Re-engineering needs Generic Programming Language Technology

Mark van den Brand[1], Paul Klint[2,1], Chris Verhoef[1] *

[1]University of Amsterdam, Programming Research Group
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

[2]CWI, Department of Software Technology
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

markvdb@wins.uva.nl, paulk@cwi.nl, x@wins.uva.nl

Abstract

Generic language technology and compiler construction techniques are a prerequisite to build analysis and conversion tools that are needed for the re-engineering of large software systems. We argue that generic language technology is a crucial means to do fundamental re-engineering. Furthermore, we address the issue that the application of compiler construction techniques in re-engineering generates new research questions in the field of compiler construction.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.3.2 [**Programming Languages**]: Language Classifications—Specialized application languages;

Additional Key Words and Phrases: re-engineering, reverse engineering, system renovation, intermediate data representation, compiler construction techniques, generic language technology, programming environment generator

1 Introduction

In 1977, Mathew Hecht wrote in his book [Hec77] on flow analysis of computer programs "Flow analysis can be used to derive information of use to human beings about a computer program", in fact he was referring to what we nowadays call *program understanding* or *reverse engineering*. He further motivated the use of flow analysis by stating that "some automatic program restructuring may be possible" and that "perhaps remodularization could be accomodated", techniques that are relevant to restructure and remodularize *legacy* systems. So, it comes hardly as a surprise that we will argue here that classical compiler construction techniques are extremely useful to aid in re-engineering.

Re-engineering is becoming more and more important. There is a constant need for updating and renovating business-critical software systems for many and diverse reasons: business requirements change, technological infrastructure is modernized, governments change laws, or the third millennium approaches, to mention a few. So, in the area of software engineering the subjects of program understanding and system renovation become more and more important. The interest in such subjects originates from the difficulties that one encounters when attempting to maintain large, old, software systems. It is not hard to understand that it is very difficult—if not impossible—to renovate such legacy systems.

The purpose of this paper is to show that a substantial part of the technology used in re-engineering often originates from these fields. We want to make researchers in the field of compiler construction and generic language technology aware of the application of their techniques in the field of re-engineering. Furthermore, we will identify topics for further research that are particularly relevant for re-engineering.

In [BKV96b] generic language technology is used as a core technology for re-engineering. For more information on the subject of re-engineering we refer to the annotated bibliographies [Arn93] and [BKV96a].

*The authors were all in part sponsored by bank ABN AMRO, software house DPFinance, and the Dutch Ministry of Economical Affairs via the Senter Project #ITU95017 "SOS Resolver". Chris Verhoef was also supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), project *Interactive tools for program understanding*, 612-33-002.

2 Reverse Engineering and System Renovation Terminology

The term reverse engineering finds its origins in hardware technology and denotes the process of obtaining the specification of complex hardware systems. Now the meaning of this notion has shifted to software. As far as we know there is not (yet) a standard definition of what reverse engineering is but in [CC90] we can read:

> "Reverse engineering is the process of analyzing a subject system to identify the system's components and their inter-relationships, and to create representations of the system in another form at higher levels of abstraction."

According to [CC90] the following six terms characterize system renovation:

- Forward engineering.
- Reverse engineering.
- Redocumentation.
- Design recovery.
- Restructuring.
- Re-engineering (or renovation).

Forward engineering moves from a high-level abstraction and design to a low-level implementation. *Reverse engineering* can be seen as the inverse process. It can be characterized as analysing a software system in order to, firstly, identify the system components and their interactions, and to, secondly, make representations of the system on a different, possible higher, level of abstraction. This can be seen as a form of decompilation. It may be necessary to move even from assembler (or from the executables) level to a higher level.

Reverse engineering restricts itself to *investigating* a system. Adaptation of a system is beyond reverse engineering but within the scope of system renovation. *Redocumentation* focuses on making a semantically equivalent description at the same level of abstraction. It is in fact a simple form of reverse engineering. Tools for redocumentation include, among others, pretty printers, diagram generators, and cross-reference listing generators. In *design recovery* domain knowledge and external information is used to make an equivalent description of a system at a higher level of abstraction. So, more information than the source code of the system is used. The notion *restructuring* amounts to transforming a system from one representation to another one at the same level of abstraction. An essential aspect of restructuring is that the semantic behaviour of the original system and the new one should remain the same; no modifications of the functionality is involved. The purpose of *re-engineering* or *renovation* is to study the system, by making a specification at a higher abstraction level, adding new functionality to this specification and develop a completely new system on the basis of the original one by using forward engineering techniques.

