Towards a user-controlled software renovation factory

Jacob Brunekreefa,b, Bob Diertensa,a,*

aProgramming Research Group, University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
bTriLoc Software Engineering bv, Randstad 20-25, 1314 BB Almere-Stad, The Netherlands

Abstract

Part of software maintenance consists in applying program transformations system-wide. In a number of recent papers, a factory approach has been advocated in which one program after another is fed to an assembly line that consists of a sequence of transformation tools. The general feeling seems to be that such factories have to be constructed and operated by specialists (the ‘vendors’). We think this is an undesirable situation. In this paper we present a software renovation factory which is, as much as possible, user controlled. The factory is controlled by means of a graphical user interface. Two modes of control are distinguished: an architectural mode where an operational renovation factory is constructed out of a set of available tools (parsers, unparsers, transformation modules), and an execution mode where the operational factory is applied for renovation purposes. We report about an experiment with a COBOL transformation factory which has been used for the conversion of a real-world business application system.

© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Software renovation; Re-engineering; Software factory; Practice report

1. Introduction

Maintenance of big software systems sometimes requires the same (sequence of) transformations for all programs of the system. Performing these transformations by hand is dull and error-prone. In recent papers [5,6,3], a factory approach has been advocated as a solution for this problem. In this approach the required transformations are modeled as working stations along a conveyor belt. Each working station performs a

* Corresponding author.
E-mail address: bobd@science.uva.nl (B. Diertens).

0167-6423/02/$-see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0167-6423(02)00060-6
well-defined transformation task on an input program and produces an output program. By cascading the working stations along an assembly line, and by feeding a set of programs batch-wise to the first working station, a process is defined in which a big system is transformed batch-wise, without the need of any user-interaction.

Both the construction and the operation of such a transformation factory is a non-trivial job. In the cited papers we read the opinion that this job has to be done by the vendor of the factory, it cannot be left for the average maintenance programmer without additional support. We think this is an undesirable point of view. In our experience, a company that owns legacy software is not always keen on moving parts of a system far away from the run-time environment, and, after some time, integrating the renovated code (coming from outside the company!) in the run-time environment again. Both technical arguments and security arguments play a role. Recent reports about the Y2K market for outsourced code seem to support this point of view.

We claim that it is not necessary to put everything under control of the factory owner/vendor. In this paper we present a factory control unit, with which an operational renovation factory can be constructed and operated without any specific knowledge of the underlying tools. The tools still will have to be developed by specialists, with a strong emphasis on reliability, compositionality and ease-of-use. Putting them together in a factory context can be done by maintenance programmers. We also present a language set-up unit and a tool set-up unit which make it possible to hide details about parser, unparsers, pre- and post-processors, and the transformation tools from the user.

The paper is organized as follows. In Section 2 we recall from other papers an overall picture of a renovation factory, and we identify its various components. Two user modes are identified: one related to the construction of an operational factory, the other related to the execution of transformation runs. The language set-up unit and the tool set-up unit are presented in the Sections 3 and 4. In Section 5 we present a factory control unit, a graphical user interface for both user modes. In Section 6 we report about a COBOL convertor that has been built with the factory control unit. This convertor has been used for the conversion of a 110 KLOC COBOL system. A section with concluding remarks completes the paper.

There are more tools for program analysis and/or transformation. We refer to the papers of Bellay and Gall [2] and Armstrong and Trudeau [1] for an overview and evaluation of the current tool spectrum. Most of the tools that are discussed are first of all meant for program understanding, not program transformation. Sneed [8] reports about a commercial reengineering workbench that contains both diagnostic tools and reconstruction tools.

2. A software renovation factory

From [5,6,3] we adopt the factory paradigm for renovation tasks on big software systems. Today, renovation factories are mainly used for Y2K conversion and Euro conversion. However, the factory concept applies to any system-wide transformation task.
A renovation factory is depicted as an assembly line with a conveyor belt transporting programs from one working station to another. The programs of a software system are fed to the factory one after another. Big systems are processed batch-wise, with a minimum of user interaction.

