

Towards Automated Modification of Legacy Assets

Chris Verhoef

Programming Research Group

University of Amsterdam

Kruislaan 403

1098 SJ Amsterdam

The Netherlands

Phone: +31-20-525-7581

Fax: +31-20-525-7490

Email: x@wins.uva.nl

Abstract

In this paper we argue that there is a necessity for automating modifications to legacy assets. We propose a five layered process for the introduction and employment of tool support that enables automated modification to entire legacy systems. Furthermore, we elaborately discuss each layer on a conceptual level, and we make appropriate references to sources where technical contributions supporting that particular layer can be found. We sketch the perspective that more and more people working in the software engineering area will be contributing to working on existing systems and/or tools to support such work.

Categories and Subject Description:

D.2.6 [Software Engineering]: Programming Environments—Interactive;

D.2.7 [Software Engineering]: Distribution and Maintenance—Restructuring;

D.3.4. [Processors]: Parsing.

Additional Key Words and Phrases:

Reengineering, System renovation, Software renovation factories, Legacy systems, Legacy assets.

1 INTRODUCTION

In the software IT industry, most development is about enhancing existing systems, for instance, providing new front-ends to established back-ends, capitalizing on existing relational technology for data storage, and building new interfaces to existing software assets. Organizations are more and more trying to find effective ways to reuse their investments in existing packages, data bases, and legacy assets. Componentization plays a crucial role in the opening up of reusing packages and data base technology [Allen and Frost 1998]. Realistic component-based development software engineering approaches recognize the importance of the rich body of existing software and try to bring the installed base of software in the process as an asset rather than ignoring it as a liability. One of the first books on component-based software processes was written by various authors from the Software Engineering Institute [Brown 1996]. Maybe half of the content of that book is devoted to architectural analysis, recovery, and program understanding. Also the excellent textbook [Allen and Frost 1998] on component-based development contains an entire chapter that addresses legacy systems. In such textbooks the assumption is made that technology is available to unlock legacy assets to enable reuse within the component-based software development process. Indeed software renovation plays an important role in the unlocking of legacy assets. It is the key to enable changes to software, which is often a necessity to keep in pace with the rapidly changing business goals of companies that own large information systems. Software renovation is the area where from existing systems new systems are created rather than starting from scratch. A good introduction to software renovation terminology is [Chikofsky and Cross 1990]. An annotated bibliography on the subject can be found in [Brand *et al.* 1997b]. The latest issues in reverse engineering, maintenance, and program comprehension can be found in the following conference proceedings [Blaha *et al.* 1998; Tilley and Visaggio 1998; Bennett and Khoshgoftaar 1998; Nesi and Verhoef 1999].

1.1 The Current Situation

The reality of every-day software is that the installed base is much more brittle than we could even dream of. Aging legacy information systems are very likely to be poorly structured, and are being fixed under severe schedule pressure often by inexperienced personnel. Such systems tend to have a high bad fix injection rate. No-one dares to touch such software anymore, and when this does happen, it takes a lot of time to make the modifications. Let's take a brief look at the current situation in the IT industry. Existing software is expensive. The situation in the United States provides us with some insight: on the average, the software maintenance budget of U.S. enterprises equals or exceeds 45 percent of the *total* budget [Jones 1994].

We say that a software production library where the costs and resources for maintenance and enhancement are greater than for new development has high maintenance costs. More than 60 percent of Fortune 500 U.S. enterprises have high maintenance costs [Jones 1994, p. 146]. The costs of maintenance and renovation are still increasing. The expectation is that maintenance and enhancement will become the primary work of the software industry since virtually every organization will be or is computerized. Therefore, it is not a surprise that the number of people working on maintenance and enhancement of existing systems outnumber people working on development of brand-new software systems. In the current decade four out of seven software engineers are working on repair and enhancement of existing software, and the forecasts are that this trend will continue. Due to the Euro conversion work and the Year 2000 repair efforts it is estimated that in 1999 about 80 percent of the software professionals will be engaged in various maintenance and enhancement activities [Jones 1998a, p. 595].

The most common specialism in IT industry is maintenance and/or enhancement specialist. About 25 percent of software engineers are *specialist* in maintenance and enhancement [Jones 1996, p. 223]. If we focus on the information systems world, this percentage is even higher. About 16% of the specialists is expert in the enhancement to legacy systems, 9% is maintenance specialist for defect repairs, about 5% is specialist in geriatric care of legacy assets. This totals about 30% of the specialists who are dealing with aging legacy systems. Compare this to 1% of the specialists working on rapid application development in the IS community [Jones 1996, p. 236–7]. These numbers should not be surprising, after all, the information systems community was one of the first industries to apply computer technology to their business operations, so this industry has large portfolios of aging legacy assets.

Development of, say a car, is difficult, whereas maintenance of such hardware is less a problem. In the software world the situation is different. Maintenance and enhancement are difficult issues. Gallagher and Lyle [Gallagher and Lyle 1991] postulate

While some may view software maintenance as a less intellectually demanding activity than development, the central premise [...] is that software maintenance is more demanding.

Another indication of the difficulties that come into play with software renovation is the following metaphor: software renovation is about as easy as reconstructing a pig from a sausage [Eastwood 1992]. In [Adams 1996, p. 73] we read that a good example for which you can confidently predict failure is *any* large-scale reengineering effort.

Cost estimation of software activities is a crucial aspect in the information systems community. Already in the seventies the foundations for the function point techniques were introduced by Albrecht [Albrecht 1979]. Also in this area, maintenance and enhancement is the most critical aspect of software cost estimation. Moreover such cost estimation is much more difficult than estimating new software projects [Jones

1998a, p. 595].

1.2 Why is there a problem?

Despite the overwhelming body of evidence that maintenance and enhancement needs more attention and is more difficult than software development from scratch, there is little focus on these topics. For instance in the software engineering research area, there is a clear need for extensive research on maintenance and enhancement topics. However, there are not many books that deal with the subject. About one out of every hundred books on software engineering discusses maintenance and enhancement as a separate topic [Jones 1998a, p. 595–6]. In fact, one of the most common 60 software risks that have been discussed in [Jones 1994] is inadequate curricula for software engineering. Courses in maintenance and enhancement of aging software is one of the recommended methods to prevent such risks.

Possible explanations of this phenomenon are of a social nature. Working on software modifications is not perceived as a challenging task. In the 1970s Danziger investigated the adoption of computers to local governments in the United States, who applied them to data-handling tasks such as accounting, issuing payrolls, and record keeping. Although the functionality of such data processing software is similar for many local governments, the so-called Not Invented Here syndrome was omnipresent. Computer programmers working for a local government thought it was more fun to re-invent a software program than simply to reuse it from another local government or to purchase it. The latter options were seen as unstimulating drudgery. Also the relatively small differences between the software that was developed in the twelve cities that were subject to this research effort were stressed as unique features, and major improvements [Danziger 1977]. This situation is still common.

