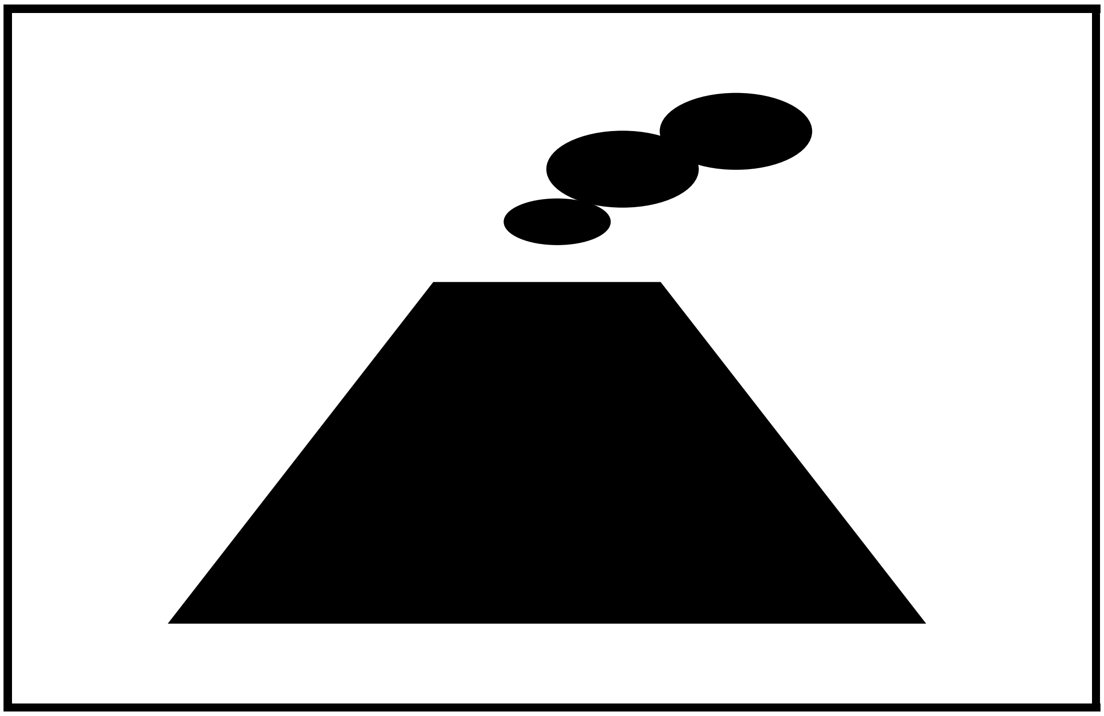
University of Amsterdam
Programming Research Group



Research Issues in the Renovation of Legacy Systems

Arie van Deursen
Paul Klint
Chris Verhoef

Report P9902 april 1999

University of Amsterdam
Department of Computer Science
Programming Research Group

Research Issues in the Renovation of Legacy Systems

Arie van Deursen
Paul Klint
Chris Verhoef

Report P9902 april 1999

A. van Deursen

CWI

P.O.Box 94079
1090 GB Amsterdam
The Netherlands

tel. +31 20 592 4075
e-mail: arie@cwi.nl

P. Klint

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7585
e-mail: paulk@wins.uva.nl

CWI

P.O.Box 94079
1090 GB Amsterdam
The Netherlands

tel. +31 20 592 4126
e-mail: paulk@cwi.nl

C. Verhoef

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7581
e-mail: x@wins.uva.nl

Universiteit van Amsterdam, 1999

Research Issues in the Renovation of Legacy Systems

Arie van Deursen[1], Paul Klint[1,2], and Chris Verhoef[2]

[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[2] University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

`arie@cwi.nl, http://www.cwi.nl/˜arie`
`paulk@cwi.nl, http://www.cwi.nl/˜paulk`
`x@wins.uva.nl, http://adam.wins.uva.nl/˜x`

Abstract. The goals of this tutorial are to: (*i*) give the reader a quick introduction to the field of software renovation as a whole; (*ii*) show that many techniques from compiler technology and formal methods can be applied; (*iii*) demonstrate that research should be driven by real-life, industrial, case studies; and (*iv*) indicate that many challenging problems are still unsolved. During the presentation of this turorial, demonstrations will be given of several of the case studies discussed here.

1 Introduction

Software renovation is using tomorrow's technology to bring yesterday's software to the level of today. In this paper, we provide an overview of this research area. We start (in this section) by exploring the need for software renovation. Moreover, we provide definitions of the basic terminology, and pointers to the most important literature. We then proceed to discuss two aspects of software renovation in more detail. In Section 2 we study how we can increase our understanding of a given legacy system, and how we can apply this knowledge for the purpose of migrating the legacy to object technology. Techniques like type inference and concept analysis play an essential role here. In Section 3 we deal with ways of building renovation factories that are capable of restructuring legacy systems of millions of lines of code in an entirely automatic manner. In Section 4 we conclude this tutorial and present some findings based on our experience in software renovation research.

Initially triggered by concerns regarding the renovation of our own software, we have since 1996 closely cooperated with several Dutch banks, and (inter)national software houses and telecommunications firms on the question how to prepare their software system assets for future flexibility. The work presented here is directly driven by their industrial needs.

1.1 Setting the stage

Is there enough legacy software in the world to justify investments in software renovation technology? It turns out that we are living on a software *volcano*: large numbers of

Language	Statements/FP
Assembler	320
C	128
Fortran77	107
Cobol85	107
C++	64
Visual Basic	32
Perl	27
Smalltalk	21
SQL	13

(a)

Language	Used in % of total
COBOL	30
Assembler	10
C	10
C++	10
500 other languages	40

(b)

Table 1. (a) Function Points *versus* Lines of Code; (b) Distribution of languages

new and old software systems control our lives. We admire the sheer bulk of the magnificent volcano, benefit from the fertile grounds surrounding it, yet at the same are suffering from frequent eruptions of lava, steam, and poisonous gas, uncertain what is going on within the volcano, and when the next large eruption will be.

The figures collected by Jones [31] provide insight in the size of the problem. He uses the *function point* (FP) as unit of measurement for software. It abstracts from specific programming languages and specific presentation styles of programs. The correlation between function points with the measurement in lines of code differs per programming language, and is summarized in Table 1(a). Another point of reference is that the size of Windows 95 is equal to 8.5×10^4 FP.

The *total volume of software* is estimated at 7×10^9 FP (7 *Giga-FP*). The distribution of the various programming languages used to implement all these function points is summarized in Table 1(b). Older languages dominate the scene: even today 30% of the 7 Giga-FP is written in COBOL. If we (hypothetically) assume that all software is written in COBOL we get an estimation (via 107 COBOL statements per FP) of 6.4×10^{11} COBOL statements for the total volume of software.

As measure of software quality (or rather, the lack of it), Jones has estimated that on average 5 errors occur per function point. This includes errors in requirements, design, coding, documentation and bad fixes. The result is a frightening figure of 35×10^9 programming errors (35 *Giga-bugs*) waiting for a chance to burst out sooner or later.

Developing better ways of developing *new* software will not solve this problem. When an industry approaches 50 years of age—as is the case with computer science—it takes more workers to perform maintenance than to build new products. Based on current data, Table 2 shows extrapolations for the number of programmers working on new projects, enhancements and repairs. In the current decade, four out of seven programmers are working on enhancement and repair projects. The forecasts predict that by 2020 only one third of all programmers will be working on projects involving the construction of new software.