3 Specific Languages and Re-engineering

In this section we will give the reader an impression on the relation between specific programming languages and re-engineering.

Since many business-critial systems that need re-engineering are written in COBOL there are quite a number of papers available that discuss methods and techniques that focus on COBOL. For instance, in [GBW95] the re-engineering of 50,000 lines of COBOL to Ada is described. The goal was to do it as automatically as possible using compiler construction techniques. An example of the use of program slicing to aid in re-engineering COBOL can be found in [NEK93] and [CFV93]. In [NM93] Software Refinery, a tool originally developed primarily to generate programming environments, is used to build a modularization tool for COBOL. In [Zuy93] aspects of re-engineering and their relation to language technology are discussed. We mention the compilation of COBOL code to equational specifications, their restructuring and simplification, and regeneration of COBOL code from them. Moreover COBOL is compiled to an intermediate language supporting both all the features of COBOL as well as those of JCL. Various tools support these compilations. In [BFK+94] and [BFKO93] denotational semantics is advocated as a formal foundation for understanding COBOL programs. These ideas are implemented in a tool for the reverse engineering of COBOL-74 programs.

Not only COBOL is subject to reverse engineering. We mention [OC93] in which a tool combining static and dynamic information for analyzing C programs is described. In [CCC93] a method is described to produce design level documents by static analysis of Ada programs using data flow analysis. Finally, in [Byr91] we can find a method to convert Fortran programs into Ada code. This is done via analysis of the Fortran code followed by a reimplemention of the extracted design in Ada. Needless to say that in the above cases generic language technology and compiler construction techniques play an important role.

4 Compiler Construction Techniques and Re-engineering

Many re-engineering tools use compiler construction techniques. When constructing a compiler these techniques are used to go from a high-level language to a low-level implementation. When re-engineering a legacy system those techniques are used to move from a low-level implementation to a more abstract level. In compiler construction terms we could say that re-engineering amounts to the decompilation of source code into its specification.

A number of standard techniques in compiler construction are listed below together with their applications in the field of re-engineering.

- Scanning. Usually a scanner performs the lexical analysis of a program, it tokenizes programs to be fed to the parser. In re-engineering, the technique of lexical analysis serves the purpose of program understanding. It can be used to locate, for instance, a specific identifier in the source code and it can thus be considered as an intelligent `grep(1)` facility. Many so-called Y2K-tools[1] like SEEC COBOL Analist [SEE96] use scanning techniques to find date related identifiers and variables, e.g., `YEAR`, `YY`, `MONTH`, `CENTURY`, etc., in the code of legacy systems.

  The usefulness of scanner generators for re-engineering is quite obvious. For the COBOL language numerous dialects exists among which even non official ones and the development of scanners for each of these dialects is too expensive. Since the scanners are only slightly different, a generic approach using a scanner generator is appropriate.

- Parsing. Usually a parser is used to determine whether or not a string of tokens could be generated by a grammar. Re-engineering tools work on the (abstract) syntax trees yielded by the parser. They can be used to calculate, for instance, the McGabe and McClure cyclometric complexity measures. Such metrics characterize the complexity of programs and give an indication of the costs to re-engineer them.

  At the syntactical level, there are many variations in the various COBOL dialects, and it is time-consuming to develop specific parsers for them from scratch[2]. The use of parser generators ensures the correctness of parsers and gives a considerable reduction in implementation effort.

- Type checking. One of the standard static checks of a compiler is type checking. In the realm of re-engineering type checking results can be stored in the (abstract) syntax tree to be used for inspection later on. It can also be used to locate variables of the same type, for instance, variables of the type `PIC 99` in COBOL programs, in order to locate possible date related variables that may give rise to year 2000 problems.

  Type checkers can be defined using syntax directed translation mechanisms, such as attribute grammars [AM91] or algebraic formalisms [DHK96]. The benefits of these formalisms are the strong relationship

[1] Y2K stands for year 2000.

