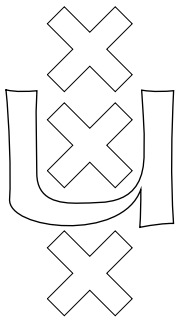
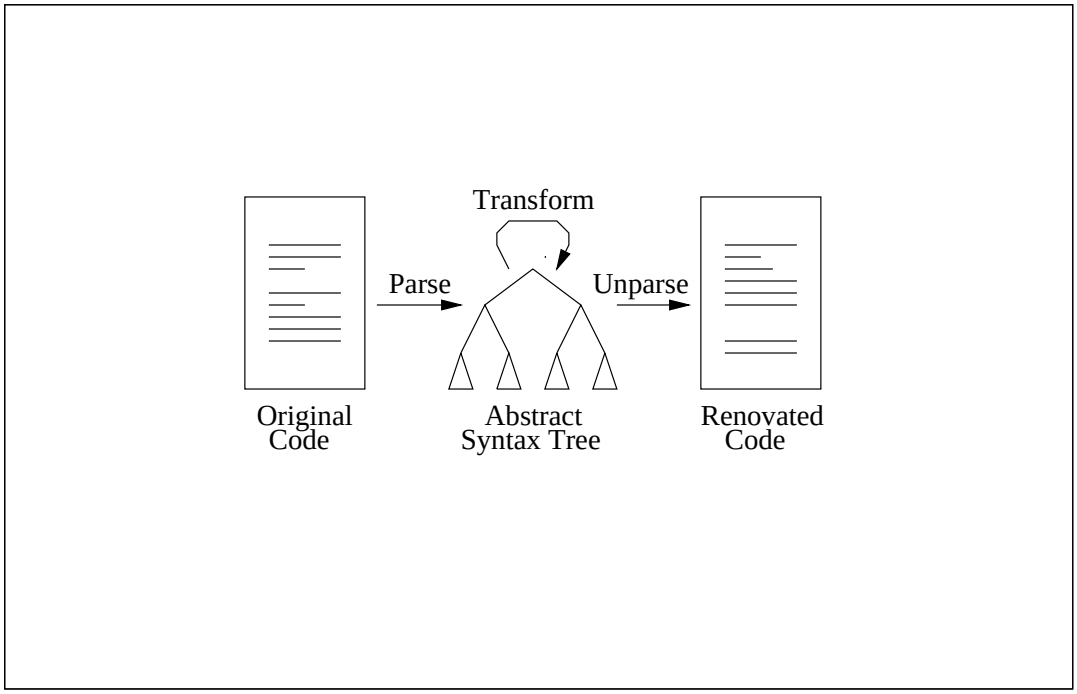


University of Amsterdam

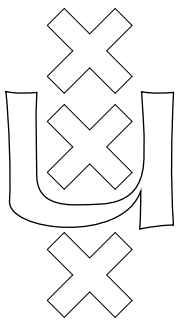
Programming Research Group



Generation of Components for Software Renovation Factories from Context-free Grammars

Mark van den Brand
Alex Sellink
Chris Verhoef

Report P9705 June 1997



University of Amsterdam

Department of Computer Science

Programming Research Group

Generation of components for software renovation factories from context-free grammars

Mark van den Brand
Alex Sellink
Chris Verhoef

Report P9705 June 1997

M.G.J. van den Brand

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7593
e-mail: markvdb@fwi.uva.nl

M.P.A. Sellink

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7583
e-mail: alex@fwi.uva.nl

C. Verhoef

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

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

Universiteit van Amsterdam, 1997

Generation of Components for Software Renovation Factories from Context-free Grammars

Mark van den Brand, Alex Sellink, Chris Verhoef *

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

markvdb@wins.uva.nl, alex@wins.uva.nl, x@wins.uva.nl

Abstract

We present an approach for the generation of components for a software renovation factory. These components are generated from a context-free grammar definition that recognizes the code that has to be renovated. We generate analysis and transformation components that can be instantiated with a specific transformation or analysis task. We apply our approach to COBOL and we discuss the construction of realistic software renovation components using our approach.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.2.m [**Software Engineering**]: Miscellaneous—Rapid prototyping
Additional Key Words and Phrases: Reengineering, System renovation, Restructuring, Language migration, COBOL

1 Introduction

Software engineers are faced with serious problems when dealing with the renovation of large amounts of legacy code. Manual approaches are not feasible, in general, both due to the amount of code and since it is usually unreliable to renovate by hand. Nowadays, it is more and more recognized that a factory-like approach to renovate legacy code is a sensible paradigm. Before we can use such a factory we need to construct its components. Since there are many and diverse renovation tasks for a myriad of languages and their dialects, there are as many components imaginable. These components should preferably be reliable, maintainable, reusable, and easy to construct. So, in our opinion, there is a need for a construction technique of components that are necessary in software renovation factories. In this paper we propose a method to generate such components from the context-free grammar that recognizes the code that has to be renovated (in [BSV97] a method is discussed to obtain grammars from legacy code). Due to our generative approach and the presence of a grammar the components are reliable, maintainable, maximally reusable, and their implementation is measured in minutes rather than hours. At the time of writing this paper we are in a preliminary phase to design a software renovation factory to restructure legacy

*Chris Verhoef was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), project *Interactive tools for program understanding*, 612-33-002.

code from a large financial company and we are using the methods we report on in this paper to construct its components.

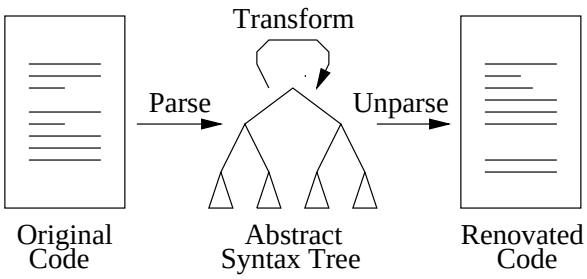


Figure 1: Schematic software renovation factory.

We think of a software renovation factory as the batch oriented transformation of massive amounts of code (see Figure 1). Roughly, such code is processed in three phases. First, the code is translated by a parser into an annotated abstract syntax tree. Then the annotated abstract syntax tree is manipulated, e.g. transformed or restructured, according to the desired renovation strategy. Finally, an unparser translates the abstract syntax tree back to text. So, parsers, transformations, and unparsers form the main parts of a software renovation factory. Parsing and unparsing components can be obtained by powerful techniques: they can be generated from the context-free grammar of the language to be parsed or unparsed. Lex and Yacc [Joh75, LS86] are well-known examples of scanner generator and parser generators, respectively. For the generation of unparsers from a context-free grammar we refer to [BV96]. The front-end and back-end of a software renovation factory can thus be generated. We will show that it is also possible to generate the other components of a software renovation factory.