Another important issue with respect to maintenance and enhancement is that it is preventive. Prevention of deterioration of software has an impact that is difficult to measure. The nature of preventive measures is to lower the probability that some future unwanted event will occur. The unwanted future event might not happen anyway, even without adoption of the measure, and so the benefits of adoption are not clear-cut. Also the prevented events, by definition, do not occur, and so they cannot be observed or counted [Rogers 1995, p. 70]. That is why contraceptives are one of the most difficult types of innovations to diffuse [Rogers 1973]. An interesting phenomenon accompanying preventive issues is known as the KAP-gap. This is the relative long time between Knowledge and positive Attitude towards a preventive measure and its actual use in Practice [Rogers 1995, p. 71]. This phenomenon is also known in the software arena: the awareness about the Year 2000 problem is rather high at the moment. Also a positive attitude towards the problem is measured. However, many companies have not taken any preventive action to avoid the

problem. In other preventive software engineering practices the same pattern is at work. We briefly mention the important invention of software inspections by Fagan [Fagan 1976; Fagan 1986]. Software inspection is also a preventive measure: prevention of design errors to avoid high costs in correcting them in later stages. Gilb and Graham write in their textbook on software inspection that not everybody will be happy with this preventive technology. They speak of “resentment or even sabotage (we’ll make sure it doesn’t work here)” [Gilb and Graham 1993]. This technology also diffuses very slowly. To give an indication, Jones who has probably the largest knowledge base in the world (7000+ projects from 600+ clients), has 75 clients using formal inspections and 200 clients use some sort of more informal review process [Jones 1997, p. 215]. So despite the fact that 25 years of continuous data is available, about 30% of Jones’ clients use inspections. We believe that maintenance is similar to the abovementioned cases: it is difficult to measure its relative advantage. Therefore, there is not clear-cut focus on the subject.

1.3 What can be done about the problems?

The total amount of installed software measured in function points [Albrecht 1979] is estimated at about 7 Giga function points [Jones 1993]. For people who are not familiar with function points, this amounts to 640 billion logical COBOL statements. Other estimates confirm the magnitude of the amount of software: in 1990 the installed base of software was estimated on 120 billion lines of source code [Ulrich 1990]. Variations in the above estimates are easily clarified by variations in methods of line counting. Depending on the methods used, a variation of one to five is possible [Jones 1986, p. 15]. For us it is important to realize the magnitude, not exact figures. Such huge amounts of software cannot simply be discarded, and they need modifications on a daily basis [Sellink and Verhoef 1999a]. It is no longer possible to do all the maintenance and enhancement work by hand. First of all there is an endemic shortage for software personnel [Rubin 1997]. Second, when the amounts of software are above a certain range, making changes by hand is not recommended [Hall 1996; Jones 1998c].

To overcome such problems, a subindustry in the software engineering area is emerging. Large investments are done in so-called software renovation factories. Such initiatives are catalyzed by the pervasiveness and size of the Year 2000 problem and the Euro conversion problem. Their employment is becoming more and more necessary in the IT industry.

1.4 Related work

Using automated tools for software modifications is complex. Many issues play a role. In this paper we focus on the technical infrastructure and we briefly mention the kind of personnel that is necessary to

create, use and operate software renovation factories. For a more organizational view on software factories we refer the reader to [Basili *et al.* 1992]. We briefly mention the most important implementation platforms that enable the construction and operation of automated software modifications. It is possible to implement automated renovation tools using technologies like REFINE [Smith *et al.* 1985; REFINE 1992], COSMOS [COSMOS 1998], the ASF+SDF Meta-Environment [Heering *et al.* 1986; Klint 1993], RainCode [Blasband 1998], Elegant [Augusteijn 1993; Elegant 1993], TXL [Cordy *et al.* 1991], and many others.

1.5 Organization

The premise of this paper is to provide a state of the art insight into the world of automated software modifications. The remainder of this paper is organized as follows. In Section 2 we give an overall view on the topic of software renovation factories. They are the vehicles enabling automated modification. Then in the next five sections we discuss the pillars of software renovation factories. In Section 3 we discuss how we can parse the code that needs modification. Then in Section 4 we discuss how we can generate product-lines [Bass *et al.* 1998] for software renovation: we discuss a Factory Generator. In Section 5 we provide examples of typical renovation and maintenance components that can be rapidly developed and tested using a generated factory. In Section 6 we consider the issue of putting all the components together into assembly lines that make the changes completely automatic. In Section 7 we discuss the factory operators who are the people working in the factories. Finally, we place the issue of automated modifications to large software portfolios in perspective in Section 8.

1.6 Acknowledgements

This paper served as a lecture note for a keynote speech at the Joint third World Multiconference on Systemics, Cybernetics and Informatics and the fifth International Conference on Information Systems Analysis and Synthesis (SCI/ISAS'99). We are very grateful for the valuable feedback that we obtained from the attendees. We thank the anonymous reviewers for their valuable observations.

2 SOFTWARE RENOVATION FACTORIES

Before we start discussing details, we provide an overview of the relevant topics that play a role in automated modification. The ultimate goal is to automate maintenance and renovation tasks so that when we turn the handle, the particular task is performed without any further human interaction. We realize that this goal is rather challenging and might not be possible for every maintenance or enhancement task. On

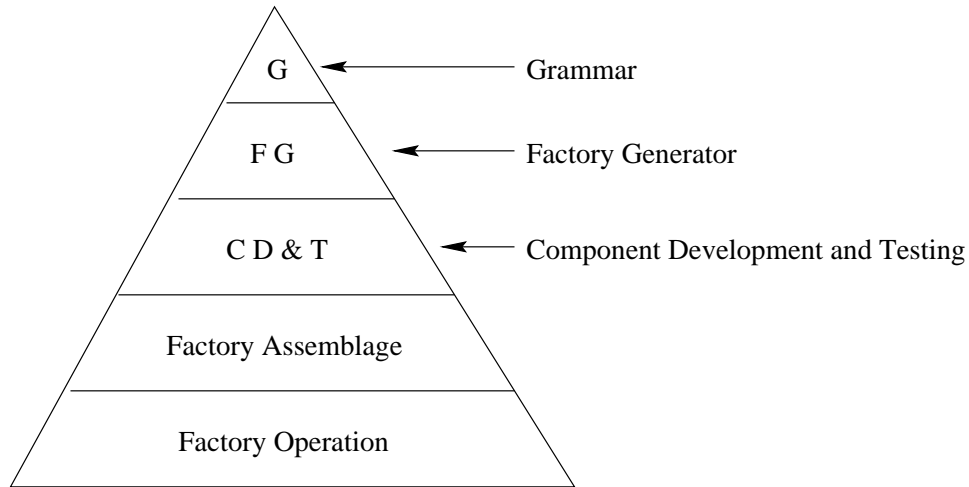


Figure 1: The Factory Pyramid.

the other hand, the massive problems with maintenance and enhancement brought some enterprises already in 1993 at the level where their entire workload is related to updating, enhancing and fixing problems in existing legacy applications [Jones 1994, p. 146]. There is no reason to believe that this situation will not become the primary one for the entire software industry.

In the literature focussed on large scale software engineering we can find a few characterizations of software renovation factories. The Gartner Group defines a software renovation factory as follows: a software renovation factory is a set of tools operated by a vendor (who usually owns the factory). The vendor's employees operate the technology, either by setting up the factory on-site or at a central facility [Hall 1996; Jones 1998c]. Another characterization of the factory approach is given by Jones [Jones 1994, p. 608]: the software factory concept envisions software being produced like a manufactured product more or less following the assembly line technique. Software factories are a concept that originates from Japan. Most of those factories were founded in the 1970s. For an overview of Japanese software factories we encourage the reader to consult [Matsumoto 1989]. In [Fokkink and Verhoef 1999] a formal definition is given for a software renovation factory.