Therefore, we must conclude that the importance of maintenance and gradual improvement of software is ever increasing and deserves more and more attention both in computer science education and research.

Year	New projects	Enhancements	Repairs	Total
1950	90	3	7	100
1960	8500	500	1000	10000
1970	65000	15000	20000	100000
1980	1200000	600000	200000	2000000
1990	**3000000**	**3000000**	**1000000**	**7000000**
2000	4000000	4500000	1500000	10000000
2010	5000000	7000000	2000000	14000000
2020	7000000	11000000	3000000	21000000

Table 2. Forecasts for numbers of programmers (worldwide) and distribution of their activities

1.2 Goals of this tutorial

The goals of this tutorial are to

- give the reader a quick introduction to the field of software renovation as a whole;
- show that many techniques from compiler technology and formal methods can be applied;
- demonstrate that research should be driven by real-life, industrial, case studies;
- indicate that many challenging problems are still unsolved.

We will present the approach we have taken in Amsterdam to solve a variety of problems in the area of system renovation. In the remainder of this introduction we will now first define what software renovation is (Section 1.3), sketch the technological infrastructure we use (Section 1.4), and give pointers for further reading (Section 1.5).

1.3 What is software renovation?

Chikofsky and Cross [17] have proposed a terminology for the field of re-engineering. The term *reverse engineering* has its origins in hardware technology and denotes the process of obtaining the specification of a complex hardware system. Today the notion of reverse engineering is also applied to software. While *forward engineering* goes from a high-level design to a low-level implementation, *reverse engineering* can be seen as the inverse process. It amounts to analyzing a software system to both identify the system's components and their interactions and to represent the system on a higher level of abstraction.

This higher level of abstraction can be achieved by filtering out irrelevant technical detail, or by combining legacy code elements in novel ways. Alternatively, it can be realized by recognizing instances of a library of higher level *plans* in the program code [44, 55, 23]. This latter technique is in particular applied to the problem of *program comprehension*, which aims at explaining pieces of source code to (maintenance) programmers. Techniques from the debugging and program analysis area, such as *slicing* [53], can be used for this purpose. The problem of explaining the overall *architecture* of a legacy system, indicating all the components and their interrelationships, is referred to as *system understanding*.

Adaptation of a system is the goal of *system renovation*. This can be done in one of two ways. The first is *system restructuring*, which amounts to transforming a system from one representation to another at the same level of abstraction. An essential aspect of restructuring is that the semantic behaviour of the original system and the new one remain the same; no modification of the functionality is involved. The alternative way is to perform the renovation via a reverse engineering step, which is called *re-engineering*: first a specification on a higher level of abstraction is constructed, then a transformation is applied on the design level, followed by a forward engineering step based on the improved design. A *renovation factory* is a collection of software tools that aim at the fully automatic renovation of legacy systems by organizing the renovation process as an assembly line of smaller, consecutive, renovation steps.

Last but not least, one can distinguish *methodology* and *technology* for system renovation. The former deals with process and management aspects of renovation and typically identifies phases like system inventory, strategy determination, impact analysis, detailed planning, and conversion. The latter provides the necessary techniques to implement the steps prescribed by the methodology. Although methodology and technology form a symbiosis, we will here mostly concentrate on the technological aspects of system renovation. In this tutorial, we will deal with system understanding (Section 2) as well as system renovation (Section 3).

1.4 ASF+SDF

The technical infrastructure we will use for renovation is primarily based on the ASF+SDF Meta-Environment. The specification formalism ASF+SDF [3, 29, 19] is a combination of the algebraic specification formalism ASF and the syntax definition formalism SDF. ASF+SDF specifications consist of modules, each module has an SDF-part (defining lexical and context-free syntax) and an ASF-part (defining equations). The SDF part corresponds to signatures in ordinary algebraic specification formalisms. However, syntax is not restricted to plain prefix notation since arbitrary context-free grammars can be defined. The syntax defined in the SDF-part of a module can be used immediately when defining equations, the syntax in equations is thus *user-defined*. The equations in an ASF+SDF specification have the following distinctive features:

- Conditional equations with positive and negative conditions.
- Non left-linear equations.
- List matching.
- Default equations.

It is possible to execute specifications by interpreting the equations as conditional rewrite rules. The semantics of ASF+SDF is based on innermost rewriting. Default equations are tried when all other applicable equations have failed, because either the arguments did not match or one of the conditions failed.

One of the powerful features of the ASF+SDF specification language is list matching. The implementation of list matching may involve backtracking to find a match that satisfies the left-hand side of the rewrite rule as well as all its conditions. There is only backtracking within the scope of a rewrite rule, so if the right-hand side of the rewrite rule is normalized and this normalization fails *no* backtracking is performed to find a new match.

The development of ASF+SDF specifications is supported by an interactive programming environment, the ASF+SDF Meta-Environment [32]. In this environment specifications can be developed and tested. It provides syntax-directed editors, a parser generator, a pretty printer generator, and a rewrite engine. Given this rewrite engine terms can be reduced by interpreting the equations as rewrite rules. An overview of industrial applications of the system can be found in [6, 10].

1.5 Further Reading

General introductions to renovation are books on data reverse engineering [1], the migration of legacy systems [14], the transition to object technology [49], and software maintenance [43]. An annotated bibliography of current renovation literature can be found in [9]. The following workshops and conferences are of interest:

- *Working Conferences on Reverse Engineering* [5].
- *European Conferences on Maintenance and Reengineering* [39].
- *International Workshops on Program Comprehension* [52].
- *International Conferences on Software Maintenance* [2].

Regularly, sessions on maintenance and renovation occur in general software engineering conferences. Another useful source is the *Journal of Software Maintenance*. Other relevant information includes:

- An on-line database of publications on renovation (including abstracts)[1].
- A CASE tool vendor list[2] (useful for tools that can be used in reverse engineering).
- The home page of the SEI Reengineering Centre [3].
- An overview[4] of the Georgia Tech reverse engineering group presenting papers, tools and pointers to other groups.
- A description[5] of the research activities related to program comprehension and reengineering performed at the CARE (Computer-Aided Reengineering) Laboratory in the Computer Science Department of Tennessee Technological University.

2 Object Identification

A key aspect of software renovation is *modernization*: letting a legacy system, developed using the technology of decades ago, benefit from current advancements in programming languages. In this section, we will look at techniques that can help when going from a system developed following the traditional, procedural methodology, towards a system set up according to the principles of object orientation.

---

[1] `http://www.informatik.uni-stuttgart.de/ifi/ps/reengineering/`

[2] `http://www.qucis.queensu.ca/Software-Engineering/vendor.html`

[3] `http://www.sei.cmu.edu/reengineering/`

[4] `http://www.cc.gatech.edu/reverse/`

[5] `http://www.csc.tntech.edu/~linos/`

2.1 Challenges

A transition from a traditional COBOL environment to an object oriented platform should enhance a system's correctness, robustness, extendibility, and reusability, the key factors affecting software quality[6]. Moreover, object technology is an enabler for *componentization*: splitting a large application into reusable components, such that they can be accessed independently, and possibly replaced incrementally by commercial off-the-shelf components. Sneed discusses several other, highly practical, reasons for renovation[7] [49]. In short, finding objects[8] in legacy systems is a key research area in software renovation.

The literature reports several systematic approaches to object identification, some of which can be partially automated, such as [42, 36, 18, 40, 26, 54]. These typically involve three steps: (1) identify legacy records as candidate classes; (2) identify legacy procedures or programs as candidate methods; (3) determine the best class for each method via some form of cluster analysis [34].