The formalisms ASF and SDF [BHK89, HHKR89, DHK96] and their support environment the ASF+SDF Meta-Environment [Kli93] have been used to generate programming environments from a context-free grammar. This includes the generation of, parsers, unparsers, structured editors, etc. SDF stands for *syntax definition formalism* and can be used to define a context-free grammar ASF means *algebraic specification formalism* and can be used to define arbitrary behaviour over a given grammar. For example, the syntax and semantics of a programming language can be defined using SDF and ASF, respectively. We will use the same technology to generate components for software renovation factories from a given context-free grammar. The generic techniques we use to do this are similar to the ones that were used to construct an unparser generator, see [BV96]. For more information on connections between reengineering and generic language technology we refer to [BKV97]. To give the reader an idea why it is possible to generate components for software renovation factories we will give some intuition. For the functions generated by a parser generator the result type is always an abstract syntax tree, for the unparser functions the result type is text, for an occurrence testing component it is a Boolean, for a software metric component it is a (natural) number, for a restructuring component the input type equals the result type, and so on. As soon as it is possible to predict the result types of functions that you want to generate it seems often possible to generate the major part of such a component. It is only necessary to fill in the details that were not predictable. Consider, for instance, the COBOL74 dialect OS/VS COBOL `EXHIBIT` statement which is not supported by COBOL/370 (a COBOL85 dialect). In a COBOL migration project we need a component that takes care of the `EXHIBIT` statements. The only part of the generated component that we cannot predict is what the `EXHIBIT` equivalent in COBOL/370 should be. Those few lines have to be done by hand.

We discuss the various ways of constructing components for a software renovation factory. Given a specific renovation problem in a specific language for which a component has to be developed, several approaches are possible to construct the component. We introduce the following abbreviations LP, LS, PP, and PS standing for *language parameterized*, *language specific*, *problem parameterized*, and *problem specific*, respectively. The classical way of constructing a component for a given problem in a given language is handcrafted construction of the component, that is, the LS-PS approach. A serious problem in renovation is that every new customer uses a new dialect so the component has to be re-implemented. A more generative approach can solve that problem. Known generic examples are LP-PS components: parser generators [Joh75, LS86] and unparser generators [BV96], which take the language as a parameter. We are not aware of of LS-PP components in the litera-

ture. We discuss them in this paper: generic components for COBOL which can be instantiated for a specific problem, e.g., scope termination transformations and current date transformations (see Section 3). Examples of LP-PP components are generic component generators that can be instantiated with a specific problem in a specific language to obtain a component solving the specific problem. This approach is also new. Next, we will give the reader an idea of this approach.

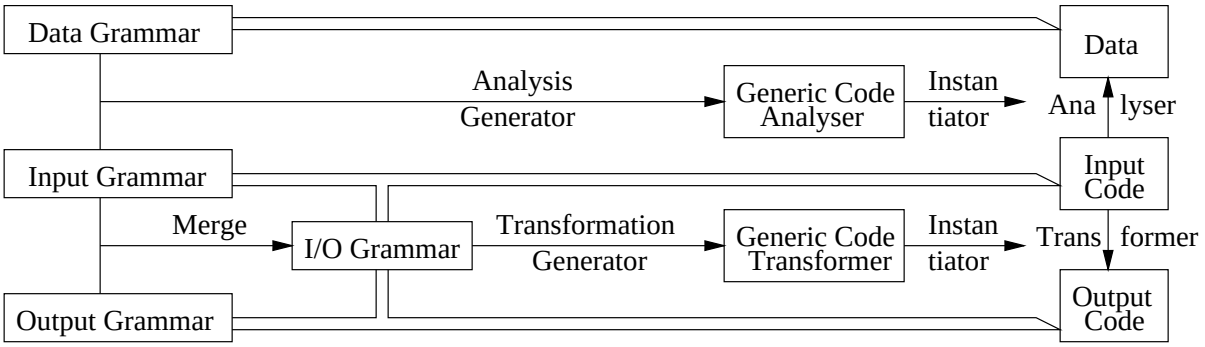


Figure 2: Generation of components for software renovation factories.

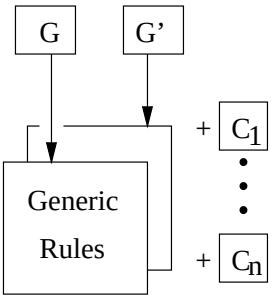


Figure 3: Maximal reuse of actual components.

The LP-PP approach to construct a component is depicted in Figure 2. In this figure we assume the presence of so-called input code in some input language and some given reengineering problem, for instance, an analysis or a transformation task. The figure describes a method to obtain components to perform the reengineering task. First we use a generator that takes the grammar parameter as input. This generator generates a generic component that can be instantiated with the specific problem thus obtaining the component that we need to solve the given problem. We have four grammars: one that recognizes the input code, one that recognizes the output code, one that reconizes both, and one that recognizes the results of an analysis. Recognition of code is expressed with the asymmetric open arrows (see [BSV97] for a general method on how to obtain grammars from legacy code). A component that transforms input code into output code should understand both grammars, in order to be able to combine transformation components sequentially. This is expressed in Figure 2 by the merge operator. Note that the implementation should be able to merge grammars in a convenient way without having reduce/reduce and shift/reduce conflicts, so Lex+Yacc approaches are not always satisfactory. We do not have these problems, since we use generalized LR parsing that can handle general context-free grammars [BSV97]. Thus, we obtain a merged grammar: the I/O grammar that understands both the input and output code. Given this merged grammar a generator generates a generic code transformer. A component that performs an analysis should have knowledge of the input code and the types in which the data is presented, for instance, a Boolean value or a (natural) number. So, the analyzer generator takes as arguments the input grammar and the result data type to generate the generic analysis functions. Now we come to the PP part: for a specific reengineering task, say a transformation or an analysis,

we can instantiate the generic components to obtain a specific component that implements the reengineering task. This instantiation consists of instantiating the non predictable parts of the component. In this way we can generate the various components that are necessary in a software renovation factory.