2.1 The Factory Pyramid

When software is to be processed by a software renovation factory, this implies that the factory must have detailed knowledge of the languages involved in the system that needs modification, in the same vein as a compiler needs to know all the details on the language used. Obviously, parsers are necessary to implement this. The usual way to construct parsers is to use parser generation technology [Aho *et al.* 1986]. That

is, the grammar of the language is expressed in some formal way and from that a parser generator tool constructs a program that indeed parses the source text and turns it into a tree for further processing. It can easily be imagined that many more language-based components play an important role in the employment of software renovation factories. For instance, when the code has been renovated, it needs to be turned into source text again, using a formatter, also known as an unparser. We can use the same grammar description that generated a parser, to generate an unparser [Brand and Visser 1996]. In fact, it is possible to generate almost all generic functionality for software renovation factories from grammar descriptions. Therefore, the grammar of the code that needs renovation is one of the most valuable assets to enable automated renovation practice.

We proceed to discuss the so-called Factory Pyramid (see Figure 1). This is a five-layered view of the kind of activity necessary for the creation, development, and operation of software renovation factories. It is a pyramid to express that the lower we get in the hierarchy the more people can be employed. The top of the pyramid is labeled with a G , short for *Grammar*. In that layer, language engineers are employed who construct grammars for the myriad of languages involved with renovation plus their dialects. In Section 3 we will more elaborately discuss the technology they use. The profile of such people is that they are professional software engineers with a background in compiler construction and knowledge of formal language theory. The output of the language engineers consists of grammars. The second layer is abbreviated $F G$, which stands for *Factory Generation*. What happens here is that the grammars are used as input for parser generators, unparser generators, and other generators that we will discuss in Section 4. In this phase the core asset architecture supporting actual software renovation tasks is constructed. The people working in this layer should also be knowledgeable in compiler construction and the like. Moreover, they must be able to interpret the grammars and discuss them with the language engineers. The third layer of the pyramid is the part where *Component Development and Testing* (abbreviated $C D \& T$) occurs. In this layer, the core asset architecture is used to develop components that can carry out certain renovation tasks. In Section 5 we will discuss a string of examples of components that are useful for renovation. In fact, at this level a product-line for the rapid development of renovation components has been established. The people working in this layer should have a thorough knowledge of the problem domain so that they can develop the necessary renovation components fairly easy. They can be former senior developers employed at organizations that normally produce the code that is now under renovation. Then we enter the layer where assembly lines need to be integrated and where entire software renovation factories are assembled. In Section 6 we discuss examples of such assembly processes. The components that have been prototyped and tested should now be implemented efficiently, moreover their coordination has to be programmed. The latter is important so that manual intervention becomes minimal. The personnel that is necessary for such tasks

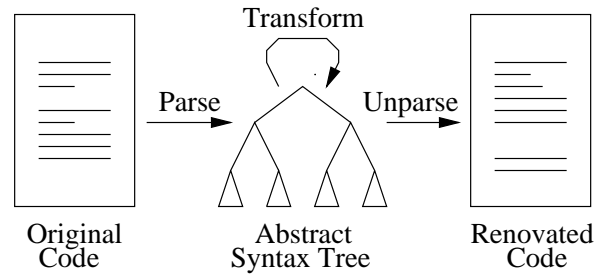


Figure 2: Schematic software renovation factory.

are systems programmers. They do not necessarily need domain knowledge of the renovation problems. In the fifth layer we operate the software renovation factory. This means, intake of code, put it at the right location, start assembly lines in the renovation factory, keep track of the progress of the modifications, suggest improvements, suggest new tools, and so on. These are typical operator tasks and they can be done by people with less formal software engineering educations as the layers one to four.

2.2 The Basic Picture

The basic picture of software renovation is the batch oriented processing of massive amounts of code as depicted in Figure 2. A system is processed in three phases. First, the code is parsed. Then the parsed code is manipulated, e.g. transformed and analyzed. Finally, an unparser translates the parsed code back to text. This rough indication of the software renovation process gives rise to a number of more specific research topics that we will briefly discuss below.

We elaborate more on the internals of the Factory Pyramid that we depicted in Figure 1. The first major issue is to obtain extensive tool support for the language engineers. They need tools for the rapid development and reengineering of grammars. This area is called Computer Aided Language Engineering (CALE), and is discussed in Section 3. For the second layer in the Factory Pyramid, also extensive generative tool support is mandatory. We discuss some of the factory generator issues in Section 4. The resulting core asset architecture is then used by component developers. We discuss a string of examples that make apparent what kind of components are important. We provide some real-world applications from the information systems area. It is also crucial that component developers have a prototyping environment at hand. We discuss issues connected to prototyping in Section 5. Finally, the prototypes must be turned into an efficient production environment: the software renovation factory is ready. We illustrate issues connected to factory assemblage in Section 6.

3 COMPUTER AIDED LANGUAGE ENGINEERING

One of the characteristics of software renovation is that often many different languages are involved. To give an example, the Year 2000 problem resides probably in systems written in 500 languages [Jones 1998b] plus another 200 proprietary languages [Jones 1998a]. To give the reader an idea, Jones reports Year 2000 search engines support for less than 50 languages and Year 2000 repair engines are available for about 10 languages [Jones 1998b, loc. cit. p. 325]. As can be seen, there is a dire need for more support, and we think that CALE technology is a good candidate for support in the Year 2000 area.

It will be clear that supporting technology like sophisticated parser generator technology and/or sophisticated parsing algorithms is mandatory. Technology that is established in the compiler construction community, like LEX [Lesk and Schmidt 1986] and YACC [Johnson 1975] breaks down for renovation. There are a few reasons that support our argument. First of all, the source code that is subject to renovation is normally not equal to the code that is processed by a compiler. Consider comments in the code that need to be taken into account. Another problem is the use of embedded languages that are taken care of by preprocessors before actual compilation takes place. During a renovation, comments and embedded languages cannot be removed or expanded, they should remain intact. The abovementioned issues make clear that main stream technology is not at all sufficient. For more information on the use of parser generator technology we refer to [Brand *et al.* 1998d].

Technology that enables obtaining grammars for languages in a cost-effective manner is also necessary. In fact, what we call computer aided language engineering is called *lingware engineering* in natural language processing [Koster 1991; Nederhof *et al.* 1992]. The relevant issue is that we try to reuse and/or retarget grammars so that their construction time is reduced significantly. Normally the grammar of a language is discussed in some formal form. We mention standards containing language descriptions, we mention manuals of proprietary languages, we mention source code of tools containing grammar information, such as pretty printers, complexity analysis tools, and compilers. Most of the times such grammars are expressed in a dialect of the Backus Naur Formalism [Backus 1960]. Using CALE tools we can extract this knowledge and we can retarget the information into a usable form for renovation.

As an example, the normal productivity that we measured for grammar construction by hand is about 300 production rules per month [Brand *et al.* 1997d]. Using CALE technology we generated about 3000 context-free production rules for a huge proprietary language for real time programming. The effort took half a day. For more information on this kind of work we refer to [Sellink and Verhoef 1998a; Sellink and Verhoef 1999b].

4 FACTORY GENERATION

Extensive maintenance or renovation becomes cost-effective when (parts of) such tasks can be automated. This is where the use of generic language technology to facilitate the construction of software renovation factories comes into play [Brand *et al.* 1997a].