There are several problems, however, with the application of these approaches to actual systems.

1. Legacy systems greatly vary in source language, application domain, database system used, etc. It is not easy to select the identification approach best-suited for the legacy system at hand.
2. It is impossible to select a *single* object identification approach, since legacy systems typically are heterogeneous, using various languages, database systems, transaction monitors, and so on.
3. There is limited experience with actual object identification projects, making it likely that new migration projects will reveal problems not encountered before.

Thus, when embarking upon an object identification project, one will have to select and compose one's own blend of object identification techniques. Moreover, during the project, new problems will have to be solved.

In this section, we will look at three object identification issues in more detail: support for providing legacy system understanding (Section 2.2), ways to find types in a COBOL system (Section 2.3), and techniques for combining the types found with selected legacy functionality, thus arriving at candidate classes (Section 2.4).

2.2 System Understanding

Finding meaningful objects in a fully automatic way is impossible. The higher the level of automation, the stronger the components found will rely on the actual technical im-

---

[6] Of course, object-technology will create its own renovation problems since features like multiple inheritance, unwieldy class hierarchies, and polymorphism severely complicate program analysis. In this tutorial, we will not further explore this interesting topic.

[7] As an example, an additional reason is that it will become more and more difficult to find mainframe maintenance programmers, since young programmers coming from university are not willing to learn about old technology, but want to work using modern languages instead.

[8] Strictly speaking, we search for *classes*. The term *object identification*, however, is so commonly used that we stick to this terminology.

[P9809] R. Wedemeijer. *Introduction & Basic Tooling for* CASL *using* ASF+SDF.

[P9808] J.A. Bergstra and A. Ponse. *Two Recursive Generalizations of Iteration in Process Algebra.*

[P9807] M.P.A. Sellink and C. Verhoef. *An Architecture for Software Maintenance.*

[P9806] M. B. van der Zwaag. *Some Verifications in Process Algebra with Iota.*

[P9805] M.P.A. Sellink and C. Verhoef. *Development, Assessment, and Reengineering of Language Descriptions.*

[P9804] M.P.A. Sellink and C. Verhoef. *Native Patterns.*

[P9803] J.J. Brunekreef. *Annotated Algebraic Specification of the Syntax and Semantics of the Programming Language* Alma-0.

[P9802] W. Fokkink and C. Verhoef. *Conservative Extension in Positive/Negative Conditional Term Rewriting with Applications to Software Renovation Factories.*

[P9801] P. Klint and C. Verhoef. *Evolutionary Software Engineering: A Component-based Approach.*

[P9726] J.A. Bergstra and A. Ponse. *Grid Protocol Specifications.*

[P9725] J.A. Bergstra and A. Ponse. *Process Algebra Primitives for File Transfer.*

[P9724] J.A. Bergstra and A. Ponse. *Process Algebra with Four-Valued Logic.*

[P9723] J.A. Bergstra and A. Ponse. *Kleene's Three-Valued Logic and Process Algebra.*

[P9722] J.A. Bergstra and A. Ponse. *Bochvar-McCarthy Logic and Process Algebra.*

[P9721] M.P.A. Sellink and C. Verhoef. *Reflections on the Evolution of COBOL.*

[P9720] M. van der Graaf. *A Specification of Box to HTML in ASF+SDF.*

[P9719] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.*

B. Diertens. *Simulation and Animation of Process Algebra Specifications.*

[P9712] V. Partington. *Implementation of an Imperative Programming Language with Backtracking.*

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification of Rewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless Generalized-LR Parsing.*

[P9706] E. Visser. *A Family of Syntax Definition Formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

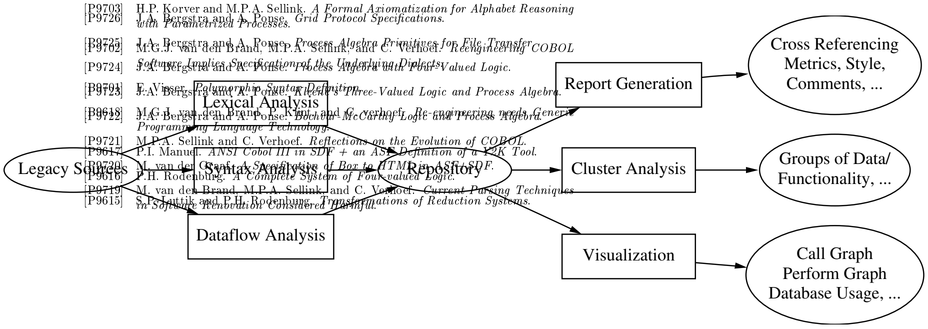
[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*



ways of program visualization are discussed by Eick [24], who, for example, is able to visualize extremely large code portfolios by representing each source line by just one colored pixel.

The main benefit of this three-phase tooling approach is that the repository permits arbitrary querying, making it possible to apply the tool set to a wide variety of renovation problems. Generally speaking, a system understanding session will iterate through these three phases, using, for example, visualization to see which filtering techniques to apply in the next iteration.

Obviously, the application of system understanding tools goes beyond mere object identification: other possibilities include generation of (interactive) documentation, quality assessment, and introducing novice programmers to a legacy application.

2.3 Type Inference

Many object identification methods work by grouping procedures based on the type of the arguments they process [34]. Unfortunately, COBOL is an untyped language, blocking this route for object identification purposes. To remedy this problem, we propose to infer types for COBOL variables based on their actual usage [22]

At first sight COBOL may *appear* to be a typed language. Every variable occurring in the statements of the procedure division, must be declared in the data division first. A typical declaration may look as follows:

```
01 TAB100.
   05 TAB100-POS    PIC  X(01) OCCURS 40.
   05 TAB100-FILLED PIC S9(03) VALUE 0.
```

Here, three variables are declared. The first is `TAB100`, which is the name of a record consisting of two fields: `TAB100-POS`, which is a single character byte (picture “`X`”) occurring 40 times, i.e., an array of length 40; and `TAB100-FILLED`, which is an integer (picture “`9`”) comprising three bytes initialized with value zero.

Unfortunately, the variable declarations in the data division suffer from a number of problems, making them unsuitable to fulfill the role of types. First of all, since it is not possible to separate type definitions from variable declarations, when two variables for the same record structure are needed, the full record construction needs to be repeated. This violates the principle that the type hides the actual representation chosen.

Besides that, the absence of type definitions makes it difficult to group variables that represent the same kind of entities. Although it might well be possible that such variables have the same byte representation, the converse does not hold: One cannot conclude that whenever two variables share the same byte representation, they must represent the same kind of entity.

In addition to these important problems pertaining to type definitions, COBOL only has limited means to accurately indicate the allowed set of values for a variable (i.e., there are no ranges or enumeration types). Moreover, in COBOL, sections or paragraphs that are used as procedures are type-less, and have no explicit parameter declarations.

To solve these problems, we have proposed the use of *type inference* to find types for COBOL variables based on their actual *usage* [22]. We start with the situation that every variable is of a unique primitive type. We then generate equivalences between these types based on their usage: if variables are compared using some relational operator, we infer that they must belong to the same type; and if an expression is assigned to a variable, the type of the expression must be a subtype of that of the expression.

Primitive Types We distinguish three primitive types: (1) elementary types such as numeric values or strings; (2) arrays; and (3) records. Initially every declared variable gets a unique primitive type. Types are made unique by encoding a label into them: since variable occurrences must have unique names in a COBOL program (module), the variable names can be used for this. We qualify variable names with program names to obtain uniqueness at the system level. We use T_A to denote the primitive type of variable A.