Like most of the renovation tools, we apply our transformations on an abstract syntax tree representation of a program, not on a textual (lexical) representation. We refer to [3] for a motivation of this approach. The abstract syntax tree representation of a program is produced, in an initial step, by a parser station. The transformation process consists of a sequence of program transformations, depending on user requirements. Representing each transformation by a single working station, we have a cascade of transformation stations with a variable length. At the end of this cascade we have a transformed program that is still in abstract syntax tree representation. So, at the end of the transformation cascade there is an unparsers station, which we will also call a prettyprinter station. For various reasons lexical pre-processing of the initial programs and/or lexical post-processing of the resulting programs may be required. This leads to a pre-processing station at the beginning of the assembly line and a post-processing station at the end.

The various components of a renovation factory are controlled by what we call a factory control unit. In its most simple shape, this is a script that calls the factory components in the right order, with the right input. In this paper we present a flexible, user-controlled, factory control unit that controls the construction of an operational renovation factory as well as the execution of a transformation run.

In Fig. 1 we depict the renovation factory with a conveyor belt, working stations and a factory control unit.

The development of the various factory components—a parser/unparser, the transformation tools—requires knowledge and skills with respect to grammars and program transformations. This kind of work is outside the scope of a maintenance programmer, it has to be done by specialists. In this paper we pay no attention to component generation, we refer to [3] for an extensive treatment of this topic. Once a set of factory components is available, we feel it should be possible for a maintenance programmer to construct his own assembly line and to execute his own transformation run, without inside knowledge of the techniques that are used for parsing, executing transformations, etc. In Section 5 we present a factory control unit that can be used by maintenance programmers in order to construct and operate their own renovation factory.

Fig. 1. A software renovation factory.
3. The language set-up unit

In the language set-up unit one can define language set-ups consisting of a parser and a unparser (pretty-printer), and additional pre- and post-processors. Such a set-up can be selected in the factory control unit. Working with a language set-up instead of a direct choice of a language makes it possible to work with different dialects of a particular language (e.g. IBM COBOL, MicroFocus COBOL, COBOL with embedded CICS). Moreover, an ‘inter-language’ transformation sequence with an input program in language A and an output program in language B can also be handled by creating a language directory with a parser for A and an unparser for B.

A language set-up requires the following steps:

- Selection of the parser and unparser (pretty-printer) to be used.
- Optionally: selection of the pre-processor and selection of the pre-processor set-up program for setting the pre-processor options.
- Optionally: selection of the post-processor and selection of the post-processor set-up program for setting the post-processor options.

Fig. 2 shows the dialogue window of the language set-up. In case of a new set-up, all dialogue boxes are empty. When an existing set-up is selected (by means of the drop-down list in the Name-box), the information of this set-up is displayed in the dialogue boxes. By editing the boxes a set-up can be defined or modified.

A language set-up can be saved under a unique name. This name is used in the definition of a factory set-up (see Section 5.1). When changes are required, it can be reloaded and edited.

Fig. 2. Window language set-up unit.
4. The tool set-up unit

In the tool set-up unit one can define set-ups for a set of tools, so that they can be selected in the factory control unit in a convenient way.

At this moment defining a tool set-up requires one step: the selection of the tool. Future developments of the factory may ask for more information on a tool, such as its purpose, type of input and output, etc. (see also Section 7).

Fig. 3 shows the dialogue window of the tool set-up unit. In case of a new set-up, the dialogue boxes are empty. When an existing set-up is selected, the information of this set-up is displayed in the dialogue boxes. By editing the boxes a set-up can be defined or modified.

A tool set-up can be saved under a unique name. This name is used in the definition of a factory set-up (see Section 5.1). When changes are required, it can be reloaded and edited.

5. The factory control unit

In the factory control unit two different user modes are distinguished. In the set-up mode, an operational renovation factory set-up can be created or modified. In the operational mode, a transformation run can be defined and executed, using a previously defined factory set-up.

When the factory control unit is activated, a window appears with a pull-down Control menu, see Fig. 4. From this menu one of the user modes can be selected.
The definition of an operational factory and the definition of a transformation run can be stored for future use. Previously stored definitions are automatically read at start-up time by the factory control unit.