An, in our opinion, very important consequence of the LP PP approach is that the generated components are maximally reusable and thus easily maintainable. We sketched the situation in Figure 3. In fact, components consist of a shared part that is generated from a grammar G containing the generic rules and usually small handcrafted parts $C_1, \ldots, C_n$ containing problem specific rules. Now suppose that we want to reuse the components $C_1, \ldots, C_n$ in a different dialect G'. Then we usually only have to regenerate the generated part and we do not have to modify any of the handcrafted parts. Let us give an example to make this apparent. Suppose we have a renovation factory to migrate certain company specific COBOL74 code to a dialect of COBOL85. Now for another company we have the same problem to solve. However, in the new code we find in the `PROCEDURE DIVISION` a `DECLARATIVES` section. We extend the grammar G with this knowledge thus obtaining G'. Then we regenerate the generic part of the components. Since we are sure that the components$C_1, \ldots, C_n$ do not affect the `DECLARATIVES` keyword (for, it was not even in the grammar G) they do not need to be adapted. They use the newly generated functions and the original handcrafted parts and work now on the new grammar. So they are maximally reusable and easily maintainable.

1.1 Related Work

The techniques we use to generate components are similar to generation techniques for unparsers. These techniques are described in [BV96].

In attribute grammars [AM91] an implicit tree traversal function is present. It is comparable to our generated traversal functions. Since they are implicit it is not possible to manipulate syntax trees so it is difficult to express program transformations using that technology. Higher order attribute grammars [VSK89] is an extended form of attribute grammars in which the traversal functions are no longer implicit. In principle, we think that it should be possible to implement our generic approach using higher order grammars. We have not seen any publication that discusses the generic construction of components.

We estimate that with REFINE [Rea92] it is possible to develop components with similar functionality as we develop using our generic approach but we are not aware of a generative approach to constructing components using REFINE.

1.2 Organization of the Paper

We will explain in Section 2 our generic approach, in Section 3 we will apply it to COBOL, in Section 4 we will discuss implementation details, and in Section 5 we draw some conclusions.

2 Generic Components

It is feasible to generate analysis and transformation components, when the result types of the functions which implement these components are known in advance. In this section we describe the generation of these two types of generic components, we describe how to instantiate them, and we pay attention to their reusability.

2.1 Generating Generic Components

As we already announced in the introduction we are able to generate generic components of various types. In Figure 2 we depicted two such generic components: a generic code transformer and a generic code analyzer. Let us make clear that we think of program transformation as a batch oriented process and not as an interactive process. In this section we will give some details on to generate generic components.

Just as with parser generators and unparser generators it is possible to develop generators for reengineering purposes. Given a grammar in some syntax definition formalism, say SDF [HHKR89], we are able to generate a generic code transformer and a generic code analyzer. These generated generic components consist of a collection of functions capable of recursively traversing a syntax

tree. In our case they are either geared towards being instantiated with a specific code transformation problem or are precooked to perform a specific analysis task.

Next, we will explain how the generation process works by giving an example of an abstract grammar rule and showing the output of both the transformer and the analyzer generator. Then we will give a more elaborate example.

Suppose we have the following (abstract) context-free grammar rule in SDF-style: `L1 S1 L2 S2 L3 -> S3` Where `L1, L2, L3` stand for literals or keywords like `begin`, `end`, or `if`. `S1, S2, S3` are nonterminals, for example, `declaration`, `statement`, or `program`. In BNF-style the rule would be: `S3 ::= L1 S1 L2 S2 L3`. In the next two paragraphs we give a short characterization of both generic components and their generated output of the above grammar rule.

Generic Code Transformer We generate traversal functions for a given grammar. Their purpose can be language migration, dialect conversion, code restructuring, etc. For each grammar rule we generate one ASF equation (we recall that ASF [BHK89] stands for algebraic specification formalism). We note that functions in algebraic formalisms are defined by means of equations. See Section 4 for more details on the underlying implementation. So we generate for the context-free grammar rule presented above the following ASF equation:

```
T_S3(L1 var_S1 L2 var_S2 L3)^{Attr*} =
  L1 T_S1(var_S1)^{Attr*}
  L2 T_S2(var_S2)^{Attr*}
  L3
```

Where `T_S3` is the top traversal function and `T_S1` and `T_S2` are traversal functions for the body. Moreover, `var_S1` is a variable representing the subtree with root `S1`, `var_S2` a variable for the subtree `S2` and `Attr*` a variable which represents a list of attributes which may contain values needed for specific transformations such as the introduction of new variables. Note that in this situation we leave the keywords (`L1-3`) untouched and that the traversal function traverses the abstract syntax tree. In fact, `T_S3` is suggestive syntax: the `_S3` is a postfix operator and `T` is a variable for which we must substitute some function name. By instantiating this variable we construct the actual components, as we will see in Section 2.2.

Generic Code Analyzer Here we generate slightly different recursive functions. The purpose of these components is to check for something, to perform some software measurement task, etc. We generate for our example context-free grammar rule the following ASF equation:

```
A_S3(Op,Data,L1 var_S1 L2 var_S2 L3)^{Attr*} =
  A_S1(var_S1)^{Attr*} Op A_S2(var_S2)^{Attr*}
```

Here `A_S3` is the top analysis function and the others are for the body; `Op` is an operator variable, like `+`, or `and`; `Data` is a variable representing some default value, like `true`; the other symbols are as with the transform function. The difference with the transform function is that the analyze function does not preserve the structure of the context-free rule: the keywords (`L1-3`) are omitted. Another difference is that the variable `Op` representing an operator is an extra argument representing what has to be applied to the resulting analysis functions. Again, `A_S3` is suggestive syntax: the `_S3` is a postfix operator and `A` is a variable for which we must substitute some function name. If we instantiate `A` we construct an actual analyzer.

Concrete Example Now we give a more elaborate example that serves as a running example for the remainder of this section. Consider the following simple production rules representing a fragment of a language containing some basic constructs:

```
STAT            -> SERIES
STAT ";" SERIES -> SERIES

ID ":=" EXP                              -> STAT
"if" EXP "then" SERIES "else" SERIES "fi" -> STAT
```

The generated output of the generic code analyzer is:

```
[1] A_SERIES(Op,Data,var_STAT)^{Attr*} =
      A_STAT(Op,Data,var_STAT)^{Attr*}
[2] A_SERIES(Op,Data,var_STAT;var_SERIES)^{Attr*} =
```

```
    A_STAT(Op,Data,var_STAT)^{Attr*} Op
      A_SERIES(Op,Data,var_SERIES)^{Attr*}
[3] A_STAT(Op,Data,var_ID := var_EXP)^{Attr*} =
      A_ID(Op,Data,var_ID)^{Attr*} Op
        A_EXP(Op,Data,var_EXP)^{Attr*}
[4] A_STAT(Op,Data,if var_EXP
                   then var_SERIES1
                   else var_SERIES2 fi)^{Attr*} =
      A_EXP(Op,Data,var_EXP)^{Attr*} Op
        A_SERIES(Op,Data,var_SERIES1)^{Attr*} Op
        A_SERIES(Op,Data,var_SERIES2)^{Attr*}
```

We see that four ASF equations are generated. For each context-free grammar rule one equation is generated. Consider, for instance equation `[3]`, its left-hand side consists of the function name `A_STAT` and three arguments, the operator variable, the data variable, and the assignment statement. The nonterminals in the assignment statement are translated to function calls in the right-hand side of the equation. This right-hand side consists of a function applied to the left-hand side of the assignment, an application of the operator `Op`, and a function applied to the right-hand side of the assignment. The operator `Op` connects the parts of the right-hand side. In the next section we will show how to obtain an actual component from this generic one.