Type Relations By looking at the *expressions* occurring in statements, an *equivalence relation* between primitive types can be inferred. The following cases are distinguished:

- For relational expressions such as $v = u$ or $v \leq u$, an equivalence between T_v and T_u is inferred.
- For arithmetic expression such as $v + u$ or $v * u$, an equivalence between T_u and T_v is inferred.
- For two different array accesses $a[v]$ and $a[u]$ an equivalence between T_v and T_u is inferred.
- From an assignment $v := u$ we infer that T_u is a *subtype* of T_v,

By *type*, we will generally mean an *equivalence class of primitive types*. Subtyping is important to avoid the problem of *pollution*, the derivation of counter-intuitive equivalences due to commutativity and transitivity of the equivalence relation [22].

System-Level Analysis In addition to inferring type relations within individual programs, we infer type relations at the system-wide level. Such relations ensure that if a variable is declared in a copybook (include file), its type is the same in all the different programs that copybook is included in. Furthermore, we infer that the types of the actual parameters of a program call at the module level (listed in the `USING` clause) are subtypes of the formal parameters (listed in the linkage section), and that variables read from or written to the same databases have equivalent types.

Related Work Type inference for COBOL is related to earlier work on type inference for C [41] and on various approaches for detecting and correcting year 2000 problems [28]. An approach for dealing with the year 2000 problem based on type theory is presented by [25]. A detailed overview of related work is given in [22].

Clearly, type inference for COBOL has applications beyond mere object identification: in [22] we explain how types can also be used to replace literal values by symbolic constants, year 2000 and Euro conversions, language migrations, and maintenance monitoring.

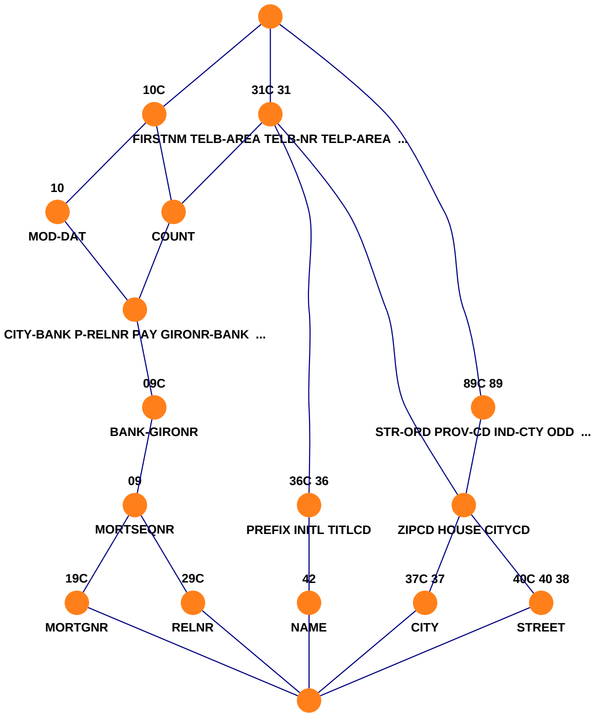


Fig. 2. Concept lattice combining data fields with programs

2.4 Concept Analysis

For many business applications written in COBOL, the data stored and processed represent the core of the system. For that reason, the data records used in COBOL programs are the starting point for many object identification approaches (such as [18, 40, 26]). These records are then in turn combined with procedures or programs, thus arriving at candidate classes. A common way of finding the desired combinations is to use *cluster analysis* [34].

Recently, the use of mathematical *concept analysis* has been proposed as an alternative to the use of cluster analysis for the purpose of legacy code analysis [35, 48, 50, 51]. As has been argued in [21], concept analysis avoids some of the problems encountered when using cluster analysis for the purpose of object identification.

Concept analysis starts with a table indicating the *features* of a given set of *items*. It then builds up so-called *concepts* which are maximal sets of items sharing certain features. All possible concepts can be grouped into a single lattice, the so-called *concept* *lattice*. The smallest concepts consist of few items having potentially many different features, the largest concepts consist of many different items that have only few features in common.

An example concept lattice is shown in Figure 2. This lattice was derived automatically from a 100,000 LOC COBOL case study. The *items* in this lattice are *fields* of records that are read from or written to file. They are shown as names below each concept in Figure 2. The *features* are based on the use of fields in *programs*: If a field (or in fact, the *type* inferred for that field) is used in a given program, this program becomes a feature of that field. The programs are written above each concept in Figure 2. Each concept in that figure corresponds to a combination of fields and the programs using them, as occurring in the legacy system. Each concept is a candidate class. Connections between classes correspond to aggregation, association, or inheritance.

To see how the lattice of Figure 2 can help to find objects, let us browse through some of the concepts. The row just above the bottom element consists of five separate concepts, each containing a single field. As an example, the leftmost concept deals with *mortgage numbers* stored in the field `MORTGNR`. With it is associated program `19C`, which according to the comment lines at the beginning of this program performs certain checks on the validity of mortgage numbers. This program *only* uses the field `MORTGNR`, and no other ones. As another example, the concept `STREET` (at the bottom right) has three different programs directly associated with it. Of these, `40` and `40C` compute a certain standardized extract from a street, while program `38` takes care of standardizing street names.

If we move up in the lattice, the concepts become larger, i.e., contain more items. The leftmost concept at the second row contains *three* different fields: the *mortgage sequence number* `MORTSEQNR` written directly at the node, as well as the two fields from the lower concepts connected to it, `MORTGNR` and `RELNR`. Program `09` uses all three fields to search for full mortgage and relation records.

Another concept of interest is the last one of the second row. It represents the combination of the fields `ZIPCD` (zip code), `HOUSE` (house number), and `CITYCD` (city code), together with `STREET` and `CITY`. This combination of five is a separate concept, because it actually occurs in four different programs (`89C`, `89`, `31C`, `31`). However, there are no programs that *only* use these variables, and hence this concept has no program associated with it. It corresponds to a common superclass for both the `89C,89` and the `31,31C` concepts.

In short, the lattice provides insight into the organization of the legacy system, and gives suggestions for grouping programs and fields into classes. The human re-engineer can use this information to select initial candidate classes based on the data and functionality available in the legacy.

The crucial step with both cluster and concept analysis is to apply the correct filtering criteria, in order to reduce the overwhelming number of variables, sections, programs, databases, and so on, to the relevant ones. Such selection criteria may differ from system to system, and can only be found by trying several alternatives on the system being investigated—this is exactly where the system understanding tool set discussed in Section 2.2 comes in. The selection criteria used to arrive at Figure 2 are based on persistent data and metrics derived from the call relation, as discussed in [20, 21].