[2] So, it is not suprising that in `comp.compilers` (a Usenet newsgroup on compilers) frequent requests appear for public domain COBOL grammars in some standard format, e.g., LEX+YACC [Joh86, LS86] or BNF, probably to be used in formal analysis tools. Until now there has been no positive reply.

between syntax and semantics, and the ease of constructing such specifications. Formalisms that support some form of modularity provide facilities for re-usability in case of dialects.

- Control flow analysis. The purpose of this analysis is to encode the flow of control of a program for use in the ensuing data flow analysis. In the field of program understanding it is used to make the structure of a program apparent. A number of interesting papers on the subject of control flow and re-engineering are, for instance, [Amm92], [CNR90], and [Oul82].

- Data flow analysis. Usually data flow analysis is the process of extracting information from a computer program about the possible run-time modification, preservation and usage of certain quantities in it. In re-engineering such techniques are useful to detect dependencies between variables such as def-use chains. A typical example is to locate all the variables that are dependent on date variables to aid in making software year 2000 compliant. More information on data flow analysis can be found in [MR90].

- Abstract interpretation. A classical application of abstract interpretation is the nonstandard exectution of a program by casting out nines to check arithmetic computations in that program. It is used for program validation and analysis. In re-engineering abstract interpretation can be used for analysis as well, for instance, to do range checks on certain variables to see whether or not they remain in a certain range.

- Program slicing. Program slicing is a technique to identify the minimal amount of executable code that is needed to give a certain variable its value. Slicing can also be used to calculate the pieces of a program that depend on a given variable, and it can be used to debug a program. In re-engineering it can be used for the same kind of analysis, for instance, to identify parts of code that are responsible for date related calculations. Tip [Tip95] gives an overview of program slicing techniques. A few papers on re-engineering and program slicing are [BE93], [GL91], [GHS92], and [Hal95].

We saw above that there are many applications of compiler construction technology in re-engineering. Several phases in which source code is processed by a compiler can be related to the phases that such code will pass during re-engineering. As is well-known (see, e.g., [ASU86] or [WM95]) we can distinguish the following analytic phases in a compiler: the lexical, syntactic, and semantic analyzer where the bulk of the analysis is taken care of. In re-engineering we have exactly the same phases that are also meant for analyzing the source code: the lexical phase for a rough inspection of the code, the syntax analysis for both composing a parse tree and for more involved analysis and the semantic analysis for even more involved analyses as we discussed above.

Of course the target of a compiler is to generate code from a source program. In that respect re-engineering and compiler construction differ, however, in [CM96] code generation is used for binary translation of systems from a (legacy) architecture to a modern architecture. Note that code optimization techniques are used in re-engineering as well. To generate optimal code it is necessary that the structure of a program and the dependencies between the variables in the program are made clear. To make a program understandable for human beings its structure and dependencies have to be clear as well. So, it is not surprizing that the compiler optimization techniques such as control flow analysis, data flow analysis and abstract interpretation, are also relevant in re-engineering (see above).

We conclude that re-engineering techniques benefit from compiler construction techniques and that the other well-established techniques that are available in the compiler design field could be fruitfully applied in re-engineering as well. We believe that these techniques will attract new interest by their application in re-engineering.

5 Generic Programming Language Technology and Re-engineering

To what extent depend various re-engineering tools on specific programming languages? Many tools are geared towards COBOL (or some of its dialects), but some re-engineering tools claim to be language independent.

We will classify language (in)dependence in the following categories:

- We call a system *language-independent* if it has *no* built-in knowledge of a specific language. An example is the UNIX[3] tool `grep(1)`, that can be used for simple textual searches in source files.
- We call a system *language-dependent* if the knowledge of a language is hard-wired in the system, e.g., a C-compiler. This knowledge can be implemented in the system by hand, via a generator, or via a combination of these approaches.
- We call a system *language parameterized* (or *generic*) if the language is a parameter of the system and upon instantiation with a language definition a language-specific system is obtained. Examples are the Synthesizer Generator [RT89] and the ASF+SDF Meta-Environment [Kli93, DHK96].

Generic language technology developed during the eigthies and embodied in programming environment generators such as, for instance, the Synthesizer Generator [RT89], Software Refinery [Rea92], the PSG system [BS86], CENTAUR [BCD+89] and the ASF+SDF Meta-Environment [Kli93], forms now the basis for interactive re-engineering systems. Such systems provide facilities for program analysis, visualization, code restructuring, and automatic language conversions.