4.1 Parser Generation

One issue that was already mentioned is the support on the development of sophisticated parser generation technology. Using such technology it will become feasible to deal with grammars suited for reengineering without creating a maintenance problem for those grammars. Renovation grammars normally change very frequently, and simple parser generator technology will create enormous maintenance burdens on such grammars. It is a common experience within the reengineering area that when it is possible to parse say, COBOL, still every new project the parsers need adaptations. This is due to the myriad of dialects. Also for C there are over 20 dialects. Established compiler construction parser generation technology like LEX/YACC [Lesk and Schmidt 1986; Johnson 1975] is not meant for continuous change. When a simple change is made to such a grammar it is not uncommon that solving problems like shift/reduce and reduce/reduce problems takes a long time (see [Levine *et al.* 1992] for details on these terms). Moreover, change after change will make the grammars unmaintainable. A solution to these problems is the use of generalized LR parsing [Lang 1974; Tomita 1986; Rekers 1992; Visser 1997]. This is a type of parsing that originates from natural language processing. It can handle local ambiguities very well, thereby the need to solve the abovementioned conflicts is not needed anymore. It is not as efficient as the generated parsers in the compiler construction area, but we should also add the construction time for parsers, which is very short using the generalized technology. See [Aycock and Horspool 1999] for faster generalized LR parser technology. We refer to [Brand *et al.* 1998d] for more details on the combination of parsing and reengineering.

4.2 Unparser Generation

Another issue that is important is the generation of unparser or formatter generation technology. In fact, in the paper [Sneed 1998] the formatting of source code is mentioned as a first step towards reengineering source programs. Also during the development of renovation components, and as final output of a renovation factory, it is important that the code to be delivered is turned from a parse tree into readable text again. It is also useful to have source code in a browsable form for inspection. These tools can all be provided using unparser generation technology. We refer to [Brand and Visser 1996] for information on formatter generator

technology. For the generation of hypertext support to support software renovation we refer to [Graaf 1997; Sellink and Verhoef 1999b].

4.3 Scaffolding Generation

It is of utter importance to have complete control over what steps are taken during an (automated) renovation process. Therefore, it is crucial to have a mechanism available that deals with persistent intermediate results. A very common concept in the development of systems is the use of code, data, or entire programs that are built for debugging or tracing purposes, but never intended to be in the final product. This technology is also known as scaffolding. See for instance books on good coding practice that mention scaffolding [Knuth 1968; Brooks Jr. 1995; Bentley 1988; McConnell 1993]. Since scaffolding is usually removed when a software product is put into production, it seems natural to us to bring back such knowledge in the source code while renovating it. Such knowledge can be control-flow or data-flow information, or more specific information. We observed that for automated renovation of software, the source code manipulations are so complex, that intermediate results of calculations are mandatory in order to keep track of the modification process. We have found it extremely useful to include such scaffolding in the source text so that easy and immediate inspection and/or modification of the results is viable [Sellink and Verhoef 1999c]. We can generate special grammars that can also parse scaffolding information to support renovation of sources. For an elaborate treatment of scaffolding and its connection with software renovation we direct the reader to [Sellink and Verhoef 1999c].

4.4 Transformation and Analysis Generation

An important issue is how to deal with generic parts of components for analysis and renovation. The parts of such components that are depending on the language, in a very structured way are candidates for automatic generation. We mention traversal of parse trees. Searching in source code means that the parse tree is traversed. Such traversals of trees are part of almost every renovation component. At this moment we have two flavors: tree traversal generation and tree analysis generation. As we have explained, grammars change often. This is reflected in the renovation components as well: when such components traverse a tree, and the tree definition changes, they also have to change. Therefore, if the renovation components have to change each time the grammar is modified, the renovation components will be subject to a maintenance problem. Using a generative approach towards component development enables components that are reliable, maintainable, maximally reusable, and their implementation is usually measured in minutes rather than hours. This topic is elaborately discussed in [Brand *et al.* 1999]. In that paper we not only show the generation process but we also illustrate that the components are reliable, maintainable, maximally reusable.

4.5 Pattern language generation

Of course, it is important to be able to recognize certain patterns in code to facilitate analyzing and modifying code. As we indicated in the Factory Pyramid (see Figure 1), in the third layer people turn the actual renovation tasks into components. The work consists of constructing patterns, constructing replacements, and testing the ensuing transformations. The qualifications that many potential component developers have, are that they know the languages in which the code has been written and that they can think of solutions for domain specific problems. For example, a COBOL programmer is perfectly capable of solving an individual Year 2000 problem when confronted with problematic code (possibly after a short introductory course in the problem area).

We cannot expect that domain experts are also knowledgeable on the subject of pattern languages and so on. For example: a COBOL renovation factory should be such that a domain expert, like an experienced COBOL developer is able to work in it. Since the used programming language is a substantial factor for productivity [Jones 1986], it is crucial to design the pattern language that should be used in this factory as carefully as possible. Our solution is that we generate a so-called native pattern language from the grammar of the code that has to be reengineered. An example of a native pattern is a real code fragment. So, for instance a COBOL code fragment is a native pattern. To create more general native patterns we allow the use of variables in exactly the way they are defined in the language reference manuals. So for instance, in the COBOL manuals (like the ANSI Standard [ANSI-COBOL 1985] or the IBM COBOL manual [IBM-COBOL 1997]), the use of variables like `Statement1`, `Statement3+`, and `Statement5*` is allowed. The interpretation of `Statement1` is that it matches exactly one arbitrary COBOL statement, `Statement1+` matches one or more COBOL statements, and `Statement1*` zero or more. Note that the convention of numbers is common in language reference manuals to explain the language. Domain experts have seen such language reference manuals, so the learning curve for a native pattern language is close to zero. Details on the generation of native pattern languages and their use can be found in [Sellink and Verhoef 1998c].

4.6 Putting it all Together

We indicated in Figure 2 what the basic contents of a software renovation factory comprises. In this section we discussed a number of components that together form the core asset architecture to construct such a factory. The generated parser takes care of building the abstract syntax trees. The generic transformation and analysis functionality enable rapid development of tools that perform actual renovation tasks. The generated native pattern language plus its documentation helps in specifying these tools. When transformations or analyses become complex, the generated scaffolding aids are indispensable for storing intermediate results.

We discuss the ensuing architecture in more detail in [Sellink and Verhoef 1999a].

5 COMPONENT PROTOTYPING

It is impossible to create a component library that solves all the renovation problems. Therefore, it is always necessary to develop new components. It is our experience that many and diverse tasks can be split into very small but clear components that can be reused many times. The reason for this is that in many cases the problems are similar but not that similar that a monolithic approach pays off. As a consequence, the components that we like to develop have small but very clear behavior. An advantage of this approach is that many small completely understandable steps can lead to very intricate transformations of the code in question. Another advantage is that the small steps make the process controllable. This is a major advantage for component prototyping and testing. We will provide a few typical examples of tasks that have been accomplished using many small components. We will not discuss the components themselves in detail, but rather the kind of application so that the reader obtains a clear idea of what can be expected from automated renovation of legacy assets.