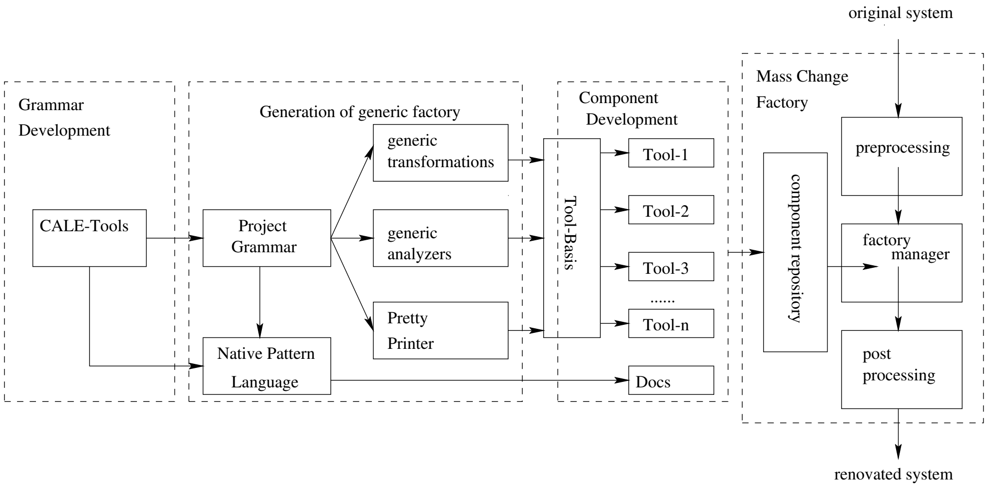


Fig. 3. Four phases in the creation and use of software renovation factories

3 Renovation Factories

As soon as the architecture of a legacy system has been recovered and the overall understanding of the system has been increased using the techniques described in the previous sections, we are in a position to determine what should be *done* with it. Should it be abandoned or renovated? In reality, there is a close interplay between analysis and renovation since the analysis of well-structured code will yield more precise analysis results than the analysis of badly structured code. It is not uncommon that analysis/renovation is an iterative process. For simplicity we treat them in this section as sequential steps and concentrate on a factory-like approach to renovation (Section 3.1). Next, we discuss two examples (Section 3.2).

Unlike system understanding, which is inherently interactive, here the ultimate goal is a completely automated renovation factory, which can process millions of lines of code without human interaction.

3.1 Creation and use of renovation factories

In practice, there are many needs for program transformations and program restructuring like simple global changes in calling conventions, migrations to new language dialects, goto elimination, and control flow restructuring. Since these transformations will affect millions of lines of code, a factory-like approach, with minimal human intervention, is desired to achieve a cost-effective, high-quality, solution. Recall from Section 1.3, that we define a renovation factory as a collection of software tools that aim at the fully automatic renovation of legacy systems by organizing the renovation process as an assembly line of smaller, consecutive, renovation steps.

Legacy systems show a lot of variety regarding their overall architecture, programming languages used, database organization, error handling, calling conventions, and the like. Experience shows that each renovation project is unique and requires an extensively tailored approach. The generation and customization of renovation factories is therefore a major issue. In order to promote flexibility and reusability, renovation factories should be built-up from from individual tools that are as small and general as possible. Our approach is illustrated in Figure 3 and consists of four phases:

- Grammar development: determine the project grammar.
- Factory generation: generate a generic framework for the renovation factory. This amounts to automatically deriving generic tools (parser, pretty printer, frameworks for analysis and transformation) from the project grammar.
- Component development: develop dedicated tools for the factory that are specific for this project.
- Factory production environment: configure and run the factory.

We will now discuss these phases in turn.

Grammar development Grammars form the basis for our factory generation approach. However, in practice it is completely non-trivial to obtain and manage such grammars. The grammar may be included in an international standard, it may be contained in a manual for a proprietary language, or it may be embedded in the source code of existing tools such as compilers or pretty printers [46]. Another problem is that these grammars tend to be huge (several thousands of grammars rules) since they cover various language dialects as well as various local language extensions. From a grammar maintenance perspective, standard LR parsing techniques become unsuitable here, since they tend to generate too many shift/reduce conflicts after each modification to the grammar. We completely depend on the Generalized LR parsing method provided by the ASF+SDF Meta-Environment which is capable of handling arbitrary context-free grammars. It is a research issue how to obtain grammars for languages in a cost-effective manner. We have labeled this activity *Computer Aided Language Engineering* (CALE) and have a collection of tools for grammar extraction and manipulation.

Factory Generation Once the project grammar has been determined, we can focus on the automatic generation of components that are language parameterized such as frameworks for generic analysis, transformation, and pretty printing [13]. Since we are working in a first-order setting, we cannot use higher-order analysis and transformation functions. Instead, we generate from the grammar specific analysis and transformation functions which perform a default operation for each language construct. This default behaviour can be overruled by writing conditional equations that define the desired behaviour for specific language constructs.

This immediately raises the question, who should write these equations: a formal specification expert or a programmer knowledgeable in the legacy languages of the project? It is clearly necessary to be able to describe certain patterns in the code to facilitate analysis and automatic change. The approach we take is to automatically generate a *pattern language* [47] from the project grammar that resembles the language defined by the project grammar as much as possible. Thanks to the tight integration between abstract and concrete syntax in ASF+SDF, we can express patterns over the abstract syntax tree using the concrete syntax of the language being reengineered. In this way, a programmer knowledgeable in that language can specify search and replacement patterns in conditional equations without needing to know all the formal machinery that is used in the factory as a whole.

It is important to have complete control over what steps are taken during a (automated) renovation process and over how their intermediate results are handled. We automatically generate support for a form of "scaffolding" which is similar to the inclusion of pseudo-code or assertions in comments during traditional code development. In our case, we see a scaffolding as an extension to the project grammar that allows the representation of intermediate results of analysis or transformation as annotations of the program code.

Component prototyping As already mentioned, renovation factories should be built-up from individual tools that are as small and general as possible. This results not only in better flexibility and reuse, but also in increased control over and understanding of each tool. Apart from general tools for bringing program parts in certain standard forms and for performing program simplifications, we have developed tools for upgrading embedded SQL, for normalizing the control flow of embedded CICS, and for step by step restructuring of COBOL code [11, 45].

Factory Production Environment The final phase in our approach is actually building the renovation factory. The components that have been developed and prototyped need to be put into operation in an efficient production environment.

Given the size of legacy systems, issues like scalability and multi-processor execution of the factory are now also coming into play. Important supporting technologies are compilers that turn prototype specifications into efficient stand-alone C programs [7] and middleware that is optimized for the connection of language-oriented tools [4]. References [8, 33, 10, 15] give overall pictures and further discussions of renovation factories.

3.2 Two examples

We will now discuss two examples. The first example concerns the grammar extraction and factory generation for a proprietary language for switching systems. The second example illustrates the use of a COBOL factory for extracting business logic from programs by separating computation from coordination.

Grammar development for a switching system's language Most software and hardware for switching systems is proprietary. This includes central processing units, compilers and operating systems that have been developed in-house. For instance, Lucent Technologies, uses UNIX and C targeted towards their own proprietary processor. They have to maintain their own UNIX and their own C compiler. The same phenomenon occurs at Ericsson: they have developed their own central processor and their own operating systems, languages and compilers. The difference between the two situations is that

Ericsson uses tools that are *not* widely used for other processors. As a result, software renovation tools are not available in large amounts for their software. The Ericsson case is therefore an ideal test case for our approach.

As described in the previous section, the first step in generating a renovation factory is the development of a relevant project grammar. In this case, it concerns the proprietary language SSL (Switching System Language). A program in SSL is in fact a collection of programs in 20 different languages that are combined into a single file.

The only complete and reliable source for the SSL grammar is the source code of the SSL compiler. Several steps are needed to extract this grammar. First, the compiler source code is reverse engineered so that the grammar part is identified. Then the essential grammar information is extracted from this part of the compiler. This resulted in the extraction of more than 3000 lexical and context-free production rules in Backus-Naur Form (BNF). Unfortunately, this grammar is not yet usable for reengineering since it is too heavily oriented towards compilation. For instance, the compilation-oriented grammar removes comments, whereas during renovation, the comments should be seen as part of the source since they have to be retained in the final result of the renovation.

