Scaffolding for Software Renovation

Alex Sellink and Chris Verhoef
University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
alex@wins.uva.nl, x@wins.uva.nl

Abstract

We discuss an approach that explores the use of scaffolding of source code to facilitate its renovation. We show that scaffolding is a useful paradigm for software renovation. We designed syntax and semantics for scaffolding, that enables all relevant applications of scaffolding. The automatic generation of extensions to a normal grammar, so that the resulting extension grammar can parse code with scaffolding, is discussed. We used the scaffolding paradigm itself to implement the generation process, thereby showing that our approach towards scaffolding is also useful in software development. Finally, we discuss real-world applications of scaffolding for software renovation, in both our own work and work from people in the reengineering IT industry.

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, Language description development, Grammar reengineering, Scaffolding, Computer aided language engineering (CALE).

1 Introduction

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*. Knuth [24, p. 189] mentions that the "most effective debugging techniques seem to be those which are designed and built into the program itself—many of today's best programmers will devote nearly half of there programs to facilitating the debugging process on the other half; the first half, which usually consists of fairly straightforward routines, will eventually be thrown away, but the net result is a surprising gain in productivity." While Knuth focuses on programming in the small, these figures are also mentioned by people discussing programming in the large. For instance, Brooks [12, p. 148] wrote in 1975 that it is "not unreasonable for there to be half as much code in scaffolding as there is in the [final] product." It will be not a surprise that in books on good coding practices, scaffolding is included. See for instance, [3] and [28]. Also in the cost estimation area the phenomenon of code that is not in the final product has been noted [20, p. 18].

The scaffolding around a building provides access to components that workers couldn't otherwise reach. Similarly, software scaffolding gives programmers access to parts that they can otherwise not reach [3]. Often such scaffolding code shows intermediate results in complex calculations and/or manipulations of data. So, scaffolding code is included to understand software. 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. This inspection is not only meant for humans but also for automated transformations to the software. For example, in a first pass, a program is analyzed and the results of the analysis are put in a scaffolding so that we can see that the analysis provides the right information. Then in a second pass, the program can be transformed using the results of the analysis. Recall that scaffolding for development can be parsed by the compiler, for it is built in the programs. In our case scaffolding can also be parsed. However, our scaffolding normally is not part of the language that is used in the source programs. This means that we have to incorporate the scaffolding inside the existing language so that we can parse both the original code plus the scaffolding.

We have developed a systematic approach to the use of scaffolding in software renovation. We developed tools that turn a given context-free grammar into a grammar that incorporates scaffolding. Those tools are part of our Factory Generator. This is a piece of software that generates from a context-free grammar, an architecture that we call a generic software renovation factory for that language. A generic software renovation factory enables rapid development of analysis and transformation components that can use scaffolding: to add scaffolding to the source code, to analyze this scaffolding, and to use the analysis results for making the necessary transformations.

Organization The remainder of this paper is organized as follows. In Section 2 we first give an idea of the use of scaffolding in renovation. In Section 3 we discuss the idea of scaffolding in a grammar context. We give a rough overview of the form that these extended grammars take. Moreover with the aid of simple context-free rules we make the process of obtaining extended grammars transparent. Then we are ready for a full treat of the syntax and semantics of scaffolding in Section 4. In Section 5 we elaborate on the generation process leading to grammars extended with scaffolding. Section 6 discusses applications of scaffolding both in our own work, and work done in IT industry. In Section 7 we conclude by summarizing the main points of the paper.

Related work There is a rich body of work in the realm of scaffolding abstract syntax trees (ASTs). We mention attribute grammars [1, 40] where in addition to grammar rules, also rules for attributing the nodes of the ASTs can be defined. In fact, this could be called scaffolding as well albeit that the underlying AST is decorated and initially the source code itself is not scaffolded. In many and diverse reverse engineering contributions we encountered similar scaffolding of the underlying ASTs of source programs with standard information like control-flow and data-flow information, but not of the source programs themselves. See for instance the literature on program plan recognition [42, 43, 44]. We do not scaffold the AST, but we scaffold the source text, then we parse the resulting scaffolded code. After a (small) transformation step the code and the scaffolding can be unparsed, inspected and/or modified if necessary. Moreover, lexical tools and humans can easily add scaffolding to source code, which is hard on an intermediate AST format. So in our case there is just more code in the form of scaffolding. As far as we know, the use of source-based scaffolding that is *not* source code or comment is new. Of course the concept of using real source code as scaffolding during development is as old as programming, as we already mentioned.

We realize that complex transformations that we carry out using scaffolding can also be implemented using many other transformation systems. In fact, *everything* we do can also be implemented in raw machine language. In our opinion, the relevant issue is not whether it is possible to implement renovation problems using technologies like REFINE [34], COSMOS [15], the ASF+SDF Meta-Environment [23], RainCode [33], Elegant [2, 32], TXL [14], or still other systems. The relevant issue is whether it can be done in a convenient way. Our approach emphasizes how transformations can be implemented as easy as possible. Let us illustrate our point with an example. In the paper [26] a complex migration from a proprietary language to procedural C++ is carried out. One of the issues that is mentioned in this paper is that "it has become apparent from this project, that the transformation logic should be as modular and localized as possible (i.e. different transformation programs for each language construct)". We contacted Kostas Kontogiannis [25] about this project, and he told us that he will never ever implement such tools again. The reason was that implementing such transformations was a too complex task, and it was very difficult to keep track of the activities. We believe that scaffolding contributes to making transformations more easy, just like scaffolding also makes development more easy.

Acknowledgements We thank Pieter Bloemendaal of Triloc Software Engineering Europe for his examples on the use of scaffolding. We present those examples in Section 6.2. We thank Joost Visser (University of Amsterdam) for his comments on attribute grammars.

2 Scaffolding in Action

In this section we give some examples illustrating the use of scaffolding. We do this in order to give the reader an idea before the more formal discussions commence. We recall that in Section 6 more examples are discussed. We will briefly touch upon the issues of context-sensitive transformations, infinitesimal small transformations, phasing of transformations, performance improvement by the use of scaffolding, reuse, the use of lexical tools in combination with scaffolding, and the use of our version of scaffolding in software development.

We start with a COBOL example containing a typical scaffolding. The syntax for scaffolding is to use the keyword `SCAFFOLD` with text brackets, and then use a self-defined type (e.g. `WINDOW`), and again text brackets containing `DATA` (in this case a variable `YY` of type `Data-name`).

```
COMPUTE XX = ZZ - 80.
SCAFFOLD [ WINDOW [ YY : Data-name ] ]
COMPUTE AGE = YY - 80.
```

Context sensitive transformations The above example shows the possible output of a Y2K analysis engine or a hand-written output by an analyst. As can be seen, similar patterns are used in a single program for two tasks. One is a normal calculation, but the other is a date related calculation that will fail after the Year 2000. So in a similar context, the first code does not need to be changed but the second statement needs repair. The recommendation is to use a windowing strategy. A transformation engine that can also parse the scaffolding can now use the information to automatically change it in the right way for each occurrence. The transformed code might look as follows (after the scaffolding has been removed).

```
COMPUTE XX = ZZ - 80.
IF YY > 50
    COMPUTE AGE = YY - 80
  ELSE
    COMPUTE AGE = YY + 100 - 80.
```

We stress that this kind of scaffolding has an important use. Best-in-class Y2K analysis engines, like COSMOS [15, 17, 21], keep track of line and column information. This lexically oriented information can be used to make simple safe changes to the analyzed software like the addition of scaffolding. Then using a context-free change engine, the actual changes can be safely made.