5.1 Mass maintenance

During software maintenance in large companies, it appears to be the case that many and diverse changes to entire systems have to be made on a daily basis. Those tasks take a lot of precious time and their automation would save large amounts of money. Since the work is often very company specific, there are no tools. So, this type of mass maintenance has to be done by hand. A typical task is the addition of explicit scope terminators, like `END-IF`, `END-PERFORM`, and such in COBOL applications for improving the readability of code. We learned from reliable sources that a software engineer at a Dutch bank worked a year on adding explicit scope terminators manually. We developed a component in about an hour that added in a mortgage system approximately 27 `END-IFs` per minute using a COBOL factory. More important, the automated process does not miss cases, makes no typographical errors, and provides a uniform layout.

Another typical task is a request for restructuring a system containing certain hard-wired coded error handling processes after an SQL statement has approached a DB2-table. An update of the IBM product DB2 provided new return codes that had to be taken into account. Therefore, the return codes had to be removed and stored in a separate program that has to be called. Another example is that over time, the structure of the error handling procedures themselves, has been modified. As a consequence, the control-flow deteriorated. With a mass maintenance transformation the control-flow could easily be updated so that the bad decision structures became natural again. For more information on such mass maintenance tasks we

refer to [Sellink and Verhoef 1999a].

5.2 Control-flow normalization

Another important modification to software systems is to remove jump instructions, in order to improve maintainability. Normal algorithms to remove jumps fail in practice. First of all since a lot of code duplication will ensue, which does not improve maintainability. Second, some control-flow problems are so intricate that such algorithms cannot handle them. We mention so-called implicit jump instructions in an embedded language CICS that can be written inside COBOL source programs (also in PL/I, C, and Assembly/370 programs). There is no `GO TO` statement, but under the surface there is, due to the CICS code. We developed components in order to take care of intricate problematic control-flow problems that reside in CICS code embedded in COBOL. The algorithm that we published in [Brand *et al.* 1998c] has been implemented by us and also in a commercial reengineering workbench, since this task is more often necessary. More details on the components that we developed can be found in [Brand *et al.* 1998c]. The algorithm to normalize implicit jump instructions has been commercially implemented in Sneed's reengineering workbench [Sneed 1998].

5.3 Extracting business logic

It is often very difficult to extract the real business logic from legacy applications. If we focus on the IS area, then it is normal to have systems written in COBOL. In fact, this is not COBOL, but also SQL, for dealing with the data bases, and CICS for dealing with the transaction processing. Roughly, one-third of the code is COBOL, one-third is SQL, and one-third is CICS. The SQL code and the CICS code are *embedded* in the COBOL programs. So the business logic is scattered over the code. Still it is of crucial interest to extract the business logic. This information can be used for re-implementation, but also for wrapping the old code, so that we can use the old code as a component in a new system. In order to wrap such code, we have to factor out the business logic. We have developed an algorithm to remove the CICS code in a COBOL/CICS system, and we developed an algorithm that separates the control-flow of programs from the actual computations. The latter algorithm makes use of very complicated restructuring components that we developed. The resulting code is a list of computations (all candidates for business logic) and a control-flow part that only tells which computation to perform next. In this way it becomes feasible to enable modifications to brittle systems with scattered business logic. For an elaborate treatment of these issues we refer to [Sellink *et al.* 1999]. In that paper we also discuss some of the developed components applied to code of a Swiss bank. The Swiss bank was so kind to give us permission to publish one program with all the intermediate modification steps. This can be downloaded from the Internet [Sellink *et al.* 1998].

There are many more component development projects that one can think of. We will not mention them here, since the goal of the listing was not to be exhaustive, but to provide insight in the type of tasks that can be accomplished completely automated.

6 FACTORY PRODUCTION ENVIRONMENT

We discuss the fourth layer in the factory pyramid that we depicted in Figure 1, it concerns the assembly of software renovation factories. At this point we have developed and tested all the necessary components, and we fleshed out their coordination. Sometimes the coordination of the components is very simple. For instance when we need a few changes to code, comprising a few simple transformations that are to be carried out as a sequence, the coordination is a simple pipeline. In other cases, the coordination of the components is very complex. For instance in the business rules extractor, we apply a number of components in various loops so that gradually the logic is extracted. The important issue here is that the coordination and the computations of the renovation tasks have been separated carefully during the component development phase. The factory assembly process deals with the actual construction of conveyor belts that carry out the complete tasks fully automatically.

The components need to be made efficient. Sometimes this means that a developed prototype needs to be reimplemented using some scripting language like `perl` [Wall and Schwartz 1991]. We advise to do this only when the component is very simple, and a lexical approach does not harm at all. In our case the developed components can be compiled directly to efficient C code [Kernighan and Ritchie 1978]. Important supporting technology is the use of an expert compiler that turns our formally specified components into efficient stand-alone C programs [Brand *et al.* 1998b]. This is all necessary since issues like scalability, multi-processor calculations and such play an important role when millions of lines of code have to be processed. The factory assembler also takes care of the coordination of the renovation components. Sometimes this implies a separate program that takes care of the coordination, like a `perl` script or a C program. In other cases, we use middleware to connect the components. Important supporting technology for such coordination is middleware that is geared towards the connection of language oriented tools, called the ToolBus [Bergstra and Klint 1996b; Bergstra and Klint 1996a; Bergstra and Klint 1998]. The expert compiler generates C code that can be directly connected to the ToolBus so that it is not necessary to change the code in order to connect it to middleware. However, if there is no need to use middleware, the code can also run without the ToolBus.

Another task of the factory assemblage team is to take care of the releases of the factory with all its assembly lines. To give the reader an idea, in one case a Y2K factory of a reengineering company has a

weekly release. Then a lot of test-runs have to be performed, version management, and so on all come into play.

To give the reader an idea of the state-of-the-art with respect to rapid delivery of efficient renovation factories, we mention that we have a COBOL renovation factory architecture for which prototyped components can readily be compiled and used in a production factory. The factory can deal with many COBOL dialects, with embedded CICS and/or SQL. There is sophisticated pre- and postprocessing of the code. This includes issues like temporary expansion of so-called copy books—comparable to an include file in C. In [Brunekreef and Diertens 1999] a production environment has been developed after a set of components had been prototyped for a specific project. In this case the project was a feasibility study for a large bank to see whether it was possible to convert a modern COBOL dialect (from 1985) back to an older dialect (a COBOL dialect from 1974). Note that the backward transformation is not a typographical error. There was a very good reason for converting the code back to an older dialect, see [Brunekreef and Diertens 1999] for details. To give the reader an idea of the work that is necessary during the factory assembly stage, it took not more than an hour to compile efficient components from the prototypes and since their coordination was a pipeline, there was no effort in the coordination. So in about half a day, we could move from the third layer to the fourth layer. In general, when there is no executable prototype, or no automatic means to compile the prototype into efficient code, the factory assembly phase boils down to a classical software development project. In this case the actual implementation will presumably take more than half a day.

The real-world case study [Brunekreef and Diertens 1999] also indicates that academics cannot invent their own reengineering case studies: they have to come from IT industry. For more information on the use and construction of software renovation factories we refer to the papers [Brand *et al.* 1996; Klint and Verhoef 1998; Brand *et al.* 1998a]. For academic readers interested in the difficulties and problems to overcome when dealing with IT industry, we refer to [Brand *et al.* 1998a].