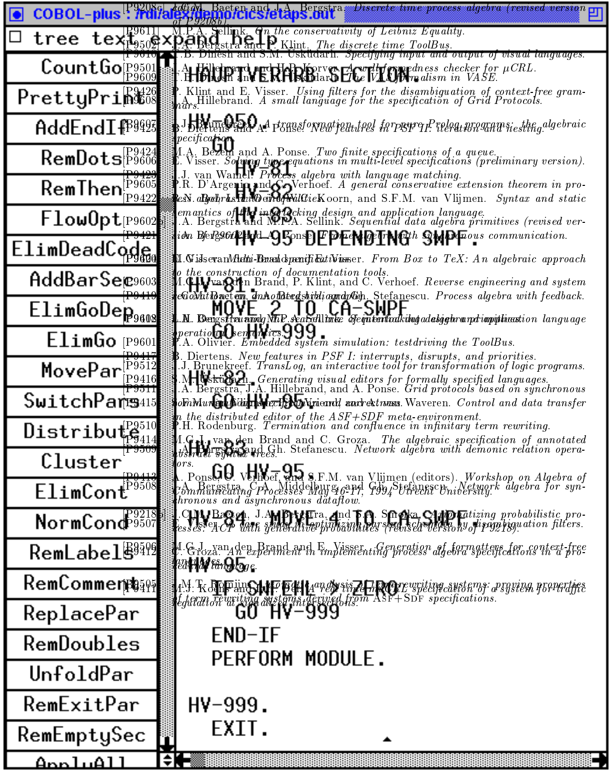
As a preparatory step for the reengineering of this intermediate SSL grammar, we have generated a renovation factory for the language BNF itself and have added a number of components that are useful for the reengineering of grammars, such as a transformation of BNF into SDF (the syntax definition language of the ASF+SDF Meta-Environment). This BNF-factory has been used to retarget the extracted SSL grammars.

Using the BNF-factory, we could transform the grammar of one of the sublanguages of SSL in 7 minutes processing time (on a standard workstation) from its initial BNF version into an SDF version that is usable for renovation purposes. Next, we were able to parse about 1 MLOC in this sublanguage in about 70 minutes processing time. Finally, a complete renovation factory for this sublanguage could be generated. When completed with the desired renovation components, it can be used for the factory-like renovation of programs in this particular SSL subset.

Currently, the complete SSL grammar is being assembled by combining the 20 embedded subgrammars. As a next step, we will generate a complete SSL-factory. When that SSL-factory is ready, we have in fact generated from the compiler source code a production line for the rapid development of component-based tools that facilitate the automated solution of software reengineering problems.

Separating coordination and computation in COBOL A typical renovation problem is to migrate transaction systems from a mainframe to a networked PC environment. Such systems are strongly tied to mainframes by the use of embedded languages like CICS that deal with transaction monitors and the coordination of data. In order to migrate to a client/server architecture, it is therefore necessary to remove this embedded CICS code. Once that is done, the code needs to be made more maintainable by removing all traces of the platform migration from the source code.

In the example that we will now discuss, a German company (SES GmbH) had already eliminated some dangerous CICS code, and we have constructed a migration tool to separate coordination from computation so that all CICS code could be removed. One of the main problems was to eliminate jump instructions from the code. Figure 4 shows



```
HAUPTVERARB SECTION.

HV-050.
    GO
      HV-81
      HV-82
      HV-83
      HV-95 DEPENDING SWPF.

HV-81.
    MOVE 2 TO CA-SWPF
    GO HV-999.

HV-82.
    GO HV-95.

HV-83.
    GO HV-95

HV-84. MOVE 4 TO CA-SWPF.

HV-95.
    IF SWFEHL > ZERO
      GO HV-999
    END-IF
    PERFORM MODULE.

HV-999.
    EXIT.
```

Fig. 4. Original COBOL code with `GO` statements

a strongly simplified version of the "spaghetti" code of the original program. The code fragment is representative of the overall quality of the code that can be found on mainframes. The first `GO ... DEPENDING ...` is actually a case statement, and contains four cases that are all `GO`s. So in fact in this small fragment we have 8 jump instructions.

Using a restructuring method based on a systolic algorithm we can remove the `GO`s in about 25 steps in such a way that the coordination and computation are separated, and the logic of the program becomes more clear. In Figure 5 the final output is shown. As can be seen, the code has been changed dramatically. Superfluous code is gone and coordination is separated from computation.

Paragraph `HV-050` provides the coordination part of the program and resembles a `main` program in C. The `EVALUATE ... END-EVALUATE` statement is the result of migrating the original `GO ... DEPENDING ...` statement; the original four cases could be collapsed into two cases. In the `HAUPTVERARB-SUBROUTINES` section, we can see that two computations are present. They are only reachable via the coordination part, since the `STOP RUN` blocks other access. This is comparable to a preamble in say Pascal where procedures are declared.

[illegible]

[P9406] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. *A propositional logic with 4 values:* [illegible]

[P9316] [illegible] *algebraic specification of a model factory, part IV.*

[P9211b] J.A. Hillebrand. *The ABP and the CABP - a comparison of performances in real* [illegible]

[P9315] [illegible] *losing comments.*

[illegible]

[P9305b] J.C.M. Baeten and J.A. Bergstra. *On sequential composition, action prefixes and process prefix* [illegible]

[illegible]

[P9308] J.F.A.K. van Benthem and J. [illegible] *systems.*

[P9330] K.M.M. de Leeuw and H. van der Meer. *A turning grille from the ancestral castle of the Dutch Stadtholders.*

[illegible]

[P9329] J.C.M. Baeten and J.A. Bergstra. *On sequential composition, action prefixes and process prefix.*

[P9304] S.F.M. van Vlijmen and A. van Waveren. *On generating synchronous interworkings* [illegible]

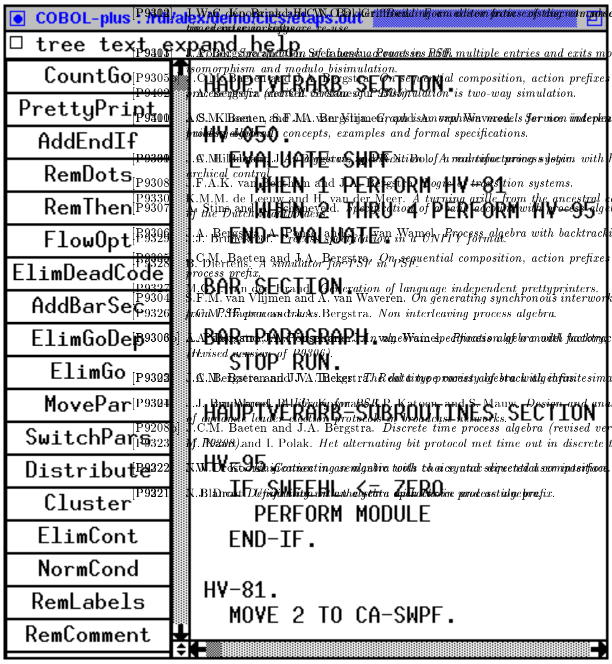
[P9326] [illegible] Bergstra. *Non interleaving process algebra.*

[illegible]

[P9208b] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version* [illegible]

[P9323] [illegible] and I. Polak. *Het alternating bit protocol met time out in discrete tijd.*

[illegible]



```
 HAUPTVERARB SECTION.

 HV-050.
   EVALUATE SWPF
   WHEN 1 PERFORM HV-81
   WHEN 2 THRU 4 PERFORM HV-95
   END-EVALUATE.

 BAR SECTION.

 BAR-PARAGRAPH.
   STOP RUN.

 HAUPTVERARB-SUBROUTINES SECTION.

 HV-95.
   IF SWEEHL <= ZERO
     PERFORM MODULE
   END-IF.

 HV-81.
   MOVE 2 TO CA-SWPF.
```

Fig. 5. Final COBOL code with all GO's removed

An original program and all intermediate steps that are carried out to remove the jump instructions and to separate computations from coordination are available on Internet.[9] We discuss this kind of restructuring in more detail in [45].