2.2 Instantiating Generic Components

The construction of components which perform reengineering tasks is rather straightforward when using our approach. It consists of two phases. Modifying the generated functions, and instantiating the variables in the generated functions. We will show this using the example above, and then we will show that the actual component is maximally reusable. Note that in Section 3 we will discuss several useful components which can be integrated in a renovation factory that have been constructed in the same way.

Our target is to construct a component for the language in the above concrete example. It counts the number of assignment statements in the code. This means that we overwrite equation `[3]`, which is responsible for the generic analysis of assignments, by substituting `cnt` for `A` and replacing the right hand side of the equation by `1`.

```
[3] cnt_STAT(Op,Data,var_ID := var_EXP)^{Attr*} = 1
```

Now we have to instantiate the variables `Op` and `Data`. In order to get the actual analyze component for a complete program we instantiate `cnt` on `SERIES`:

```
[0] count-assignments(var_SERIES) =
      cnt_SERIES(+,0,var_SERIES)^{}
```

We instantiated the variable `A` with `cnt` again, we instantiated the variable `Op` with `+`, and the variable `Data` with the default value `0`. This results in a component which can count the number of assignments in a program: `[3]` returns 1 as soon as an assignment is found and `[0]` combines found values and adds them.

Of course, this is an artificial example, but it shows exactly what has to be done in order to construct an actual component. The construction of transformation components can be done in a similar way. We see more examples of transformations in Section 3.

Reusability Next we will show that our approach is maximally reusable. That is, we can completely reuse the component as long as the grammar of the new language does not affect the part of the original grammar on which the component is working. As an example, suppose that we have a new customer that has the above language extended with a while construct:

```
"while" EXP "do" SERIES "od" -> STAT
```

We press the button and generate a new set of generic functions. Consisting of the four equations we already had and a new one:

```
[5] A_STAT(Op,Data,while var_EXP
                   do var_SERIES od)^{Attr*} =
      A_EXP(Op,Data,var_EXP)^{Attr*} Op
        A_SERIES(Op,Data,var_SERIES)^{Attr*}
```

Without changing the count-assignments component, it will now work correctly on the extended language. Only, if the new language would have different syntax for assignments, it was necessary to change the component by hand. Again, this example is artificial but it shows that when dealing with different dialects of languages, we can make maximal reuse of actual components.

3 Generating COBOL Reengineering Components

We will apply the approach that we proposed to construct reengineering components from a COBOL grammar. This COBOL grammar is based on real legacy code, see [BSV97] for more details. We will show how to construct COBOL reengineering components from the generated generic components. More precisely, we will discuss a number of real life components that are useful in restructuring COBOL code and we discuss components to migrate COBOL74 code to COBOL85 code. So, we have components to analyze, components to restructure, and components to migrate. We use analysis components to decide whether or not certain migration or restructuring components should be activated, e.g., if the COBOL74 special register CURRENT-DATE does not appear in the code, it is not necessary to perform a CURRENT-DATE transformation. What all these components have in common is that they are generated and that their specification is straightforward using the methods discussed in Section 2.2. The effort to construct them is measured in minutes rather than hours. Moreover, they are maximally reusable.

In Section 3.1 we will discuss the construction of two simple components, in Section 3.2 we will describe a restructuring which inserts missing END-IF in COBOL code. In Section 3.3 a migration component which deals with the CURRENT-DATE transformation is discussed and in Section 3.4 we compare the effort when constructing components from scratch or using our generic approach.

3.1 Two Simple Components

We will first treat two simple components: a restructuring component and a migration component. In the next subsections we will discuss two complex restructuring and migration components.

We discuss a component that restructures a MOVE CORRESPONDING statement with more than one receiving field into a MOVE statement containing the OS/VS COBOL statement connector THEN. It takes two equations by hand to construct this component. The other functions have default behaviour so the generated part takes care of that.

```
[1] mct_Stat(MOVE CORR B-exp
             TO Data-name Data-name-p)^{Attr*} =
    MOVE B-exp TO Data-name THEN
    mct_Stat(MOVE CORR B-exp TO Data-name-p)^{Attr*}
[2] mct_Stat(MOVE CORR B-exp TO Data-name)^{Attr*} =
    MOVE B-exp TO Data-name
```

mct stands for "MOVE CORR to THEN". Note the use of the synonym CORR for CORRESPONDING. In our grammar we used this one and we substituted all occurrences of the other one to CORR by preprocessing. See [BSV97] for detailed discussions on preprocessing COBOL source. B-exp stands for basic expression which can be a variable, a string, a constant, etc. Data-name stands for a variable name. From now on -p after a variable means one or more occurrences of it.

Only at the statement level (Stat) we want special behavior: a MOVE CORR with one or more Date-names should be changed into a MOVE with statement connectors THEN. This is recursively defined in two equations. The first one matches exactly this case. It removes the CORR, it produces a statement connector THEN, and recursively processes the body of the MOVE CORR until the last one. The last case is treated with equation [2]: only the CORR needs to be removed.