7 FACTORY OPERATION

According to legend, around 1800 in Leicestershire, a young man called Ned Ludd, broke into a factory and—in a fit of insane rage—destroyed two knitting machines. This was not simply an act of vandalism but also a protest against the factory owners and their use of the new machines. This story is a possible clarification for the etymology of the word luddite, which stands for one who is opposed to especially technological change. While some might think that opposition against machines and factories is due to men who failed to see the ineffable benefits of them, this phenomenon is nowadays seen in a different perspective. In a reinterpretation of luddism in Britain [Randall 1997] we read that psychologists tell us that the loss

of one's work, in particular the loss of an occupation requiring manual or mechanical skill, is a devastating blow to the individual's self-esteem, exceeded only by the shock of bereavement. Therefore, it is not obvious to us that people who normally modify code by hand, will immediately like the idea that a machine can take over.

It is also known that programmers working on a system for some time, typically maintenance programmers, consider the code to be their own [Weinberg 1971]. Therefore, it is not so easy to outsource maintenance. If this happens, there is a serious danger that the maintenance team will reject the code that is returned to them. For, they are deprived of their code, indicating that they do not do a proper job. Then strangers touch it, and they break it. Sneed mentioned (in a keynote address to the fourth Working Conference on Reverse Engineering [Baxter *et al.* 1997]) that he experienced this phenomenon in an off-shore outsourced Year 2000 conversion [Sneed. 1997]. Such hostile attitudes towards maintenance outsourcing should be taken into account when dealing with automated tools. Therefore, our approach is to bring factories to the maintenance teams so that *they* can operate them.

One of the reasons that there is a lack of advanced aids for maintenance and enhancements, is that people are naturally learners. Therefore, the competence of a maintenance team to maintain a particular systems grows over time, possibly masking the deterioration of the system [Weinberg 1992, p. 243]. Their competence also hides the fact that many tools are necessary. Over time a lot of knowledge is being put in the heads of the teams. If this knowledge is put in automated tools, a lot of such specific knowledge is put inside the tools. If that happens, a sudden exodus of the maintenance crew is less disruptive for the maintenance assignment scope of novices than usual. Therefore, intensive communication between the maintenance team and renovation component developers is necessary.

The intensive communication turns the usual fear of the factory—also known as mechanophobia—with its regularity, order, control and discipline into a blessing that frees the maintenance teams to perform error-prone and repetitive tasks, because the factory will do the repetitive work. In the mean time they can order the tools necessary while communicating with the factory team. This is not only a hope for the future, but this is actually happening. We learned from two major reengineering companies who employ many Y2K factories on four continents, that the people working in these factories feel empowered by the flexibility of the approach taken in the factories. The problem reports and change requests have lead to a weekly release of their Y2K factory. This intensive interaction between factory workers and factory constructors makes that the maintenance programmers like their work instead of showing luddite traces in their behavior.

One way to encourage a factory approach towards modifications of software portfolios is to educate enhancement programmers, with tools, training, and resources. Also maintenance teams should be paid better. However, compensation plans for maintenance programmers is out of scope for this paper. We stress

that these issues, as well as encouraging people to participate in education is a managerial responsibility. We encourage managers to take note of the book [Weinberg 1971] for learning more about the psychology of computer programming, and the book [Weinberg 1988] for understanding more of the professional programmer.

Summarizing, migrating handwork of maintenance programmers to a factory approach gives them the unique opportunity to discuss their problems with others, the factory constructors. They in turn construct assembly lines that unleash the knowledge that would normally not be outside the heads of the maintenance crew. This is a corporate asset. Both the interaction and tools give the factory operators the (correct) feeling that their work is important, difficult and appreciated. Management should support such changes in the organization. Help with managing the introduction of automated tools in an organization can be found in books like [Pressman 1988; Bouldin 1989].

8 PERSPECTIVE

One of the most common 60 risks that have been discussed in [Jones 1994] is inadequate curricula for software engineering. In the United States the severity on the average is that graduates need more than a year of remedial on-the-job training. One of the issues that could help solving this problem is more focus on maintenance and enhancement of aging legacy systems. As we indicated earlier, more than half of the professional programmers and software engineers are employed on these issues. Renovation is a difficult and academic topic, so it is only logical that software engineers extend their work to also capture software renovation. Also undergraduates should be taught issues like software entropy [Belady and Lehman 1976], specialized tools that keep aging software operational, e.g., complexity analysis tools [McCabe 1976], restructuring, reverse engineering and reengineering tools. Software engineers should also learn old languages, old operating systems, and old hardware. As far back as the early 1970s, Gerald Weinberg called this software archeology [Weinberg 1971]. It is also crucial that more people are educated on software renovation factories: on building them, and solving problems using them. Just like the number of geriatric medical specialists will grow since more and more people are getting older and older, there will be more and more geriatric software specialists necessary for similar reasons. Of course, geriatric medical specialists are not tied to using ancient tools because their subjects are old, on the contrary, they need the most sophisticated material. Similarly, geriatric software specialists are not tied to ancient tools, just because that is their input. In this paper we hope to have shown the following general observation on software renovation:

Software renovation is using tomorrow's technology to bring yesterday's software to the level of today.

REFERENCES

- Adams, S. (1996), *The Dilbert Principle*, MacMillan Publishers Ltd.
- Aho, A., R. Sethi, and J. Ullman (1986), *Compilers. Principles, Techniques and Tools*, Addison-Wesley.
- Albrecht, A. (1979), "Measuring Application Development Productivity," In *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, pp. 83–92.
- Allen, P. and S. Frost (1998), *Component-Based Development for Enterprise Systems*, Cambridge University Press.
- ANSI-COBOL (1985), *Programming Language – COBOL*, American National Standards Institute, Inc., ANSI X3.23–1985 Edition.
- Augusteijn, L. (1993), "Functional Programming, Program Transformations and Compiler Construction," Ph.D. thesis, Eindhoven University of Technology.
- Aycock, J. and N. Horspool (1999), "Faster Generalized LR Parsing," In *Proceedings of the eight International Conference on Compiler Construction*, S. Jähnichen, Ed., volume 1575 of *LNCS*, Springer-Verlag, pp. 32–46.
- Backus, J. (1960), "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," In *Proceedings of the International Conference on Information Processing*, S. de Picciotto, Ed., Unesco, Paris, pp. 125–131.
- Basili, V., G. Caldiera, and G. Cantone (1992), "A reference Architecture for the Component Factory," *ACM TOSEM* 1, 1, 53–80.
- Bass, L., P. Clements, and R. Kazman (1998), *Software Architecture in Practice*, Addison-Wesley.
- Baxter, I., A. Quilici, and C. Verhoef, Eds. (1997), *Proceedings of the Fourth Working Conference on Reverse Engineering*, IEEE Computer Society Press.
- Belady, B. and M. Lehman (1976), "A Model of Large Program Development," *IBM Systems Journal* 15, 3, 225–252.
- Bennett, K. and T. Khoshgoftaar, Eds. (1998), *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press.
- Bentley, J. (1988), *More Programming Pearls – Confessions of a Coder*, Addison-Wesley.
- Bergstra, J. and P. Klint (1996a), "The TOOLBUS coordination architecture," In *Coordination Languages and Models*, P. Ciancarini and C. Hankin, Eds., volume 1061 of *Lecture Notes in Computer Science*, pp. 75–88.
- Bergstra, J. and P. Klint (1996b), "The Discrete Time TOOLBUS," In *Algebraic Methodology and Software Technology*, M. Wirsing and M. Nivat, Eds., volume 1101 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 286–305.
- Bergstra, J. and P. Klint (1998), "The discrete time TOOLBUS—A software coordination architecture," *Science of Computer Programming* 31, 205–229.
- Blaha, M., A. Quilici, and C. Verhoef, Eds. (1998), *Proceedings of the Fifth Working Conference on Reverse Engineering*, IEEE Computer Society Press.
- Blasband, D. (1998), *RainCode*, RainCode, Brussels, Belgium, First Edition, <ftp://ftp.raincode.com/cobrc.ps>.
- Bouldin, B. (1989), *Agents of Change – Managing the Introduction of Automated Tools*, Yourdon-Press.
- Brand, M., P. Klint, and C. Verhoef (1996), "Core Technologies for System Renovation," In *SOFSEM'96: Theory and Practice of Informatics*, K. Jeffery, J. Král, and M. Bartošek, Eds., volume 1175 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 235–255.
- Brand, M., P. Klint, and C. Verhoef (1997a), "Re-engineering needs Generic Programming Language Technology," *ACM SIGPLAN Notices* 32, 2, 54–61, Available at <http://adam.wins.uva.nl/~x/sigplan/plan.html>.
- Brand, M., P. Klint, and C. Verhoef (1997b), "Reverse Engineering and System Renovation – An Annotated Bibliography," *ACM Software Engineering Notes* 22, 1, 57–68, Available at <http://adam.wins.uva.nl/~x/reeng/REanno.html>.

