Heterogeneous Software. Fachbericht Informatik 6/97, Universität Koblenz-Landau, Institut für Informatik, Koblenz, 1997.

[11] J. Ebert, A. Winter, P. Dahm, A. Franzke, and R. Süttenbach. Graph Based Modeling and Implementation with EER/GRAL. In B. Thalheim, editor, *15th International Conference on Conceptual Modeling (ER'96), Proceedings*, number 1157 in LNCS, pages 163–178, Berlin, 1996. Springer.

[12] M. Kamp. GReQL - eine Anfragesprache für das GUPRO–Repository 1.1. Interner Projektbericht 8/96, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, Januar 1996.

[13] C. McClure. *The Three Rs of Software Automation: Reengineering, Repository, and Reusability*. Engelwood Cliffs, New Jersey, 1992.

[14] I. Moore. Guru - a tool for automatic restructuring of self inheritance hierarchies. In *Tools USA' 95 (Technology of Object-Oriented Languages and Systems)*, pages 267–275, 1995.

[15] T. Reps and T. Teitelbaum. The synthesizer generator. *ACM SIGPLAN Notices*, 19(5):42–48, 1984.

[16] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, Aug. 1988.

[17] D. Yu. A view On Three R's (3Rs): Reuse, Re-Engineering, and Reverse Engineering. *ACM SIGSOFT Software Engineering Notes*, 16(3):69, 1991.

[18] H. Zima and B. Chapman. *Supercompilers for Parallel and vector Computers*. ACM Frontier Series. Addison-Wesley, 1991.