Infinitesimal small transformations Another important purpose of scaffolding is what we call the infinitesimal small transformation. In complex code transformations it is often convenient to be able to access intermediate results. Like in development, where code is scaffolded to keep track of such intermediate results, we use scaffolding to keep track of intermediate transformation steps. Also we can construct the required analysis results step by step before we commence with an actual transformation. We noticed that implementations that make extensive use of transformation technology can become quite complex. Using scaffolding, it is always possible to decompose difficult calculations into smaller steps. Scaffolding enables us to make the steps so small that they seem infinitesimally small. We give a typical example. Suppose that we wish to restructure Assembler/370 programs to improve their maintainability or to migrate it to, say, OS/VS COBOL. In Assembler/370, it is possible to use jumps using a hard coded number of bytes instead of using a label. We provide some abstract code plus the transformed code below. Below, `GO` stands for the jump instruction, `S1`–`S4` are arbitrary Assembler/370 instructions, and `LAB` is an Assembler/370 label. Furthermore, we assume that the number of bytes necessary to store instructions `S1` and `S2` equals 24. So the transformation below is obvious.

```
GO *+24                            GO LAB
S1                                 S1
S2             --->                S2
S3                            LAB  S3
S4                                 S4
```

In order to perform such a transformation, four questions need to be answered.

1. What is the size of the instructions `S1`, ..., `S4`?
2. What is the cumulative byte distance between statements?
3. When is this size equal to 24?
4. Where to put the label `LAB`?

Scaffolding can be used to keep the mental steps in this process as small as necessary. Let us show how we can solve this problem with the aid of scaffolding. First we scaffold the code by providing the byte size of all the instructions. This answers the first question, and puts the result in the code, ready for human inspection.

```
GO *+24
S1 SCAFFOLD [ SIZE [ 8 ] ]
S2 SCAFFOLD [ SIZE [ 16 ] ]
S3 SCAFFOLD [ SIZE [ 8 ] ]
S4 SCAFFOLD [ SIZE [ 4 ] ]
```

The problem of providing the cumulative distance between statements is now reduced to simple addition of numbers. So we use the scaffolding code itself, and we produce the results of this calculation a new type of scaffolding (for the sake of clarity we omit the `SIZE` scaffolding):

```
GO *+24
S1 SCAFFOLD [ CUM_DIST [ 0 ] ]
S2 SCAFFOLD [ CUM_DIST [ 8 ] ]
S3 SCAFFOLD [ CUM_DIST [ 24 ] ]
S4 SCAFFOLD [ CUM_DIST [ 32 ] ]
```

At which statement the byte size equals 24 is now visible in the code for inspection. Moreover, the scaffolding can simply be matched by a transformation, so we can now easily add the label and replace the byte addressing:

```
    GO LAB
    S1 SCAFFOLD [ CUM_DIST [ 0 ] ]
    S2 SCAFFOLD [ CUM_DIST [ 8 ] ]
LAB S3 SCAFFOLD [ CUM_DIST [ 24 ] ]
    S4 SCAFFOLD [ CUM_DIST [ 32 ] ]
```

Finally we can remove the scaffolding:

```
    GO LAB
    S1
    S2
LAB S3
    S4
```

This example illustrates that problems can be decomposed into very small and understandable steps. Moreover, intermediate results of the calculations can be inspected by humans and used by transformations to make the necessary changes.

Phasing of transformations The use of scaffolding can not only represent intermediate results, but can also provide the possibility of delaying the execution of certain transformations. We usually discriminate three phases in a transformation process. First during certain calculations scaffolding is added to mark that something needs to be done. When all the calculations of a certain type are finished, the second phase starts: we collect the scaffolded information and analyze the scaffolding so that a transformation can be prepared. Then in the last phase, we carry out the actual transformations. typical examples of such three phase processes are transformations that require fresh variables. In many transformations we need auxiliary variables. This happens, e.g., when eliminating `GO TO` statements in source code. We can avoid code duplication by introducing so-called (fresh) switch variables. See for instance [8] for a string of such examples. We can first add the variables while eliminating the jump instructions. We then create scaffolding stating that a new variable is made, together with its name. When all `GO TO` statements have been eliminated, we collect the scaffolding and put it at the top of the program. Subsequently, we add all variables in the `DATA DIVISION` in one fell swoop. Finally, we can check the freshness of the variables in a single phase.

Performance In many cases information that has been calculated once, can be used many times. Such information can be stored in a scaffolding so that we can reuse the result of the calculations. For instance, we have implemented a systolic algorithm for eliminating `GO TO` statements in [36]. In this algorithm we move parts of the code that are free of `GO TO`s to a simulated subroutine area. We can scaffold those COBOL paragraphs with the number of `GO TO`s in it. As soon as this number is zero we can ship that part to the simulated subroutine area. We will come back on this issue in Section 6.

Reuse We have seen that we can reuse the results of calculations, which gives better performance. We can also reuse the small steps themselves. Since the steps we make in transforming code are small, the meaning of those steps is often simple and clear, and more important not polluted with idiosyncrasies of large transformations. This encourages reuse of existing transformations and analyses. For instance, the transformations to calculate the number of `GO TO`s in COBOL are the same as the ones to calculate the cumulative byte distance in the Assembler/370 code. They are both simple additions of natural numbers.

AST-directed lexical operations Since we parse all the code, and make changes on the AST level, we also have the power to unparse the code and give it a specific AST-directed layout, possibly enriched with scaffolding. Next, a simple lexical tool like `perl` [41] can be used safely. On the one hand by the fact that we use an unparser, and on the other hand, since we can attach scaffolding to the right places we accomplish that no false positives can be created by the lexical tools. Here we can see an advantage of using scaffolding in the program text instead of in the parse tree representation, as is usually done in the reverse engineering area. In Section 6.2 we will see many more applications of the combination of lexical tools and scaffolding.

Scaffolding and development As noted in the introduction, scaffolding is commonly used in development. Also our form of scaffolding is useful for development. For example, we used it in order to build the transformations that generate the extended grammar that incorporates scaffolding. We will illustrate this in Section 5. So the use of scaffolding as we developed it is not limited to renovation, but can also be used conveniently for development. It is possible to put architectural information in scaffolding. We can think of information residing in the source code that will be inspected by the build process, so that architectural rules can be checked to hold. This idea was born while working on the paper [27] where a process is proposed that includes such scaffolding in the architecture, so that architectural transformations can be carried out. When such scaffolding is in the systems from the start, it can be used to check whether architectural rules are violated.