- Brand, M., P. Klint, and C. Verhoef (1998a), "Term Rewriting for Sale," In *Second International Workshop on Rewriting Logic and its Applications*, C. Kirchner and H. Kirchner, Eds., Electronic Notes in Theoretical Computer Science, Springer-Verlag, Available at: <http://adam.wins.uva.nl/~x/sale/sale.html>.
- Brand, M., P. Olivier, J. Heering, and P. Klint (1998b), "Compiling Rewrite Systems: The ASF+SDF Compiler," Technical report, CWI/University of Amsterdam, In preparation.
- Brand, M., M. Sellink, and C. Verhoef (1997c), "Generation of Components for Software Renovation Factories from Context-free Grammars," In *Proceedings Fourth Working Conference on Reverse Engineering*, I. Baxter, A. Quilici, and C. Verhoef, Eds., pp. 144–153, Available at <http://adam.wins.uva.nl/~x/trans/trans.html>.
- Brand, M., M. Sellink, and C. Verhoef (1997d), "Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes," In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, M. Sellink, Ed., electronic Workshops in Computing, Springer verlag, Available at <http://adam.wins.uva.nl/~x/coboldef/coboldef.html>.
- Brand, M., M. Sellink, and C. Verhoef (1998c), "Control Flow Normalization for COBOL/CICS Legacy Systems," In *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, P. Nesi and F. Lehner, Eds., pp. 11–19, Available at <http://adam.wins.uva.nl/~x/cfn/cfn.html>.
- Brand, M., M. Sellink, and C. Verhoef (1998d), "Current Parsing Techniques in Software Renovation Considered Harmful," In *Proceedings of the sixth International Workshop on Program Comprehension*, S. Tilley and G. Visaggio, Eds., pp. 108–117, Available at <http://adam.wins.uva.nl/~x/ref/ref.html>.
- Brand, M., M. Sellink, and C. Verhoef (1999), "Generation of Components for Software Renovation Factories from Context-free Grammars," Accepted for publication in *Science of Computer Programming*. Available at <http://adam.wins.uva.nl/~x/scp/scp.html>. An extended abstract with the same title appeared earlier: [Brand *et al.* 1997c].
- Brand, M. and E. Visser (1996), "Generation of Formatters for Context-free Languages," *ACM Transactions on Software Engineering and Methodology* 5, 1–41.
- Brooks Jr., F. (1995), *The Mythical Man-Month – Essays on Software Engineering*, Addison-Wesley, Anniversary Edition.
- Brown, A., Ed. (1996), *Component-Based Software Engineering*, IEEE Computer Society Press.
- Brunekreef, J. and B. Diertens (1999), "Towards a User-Controlled Software Renovation Factory," In *Proceedings of the Third European Conference on Maintenance and Reengineering*, P. Nesi and C. Verhoef, Eds., IEEE Computer Society Press, pp. 83–90.
- Chikofsky, E. and J. Cross (1990), "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software* 7, 1, 13–17.
- Cordy, J., C. Halpern-Hamu, and E. Promislow (1991), "TXL: A Rapid Prototyping System for Programming Language Dialects," *Computer Languages* 16, 1, 97–107.
- COSMOS (1998), *Emendo Y2K White paper*, Emendo Software Group, The Netherlands, Available at <http://www.emendo.com/>.
- Danziger, J. (1977), "Computers, Local Governments and the Litany to EDP," *Public Administration Review* 37, 28–37.
- Eastwood, A. (1992), "It's a hard sell - and hard work too. (software reengineering)," *Computing Canada* 18, 22, 35.
- Elegant (1993), *The Elegant Home Page*, Philips Electronics B.V., The Netherlands, <http://www.research.philips.com/generalinfo/special/elegant/elegant.html>.
- Fagan, M. (1976), "Design and code inspections to reduce errors in programs," *IBM Systems Journal* 15, 3, 182–211.
- Fagan, M. (1986), "Advances in Software Inspections," *IEEE Transactions on Software Engineering SE-12*, 7, 744–751.
- Fokkink, W. and C. Verhoef (1999), "Conservative extension in positive/negative conditional term rewriting with applications to software renovation factories," In *Proceedings 2nd Conference on Fundamental Approaches to Software Engineering*, J.-P. Finance, Ed., volume 1577 of *Lecture Notes in Computer Science*, Springer-Verlag, Amsterdam, pp. 98–113.
- Gallagher, K. and J. Lyle (1991), "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering* 17, 8, 751–761.