5.1. Constructing an operational factory

The construction of an operational renovation factory requires two steps:

- The selection of a programming language, i.e. the selection of the parser and prettyprinter to be used. In the factory control unit this is done by selecting a language set-up. This set-up is supposed to contain both the parser and the prettyprinter, as well as the scripts for pre-processing and post-processing if required (see Section 3).
- The definition of a transformation assembly line. The assembly line determines which transformations are performed in which order. In the factory control unit this is done by the selection of a tool-set, followed by the selection of tools for the assembly line from the transformation tools present in the selected tool-set. The order of selection determines which transformation is carried out first. The parser and prettyprinter, and the optional pre- and post-processor, are automatically added at the front and at the end of each assembly line. It is not necessary for a user to select these tools separately.

Fig. 5 shows the dialogue window of the factory control unit in the set-up mode. In case of a new set-up all dialogue boxes are empty. When a previously defined set-up is selected, the information concerning this set-up is displayed in the dialogue boxes. By editing these boxes a set-up can be defined or modified.

A factory set-up can be saved under a unique name. This name is used in the definition of a factory run, see the next section. When changes in the set-up are required, it can be reloaded and edited in the factory set-up mode.

Fig. 5. Factory set-up definition window.
5.2. Defining and executing a factory run

With a factory run we mean the execution of the tool sequence in an assembly line as defined in a factory set-up, with as input a set of program files and as output a set of transformed files. Before a factory run can be executed, the following steps are required.

- Selection of a (pre-defined) assembly line by selecting its name from a list of available assembly lines.
- Selection of the input files by specifying a source directory and one or more files in this directory. With regular expressions like AA* or *.cbl, a selection can be made from the files in the given directory. The current version of the factory selects its input files from one directory, it is not possible to execute a factory run on files from different directories.
- Selection of a destination directory in which the output files are stored. An output file can have the same name as the corresponding input file. It is also possible to give its name an extension.
- Depending on the selected language directory, additional pre-processing and/or post-processing steps can be defined.

Fig. 6 shows the dialogue window of the factory control unit in the operation mode. As in the set-up mode, in case of a new run all dialogue boxes are empty. When a previously defined run is selected, the information concerning this run is displayed in the dialogue boxes. By editing these boxes a run can be defined or modified. A factory run can be stored under a user-defined name. When the Set-up button for pre-processing or post-processing is clicked, a separate window pops up in which the various options can be (de)selected. A transformation run is started with the run command from the Operate menu.

![Factory Control: Operate](image.png)

Fig. 6. Factory operation definition window.
As mentioned before, the definition of factory set-ups and factory runs are stored. Different set-ups and different runs are saved under different names.

6. From COBOL-85 to COBOL-74

The renovation factory has been used to convert a 110 KLOC COBOL-85 system into a COBOL-74 system. In a big financial company in the Netherlands, new applications are developed with a tool that generates COBOL-85 code. However, in May 1998 the programming language in the production environment was still COBOL-74 (IBM OS/VS COBOL). So, for the time being, there was a need to convert new developed programs to an older dialect. By compiling COBOL-85 programs with the COBOL-74 OS/VS compiler we obtained a list with error messages, from which we extracted the following list of required transformations. ¹

1. Replacement of the INITIALIZE statement. This statement is unknown in COBOL-74 and will have to be replaced by one or more MOVE statements.
2. Removal of scope-terminators: END-IF, END-ADD, END-SUBTRACT, etc. In COBOL-74 no scope terminators exist, so the program needs to be restructured.
3. Replacement of the CONTINUE statement. This statement is also unknown in COBOL-74.
4. Replacement of the data field USAGE options BINARY and PACKED-DECIMAL by the options COMP and COMP-3.
5. Replacement of the relational operators >= and <= by the semantical identical COBOL-74 constructions NOT < and NOT >.
6. Replacement of double quotes by single quotes in non-numerical literals.

The first two transformations are global transformations, they have their impact on several parts of a program. The other four transformations are local transformations: only the program element that contains the construction-to-be-modified is involved in the transformation.