3 Extended Grammars

In this section we assume that we already have a grammar available that is not geared towards the use of scaffolding. These grammars can be made by hand, as discussed in the case study [7] where an extensive COBOL grammar is made including CICS and SQL for a large bank (the tenth bank of the world). Such grammars can also be generated from the compiler source code as discussed in case study [39] where we generated a 3000 production rule grammar for Ericsson Software Technology directly from the source code of their proprietary compiler.

Before we continue, we mention that all modules are easily regenerated if the input grammar changes. Note that grammar changes are common in the reengineering world.

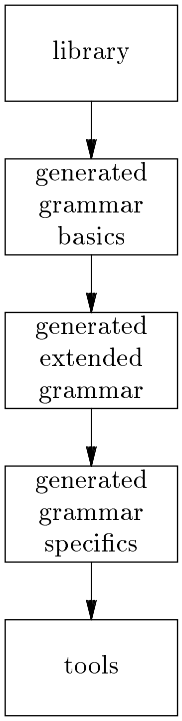

Figure 1: an extended grammar in the context of a software renovation factory.

For each new project often some changes to the grammar are necessary. We have elaborated on such issues in another paper and the interested reader is encouraged to read more in [9] on this topic. For now it is important to realize that everything has been generated from the input grammar so that changes to the input do not imply that the entire architecture has to be redone by hand.

In Figure 1 we depicted important parts of an extended grammar within the context of a generic software renovation factory architecture. First of all we have a library of modules that are language independent. This does not necessarily need to be a fixed library. To give the reader an idea, in such a library, we define a number of often used data types such as Booleans and Integers that are useful in code analyses and that can fruitfully be used as conditions in code transformations. When necessary, new data types can be added effortlessly. Furthermore, the simple scaffolding language itself is defined in the library. Its syntax and semantics are discussed in detail in Section 4. Since we wish to analyze and transform scaffolding itself, we generated for the scaffolding language an analysis and transformation framework as discussed in [10]. When the scaffolding language needs to be adapted to serve some specific need that we have not anticipated, we can easily regenerate its analysis and transformation framework.

In the second box of Figure 1, we generate from the input grammar a number of basic language specific modules. We mention a context-free grammar for dealing with lists and separated lists. Those are very useful when reengineering code (as has been shown in several other papers, for instance, [37]). Some other issues are of a more technical nature, such as a module containing all the sort declarations and such.

In the third box of Figure 1 we actually extend the input grammar with scaffolding syntax. Moreover, we isolate the list and separated list constructs. Since in this paper we focus on scaffolding we will elaborate a little bit more on that aspect and postpone discussions like list isolation to another paper. We proceed to present an input grammar fragment of COBOL. The grammar rule is in SDF [18] (Syntax Definition Formalism). The `MOVE` statement as constructed in [7] looks as follows.

```
"MOVE" Corr A-exp "TO" Data-name-p -> Statement
```

The sort name `Corr` stands for a possible occurrence of the keyword `CORRESPONDING`, the sort name `A-exp` is an arithmetical expression, and the sort name `Data-name-p` is a list of one or more data names. Of course, the sort name `Statement` represents a COBOL statement. It is our intention to be able to parse not only the COBOL code (possibly containing CICS/SQL/etc), but also scaffolding code as we have seen in the previous section. A sample of code that we might want to analyze or transform (and thus parse) is:

```
SCAFFOLD [ WINDOW [ TMP : Data-name ] ]
MOVE 19 TO TMP.
```

This code might have been produced by another tool, for instance a Y2K analysis engine. Or the code has been produced by hand by a Y2K analyist, working in a Y2K factory, such as COSMOS 2000 [15]. When we wish to further analyze and/or transform this code, we need to be able to parse the above code. This means that we have to adapt the above grammar rule so that the intermediate scaffolding code is taken into account. We inject in a structured way in the input grammar an extra or fresh sort that will take care of this. We will now show the changed grammar so that we can parse the scaffolding in the above fragment.

```
X-s "MOVE" Corr A-exp X-s "TO" Data-name-p -> Statement
```

What happened here is that *before* each terminal we have added a fresh sort name `X-s`. This stands for zero or more occurrences of sort `X`. This sort `X` is defined in the library. The `X` stands for extension, or extended. Scaffolding is of sort `X` so now we can parse the fragment. The reader might have expected another name for this sort, like `Scaffolding-s` or something similar. We put in an extra layer for the sort introduction for a good reason. Sort introduction does not only serve scaffolding but also other purposes. We mention that sort introduction also enables parsing code including its comments. For, if we wish to transform code containing comments, it is not a good idea to remove those comments. Consequently, comments are also of sort `X`, so that we can deal with comments as well. We refer the reader to [8] where transformations that include comments are being discussed in detail. Since scaffolding as well as comments can occur at virtually any location in code, their introduction in a grammar is obtained in the same way as with scaffolding. Therefore, we have chosen to not use `Scaffolding-s` as sort name. Other types of extensions to grammars that one may wish to distinguish from comments are compiler directives, such as the compiler directives that are used in CHILL [19] with their typical diamond notation (`<>`). Since they can occur also at virtually every location in the code, they are comparable to comments, but we must be able to separate them from comments.

The above introduction with `X-s` might give rise to the idea that the introduction transformation is a trivial one. This is not the case. First of all we cannot just "prefix" every sort name with `X-s`, for this would give a truly ambiguous grammar. Therefore, we only prefix the terminals. We make the same modifications to all other context-free rules. So also for the production rule that defines `Corr`. With this procedure we cover almost everything. We miss the sorts that are lexically defined and context-free used. They are in fact, terminals in disguise. They do obviously not have a context-free definition and will be missed unless we take precautions. Those sorts need sort introduction as well. For instance a `Data-name-p` is a COBOL identifier. Thereto we generate a module that takes care of the prefixing of lexically defined sorts that are used context-free. For example, for `Data-name-p` we rename the original name in the lexical syntax by `Data-name-lex-p` and we define a context-free rule in a specially created module with the following form:

```
X-s Data-name-lex-p -> Data-name-p
```

Once we have interspersed the input grammar with the extra sort `X-s`, we call this an extended grammar, or an `X` grammar. In Section 5 we will come back on how we generate `X` grammars.