Next we discuss a component that migrates the THEN statement connector (COBOL74 specific) into multiple statements. In COBOL/370 (a COBOL85 dialect) THEN as a statement connector is obsolete. We have to change four equations by hand:

```
[1] rth_Sent(var_Stat1 THEN var_Stat2 var_Sent1)^{Attr*} =
    var_Stat1 rth_Sent(var_Stat2 var_Sent1)^{Attr*}
[2] rth_Sent(var_Stat1 THEN var_Stat2.)^{Attr*} =
    var_Stat1 rth_Stat(var_Stat2)^{Attr*}.
[3] rth_Stat-p(var_Stat1 THEN var_Stat2 var_Stat-p)^{Attr*}=
    var_Stat1 rth_Stat-p(var_Stat2 var_Stat-p)^{Attr*}
[4] rth_Stat(var_Stat1 THEN var_Stat2)^{Attr*} =
    var_Stat1 rth_Stat(var_Stat2)^{Attr*}
```

rth stands for remove THEN. Equations [1-4] treat all the different cases of occurrences of THEN as a statement connector: at the beginning of a sentence [1] and at the end of a sentence [2]. Equation [3] takes care of the THEN at the beginning of a non-empty list of statements whereas equation [4] takes care of the THEN at the end of a non-empty list of statements. Note that the context-free grammar rule for THEN on the Stat level in SDF is:

```
Stat "THEN" Stat -> Stat {right}
```

Where {right} indicates that the binary operator THEN is right associative. Therefore, it is not necessary to process the var_Stat1 subtree in the right-hand sides. Recursively, the statements which form a sentence or a list of statements are processed and if a THEN connector is found it is removed.

Example We will show how mct and rth process the following code fragment and why it is useful to transform and migrate in this way.

```
                          MOVE A TO B THEN    MOVE A TO B
MOVE CORR A TO B C D. -> MOVE A TO C THEN -> MOVE A TO C
                          MOVE A TO D.        MOVE A TO D.
```

First, we see that mct transforms the COBOL74 code into COBOL74 code with a statement connector. Then rth removes the statement connectors. We use an extra phase because we want to keep components as simple as possible. It is possible to equip mct with the extra functionality so it could make from one statement an arbitrary number of statements, which is in this example from one to three, however this would make the component more complicated. Furthermore, there are other transformations which go from n to m statements as well. We chose to do this in two steps: first use THEN as a statement connector to keep the number of statements constant and in the end we use rth to remove the connectors. Another example of this use can be found in Section 3.3.

3.2 A Restructuring Component

Now that we have seen a simple restructuring component we discuss here a more involved one. Transformations that are more complex than the ones we treated thus far are, for instance, implicit to explicit scope terminating transformations in COBOL. In COBOL74 dialects there is, for instance, no way to explicitly terminating an IF statement. A dot (.) terminates all open IF clauses. When we have nested IF statements an ELSE closes a higher IF clause. Note that a dot is not possible in nested IF statements since that would close *all* open IF statements. In COBOL85 dialects an explicit scope terminator END-IF has been introduced. Although the END-IF is optional in COBOL85 dialects, much of the company code that we have seen uses this feature and abandons the implicit termination options since it is error prone.

There are not only END-IF terminators but also END-READ and END-WRITE terminators in COBOL85 dialects. We will discuss the construction of a restructuring component that transforms the implicit scope terminators for conditionals into an END-IF. We mention that other implicit to explicit scope termination components are constructed analogously, but simpler since their grammar is simpler.

First, we discuss a fragment of the COBOL grammar in SDF. These grammar rules recognize COBOL74 programs without scope terminators as well as COBOL85 programs with or without scope terminators (this is an example of an I/O grammar depicted in Figure 2). The grammar of the conditionals is rather complicated. The reason for this are the scope termination rules. We recall those from the ANSI Standard [Ame85]:

The scope of the IF statement may be terminated by any of the following:

a. An END-IF phrase at the same level of nesting.

b. If nested, by an ELSE phrase associated with an IF statement at a higher level of nesting.

c. A separator period.

We have expressed the above rules in the context-free grammar. Let us first explain that conditional expressions in COBOL85 dialects come in three flavors: the good, the bad, and the ugly. The good ones use the explicit scope terminator END-IF (rule a). The bad ones come in nested conditional COBOL constructs and seem, due to the nature of the implicit scope termination rules,

to be incomplete (rule b). The expression `IF L-exp Stat1 ELSE Stat2` is an example of a bad conditional. The ugly ones are terminated with a separator period (rule c).

We explain the grammar fragment below. Since also normal statements can occur in the body of an `IF` statement, we first created a nonterminal called `Cond-body` in which those statements and possibly a bad conditional, called `Bad-cond`, may occur at the end. Then we are able to define what a `Bad-cond` is: an `IF` statement that is not explicitly terminated and that can contain such non terminated `IF` statements as well. Then we say that with the addition of an `END-IF` we can turn an incomplete `IF` statement into a complete Statement, called `Stat`. Finally, we express in the last two rules how to complete an `IF` with the addition of a separator period. Let us recall that a COBOL sentence (`Sent`) is one or more statements followed by a separator period (note that this implies that in COBOL74 dialects only `IF` sentences exist and no `IF` statements!).

```
Stat           -> Cond-body
Bad-cond       -> Cond-body
Stat Cond-body -> Cond-body

"IF" L-exp Cond-body "ELSE" Cond-body -> Bad-cond

Bad-cond "END-IF"             -> Stat
"IF" L-exp Cond-body "END-IF" -> Stat

"IF" L-exp Sent                   -> Sent
"IF" L-exp Cond-body "ELSE" Sent  -> Sent
```

Note that the above context-free grammar rules are only a selection of a complete context-free grammar definition of COBOL in SDF. An example of a code fragment that can be parsed by the above grammar fragment and that contains all three conditional flavors is the `slight-slot` program in Figure 4; we indicated the three possible `IF` statements in there, with good, bad, and ugly tags.

Now we can discuss the transformation component that turns bad and ugly conditionals into good ones; we call it `aei` for "add `END-IF`". It is the component that inserts an `END-IF` at the appropriate places.

We have to adapt three of the generated generic equations. This is not surprising, since we have three places in the grammar where an `END-IF` is missing and an implicit scope terminator takes care of the scope of the `IF` statement. They are the one that defines `Bad-cond` (that one misses an explicit scope terminator) and the two that take care of the separator period (they use a dot as terminator instead of an `END-IF`). Furthermore, we need an auxiliary function that removes a separator period from a sentence; it is called `rsp` which stands for "remove separator period".

```
[1] aei_Cond-body(Bad-cond)^{Attr*} =
      aei_Bad-cond(Bad-cond)^{Attr*} END-IF
[2] aei_Sent(IF L-exp Sent)^{Attr*} =
      IF L-exp rsp(aei_Sent(Sent)^{Attr*}) END-IF.
[3] aei_Sent(IF L-exp Cond-body ELSE Sent)^{Attr*} =
      IF L-exp aei_Cond-body(Cond-body)^{Attr*}
      ELSE rsp(aei_Sent(Sent)^{Attr*}) END-IF.
[4] rsp_Sent(Stat.) = Stat
```

In Figure 4 we have depicted a COBOL program that prints `slight` or `slot` depending on the value of `X`. Next to it we see the result of its transformation with `aei`. Let us briefly discuss how `aei` processes the input program so that the above definition of `aei` becomes clear.