Since many legacy systems are polylingual it is important that re-engineering systems are based on generic language technology. We are confronted with programs or even complete systems written in numerous dialects of old fashioned programming languages which have to be understood and analyzed. Developing new tools for all the dialects is far too expensive and can be done more effectively using generic techniques. So, there is a strong connection between re-engineering and the fields of programming environment generators and compiler generators. The generic aspect is thus extremely valuable in re-engineering, see [PP94a].

The def-use relations in programs, for example, are in fact language independent, however their implementation is often language dependent. In [Moo96] a generic data flow language is defined which is powerful enough to do all kinds of data flow analysis. An arbitrary language is translated to this data flow language.

Various new, generic, approaches to program analysis exist. In [BDFH96] an equational logic language, PIM [Fie92], is presented which can serve as a intermediate language representation to solve problems in the field of program slicing, symbolic evaluation, and dependence analysis. It is designed to be used by compilers and analysis tools to process imperative langauges and can be used for re-engineering purposes as well. Other approaches include static shape analysis, security analysis, and the generation of static analysis algorithms from semantic definitions. An overview of recent work in these areas can be found in [HN96].

6 Research Directions

One of the challenges is how to formally define the syntax and the semantics of the old fashioned programming languages one encounters during re-engineering. For example, in [Man96] the syntax of COBOL-85 is formally defined in SDF [HHKR92] using the ANSI definition of COBOL-85 [ANS85]. Are the various language definition formalisms powerful enough to be used for this purpose?

Below we will list a number of research questions generated by applying compiler construction techniques in re-engineering.

- Higher-level techniques for lexical analysis, that permit easy extraction of information, such as call graphs, from programs written in languages for which no parser is available.
- Application of GLR parsers [Tom85, Rek92] to generate parsers for families of COBOL or PL/I language dialects. Note that conventional LALR-techniques break down (i.e., generate many shift-reduce conflicts) when various dialects have to be covered by a single parser.
- Generic data flow analysis.
- Visualization techniques for presenting the results of program analysis.
- Knowledge representation techniques for encoding certain implicit aspects of programs such as dependencies on their operating context, programming conventions and techniques, and application specific information.

[3] UNIX is a registered trademark of UNIX System Laboratories

- Query techniques to retrieve all kinds of syntactic and semantic information about a program, see [PP94b, BKV96a].
- (Automatic) program transformation techniques for dialect conversion, systematic editing and correction, and conversion to object-oriented languages.

Concluding we could say that challenging research in (generic) programming language technology is inspired by problems encountered in the field of re-engineering.

References

[AM91] H. Alblas and B. Melichar, editors. *International Summer School on Atribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[Amm92] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, 1992.

[ANS85] *ANSI Programming Language – COBOL, ANSI X3.23-1985*. American National Standards Institute, 1985.

[Arn93] R.S. Arnold. *Software Reengineering*. IEEE Computer Society Press, 1993.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[BCD+89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 24(2).

[BDFH96] J. A. Bergstra, T. B. Dinesh, J. Field, and J. Heering. A complete transformational toolkit for compilers. In H. R. Nielson, editor, *Programming Languages and Systems (ESOP '96)*, volume 1058 of *Lecture Notes in Computer Science*, pages 92–107. Springer-Verlag, 1996. Full version: Technical Report RC 20342, IBM T. J. Watson Research Center, Yorktown Heights, and Technical Report CS-R9601, Centrum voor Wiskunde en Informatica (CWI), Amsterdam. The full version will appear in TOPLAS.

[BE93] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *[WC93]*, pages 54–63, 1993.

[BFK+94] P. Baumann, J. Fässler, M. Kiser, Z. Öztürk, and L. Richter. Semantics-based reverse engineering. Technical Report 94.08, Department of Computer Science, University of Zurich, Switzerland, 1994.

[BFKO93] P. Baumann, J. Fässler, M. Kiser, and Z. Öztürk. Beauty and the Beast or A Formal Description of the Control Constructs of Cobol and its Implementation. Technical Report 93.39, Department of Computer Science, University of Zurich, Switzerland, 1993.