For our factory we have used a set of tools (working stations) that has been constructed with the ASF + SDF Meta-environment of Klint [7]. We have used a slightly adapted version of the COBOL-grammar described in [4]. From this grammar we have automatically extracted a generic transformation environment. In this environment, for each program construct a default transformation equation (rewrite rule) transforms the construct to itself. These default equations can be overruled by user-defined equations, specifying what has to be transformed in the construct at hand. In this section we give these non-default equations for the required transformations. For more details about this approach to program transformation we refer to [3].

An ASF equation (rewrite rule) has the following syntax:

\[ \text{Redex-term} = \text{Reduct-term} \]

¹As we only deal with generated code, not all divergencies between the two COBOL versions had to be covered.
A conditional rewrite rule is only applied when a set of conditions evaluates to true. The following syntax applies:

\[
\text{[tag]} \quad \text{Condition-1, \ldots, Condition-n} \\
\text{==================================================================}
\]

\[
\text{Redex-term} = \text{Reduct-term}
\]

A condition is written as \( \text{Term-1} = \text{Term-2} \). If one of the terms is an uninstantiated variable, the condition evaluates to true, as a side-effect the value of the other term is assigned to the variable.

6.1. Replacement of the INITIALIZE statement

The COBOL-85 INITIALIZE statement initializes all data fields in the argument list of the statement to a default value (zero for numerical fields, spaces for character fields). As this statement is unknown in COBOL-74, we have to replace it by one or more MOVE statements, taking care of the required initialization. We have considered two solutions:

- **Data duplication**: In the Procedure Division of a COBOL program module each INITIALIZE statement is replaced by a MOVE statement. The statement INITIALIZE AAA is replaced by the statement MOVE INIT-AAA TO AAA. The record INIT-AAA is a copy of the record AAA, with all its basic fields initialized to default values (ZERO, SPACE). The record INIT-AAA is added to the Working-Storage section of the program.
- **Statement duplication**: In the Procedure Division of a program module each INITIALIZE statement is replaced by a sequence of MOVE statements. For each basic field of the record-to-be-initialized a MOVE statement with the initial value (ZERO, SPACE) is included. In this solution the Working-Storage section remains untouched.

We have chosen the first solution, as we feel that a significant increase of the number of statements in the Procedure Division is less desirable than enlarging its Working-Storage section.

Before we present the equations that implement the transformation, a few general remarks are in order.

- In order to keep the equations readable, some technical details (e.g. the handling of COBOL-comment lines) have been left out.
- The name of a transformation function consists of two parts, separated by an underscore character. The first part denotes the required transformation (e.g. Replace-INITIALIZE), the second part denotes the syntactical level the function is applied to (e.g. Program, Data-div, Proc-div, for the Program level, Data Division level and Procedure Division level).
- The name of a syntactical construction, followed by a digit, denotes a variable that stands for the syntactical construction. So, in the equation below, the variable Ident-div1 denotes the Identification Division of a COBOL program. A star/plus character is used to denote a list of zero/one or more elements of a syntactical construction. The variable Data-name*1 denotes a list of zero or more data names.
• A transformation function may have ‘attributes’. These attributes are written between curly braces.
• In conditions we apply functions that collect information about its argument (a program construct). These functions have a syntax that is similar to the syntax of transformation functions. They only have two more arguments. We pay no attention to these arguments.

We first present the equation that specifies the transformation at program level. Lines starting with the character sequence ‘%%’ are comment lines.

```plaintext
%% Top equation for complete program. Algorithm:
%% 1. Collect data-field names from INITIALIZE statements
%% 2. Replace the sub-record names by the corresponding 01 record names
%% 3. Remove doubles from this list
%% 4. Add to the ws-section INIT-records for the list made in 3
%% 5. Convert in the procedure div. INITIALIZE statements to MOVE
[t1] Collect-Datanames_Proc-div(**, ,Proc-div1)^{} = Data-name*1,
  Collect-Topnames_Data-div(**, ,Data-div1)^ {Data-name*1} =
  Data-name*2,
  removeDoubles(Data-name*2) = Data-name*3
=========================================
Replace-INITIALIZE_Program(
  Ident-div1
  Env-div1
  Data-div1
  Proc-div1)^{} =

  Ident-div1
  Env-div1
  Add-INIT-Dd_Data-div(Data-div1)^
    {Data-name*3}
Replace-INITIALIZE_Proc-div(Proc-div1)^{}
```