4 Conclusions

In this tutorial, we have covered a variety of issues in the area of system renovation. Starting with a discussion on the economic need for maintenance and renovation, we have presented our approaches to problems like object identification, system understanding, grammar reengineering, and the creation of renovation factories.

Several observations can be made about the field of software renovation as a whole. A most challenging aspect of software renovation research is the number of different areas in which proficiency is required. These include:

- Historic programming languages and systems, such as COBOL and the mainframe environment.
- Target new programming language technologies, such as CORBA or Java, and the current and future market standards.

---

[9] An on-line demonstration of all steps involved is available at `http://adam.wins.uva.nl/~x/systolic/systolic.html`.

- Migration technology, such as the ASF+SDF Meta-Environment.
- Software engineering theory and commercial practice.
- Knowledge transfer, coping with conservatism ("COBOL is the best"), unrealistic expectations of new technology ("Java will solve everything"), and unfamiliarity with migration technology ("What did you say a grammar was?").

Clearly, this should also have an effect on the curricula for software engineering.

Concerning the migration technology used, we benefited from the use of the ASF+SDF Meta-Environment. Three of its distinctive technical properties turned out to be of great significance for renovation purposes:

- The techniques used are *generic*: they do not depend on one particular programming language, such as COBOL or PL/I, but they are parameterized with a language definition. As a result, major parts of our renovation techniques can be directly reused for different languages. To give an example, the Year 2000 problem resides probably in systems written in 500 "standard" languages [31] plus another 200 proprietary languages [30].
- The techniques used are *formal*: the underlying formal notions are many-sorted signatures and positive/negative conditional rewriting. Program conversions are, for instance, expressed as rewrite rules. This formalization increases the quality of analyses and conversions [27].
- Syntactic analysis is based on *generalized LR parsing* (GLR). This enables the construction of modular grammars and parsers for languages with large grammars and many dialects (like COBOL) [12]. More traditional parsing techniques (e.g., LALR(1) as used in parser generators like Yacc and Bison) lead to increased maintenance problems for large grammars: the time needed to add language constructs needed for new dialects becomes prohibitive.

The driving force behind software renovation is the strong need to maintain and renovate parts of the software volcano. This implies that software renovation research has to be carried out in close cooperation with industrial partners that are in the possession of problems that *have* to be solved. Fortunately, we can be confident that progress in areas like compiler and programming language technology and formal methods, will continue to help to offer the right tools at the right time. At the same time, the analysis of legacy systems provides the empirical foundation for programming language research. It uncovers the effects, both positive and negative, of years of intensive use of a programming language in the real world.

As we have tried to show here, software renovation research also reveals new challenging problems. Techniques for analysis or code generation that were satisfactory from the perspective of a traditional compiler may no longer be satisfactory from the perspective of interactive program understanding. The gigantic scale of renovation projects presents implementation problems (and opportunities) that may inspire research for many years to come.

Finally, the largest challenge we see is to try to bridge the gap between research aimed at building *new* software and research aimed at maintaining or renovating old software. We strongly believe that an integrated approach to both is the best way to proceed. This implies introducing maintenance and renovation considerations much earlier in the software construction process than is usual today. This also implies designing new languages and programming environments that are more amenable to maintenance and renovation.

Acknowledgments

Although the three of us wrote this text, it represents the work of many colleagues with whom we have cooperated or are cooperating. We thank them all for their direct or indirect contributions to the current paper. Tobias Kuipers commented on initial drafts of this tutorial.

References

1. P.H. Aiken. *Data Reverse Engineering*. McGraw-Hill, 1996.
2. K. Bennett and T.M. Khoshgoftaar, editors. *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, November 1998.
3. J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.
4. J.A. Bergstra and P. Klint. The discrete time TOOLBUS—a software coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.
5. M.H. Blaha, A. Quilici, and C. Verhoef, editors. *Proceedings of the Fifth Working Conference on Reverse Engineering*. IEEE Computer Society, October 1998.
6. M. G. J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.
7. M.G.J. van den Brand, P. Klint, and P. Olivier. Compilation and memory management for ASF+SDF. In *Proceedings of the 8th International Conference on Compiler Construction, CC'99*, LNCS. Springer-Verlag, 1999. To appear.
8. 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.
9. M.G.J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation – an annotated bibliography. *ACM Software Engineering Notes*, 22(1):57–68, 1997.
10. M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Springer-Verlag, 1998.
11. M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In P. Nesi and F. Lehner, editors, *Proc. 2nd Euromicro Conf. on Maintenance and Reengineering*, pages 11–19. IEEE Computer Society, 1998.
12. M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proc. Sixth International Workshop on Program Comprehension*, pages 108–117. IEEE Computer Society, 1998.
13. M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.
14. M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, interfaces and the incremental approach*. Morgan Kaufman Publishers, 1995.

15. J. Brunekreef and B. Diertens. Towards a user-controlled software renovation factory. In P. Nesi and C. Verhoef, editors, *Proc. Third European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 1999. To Appear.
16. 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.
17. E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.
18. A. Cimitile, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino. Identifying objects in legacy systems using design metrics. *Journal of Systems and Software*, 1998. To appear.
19. 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.
20. 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.
21. A. van Deursen and T. Kuipers. Finding objects using cluster and concept analysis. In *21st International Conference on Software Engineering, ICSE-21*. ACM, 1999. To appear.
22. A. van Deursen and L. Moonen. Type inference for COBOL systems. In I. Baxter, A. Quilici, and C. Verhoef, editors, *Proc. 5th Working Conf. on Reverse Engineering*, pages 220–230. IEEE Computer Society, 1998.
23. A. van Deursen, S. Woods, and A. Quilici. Program plan recognition for year 2000 tools. In *Proceedings 4th Working Conference on Reverse Engineering; WCRE'97*, pages 124–133. IEEE Computer Society, 1997.
24. S. G. Eick. A visualization tool for Y2K. *IEEE Computer*, 31(10):63–69, 1998.
25. P. H. Eidorff, F. Henglein, C. Mossin, H. Niss, M. H. Sorensen, and M. Tofte. Anno Domini: From type theory to Year 2000 conversion tool. In *26th Annual Symposium on Principles of Programming Languages, POPL'99*. ACM, 1999. To appear.
26. H. Fergen, P. Reichelt, and K. P. Schmidt. Bringing objects into COBOL: MOORE - a tool for migration from COBOL85 to object-oriented COBOL. In *Proc. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS 14)*, pages 435–448. Prentice-Hall, 1994.
27. W.J. Fokkink and C. Verhoef. Conservative extension in positive/negative conditional term rewriting with applications to software renovation factories. In *Fundamental Approaches to Software Engineering*, LNCS, 1999. To Appear.
28. J. Hart and A. Pizzarello. A scaleable, automated process for year 2000 system correction. In *Proceedings of the 18th International Conference on Software Engineering ICSE-18*, pages 475–484. ACM, 1996.
29. J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
30. C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.
31. Capers Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*. Addison-Wesley, 1998.
32. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.
33. P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In R.N. Horspool, editor, *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, pages 1–18. Chapman & Hall, 1998.
34. A. Lakhotia. A unified framework for expressing software subsystem classification techniques. *Journal of Systems and Software*, pages 211–231, March 1997.