We recall that all the `aei` equations are generated and that only a few of them are overruled. First, the generated function `aei_Program` is applied to the example program. Via the generated traversal functions we will reach the relevant parts of the code for `aei` where it will perform the desired transformation. Suppose we arrived at the ugly `IF` in this way. Then we recognize that this piece of code is a COBOL sentence. It matches with equation `[3]`. This results in removing the separator period by `rsp` and then adding `END-IF` followed by a separator period, see Figure 4. We continue with the remaining construction inside the ugly `IF`. This happens to be a `Cond-body` consisting of a `Stat` and a `Bad-cond`. On a simple `Stat` the function `aei` will be the identity. We arrive at the bad conditional: on the `Bad-cond` it is not the identity: we changed that one by hand. According to rule `[1]` it will add an `END-IF` and will continue with the inside of the second `IF`. In the body of that `IF` we find a simple statement and a good conditional, which is also a statement. Now the body of that conditional is examined. Since the body consists of only two simple statements we are done.

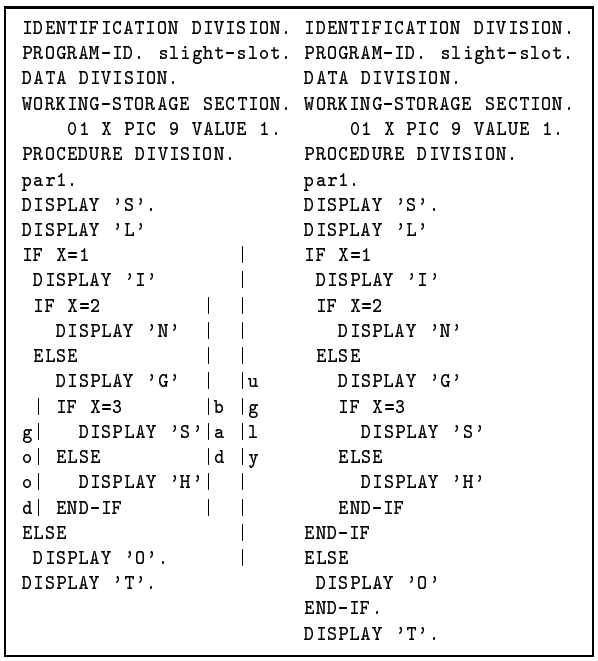


Figure 4: A program before and after `aei` transformation.

3.3 A Migration Component

In this section we will discuss a migration component with more advanced functionality than the `rth` component. This migration component transforms the OS/VS COBOL `CURRENT-DATE` special register to the special register `DATE` that is supported by both OS/VS COBOL and COBOL/370. We recall that `CURRENT-DATE` has the 8-byte alphanumeric format `MM/DD/YY` which is a common way of representing month, day, and year. `DATE` has the 6-byte alphanumeric format `YYMMDD` which is more geared towards comparison of dates. `CURRENT-DATE` is allowed as sending field in a `MOVE` statement; `DATE` is only valid as sending field in an `ACCEPT` statement. We have to deal with two issues for this component. First, we have to take care of the incompatible formats of the special registers and, second, we have to take care of the actual transformation of certain `MOVE` statements into `ACCEPT` statements. But before we can migrate, we first need to know if input source needs a `CURRENT-DATE` transformation at all. Therefore, we construct a component that analyzes a program and returns `true` if `CURRENT-DATE` occurs in it. We will first discuss this component and then the transformation component.

The component `cda` ("`CURRENT-DATE` analyzer") checks whether or not `CURRENT-DATE` occurs in a program. Next we display the overruled equations for `cda`.

```
[0] cda_Program(var_Program) =
      cda_Program(or,false,var_Program)^{}
[1] cda_Stat(o,d,MOVE CURRENT-DATE
                 TO var_Data-name-p)^{Attr*} = true
```

Equation `[0]` takes care of the correct initialization of the binary operator `o` and the default value `d` (see Section 2.2 for details). We introduce this extra equation to ensure that each application of the `cda` component uses the same operator and default value (it is not necessary to do this, only convenient). Note that the default is `false` so we assume no occurrences of `CURRENT-DATE` by default. The operator is `or` is used to combine the results of `CURRENT-DATE` tests on code parts `A` and `B` where the concatenation `A B` is an arbitrary piece of code (`A` and `B` represent, e.g., divisions, paragraphs, statements, etc.). Equation `[1]` recognizes a `MOVE CURRENT-DATE TO` expression and returns `true`.

We will use `cda` in the definition of the `cdt` component (this stands for `CURRENT-DATE` transformer). Before we display its equations, we discuss a way to perform the `CURRENT-DATE` transformation, we do this using the example program depicted in Figure 5.

The input program is the program on the left in Figure 5 with `/` omitted. This programs displays the current date as day, month, and year separately. One possible solution, proposed in [IBM92], is to change the `PIC` string of `TMP` by making it `X(6)` and to change `MOVE CURRENT-DATE` into an

```
IDENTIFICATION DIVISION.
PROGRAM-ID. cdt-test.
DATA DIVISION.
WORKING-STORAGE SECTION.
/ 01 TMP PIC X(8).
* 01 TMP PIC X(6).
01 H-DATE.
   02 H-MM   PIC XX.
   02 FILLER PIC  X.
   02 H-DD   PIC XX.
   02 FILLER PIC  X.
   02 H-YY   PIC XX.
PROCEDURE DIVISION.
/ MOVE CURRENT-DATE TO TMP
* ACCEPT TMP FROM DATE
  MOVE TMP TO H-DATE
  DISPLAY 'DAY   = ' H-DD.
  DISPLAY 'MONTH = ' H-MM.
  DISPLAY 'YEAR  = ' H-YY.
```