We process with the fourth box of Figure 1. For a start, as was seen in the example COBOL fragment, the scaffolding itself makes use of constructs of the COBOL language. In this case we mean the COBOL variable `TMP`. We have experienced that it is useful to have access to the entire language inside the scaffolding. Since we still wish to have the scaffolding language as fixed as possible, we put in this language an extra layer to deal with this phenomenon: all the code that can appear as information in scaffolding is known as `DATA`. We automatically generate a module that will make any sort of the input grammar also to be of sort `DATA`. For instance this means that for the above COBOL grammar rule we generate:

```
Corr ":Corr"                -> DATA
A-exp ":A-exp"              -> DATA
Data-name-p ":Data-name-p" -> DATA
Statement ":Statement"      -> DATA
```

The `:Data-name-p` part is present for typing reasons: then we can see that `TMP` is of type `Data-name-p`. In the fourth box we also generate a native pattern language. This is a language that enables us to write patterns that look as much as possible like the original code. We kindly refer the reader to an extensive treatment of native pattern languages in paper [37]. Furthermore, we generate some auxiliary technical functionality that makes the construction of tools more easy. Finally, we generate for the `X` grammar a generic analysis and transformation framework as discussed in great detail in [10]. The analysis and transformation framework for scaffolding is directly imported by the language dependent analysis and transformation framework.

In the last box of Figure 1, we depicted that we can build tools. When we have this complete architecture, we are in a position to construct renovation tools, like analysis and transformation components. Moreover we can use scaffolding in them. Since the focus in this paper is not on tools, but on scaffolding and its use we will not discuss tools in detail. In many other papers, we focus more on tools, and the reader is kindly invited to consult the various papers of the authors.

4 Syntax and Semantics of Scaffolding

The scaffolding language that we designed is a very simple language with a very simple semantics. The purpose of this language is that it is possible to scaffold relevant information that can be processed conveniently. Below we depict the SDF syntax [18] of the most important library module containing the heart of the scaffolding language.

```
imports Data
exports
  sorts COMMENT SCAFFOLD X X-s
  context-free syntax
    SCAFFOLD                    -> X
    COMMENT                     -> X
    X*                          -> X-s
    "SCAFFOLD" "[" DATA-s "]" -> SCAFFOLD
  variables
    "X"  [0-9]* -> X
    "X*" [0-9]* -> X *
    "X+" [0-9]* -> X +
hiddens
  variables
    "d*"  [0-9]+ -> DATA *
```

The only structure in this module is that the extension sort `X` is either of sort `SCAFFOLD` or of sort `COMMENT`. We have not implemented what `COMMENT` is, for this is language dependent and defined in a syntax module of language L if we want to renovate code written in L. Here we know that whatever the syntax of the comments is, it is also of sort `X`. The form of the scaffolding is partly defined. We chose to use prefix notation using the keyword `SCAFFOLD` and using text-brackets. The contents of the scaffolding is however language dependent. Inside a scaffolding everything is a list of zero or more objects of type `DATA`. We declare variables of sort `X`, and their list variants: `X*`, `X+` zero resp. one or more occurrences of `X`. Finally we have hidden variables that are used locally to express the semantics. We give the semantics of the above syntax in an ASF module [4]:

```
equations
[1] SCAFFOLD [ d*1 ] SCAFFOLD [ d*2 ] =
      SCAFFOLD [ d*1 d*2 ]
[2] X*1 SCAFFOLD [ ] X*2 = X*1 X*2
```

Equation `[1]` expresses that two consecutive occurrences of `SCAFFOLD` reduce to one. In equation `[2]` we express that an empty `SCAFFOLD` can be omitted. Note that `X*1`, `X*2` are lists of arbitrary `SCAFFOLD`s or `COMMENT`s. The `d*` variables represent arbitrary lists of type `DATA`. We proceed to present the `DATA` syntax and semantics. We simplified the definition slightly for the sake of clarity.

```
imports LayoutChars
exports
  sorts DATA DATA-s SCAFFOLD-TYPE
  lexical syntax
    [A-Z_]+ -> SCAFFOLD-TYPE
  context-free syntax
    SCAFFOLD-TYPE  "["  DATA-s  "]"  -> DATA
    DATA*                            -> DATA-s
hiddens
  variables
    "d*" [0-9]* -> DATA *
    "d"  [0-9]* -> DATA
    "st"        -> SCAFFOLD-TYPE
```

We import some `LayoutChars`, which we will not discuss since it just describes layout. As can be seen we define a `SCAFFOLD-TYPE` lexically. This means that we can define our own types of scaffolding. Since they are necessary for many and diverse tasks, it is not a good idea to invent a fixed number of scaffold types. We have chosen to use upper-case and underscores for `SCAFFOLD-TYPE`s; this choice is completely arbitrary and can be changed at wish. Subsequently, we define the language independent parts of the sort `DATA`. `DATA` can either be typed using a `SCAFFOLD-TYPE` and the text brackets, or untyped. We noted earlier that in order to be able to reason about language dependent issues like source code fragments in the scaffolding we generated modules so that all the sort names are of type `DATA`. The form of this syntax has been shown earlier. Finally, we define a few hidden variables in order to express the semantics of the above syntax. The semantics is very simple and discussed below.

```
equations
[1] d*1 d d*2 d d*3 = d*1 d d*2 d*3
[2] d*1 st [ ] d*2 = d*1 d*2
[3] d*1 st [ d*4 ] d*2 st [ d*5 ] d*3 =
    d*1 st [ d*4 d*5 ] d*2 d*3
```

Equation `[1]` expresses that double occurring data is removed. Equation `[2]` implies that scaffold-types that are empty can be removed. Finally, equation `[3]` collects data of the same type.

Details on semantics The ASF equations are interpreted by our implementation platform, the ASF+SDF Meta-Environment [23], as conditional term rewriting systems with positive/negative premises. The formal semantics of such systems has been subject of study in [22, 29, 30, 16]. In those papers, the meaning of negative premises in conditional term rewriting is being studied. ASF is a modular specification mechanism. This is illustrated by the import relation between the two modules `Data` and `X`. A natural question that arises is whether the defined semantics in the module `Data` is preserved when we extend the semantics with equations in module `X`. This is guaranteed by a conservative extension theorem that has been proved in [16]. All this implies that the scaffolding language is not only carefully designed but also has a well-defined formal semantics since the equations are to be interpreted as formal mathematical objects (CTRSses).

5 Generating Extended Grammars

In our opinion, the grammar of the code that needs to be renovated is the most valuable asset in order to facilitate automated tool support for renovation. A string of papers confirming this opinion has been published. We mention [11] where powerful formatters are generated from context-free grammars. We mention [35] where a GLR parser generator for interactive environments is discussed. In the study [10] an architecture for rapid development of analysis and transformation tools for renovation is discussed. We also generate so-called native pattern languages from context-free grammars [37]. We mention [39] where we generate reengineering grammars from compiler grammars. The implementation system that we use is called the ASF+SDF Meta-Environment [23]. This is a programming environment where all the functionality is generated from the context-free grammar.