The steps 1–3 of the algorithm are implemented in the three conditions of the equation. We do not show the equations that implement the functions used in the conditions. The rewrite rule does not modify the Identification Division and the Environment Division of a program (denoted by the variables Ident-div1 and Env-div1). Step 4 of the algorithm is implemented in the following equations, operating on the Data Division (Data Description part of the Working Storage Section) of a program.

```plaintext
%% Add INIT records to data-descriptions in Working-Storage Section.
%% A record name attribute (Id1) is supposed to be a 01-field.
[t2] Add-INIT-Dd_Data-desc-s(
  Data-desc*1
  Elem-num1 Id1 Dd-item*1.
)```
\textbf{Dd-body}\
\texttt{Data-desc*1^{\{Id1 Data-name*1\} = Add-INIT-Dd_Data-desc-s(\
\texttt{Data-desc*1\
\texttt{Elem-num1 Id1 Dd-item*1.\
\texttt{Dd-body*1\
\texttt{Create-INIT-record_Data-desc(\
\texttt{Elem-num1 Id1 Dd-item*1.\
\texttt{Dd-body*1)^{}}\
\texttt{Data-desc*2^{\{Data-name*1}}\}

\textbf{%% Create INIT record with INIT datanames and VALUE clauses.}\
\textbf{%% 1. For a 01 level record field.}\n[t3] \textit{init-name(Id1) = Id2,}\
\texttt{Add-VALUE-clause_Dd-item-s(Dd-item*1)^{}} = Dd-item*2\
\texttt{=================================================}\
\texttt{Create-INIT-record_Dd-header(Elem-num1 Id1 Dd-item*1.)^{} = Elem-num1 Id2 Dd-item*2.}\n
\textbf{%% 2. For a sub-record field.}\n[t4] \textit{init-name(Id1) = Id2,}\
\texttt{Add-VALUE-clause_Dd-item-s(Dd-item*1)^{}} = Dd-item*2\
\texttt{=================================================}\
\texttt{Create-INIT-record_Dd-body(Sub-elem-num1 Id1 Dd-item*1.)^{} = Sub-elem-num1 Id2 Dd-item*2.}\n
The function \textit{init-name}, used in the conditions, adds the prefix \textbf{INIT-} to an identifier. The function \textit{Add-VALUE-clause} adds a value clause (e.g. \texttt{VALUE ZERO}) to the definition of a record field. If the original definition contains a value clause, this clause is removed first.

Step 5 of the algorithm replaces an \texttt{INITIALIZE} statement by a \texttt{MOVE} statement.

\textbf{%% Replace INITIALIZE by MOVE.}\
\textbf{%% (INITIALIZE on multiple data fields is not replaced.)}\n[t5] \textit{init-name(Data-name1) = Data-name2}\
\texttt{==================================}\
\texttt{Replace-INITIALIZE_Statx(INITIALIZE Data-name1)^{} = MOVE Data-name2 TO Data-name1}\n
6.2. Removal of scope-terminators

Elimination of scope-terminators is achieved by first wrapping all arithmetic statements that contain a scope-terminator in a conditional statement without an arithmetic scope-terminator, but with the \texttt{END-IF} scope-terminator. So, the statement

\texttt{ADD \ldots END-ADD}
is replaced by

```plaintext
IF 'TRUE' ADD ... END-IF
```

In a second step all END-IFs are removed. This is done by ‘linearizing’ the program control-flow: nested IF-statements are replaced by a sequence of IF-statements (with appropriate conditions) that do not contain other IF-statements. In these IF-statements the END-IF scope-terminator is replaced by a full stop. Due to the described reconstruction, this end-of-sentence marker is harmless. Afterwards, the disjunct ‘TRUE’ is removed from the conditions, as COBOL does not support Boolean primitives. Furthermore, TRUE is a keyword reserved for other purposes.

We stress that linearizing a nested IF-construct is a non-trivial operation. The statement

```plaintext
IF Cond1
  Stat*1
  IF Cond2
    Stat*2
    END-IF
  Stat*3
END-IF.
```

cannot simply be replaced by the statement sequence

```plaintext
IF Cond1
  Stat*1.
IF Cond1 AND Cond2
  Stat*2.
IF Cond1
  Stat*3.
```

because in the execution of a statement sequence a variable that is part of a condition can be modified. We solve this problem by introducing a nesting control variable: an integer variable that counts the level of nesting. So we get

* THE INITIAL VALUE OF NESTING-CONTROL-VAR IS ZERO.