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. cdt-test.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 F-Date.
    02 F-YY PIC XX.
    02 F-MM PIC XX.
    02 F-DD PIC XX.
 01 TMP PIC X(8).
 01 H-DATE.
    02 H-MM   PIC XX.
    02 FILLER PIC  X.
    02 H-DD   PIC XX.
    02 FILLER PIC  X.
    02 H-YY   PIC XX.
 PROCEDURE DIVISION.
  ACCEPT F-Date FROM DATE
  STRING F-MM '/' F-DD '/' F-YY
    DELIMITED SIZE INTO TMP
  END-STRING.
  MOVE TMP TO H-DATE
  DISPLAY 'DAY   = ' H-DD.
  DISPLAY 'MONTH = ' H-MM.
  DISPLAY 'YEAR  = ' H-YY.
```

Figure 5: A program before and after `cdt` transformation.

`ACCEPT` statement. This is expressed in the same program by the statements marked with `/` instead of the original ones marked with `*` comment markers. This solution breaks down as soon as `TMP` is used in a context assuming the original format `X(8)`. `H-DATE` is such a context. The output of today (04/14/97) yields `DAY = 41`, `MONTH = 97`, and `YEAR =` which is unsatisfactory in our opinion. In fact, in the IBM scenario, `H-DATE` should be changed as well indicating that an expensive data flow analysis seems necessary. We propose a solution which does not need data flow analysis. The `cdt` component returns the above right-hand side program. It introduces a fresh variable `F-Date` to store `DATE` in its (fresh) subfields: `F-YY, F-MM, F-DD`. Subsequently, we simulate the format of `CURRENT-DATE` by using the `STRING` statement. In this way the format of `TMP` is exactly the same as before the migration so the above problem will not arise.

Next, we discuss the construction of `cdt`. It has four equations.

```
[0] cda_Program(var_Program0) = false
    ================================
    cdt_Program(var_Program0) = var_Program0

[1] cda_Program(var_Program0) = true,
    cdt_Program(var_Program0)^{id(F-Date)
      id(F-YY) id(F-MM) id(F-DD)} = var_Program1
    ==========================================
    cdt_Program(var_Program0) = rth_Program(var_Program1)^{}

[2] cdt_Data-desc-p(var_Data-desc-p0)^{id(var_Id0)
      id(var_Id1) id(var_Id2) id(var_Id3)} =
      01 var_Id0.
          02 var_Id1 PIC XX.
          02 var_Id2 PIC XX.
          02 var_Id3 PIC XX.
      var_Data-desc-p0