35. C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *19th International Conference on Software Engineering, ICSE-19*, pages 349–359. ACM, 1997.
36. S. S. Liu and N. Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. In *International Conference on Software Maintenance; ICSM'90*, pages 266–271. IEEE Computer Society, 1990.
37. H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance*, 5(4):181–204, 1993.
38. G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Software Engineering Methodology*, 5(3):262–292, 1996.
39. P. Nesi and C. Verhoef, editors. *Proceedings of the Third European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, March 1999.
40. P. Newcomb and G. Kottik. Reengineering procedural into object-oriented systems. In *Second Working Conference on Reverse Engineering; WCRE'95*, pages 237–249. IEEE Computer Society, 1995.
41. R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *19th International Conference on Software Engeneering; ICSE-19*. ACM, 1997.
42. C. L. Ong and W. T. Tsai. Class and object extraction from imperative code. *Journal of Object-Oriented Programming*, pages 58–68, March–April 1993.
43. T. M. Pigoski. *Practical Software Maintenance – Best Practices for Managing Your Software Investment*. John Wiley and Sons, 1997.
44. C. Rich and R. Waters. *The Programmer's Apprentice*. Frontier Series. ACM Press, Addison-Wesley, 1990.
45. M.P.A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. In P. Nesi and C. Verhoef, editors, *Proc. Third European Confrence on Software Maintenance and Reengineering*. IEEE Computer Society, 1999. To appear.
46. M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. In *Proceedings of the 13th International Automated Software Engineering Conference*, pages 314–317. IEEE Computer Society, 1998.
47. M.P.A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the 5th Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Scociety, 1998.
48. M. Siff and T. Reps. Identifying modules via concept analysis. In *International Conference on Software Maintenance, ICSM97*. IEEE Computer Society, 1997.
49. H.M. Sneed. *Objectorientierte Softwaremigration*. Addison-Wesley, 1998. In German.
50. G. Snelting. Concept analysis — a new framework for program understanding. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, 1998. SIGPLAN Notices 33(7).
51. G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Foundations of Software Engineering, FSE-6*, pages 99–110. ACM, 1998. SIGSOFT Software Engineering Notes 23(6).
52. S. Tilley and G. Visaggio, editors. *Proceedings of the Sixth International Workshop on Program Comprehension*. IEEE Computer Society, June 1998.
53. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
54. T. Wiggerts, H. Bosma, and E. Fielt. Scenarios for the identification of objects in legacy systems. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *4th Working Conference on Reverse Engineering*, pages 24–32. IEEE Computer Society, 1997.
55. S. Woods, A. Quilici, and Q. Yang. *Constraint-based Design Recovery for Software Reengineering: Theory and Experiments*. Kluwer Academic Publishers, 1997.

[P9220] A. van Waveren. *Specification of remote sensing mechanisms in real space process algebra.*

[P9219] I. Bethke and P.H. Rodenburg. *Equational Constructor Induction.*

[P9218] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. *Axiomatizing Probabilistic Processes: ACP with Generative Probabilities.*

[P9217] A. Ponse. *Process algebra and dynamic logic.*

[P9005b] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra (revised version).*

[P9216] E.E. Polak. *An efficient implementation of branching bisimulation and distinguishing formulae.*

[P9215] J.A. Bergstra and J.V. Tucker. *Equational specifications, complete term rewriting systems, and computable and semicomputable algebras.*

[P9214] J. A. Hillebrand and A. Ponse. *An algebraic specification of a model factory, part II.*

[P9213] N.J. Drost. *Unification in the algebra of sets with union and empty set.*

[P9212] J.J. van Wamel. *A study of a one bit sliding window protocol in ACP.*

[P9211] J.A. Hillebrand. *The ABP and the CABP - a comparison of performances in realtime process algebra.*

[P9210] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. *Decidability of bisimulation equivalence for processes generating context-free languages.*

[P9209] S.F.M. van Vlijmen and A. van Waveren. *An algebraic specification of a model factory.*

[P9208] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra.*

[P9207] K. de Leeuw and H. van der Meer. *A secret message in the archive of the last great pensionary of Holland.*

[P9206] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra.*

[P9205] J.J. van Wamel. *An algebraic verification of the concurrent alternating bit protocol.*

[P9204] C. Verhoef. *On induction principles.*

[P9203] J.A. Verschuren. *A simulator for mCRL in ASF+SDF.*

[P9202] J.W.C. Koorn. *A generic text and structure editor.*

[P9201] P.H. Rodenburg. *Interpolation in Equational Logic.*

[P9102b] J.J. Brunekreef. *A formal specification of three Sliding Window Protocols (revised version of P9102).*

[P9112] A. Mateescu. *Recursive systems with auxiliary variables in Basic Process Algebra.*

[P9111] J.C.M. Baeten and J.A. Bergstra. *A survey of axiom systems for process algebras.*

[P9110] J.J. van Wamel. *A formal specification of three simple protocols.*

[P9109] J.J. Brunekreef. *A formal specification of two simple protocols for Local Area Networks.*

[P9108] J.J. Brunekreef. *A formal specification of the Amoeba Transaction Protocol.*

[P9107] G.J. Veltink. *The PSF toolkit.*

[P9106] J.C.M. Baeten and J.A. Bergstra. *Asynchronous communication in real space process algebra.*

[P9105] C. Verhoef. *An operator definition principle (for process algebras).*

[P9104] J.C.M. Baeten and J.A. Bergstra. *The state operator in real time process algebra.*

[P9103] H.P. Korver. *Computing distinguishing formulas for branching bisimulation.*

[P9102] J.J. Brunekreef. *A formal specification of three Sliding Window Protocols.*

[P9101] H.P. Korver. *The current state of bisimulation tools.*

[P9009] G.J. Veltink. *From PSF to TIL.*

[P9008] J.C.M. Baeten and J.A. Bergstra. *Process algebra with signals and conditions.*

[P9007] S. Mauw and Gy. Max. *A formal specification of the Ethernet protocol.*

[P9006] G.J. Akkerman and J.C.M. Baeten. *Term rewriting analysis in process algebra.*

[P9005] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra.*

[P9004] N.J. Drost. *Algbraic formulations of trace theory.*

[P9002b] J.C.M. Baeten and J.A. Bergstra. *Process algebra with zero a object (revised version of P9002).*

[P8804b] J.A. Bergstra and J.V. Tucker. *The inescapable stack: An exercise in algebraic specification with total functions (revised version of P8804).*

[P8910b] J.C.M. Baeten and J.A. Bergstra. *Design of a specification language by abstract syntax engineering (revised version of P8910).*

[P8906b] J.C.M. Baeten, S. Mauw J.A. Bergstra, and G.J. Veltink. *A process specification formalism based on static COLD (revised version of P8906).*

[P9003] C. Verhoef. *On the register operator.*

[P8916b] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra (revised version of P8916).*

[P9002] J.C.M. Baeten and J.A. Bergstra. *Process algebra with zero object and non-determinacy.*

[P9001] L.W. Kuurman. *The jungle of process semantics.*

[P8916] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra.*

[P8915] J.C. Mulder. *The inevitable coffee machine.*