```plaintext
IF Cond1
  ADD 1 TO NESTING-CONTROL-VAR
  Stat*1.
IF NESTING-CONTROL-VAR = 1 AND Cond2
  Stat*2.
IF NESTING-CONTROL-VAR = 1
  Stat*3
  SUBTRACT 1 FROM NESTING-CONTROL-VAR.
```

Statements that cause a break in a sequential flow of control, like the outline PERFORM statement and the GOTO statement, require a fine-tuning of this solution. We do not go into the details of this fine-tuning.
As in the previous section, the first equation describes the transformation at program-level.

[t1] ElimScopeTerminators_Program(
    Ident-div1
    Env-div1
    Data-div1
    Proc-div1)^{} =

    Ident-div1
    Env-div1
    AddNestingRecord_Data-div(Data-div1)^{}
    ElimScopeTerminators_Proc-div(Proc-div1)^{}

The transformation of the Working-Storage Section in the Data-Division is straightforward: a nesting-control variable is added to the other data declarations.

[t2] AddNestingRecord_WS-sec(
    WORKING-STORAGE SECTION.
    Data-desc*1)^{} =

    WORKING-STORAGE SECTION.
    Data-desc*1
    01 NESTING-CONTROL-VAR PIC 99 VALUE ZERO.

The following equation shows how, at the Procedure-Division level, first arithmetical statements with a scope-terminator are wrapped in a conditional statement, and, second, END-IF scope-terminators are removed.

    Proc-div2,
    ElimEndIf_Proc-div(Proc-div2)^{0} = Proc-div3
    =============================================
    ElimScopeTerminators_Proc-div(Proc-div1)^{} =
    Proc-div3

The COBOL grammar we use contains 12 different production rules for arithmetical statements. So we have 12 wrapping equations. We show one.

[t11] WrapArithmeticalStats_Stat(
    ADD LorD-p1 TO Data-name-p1 Error1
    END-ADD)^{} =
    IF TRUE ADD LorD-p1 TO Data-name-p1 Error1
    END-IF

The equations concerning the elimination of the END-IFs contain a lot of low-level technical issues. Displaying these equations would require an extensive explanatory text. Therefore we have chosen to skip these equations.
6.3. Four simple transformations

The remaining four transformations are far more simple, as they affect only the statement that they are part of.

In the generated source code, a \texttt{CONTINUE} statement appears in the \texttt{THEN} branch of an \texttt{IF} statement, either as the one-and-only statement, or followed by a \texttt{GOTO} statement. In the first case we replace the \texttt{CONTINUE} statement by the \texttt{NEXT SENTENCE} statement. In the other case we simply delete it. But first we bring down the required transformation from Program level to Procedure Division level. This is shown in the following equations.

\[ t1 \] \texttt{ElimContinue\_Program(}
[Ident\-div1
Env\-div1
Data\-div1
Proc\-div1)^{} =

Ident\-div1
Env\-div1
Data\-div1
ElimContinue\_Proc\-div(Proc\-div1)^{}

\[ t2 \] \texttt{ElimContinue\_Stat(}
\texttt{IF L-exp1}
\texttt{CONTINUE}
\texttt{END-IF)}^{} =
\texttt{IF L-exp1}
\texttt{NEXT SENTENCE}
\texttt{END-IF}

\[ t3 \] \texttt{ElimContinue\_Stat(}
\texttt{IF L-exp1}
\texttt{CONTINUE}
\texttt{GOTO Lab1}
\texttt{END-IF)}^{} =
\texttt{IF L-exp1}
\texttt{GOTO Lab1}
\texttt{END-IF}

From the equations it is clear that this transformation has to be performed before the removal of the \texttt{END-IF} scope terminator.

The other transformations are specified in similar equations that operate on the required program construct (a Data Division item, a relational operator, a COBOL lexical string). We show the equations without any further comment.

\[ b2 \] \texttt{Convert-Binary\_Dd-item-s(}
\texttt{Dd-item-s1 BINARY Dd-item-s2)}^{} =
\texttt{Dd-item-s1 COMP Dd-item-s2}
Table 1
Factory timing results

<table>
<thead>
<tr>
<th>Step</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Pre-processing</td>
<td>2 min</td>
</tr>
<tr>
<td>2. Parsing</td>
<td>31 min</td>
</tr>
<tr>
<td>3. Transformations</td>
<td>42 min</td>
</tr>
<tr>
<td>4. Prettyprinting</td>
<td>32 min</td>
</tr>
<tr>
<td>5. Post-processing</td>
<td>1 min</td>
</tr>
</tbody>
</table>

\[