[3] cdt_Stat(MOVE CURRENT-DATE
             TO var_Data-name-p4)^{id(var_Id0)
      id(var_Id1) id(var_Id2) id(var_Id3)} =
      ACCEPT var_Id0 FROM DATE
      THEN
      STRING var_Id1 '/' var_Id2 '/' var_Id3
        DELIMITED SIZE INTO var_Data-name-p4
      END-STRING
```

Equation `[0]` defines `cdt_Program` under the condition that no `CURRENT-DATE` occurs in `var_Program0`: then `cdt_Program` just returns the input unchanged. Note that in ASF conditions are expressed with a double line; the condition is above it and if it is satisfied the rule below the line is applied.

Equation [1] defines cdt_Program on var_Program0 to be the rth_Program (remove THEN) of another program var_Program1 under the following conditions. First, there is a CURRENT-DATE, this is checked by cda_Program. Second, the input program is extended with a fresh record, yielding var_Program1. The function id just injects identifiers into attributes, it has no equations. We use the Attr* here for the first time, it is used as a memory so that we can both *declare* the fresh variables in the working storage section and *use* the fresh variables in the PROCEDURE DIVISION. Note that cdt_Program is an overloaded function name, it is used for the version without attributes as well as for the version with attributes attached to it (this is no problem in ASF+SDF). Equation [2] takes care of the insertion of the fresh variables in the WORKING-STORAGE SECTION. It matches the one or more present records in that section with variable var_Data-desc-p0 and it adds the fresh record F-Date to it. Finally in equation [3], on the statement level, cdt matches the MOVE CURRENT-DATE TO phrase and returns a new statement consisting of the ACCEPT part and the simulation part. Note that the THEN is removed in equation [1]. This is done to keep the number of statements a constant while transforming (see Section 3.1 for details).

Of course, cdt is not flawless: in order to migrate automatically, we have to know that the variables are fresh, so we have to check that with an analyzer. If there are nonfresh variables, we should make them fresh automatically. We will not describe this component since the scope of this paper is not to design a software renovation factory but to show how to generate its components. We are aware of the flaws of cdt and we can construct a cdt that is more sophisticated with the methods described in this paper.

Let us conclude this section with the remark that a component that takes care of the transformation of the OS/VS COBOL TIME-OF-DAY special register to the COBOL/370 TIME special register can be constructed analogously.

3.4 Handcrafted Versus Generated

In this section we will give the reader an indication of the effort it takes to conventionally construct components versus our generative approach. We took the components that we presented in Section 3 as a point of departure. In [BSV97] we argued that the effort and quality of components for reengineering would largely increase if they were made on top of a good grammar. We demonstrated that by constructing some components by hand among which aei.

We constructed the components that we generated in this paper also by hand in order to compare our approach with a conventional method. We expressed the amount of effort in the number of equations we had to specify in ASF in order to obtain the component. These figures are presented in Table 1. We explain the table. The total amount of generated equations is the same as the number of production rules in our COBOL definition, in our case we had 435 generated equations

Component	*From Scratch*	435 *Generated Equations*	
		Adapted	*Tuned*
mct	23	3	4
rth	23	4	5
aei	23	5	6
rsp	2	2	2
cda	15	2	4
cdt	28	4	5

Table 1: Handcrafted equations for some components.

Component	*From Scratch*	*Adapted*	*Tuned*
mct	23	435	24
rth	23	435	25
aei	23	435	26
rsp	2	2	2
cda	15	435	17
cdt	28	435	29

Table 2: Maximum number of used equations.

for each generic component. In the *Component* column we list the components of Section 3. The column *From Scratch* gives the number of equations that we needed to construct a component from scratch. The column *Adapted* gives the number of equations that have to be adapted or added on top of the generated equations to obtain the component. Components like `mct`, `rth`, and `aei` have a similar functionality in the sense that they only need to process the `PROCEDURE DIVISION`. If we construct a component by hand many equations are necessary to specify the location where the actual transformation should be performed. The column *Tuned* gives the total number of handcrafted equations necessary to obtain a component with a performance similar to a handcrafted one.

Next, we will discuss Table 2. The numbers presented in this table represent the maximum number of equations that can be applied when executing one of the components. In the *From Scratch* column we can see how many equations can maximally be applied to perform the transformation. In the next column we see that many more equations are involved. The functionality of the component is the same as with the handcrafted component. Note that the handcrafted component only looks at the appropriate place and the generic one looks everywhere. By adapting a few of the generated equations this is solved. This can be seen in the last column: the sum of the *used* generated equations and adapted equations equals the number equations we used to construct them by hand.

Let us discuss some typical figures. Since the `rsp` component only works on sentences, it will never traverse a whole program. So our approach does pays off in case a transformation requires the traversal of a significant part of the grammar. This becomes clear with the `cdt` component where we have to process two divisions. The handcrafted version takes more effort whereas the generic approach takes as much effort as a component that needs to process only one division like `aei`.

4 Implementation

This section discusses the implementation details that play a role in the generated components. We will briefly explain the ASF+SDF Meta-Environment and the two accompanying formalisms. For more details on ASF+SDF we refer to [BHK89, HHKR89] and for a description of the ASF+SDF Meta-Environment to [Kli93].

ASF+SDF is a modular algebraic specification formalism for the definition of syntax and semantics of (programming) languages. It is a combination of two formalisms ASF (Algebraic Specification Formalism [BHK89]), and SDF (Syntax Definition Formalism [HHKR89]). The ASF+SDF formalism is supported by an interactive programming environment, the ASF+SDF Meta-environment [Kli93]. This system is called *meta-environment* because it supports the design and development of programming environments.

ASF is based on the notion of a module consisting of a signature defining the abstract syntax of functions and a set of conditional equations defining their semantics. SDF allows the definition of concrete (i.e. lexical and context-free) syntax. Abstract syntax is automatically derived from the concrete syntax rules.

ASF+SDF has already been used for the formal definition of a variety of (programming) languages and for the specification of software engineering problems in diverse areas. See [BDK+96] for details on industrial applications.

ASF+SDF specifications can be executed by interpreting the equations as conditional rewrite rules or by compilation to C. For more information on conditional rewrite systems we refer to [Klo92] and [Kap84]. It is also possible to regard the ASF+SDF specification as a *formal* specification and to implement the described functionality in some programming language.

The generic components that we described in Section 2 are generated ASF+SDF modules. The modular structure of the underlying context-free grammar is clearly visible in the generated components. Each module in the context-free grammar corresponds with a module in the generated components. Each context-free grammar rule corresponds with an equation in a generic component. We generate so-called *default* equations. A default equation is applied when none of the other equations was successful. So in our case as soon as there is no adaptation, the system will use the generated equations, and if we add a rule by hand or adapt a generated equation and remove the `[default]` tag, it will be used instead of the default equation. The generated equations presented in the example of Section 2.1 should have been marked as default equations, e.g., `[default-1] A_SERIES(...)`. This mechanism enables us to develop components with sophisticated function-

ality by only specifying a few equations.

5 Conclusions

We have developed a powerful generative approach to construct two types of components for software renovation factories: components to analyze code and ones to transform it. In Section 2 we explained that our approach uses a context-free grammar as input and that it generates from that grammar those generic components. Such grammars should be based on the code that has to be renovated (see [BSV97] for a method to obtain such grammars). We showed that switching from one dialect to another can be done easily while maximally reusing components constructed using our approach. This is an important feature since in software renovation factories one should be able to use different dialects in a flexible way. Moreover, we have shown that our approach can be applied to large languages like COBOL: we generated components that can play a realistic role in a COBOL renovation factory. We elaborately discussed applications, both simple introductory and more advanced ones. They all served the purpose of migrating or restructuring COBOL code. The components for analyzing and transforming COBOL code were generated and then instantiated by adding and overruling a few of the generated equations. We provided a comparison of the effort it took to construct components using generic techniques and by hand. We hope to have shown that our approach allows the rapid construction of reliable components with advanced functionality. It is our estimation that a software renovation factory is an appropriate paradigm to deal with massive amounts of legacy code, and that the generation of components for such a factory is of economic importance.

References

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

[Ame85] American National Standards Insitute, Inc. *Programming Language – COBOL*, ANSI X3.23–1985 edition, 1985.

[BDK+96] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[BHK89] 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. Chapter 1.

[BKV97] M.G.J. van den Brand, P. Klint, and C. Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997.

[BSV97] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from lecagy code for reengineering purposes. Technical Report P9702, University of Amsterdam, Programming Research Group, 1997.

[BV96] 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.

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

[HHKR89] 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.

[IBM92] IBM Corporation. *COBOL/370 Migration Guide*, 1 edition, 1992.

[Joh75] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[Kap84] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2):175–193, 1984.

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

[Klo92] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume II*, pages 1–116. Oxford University Press, 1992.

[LS86] M.E. Lesk and E. Schmidt. *LEX - A lexical analyzer generator*. Bell Laboratories, unix programmer's supplementary documents, volume 1 (PS1) edition, 1986.

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

[VSK89] 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.*

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (`http://www.fwi.uva.nl/research/prog/reports/`) or by anonymous ftp to `ftp.fwi.uva.nl`, directory `pub/programming-research/reports/`.

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

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

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

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

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

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

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

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

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

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

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term rewriting.*

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*

[P9504] M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman, and E. Visser (editors). ASF+SDF *'95: a workshop on Generating Tools from Algebraic Specifications, May 11&12, 1995, CWI Amsterdam.*

[P9503] J.A. Bergstra and A. Ponse. *Frame-based process logics.*

[P9208c] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208b).*

[P9502] J.A. Bergstra and P. Klint. *The discrete time ToolBus.*

[P9501] J.A. Hillebrand and H.P. Korver. *A well-formedness checker for μCRL.*

[P9426] P. Klint and E. Visser. *Using filters for the disambiguation of context-free grammars.*

[P9425] B. Diertens and A. Ponse. *New features in PSF II: iteration and nesting.*

[P9424] M.A. Bezem and A. Ponse. *Two finite specifications of a queue.*

[P9423] J.J. van Wamel. *Process algebra with language matching.*

[P9422] R.N. Bol, L.H. Oei J.W.C. Koorn, and S.F.M. van Vlijmen. *Syntax and static semantics of the interlocking design and application language.*

[P9421] J.A. Bergstra and A. Ponse. *Frame algebra with synchronous communication.*

[P9420] M.G.J. van den Brand and E. Visser. *From Box to TeX: An algebraic approach to the construction of documentation tools.*

[P9419] J.C.M. Baeten, J.A. Bergstra, and Gh. Stefanescu. *Process algebra with feedback.*

[P9418] L.H. Oei. *Pruning the search tree of interlocking design and application language operational semantics.*

[P9417] B. Diertens. *New features in PSF I: interrupts, disrupts, and priorities.*

[P9416] S.M. Üsküdarlı. *Generating visual editors for formally specified languages.*

[P9415] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. *Control and data transfer in the distributed editor of the ASF+SDF meta-environment.*