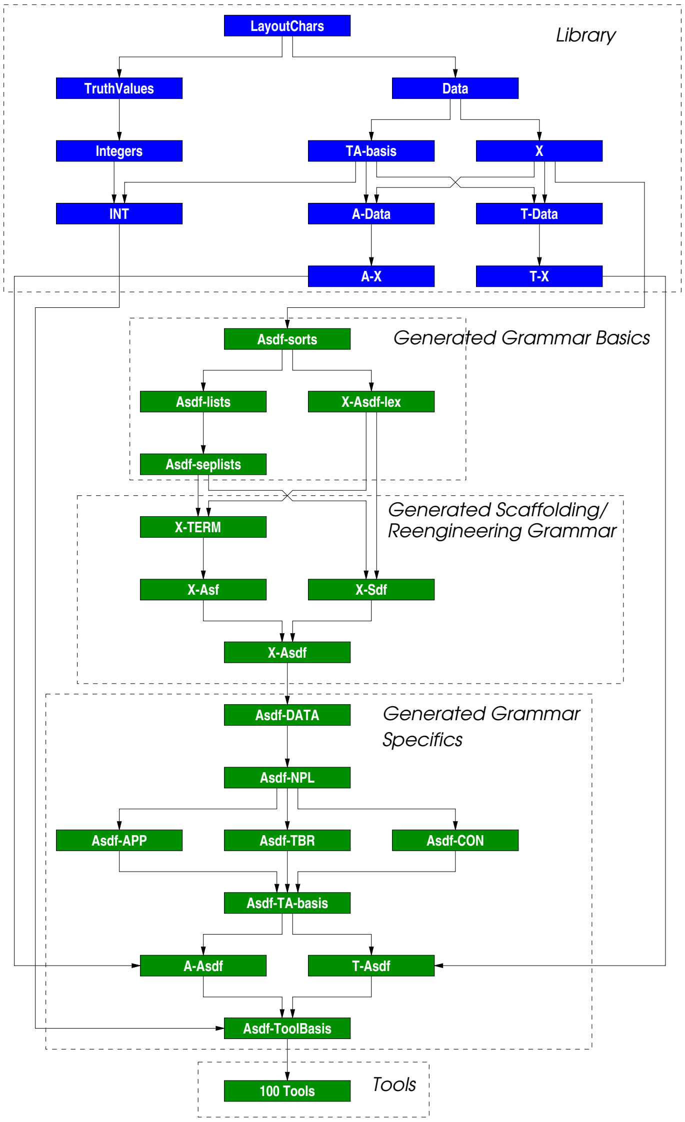

Figure 2: The Factory Generator

In this paper, we add one more result: we generate extended grammars from context-free grammars. In this section we elaborate in more detail on an example of the use of scaffolding: our Factory Generator. The Factory Generator generates most of the abovementioned issues including extended grammars from context-free grammars. In Figure 2 we depicted the import graph of the modules that form the Factory Generator. Note that the dashed rectangles are an exploded view of the boxes of Figure 1. This might be confusing, for, in Figure 1 we were discussing an architecture containing extended grammars themselves. Indeed this is the case, and the reason that this architecture comes back in our Factory Generator is that it uses scaffolding as well (as already announced in Section 2). So our Factory Generator *contains* scaffolding and generates extension grammars to *handle* scaffolding. Seen in this light it is not a surprise that the Factory Generator is bootstrapped. For, a grammar is described in a formalism that has itself a grammar. In our case the grammars are described in SDF [18]. We generate not only syntax but also semantics for the scaffolding language (as we have discussed briefly). The semantics is expressed in ASF [4]. So the input of the Factory Generator is an SDF specification of a language L, and its output is an ASF+SDF specification. We call this output an L-factory or a software renovation factory for L, or just a (software renovation) factory if no confusion about the language can arise.

We proceed to discuss Figure 2. In the library, we have reused standard library ASF modules for Booleans and Integers. We implemented the `X` grammar and the `Data` grammar that have been discussed in Section 4. The other files in the library on top of the `Data` and `X` have been generated with an earlier version of the Factory Generator. We could call that part of the library an X-factory. The next three dashed rectangles are all generated. Of course, when we started to implement this, we first had to make one extended grammar by hand. This is the grammar in the third dashed box: we reused the ASF+SDF grammar, called `Asdf`, from the ASF+SDF Meta-Environment and we carried out the transformation by hand (this resulted in the modules `X-TERM`, `X-Asf`, `X-Sdf`, `X-Asdf`. In those modules we introduced all the extensions at the right locations and we isolated the list constructs. The process was iterative: each iteration we implemented the minimal part by hand so that we could generate a full new version of the Factory Generator. After a few iterations, we reached a fixed point, meaning that when we fed the `Asdf` specification to the Factory Generator, the generated modules were exactly the ones in the three boxes on top of the library. We implemented on top of the generated architecture about 100 small tools with the functionality that they generate for *any* grammar for a language L written in SDF, an L-factory.

It is out of scope for this paper to discuss all the tools that we implemented. We restrict ourselves to discussing the tools that introduce the extended grammar. We take a very simple input grammar to clarify the process of introducing the extensions. Consider the simple SDF grammar below.

```
lexical syntax
  [A-Z] -> Character
  [0-9]+ -> Number
  Character+ -> Word
context-free syntax
  Word+ "." -> Sentence
```

```
"paragraph" Number "." Sentence+ -> Paragraph
```

The above grammar describes characters, numbers, words, sentences, and how to combine this into a paragraph. When we enter the process, the grammar has already been changed to isolate list constructions. The reason why we do this is technical: it enables more easy development of renovation components. We show only the changed syntax part, and not a number of extra modules that have been created during the process:

```
lexical syntax
  [A-Z]      -> Character
  [0-9]+     -> Number
  Character+ -> Word
context-free syntax
  Word-p "."                         -> Sentence
  "paragraph" Number "." Sentence-p -> Paragraph
```

On this grammar fragment we will apply six tools that eventually will introduce the extensions. We will focus on the use of scaffolding while doing this.

DefLexSorts This tool analyses the grammar and puts a list of used lexicals on top of the grammar in a scaffolding. We show the scaffolding below (we removed additional scaffolding not related to X-introduction for the sake of clarity):

```
SCAFFOLD [ DEF_LEX_SORTS [
    Character :Id-lex
    Number :Id-lex
    Word :Id-lex ]]
```

As can be seen in the input grammar, indeed all the sorts that have been defined lexically, are extracted in this phase, and stored in a typed scaffolding.

UsedCFSorts We also want to know which sorts are used in the context-free syntax. The reason is that only those lexically defined sorts that can actually occur context-free are amendable for X–introduction. As we can see, indeed the list below provides all the context-free used sorts.

```
SCAFFOLD [ USED_CF_SORTS [
    Word :Id-lex
    Sentence :Id-lex
    Word-s :Id-lex
    Sentence-s :Id-lex
    Word-p :Id-lex
    Number :Id-lex
    Sentence-p :Id-lex ]]
```

Note that a two extra sorts with a `-s` postfix are defined. They represent zero or more lists. They are added in an extra module so that we can also construct tools matching zero or more statements. Also now, nothing has changed to the grammar fragment itself.