[BKV96a] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. Technical Report P9614, University of Amsterdam, Programming Research Group, 1996. To appear in the proceedings of SOFSEM'96.

[BKV96b] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation – an annotated bibliography. Technical Report P9603, University of Amsterdam, Programming Research Group, 1996. To appear in ACM Software Engineering Notes.

[BS86] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[Byr91] E.J. Byrne. Software reverse engineering: A case study. *Software—Practice and Experience*, 21(12):1349–1364, 1991.

[CC90] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CCC93] G. Canfora, A. Cimitile, and U. De Carlini. A reverse engineering process for design level document production from ada code. *Information and Software Technology*, 35(1):23–34, 1993.

[CFV93] F. Cutillo, P. Fiore, and G. Visaggio. Identification and extraction of "domain independent" components in large programs. In *[WC93]*, pages 83–92, 1993.

[CM96] C. Cifuentes and V. Malhota. Binary translation: Static, dynamic, retargable? In N.F. Schneidewind, editor, *International Conference on Software Maintenance*, pages 340–349. IEEE, 1996.

[CNR90] Y-F. Chen, M.Y. Nishimoto, and C.V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, 1990.

[DHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[Fie92] J. Field. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 98–107, San Francisco, June 1992. Published as Yale University Technical Report YALEU/DCS/RR–909.

[GBW95] R. Gray, T. Bickmore, and S. Williams. Reengineering cobol systems to ada. Technical report, InVision Software Reengineering, Software Technology Center, Lockheed Palo Alto Laboratories, 1995.

[GHS92] R. Gupta, M. Harrold, and M. Soffa. An approach to regression testing using slicing. In *[Kel92]*, pages 299–308, 1992.

[GL91] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[Hal95] R.J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2:33–53, 1995.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, Amsterdam, 1977.

[HHKR92] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual,* version 6 December, 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989. ftp://ftp.cwi.nl/pub/gipe/reports/SDFmanual.ps.Z.

[HN96] C. Hankin and H.R. Nielson. Symposium on models of programming languages and computation. *ACM Computing Surveys*, 28(2):293–359, 1996.

[Joh86] S.C. Johnson. *YACC: yet another compiler-compiler*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[Kel92] M. Kellner, editor. *Proceedings Conference on Software Maintenance*. IEEE Computer Society Press, 1992.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[LS86] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, unix programmer's supplementary documents, volume 1 (ps1) edition, 1986.

[Man96] P.I. Manuel. ANSI COBOL III in SDF + an ASF definition of a Y2K tool. Master's thesis, University of Amsterdam, Programming Research Group, 1996.

[Moo96] L. Moonen. Data flow analysis for reverse engineering. Technical Report P96113, University of Amsterdam, Programming Research Group, 1996.

[MR90] Marlowe and Ryder. Properties of data flow frameworks. A unified model. *Acta Informatica*, 28:121–163, 1990.

[NEK93] J. Ning, A. Engberts, and W. Kozaczynski. Recovering reusable components from legacy systems. In *[WC93]*, pages 64–72, 1993.

[NM93] Ph. Newcomb and L. Markosian. Automating the modularization of large COBOL programs: application of an enabling technology for reengineering. In *[WC93]*, pages 222–230, 1993.

[OC93] D. Olshefski and A. Cole. A prototype system for static and dynamic program understanding. In *[WC93]*, pages 93–106, 1993.

[Oul82] G. Oulsnam. Unraveling unstructured programs. *The Computer Journal*, 25(3):379–387, 1982.

[PP94a] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[PP94b] S. Paul and A. Prakash. Supporting queries on source code: A formal framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.

[Rea92] Reasoning Systems, Palo Alto, California. *REFINE User's Guide*, 1992.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as Rek92.ps.Z.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[SEE96] SEEC, Inc., 5001 Baum Boulevard, Pittsburgh, PA 15213. *SEEC COBOL Analyst*, 1996.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[Tom85] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.

[WC93] R.C. Waters and E.J. Chikofsky, editors. *Proceedings of Working Conference on Reverse Engineering*. IEEE Computer Society Press, 1993.

[WM95] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley Publisher Ltd., 1995.

[Zuy93] H. van Zuylen, editor. *The ReDo compendium: reverse engineering for software maintenance*. Wiley, 1993.