- Gilb, T. and D. Graham (1993), *Software Inspection*, Addison-Wesley.
- Graaf, M. (1997), "A specification of Box to HTML in ASF+SDF," Technical Report P9720, University of Amsterdam, Programming Research Group, Available at <http://ftp.wins.uva.nl/pub/programming-research/reports/1997/P9720.ps.Z>.
- Hall, B. (1996), "Year 2000 tools and services," In *Symposium/ITxpo 96, The IT revolution continues: managing diversity in the 21st century*, GartnerGroup.
- Hearing, J., G. Kahn, P. Klint, and B. Lang (1986), "Generation of interactive programming environments," In *Esprit '85 - Status Report of Continuing Work 1*, T. C. of the European Communities, Ed., North-Holland, pp. 467–477.
- IBM-COBOL (1997), *IBM SAA AD/Cycle COBOL/370*, IBM Corporation, First Edition.
- Johnson, S. (1975), "YACC - Yet Another Compiler-Compiler," Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey.
- Jones, C. (1986), *Programming Productivity*, McGraw-Hill.
- Jones, C. (1993), *Software Productivity and Quality – The World Wide Perspective*, IS Management Group, Carlsbad, CA.
- Jones, C. (1994), *Assessment and Control of Software Risks*, Prentice-Hall.
- Jones, C. (1996), *Patterns of Software Systems Failure and Success*, International Thomsom Computer Press.
- Jones, C. (1997), *Software Quality – Analysis and Guidelines for Success*, International Thomsom Computer Press.
- Jones, C. (1998a), *Estimating Software Costs*, McGraw-Hill.
- Jones, C. (1998b), *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*, Addison-Wesley.
- Jones, N. (1998c), "Year 2000 Market Overview," Technical report, GartnerGroup, Stamford, CT, USA.
- Kernighan, B. and D. Ritchie (1978), *The C Programming Language*, Prentice-Hall.
- Klint, P. (1993), "A meta-environment for generating programming environments," *ACM Transactions on Software Engineering and Methodology* 2, 2, 176–201.
- Klint, P. and C. Verhoef (1998), "Evolutionary software engineering: A component-based approach," In *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, R. Horspool, Ed., Chapman & Hall, pp. 1–18, Available at: <http://adam.wins.uva.nl/~x/evol-se/evol-se.html>.
- Knuth, D. (1968), *The Art of Computer Programming – Fundamental Algorithms*, Addison-Wesley.
- Koster, C. (1991), "Affix Grammars for Programming Languages," In *International Summer School on Attribute Grammars, Applications and Systems*, H. Alblas and B. Melichar, Eds., volume 545 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 358–373.
- Lang, B. (1974), "Deterministic techniques for efficient non-deterministic parsers," In *Proceedings of the Second Colloquium on Automata, Languages and Programming*, J. Loeckx, Ed., volume 14 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 255–269.
- Lesk, M. and E. Schmidt (1986), *LEX - A lexical analyzer generator*, Bell Laboratories, UNIX Programmer's Supplementary Documents, Volume 1 (PS1) Edition.
- Levine, J., T. Mason, and D. Brown (1992), *lex & yacc*, Second Edition, O'Reilly & Associates, Inc.
- Matsumoto, Y. (1989), "An Overview of Japanese Software Factories," In *Japanese Perspectives in Software Engineering*, Y. Matsumoto and Y. Ohno, Eds., Addison-Wesley, pp. 303–320.
- McCabe, T. (1976), "A complexity measure," *IEEE Transactions on Software Engineering SE-12*, 3, 308–320.
- McConnell, S. (1993), *Code Complete*, Microsoft Press.
- Nederhof, M., C. Koster, C. Dekkers, and A. Zwol (1992), "The Grammar Workbench: A first step towards lingware engineering," In *Proceedings of the second Twente Workshop on Language Technology – Linguistic Engineering: Tools and Products*, W. Stal, A. Nijholt, and H. Akker, Eds., volume 92-29 of *Memoranda Informatica*, University of Twente, pp. 103–115.

- Nesi, P. and C. Verhoef, Eds. (1999), *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, IEEE Computer Society Press.
- Pressman, R. (1988), *Making Software Engineering Happen*, Prentice-Hall.
- Randall, A. (1997), "Reinterpreting 'Luddism': Resistance to new technology in the British Industrial Revolution," In *Resistance to New Technology: Nuclear Power, Information Technology, and Biotechnology*, M. Bauer, Ed., Cambridge University Press, pp. 57–79, reprint edition.
- REFINE (1992), *Refine User's Guide*, Reasoning Systems, Palo Alto, California.
- Rekers, J. (1992), "Parser Generation for Interactive Environments," Ph.D. thesis, University of Amsterdam, Available at <ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z>.
- Rogers, E. (1973), *Communication Strategies for Family Planning*, Free Press.
- Rogers, E. (1995), *Diffusion of Innovations*, Fourth Edition, Free Press.
- Rubin, H. (1997), *Worldwide Benchmark Report for 1997*, Pound Ridge, NY; Rubin Systems, Inc.
- Sellink, M., H. Sneed, and C. Verhoef (1998), "Systolic structuring algorithm in steps," Available at <http://adam.wins.uva.nl/~x/systolic/systolic.html>.
- Sellink, M., H. Sneed, and C. Verhoef (1999), "Restructuring of COBOL/CICS Legacy Systems," In *Proceedings of the Third European Conference on Maintenance and Reengineering*, P. Nesi and C. Verhoef, Eds., pp. 72–82, Available at <http://adam.wins.uva.nl/~x/cics/cics.html>.
- Sellink, M. and C. Verhoef (1998a), "Development, Assessment, and Reengineering of Language Descriptions," In *Proceedings of the 13th International Automated Software Engineering Conference*, pp. 314–317, Full version in [Sellink and Verhoef 1998b].
- Sellink, M. and C. Verhoef (1998b), "Development, Assessment, and Reengineering of Language Descriptions," Technical Report P9805, University of Amsterdam, Programming Research Group, Available at: <http://adam.wins.uva.nl/~x/cale/cale.html>.
- Sellink, M. and C. Verhoef (1998c), "Native patterns," In *Proceedings of the Fifth Working Conference on Reverse Engineering*, M. Blaha, A. Quilici, and C. Verhoef, Eds., IEEE Computer Society Press, pp. 89–103, Available at <http://adam.wins.uva.nl/~x/npl/npl.html>.
- Sellink, M. and C. Verhoef (1999a), "An Architecture for Automated Software Maintenance," In *Proceedings of the seventh International Workshop on Program Comprehension*, D. Smith and S. Woods, Eds., pp. 38–48, Available at <http://adam.wins.uva.nl/~x/asm/asm.html>.
- Sellink, M. and C. Verhoef (1999b), "Generation of Software Renovation Factories from Compilers," In *Proceedings of the International Conference on Software Maintenance*, H. Yang and L. White, Eds., To appear. Available via <http://adam.wins.uva.nl/~x/com/com.html>.
- Sellink, M. and C. Verhoef (1999c), "Scaffolding for Software Renovation," Technical Report P9903, University of Amsterdam, Programming Research Group, Available via <http://adam.wins.uva.nl/~x/scaf/scaf.html>.
- Smith, D., G. Kotik, and S. Westfold (1985), "Research on Knowledge-Based Software Environments at Kestrel Institute," *IEEE Transactions on Software Engineering SE-11*, 11, 1278–1295.
- Sneed, H. (1997), "Dealing with the Dual Crisis—Year 2000 and Euro—What Reverse Engineering can do to Help," Technical Report P9716, University of Amsterdam, Programming Research Group.
- Sneed, H. (1998), "Architecture and functions of a commercial software reengineering workbench," In *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, P. Nesi and F. Lehner, Eds., pp. 2–10.
- Tilley, S. and G. Visaggio, Eds. (1998), *Proceedings of the Sixth International Workshop on Program Comprehension*, IEEE Computer Society Press.

- Tomita, M. (1986), *Efficient Parsing for Natural Languages—A Fast Algorithm for Practical Systems*, Kluwer Academic Publishers.
- Ulrich, W. (1990), “The evolutionary growth of software reengineering and the decade ahead,” *American Programmer* 3, 11, 14–20.
- Visser, E. (1997), “Scannerless Generalized-LR Parsing,” Technical Report P9707, Programming Research Group, University of Amsterdam, Available at <http://www.wins.uva.nl/pub/programming-research/reports/1997/P9707.ps>.
- Wall, L. and R. Schwartz (1991), *Programming Perl*, O’Reilly & Associates, Inc.
- Weinberg, G. (1971), *The Psychology of Computer Programming*, Van Nostrand Reinhold.
- Weinberg, G. (1988), *Understanding the Professional Programmer*, Dorset House.
- Weinberg, G. (1992), *Quality Software Management: Volume 1 Systems Thinking*, Dorset House.