LexCfSorts This tool lists those lexical sorts that are used context-free. We can obtain this result by calculating the difference of the `UsedCFSorts` scaffolding and the `DefLexSorts` scaffolding. As can be seen here, the grammar itself is not analyzed again, but the calculations have migrated to the analysis results. Indeed, the added scaffolding below shows the right answer:

```
SCAFFOLD [ LEX_CF_SORTS [
    Word :Id-lex
    Number :Id-lex ]]
```

DefineLex-X At this moment we have all the information that is necessary to inject the sort `X` in the lexicals that are relevant. The tool `DefineLex-X` just takes the above scaffolding and creates a module containing the following syntax:

```
context-free syntax
  X-s Word-lex   -> Word
  X-s Number-lex -> Number
```

In Section 4 we defined that `X-s` is zero or more occurrences of `X`. This implies that all the original programs can still be parsed. Also that all comment of these programs is now recognized, and moreover, scaffolding is allowed and can be parsed. We are not ready yet.

PostFixLex Now we have introduced the new sorts `Word-lex, Number-lex` but we have not yet changed the original grammar so that the new sort names will be used. We do that using the tool that will postfix the lexical definitions in the grammar that are used context-free. The lexical syntax of the grammar fragment is changed thus:

```
lexical syntax
  [A-Z]      -> Character
  [0-9]+     -> Number-lex
  Character+ -> Word-lex
```

And indeed the sort `Character` that is not used context-free is not changed. Now the final step.

Add-X Now we can also change the context-free grammar using a very simple tool that adds the sort `X` throughout the grammar on every location just before a literal. This is the changed context-free grammar fragment:

```
context-free syntax
  Word-p X-s "."                                   -> Sentence
  X-s "paragraph" Number X-s "." Sentence-p -> Paragraph
```

As an example of what we call an infinitesimal small transformation, we provide the implementation of the last transformation.

```
imports Asdf-Toolbasis
exports
  context-free syntax
    "Add-X" -> TRANSFORM

equations
[1] Add-X_CfElem(Lit1) = X-s Lit1
```

The SDF part imports the entire Factory Generator by way of `Asdf-Toolbasis`. Since this generator is itself a factory, we have access to generic transformation components as discussed in [10]. The only thing we need to do to build a tool that can use all the generic functionality is to declare it being of sort `TRANSFORM`. After that, we have for each sort of the `Asdf`-factory a tool available. We use `Add-X_CfElem`, which is the tool that gives access to arbitrary context-free elements. The variable `Lit1` matches an arbitrary literal (a terminal). When such a literal is matched it is replaced by `X-s` followed by the same literal, as expressed in equation `[1]`.

This section did not only illustrate the generation process for extended grammars, but it also showed how scaffolding can be used during development of transformation and analysis tools. Such tools are typical for the implementation of software reengineering tasks.

Real factories We have not only used toy examples as input to our Factory Generator. We have used it to improve and extend the COBOL-factory that we already had. The extensive experience that we have with the use of our COBOL software renovation factory was a reliable guideline for implementing improvements and modifications to it. We mention a few recent references containing real-world applications of our COBOL-factory: [8, 37, 38, 36, 13]. We have reused (and enhanced) the original COBOL grammar that we made by hand [7]. This was input for the Factory Generator so that we now have an enhanced COBOL-factory. One of the enhancements is that we now can use scaffolding for COBOL analyses and transformations.

We are also implementing a software renovation factory for Ericsson Software Technology AB. Using CALE (Computer Aided Language Engineering) technology [39] we first extracted and migrated about 20 grammars of their proprietary language for programming real-time switching systems (called SSL, short for Switching System Language). The resulting overall grammar contains about 3000 production rules. This specification has been input for the Factory Generator. We have generated a first version of an SSL-factory.

It takes about five hours to generate an entire COBOL-factory. It takes about 50 hours to generate an SSL-factory. Both times are measured in an interpreted environment. We also note that our Factory Generator not only generates the scaffolding, but an entire software renovation factory architecture. We can speed up the process considerably when we compile our ASF specifications to C. We refer to [5], for a recent paper on the compilation and memory management of this compiler, and to [31] for on line benchmarks. We have not yet compiled our specifications, since we do not generate new factories quite often. In large reengineering companies it is not uncommon to have a weekly release of software renovation factories. For such situations it is of course a better idea to make use of a compiled Factory Generator. Since we still make small changes to the generation process while becoming more and more experienced with scaffolding as we implemented it, we consider it more convenient to use an interpreted environment, to enable rapid development.

6 Applications

Applications of scaffolding are usually complex. For, if an analysis or transformation would be simple, there would be no need to use scaffolding. Also this is analogous to the classical use of scaffolding: it is added to keep track of complex situations. This implies that actually explaining an entire example that makes use of scaffolding is too lengthy to present here. Instead we will sketch a number of applications that are particularly suited to use scaffolding both by us in Section 6.1 and by others in Section 6.2.

6.1 Applications for a Swiss Bank

In a recent study we implemented a systolic algorithm that structures COBOL/CICS code of a Swiss bank. The goal of that work was to separate the coordination of the programs from the computations. This study has been elaborately discussed in [36]. We will discuss one of the many transformation steps of that process in order to illustrate the use of scaffolding. One of those steps in the process is that we move COBOL paragraphs from their original location to a simulated sub-routine section. Although this so-called `MovePar` step is presented as one step in the process in paper [36], it can be decomposed into several steps that make use of scaffolding.

First we wish to show that it is not easy to move a paragraph to another place without changing the semantics of the program. Suppose our goal is to move `PAR-2` to another location. The first-order approximation of this transformation is:

```
PAR-1. Sentence1                PAR-1. Sentence1
PAR-2. Sentence2     -->               PERFORM PAR-2
PAR-3. Sentence3                PAR-3. Sentence3
```

In this example we assume that `PAR-2` is now on another location where it can be called, like has been done in the modified code. Suppose that `PAR-1` is called somewhere. Then the above transformation is not correct. For, in the original code when `PAR-1` is performed, `Sentence1` is executed and when the paragraph is finished, the control-flow goes back to the statement below the `PERFORM PAR-1`. In the transformed code both `Sentence1` and `Sentence2` are executed. So a precondition for moving paragraphs is that the previous paragraph is *not* performed somewhere in the program. We will proceed with the steps that are necessary to safely move paragraphs to other locations.

Analyze the code We scaffold the paragraphs with crucial information. First of all we only wish to move paragraphs that are free of jump instructions. Therefore, we add the following scaffolding to each paragraph (we assume that in `PAR-2` there are no `GO TO` statements):

```
SCAFFOLD [ NUMBER_OF_GO [ n1 :Integer ] ]
PAR-1. Sentence1
SCAFFOLD [ NUMBER_OF_GO [ 0 :Integer ] ]
PAR-2. Sentence2
SCAFFOLD [ NUMBER_OF_GO [ n2 :Integer ] ]
PAR-3. Sentence3
```

Furthermore, we need to know the label of the next paragraph. Therefore, a simple analysis tool takes care of that scaffold information (we omit the other scaffolding for the sake of clarity):

```
PAR-1. Sentence1
SCAFFOLD [ NUMBER_OF_GO [ 0 :Integer ]
           NEXT_PAR [ PAR-3 :Label ] ]
PAR-2. Sentence2
PAR-3. Sentence3
```

We also need to know whether the previous paragraph is being performed. So there is an analysis tool that adds this, too, to the scaffolding:

```
PAR-1. Sentence1
SCAFFOLD [ NUMBER_OF_GO [ 0 :Integer ]
           NEXT_PAR [ PAR-3 :Label ]
           PERFORMED [ FALSE :Boolean ] ]
PAR-2. Sentence2
PAR-3. Sentence3
```

We also want to know the paragraphs that jump to PAR-2. We want to know this, so that we can change those GO TO statements to PERFORMs plus another GO TO statement to the next paragraph.

```
PAR-1. Sentence1
SCAFFOLD [ NUMBER_OF_GO [ 0 :Integer ]
           NEXT_PAR [ PAR-3 :Label ]
           PERFORMED [ FALSE :Boolean ]
           GO_FROM [ PAR-Q :Label ] ]
PAR-2. Sentence2
PAR-3. Sentence3
```

In the above example PAR-Q is a paragraph on an entirely different location in the source program. The scaffolding expresses that in that paragraph a GO TO PAR-2 is made.

Analyze the scaffolding Just like the example where we made the grammar changes, we will now also analyze the ensued scaffolding. Using a simple tool we add scaffolding that provides a predicate to all paragraphs that can be moved. Obviously, PAR-2 satisfies all the conditions, so we scaffold it TB_MOVED shorthand for to be moved.

```
PAR-1. Sentence1
SCAFFOLD [ NUMBER_OF_GO [ 0 :Integer ]
           NEXT_PAR [ PAR-3 :Label ]
           PERFORMED [ FALSE :Boolean ]
           GO_FROM [ PAR-Q :Label ]
           TB_MOVED [ PAR-2 :Label ] ]
PAR-2. Sentence2
PAR-3. Sentence3
```

Then in another step we collect the TB_MOVED at the top of the program so that in a next step a transformation can use this information to move all the paragraphs that are contained in the scaffolding TB_MOVED.

GO TO shifting now we are in a position to use the scaffolding so that we can carry out a first transformation step: in all the paragraphs containing a GO TO PAR-2 we have to make a change. Recall that in our PAR-Q there was a call to PAR-2. Below we depicted the original code and the transformed code:

```
PAR-Q. Statement1*                PAR-Q. Statement1*
       IF Condition1    -->              IF Condition1
         GO TO PAR-2.                      PERFORM PAR-2
                                           GO TO PAR-3.
```

Note that we turn a GO TO in another GO TO. Therefore, we call it GO TO shifting. The idea of the algorithm is that we turn difficult GO TOs into simple ones so that eventually they all will be eliminated. See [36] for details.

Scaffold paragraphs Now we can safely move PAR-2. We do this in a few steps. First we put the paragraphs that can be moved in a scaffolding.

```
PAR-1.  Sentence1
SCAFFOLD [ NUMBER_OF_GO [ 0 :Integer ]
           NEXT_PAR [ PAR-3 :Label ]
           PERFORMED [ FALSE :Boolean ]
           GO_FROM [ PAR-1 :Label ]
           MOVE_PAR [ PAR-2. Sentence2 :Paragraph ] ]
PAR-3. Sentence3
```

We do this, since now we can eliminate GO TOs in the newly created code (for details we refer to [36]). When we have new jump instructions removed, we can reiterate the above steps and scaffold more paragraphs, and so on.

Move paragraphs When all the GO TOs are eliminated, we actually move the paragraphs to their location. We do this in two steps: first we put the paragraphs each in their own subroutine section:

```
SECTION A.
  PAR-1. Sentence1
         PERFORM PAR-2.
  PAR-3. Sentence3
BAR SECTION.
  BAR-PARAGRAPH.
    STOP RUN.
A-SUBROUTINES SECTION.
PAR-2. Sentence2
```

As can be seen, the above code is functionally equivalent to the original code. The difference is that using this algorithm we can slowly separate the business logic from the control logic.

Remove double sections We move each paragraph that can be moved to its own section. This means that as soon as we move more than one paragraph, we have double subroutine sections. We remove the double ones with a very simple tool. This means that we transform:

```
A-SUBROUTINES SECTION.          A-SUBROUTINES SECTION.
PAR-2. Sentence2         -->      PAR-2. Sentence2
A-SUBROUTINES SECTION.            PAR-4. Sentence4
PAR-4. Sentence4
```

As can be seen, all the individual steps are quite simple, but the resulting transformation is complex. Moreover, we make extensive use of scaffolding information. These transformations have all the same pattern: first we analyze the code, then we make calculations on the analysis. This leads to a conclusion, and we have the right information to carry out a number of transformations.

6.2 Applications in the Reengineering Industry

Before publication of this paper, we communicated our ideas on scaffolding source code to Triloc Software Engineering in the realm of our strategic cooperation. Triloc is working on a number of applications where scaffolding comes into play. We briefly summarize a number of their applications.

COBOL margins People familiar with COBOL know that the left and the right margins may contain comment. Most of the time we can ignore that comment since at the left-hand margin we have the line number, and the right margin reiterates the name of the program. During renovation, we can safely strip this code, and obtain so-called `666`-code (see [7] for the etymology of this name). However, sometimes the margins contain valuable maintenance records, and some COBOL compiler vendors used the margins for smart issues. In the latter case we cannot strip the code before parsing it. Now imagine this margin code: it occurs on virtual every location in the grammar. This means that the mechanism of `X`-introduction is a perfect means to parse that comment. The idea is that first using a lexical tool, the column information is put in a scaffolding, and then, a grammar that has been extended can parse the crucial information. The nice thing here is that we do not have to adapt the grammar for incorporating the margins. Of course, after unparsing a lexical tool will put the appropriate columns back to their location.

`COPY` **member expansion and collapsing** A common problem of parsing code, is that sometimes include statements, of macros must be expanded in order to get grammatically correct code. In COBOL this is the `COPY` statement. In a lexical preprocessing phase it is possible to expand the contents of the `COPY` members [13]. This can occur recursively. Of course, when we are done with the renovations, they have to be collapsed in order to obtain the original sources again. It is possible to use scaffolding to keep track of this process so that the `COPY` statements can be expanded and collapsed. In [13] the postprocessing checks whether the expanded code has changed, and if so, that it changed in all occurrences. Then the changed code is collapsed to a `COPY` statement and the code is saved to the appropriate file.

Substitution memory The above `COPY` statement has a simple form of substitution available. Its syntax is `COPY FILE REPLACING A BY B`. Scaffolding can be used to memorize the substituted value, so that when the `COPY` statement is put back it is still known what substitution has been applied.

Column corrections The unparser will pretty print expanded and collapsed `COPY` statements sometimes in a different location. For instance, when we have on a single line

```
01 COPY X. 03 FOO  PIC 99
```

there is a good chance that this layout is not recovered by the unparser. A common way of customers to checking the changes to the software is to run simple lexical comparison tools like `diff` on the original code and the transformed code. In order to avoid differences caused by the unparser, it is possible to scaffold the above code fragment with precise column information, so that during postprocessing of the code the layout idiosyncrasies will be put back in the text.

Variable length margins Sometimes COBOL programs have no left margin, sometimes there is a 6 character margin, and sometimes a 7 character margin. This kind of information needs to be known after renovation, especially when the code has been stripped (left and right margins are removed during preprocessing). In a scaffolding the amount of margin can be stored so that during postprocessing the correct margins are restored.

7 Conclusions

In this paper we have introduced the concept of scaffolding in a software renovation context. We have explained that starting from a context-free grammar suited for parsing code, we can automatically derive an extended context-free grammar that is targeted towards parsing scaffolded source code. In order to facilitate a structured approach to scaffolding source code, we have designed small but effective scaffolding syntax and semantics that is automatically interspersed with the grammar of the to-be-renovated code. We discussed the architecture, called the Factory Generator, that makes such transformations on grammars possible. We applied our research on large real-world languages: COBOL with extensions, and a huge real-time language owned by a large telecommunications manufacturer. With a string of applications, both from our own work and form the reengineering industry, we have showed that scaffolding source code for software renovation is useful.

References

[1] H. Alblas and B. Melichar, editors. *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*. Springer-Verlag, 1991.

[2] L. Augusteijn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, Eindhoven University of Technology, 1993.

[3] J. Bentley. *More Programming Pearls – Confessions of a Coder*. Addison-Wesley, 1988.

[4] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.

[5] M.G.J. van den Brand, P. Klint, and P. Olivier. Compilation and memory management for ASF+SDF. In S. Jähnichen, editor, *Proceedings of the eight International Conference on Compiler Construction*, volume 1575 of *LNCS*, pages 198–213. Springer-Verlag, 1999.

[6] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

[7] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, electronic Workshops in Computing. Springer verlag, 1997. Available at http://adam.wins.uva.nl/~x/coboldef/coboldef.html.

[8] 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, *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, pages 11–19, 1998. Available at http://adam.wins.uva.nl/~x/cfn/cfn.html.

[9] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/˜x/ref/ref.html.

[10] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 1999. To appear. Available at http://adam.wins.uva.nl/˜x/scp/scp.html. An extended abstract with the same title appeared earlier: [6].

[11] 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.

[12] F.P. Brooks Jr. *The Mythical Man-Month – Essays on Software Engineering*. Addison-Wesley, 1995. Anniversary Edition.

[13] J. Brunekreef and B. Diertens. Towards a user-controlled software renovation factory. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 83–90. IEEE Computer Society, 1999.

[14] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[15] Emendo Software Group, The Netherlands. *Emendo Y2K White paper*, 1998. Available at http://www.emendo.com/.

[16] W.J. Fokkink and C. Verhoef. Conservative extension in positive/negative conditional term rewriting with applications to software renovation factories. In J.-P. Finance, editor, *Proceedings 2nd Conference on Fundamental Approaches to Software Engineering*, volume 1577 of *LNCS*, pages 98–113, Amsterdam, 1999. Springer-Verlag.

[17] B. Hall. Year 2000 tools and services. In *Symposium/ITxpo 96, The IT revolution continues: managing diversity in the 21st century*. Gartner Group, 1996.

[18] 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.

[19] International Telecommunication Union. *Recommendation Z.200 (11/93) - CCITT High Level Language (CHILL)*, 1993.

[20] C. Jones. *Programming Productivity*. McGraw-Hill, 1986.

[21] N. Jones. Year 2000 market overview. Technical report, Gartner Group, Stamford, CT, USA, 1998.

[22] S. Kaplan. Positive/negative conditional rewriting. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, volume 308 of *LNCS*, pages 129–143. Springer-Verlag, 1988.

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

[24] D.E. Knuth. *The Art of Computer Programming – Fundamental Algorithms*. Addison-Wesley, 1968.

[25] K. Kontogiannis. University of Waterloo, Personal Communication, September 1998.

[26] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Muller, and J. Mylopoulos. Code migration through transformations: An experience report. In *Proceedings of CASCON'98*, 1998.

[27] R. Krikhaar, A. Postma, M.P.A. Sellink, M. Stroucken, and C. Verhoef. A two-phase process for software architecture improvement. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 371–380, 1999. Available at http://adam.wins.uva.nl/˜x/sai/sai.html.

[28] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[29] C. K. Mohan and M. K. Srivas. Conditional specifications with inequational assumptions. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems (CTRS '88)*, volume 308 of *Lecture Notes in Computer Science*, pages 161–178. Springer-Verlag, 1988.

[30] C. K. Mohan and M. K. Srivas. Negation with logical variables in conditional rewriting. In N. Dershowitz, editor, *Rewriting Techniques and Applications (RTA '89)*, volume 355 of *LNCS*, pages 292–310. Springer-Verlag, 1989.

[31] P. Olivier. Benchmarking of functional/algebraic language implementations, 1999. Available at http://adam.wins.uva.nl/˜olivierp/benchmark/index.html.

[32] Philips Electronics B.V., The Netherlands. *The Elegant Home Page*, 1993. http://www.research.philips.com/generalinfo/special/elegant/elegant.html.

[33] RainCode, Brussels, Belgium. *RainCode*, 1.07 edition, 1998. ftp://ftp.raincode.com/cobrc.ps.

[34] Reasoning Systems, Palo Alto, California. *Refine User's Guide*, 1992.

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

[36] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 72–82, 1999. Available at http://adam.wins.uva.nl/˜x/cics/cics.html.

[37] M.P.A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the Fifth Working Conference on Reverse Engineering*, pages 89–103. IEEE Computer Society, 1998. Available at http://adam.wins.uva.nl/˜x/npl/npl.html.

[38] M.P.A. Sellink and C. Verhoef. An architecture for automated software maintenance. In D. Smith and S.G. Woods, editors, *Proceedings of the seventh International Workshop on Program Comprehension*, 1999. To appear. Available at http://adam.wins.uva.nl/˜x/asm/asm.html.

[39] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. In H. Yang and L. White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 245–255, 1999. Available via http://adam.wins.uva.nl/˜x/com/com.html.

[40] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. Higher order attribute grammars. *SIGPLAN Notices*, 24(7):131–145, 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[41] L. Wall and R.L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[42] L. M. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, 1992.

[43] S. Woods. *A Method of Program Understanding using Constraint Satisfaction for Software Reverse Engineering*. PhD thesis, University of Waterloo, 1996.

[44] S.G. Woods, A. Quilici, and Q. Yang. *Constraint-based Design Recovery for Software Reengineering: Theory and Experiments*. Kluwer Academic Publishers, 1997.