\text{Convert-Binary\_Dd-item-s(} \backslash Dd\text{-item-s1 PACKED-DECIMAL } Dd\text{-item-s2)}^{} = Dd\text{-item-s1 COMP-3 } Dd\text{-item-s2}
\]

\[
\text{Convert-Rel\_Rel}(\leq)^{} = \text{NOT } >
\]

\[
\text{Convert-Rel\_Rel}(\geq)^{} = \text{NOT } <
\]

%% The help function dq2sq actually does the quote conversion.

\[
\text{dq2sq}(\text{Lex-Str1}) = \text{Lex-Str2}
\]

\[
\text{Convert-Quotes\_Lex-Str(Lex-Str1)}^{} = \text{Lex-Str2}
\]

6.4. A COBOL renovation factory

The six required transformations have been specified and tested in the interactive mode of the ASF+SDF Meta-environment. Furthermore, an unpars (prettyprinter) has been generated. In a next step, these tools have been compiled to a set of C-functions. These functions have been compiled to stand-alone executables. These executables are the working-stations of our renovation factory.

We have tested the renovation factory with the COBOL-85 to COBOL-74 transformations by transforming 98 source files with in total 110 KLOC. All six transformations have been applied in one big cascade. As a platform we used a Sun Ultra-5 with 128Mb internal memory. As far as possible, the duration of each step has been measured. The results are summarized in Table 1.

Some remarks:

1. The pre-processing step and the post-processing step are executed by means of a Perl-script. This is obviously very fast compared to the other steps.
2. Prettyprinting a program requires a traversal of the complete abstract syntax tree. For each node information about how it will be displayed is added. So, the complete program is restructured. This makes prettyprinting equivalent with a very complex program transformation, what is reflected in the time needed for this operation.
7. Evaluating remarks

In this paper we have presented a software renovation factory which is as much as possible user-operated. By making a clear distinction between the specialist work of component construction on one side and the non-specialist work of factory construction and factory operation on the other side, we tried to make it clear which part of the job can be handled by maintenance programmers. Our factory control unit is rather new, so we do not have much experience that supports (or refutes) our point of view. More experience, with real-world software systems and real-world companies, will be needed.

In the current factory set-up, an assembly line contains a linear sequence of transformation tools. Every program is subjected to all transformations, even if the transformation is not required, e.g., when a program does not contain the program construction that needs to be transformed. In a more efficiently organized factory, such a program would be skipped by the working station that performs the transformation. By introducing analysis tools besides transformation tools, this kind of ‘intelligence’ can be included in a transformation factory. From the work of Sellink and Verhoef we know that these tools already exist within the context of the ASF + SDF Meta-environment. The factory set-up interface of Section 5.1 will have to be extended with primitives that enable conditional branches in assembly lines: IF Construct-Present IS TRUE THEN Convert-Construct (ELSE SKIP). We expect that adding this kind of intelligence will improve the throughput figures of a factory. This subject is left for future research.

As described in the previous section, the various factory components are compiled C functions. This implies that components can be transferred from one platform to another. The current version of the factory control unit is written in Tcl/Tk and Perl, languages that also run on various platforms. This makes our renovation factory more or less platform-independent, an important property for a user-controlled tool.

Acknowledgements

We thank Alex Sellink and Chris Verhoef (University of Amsterdam) for their support on the many topics that have to be mastered when constructing a transformation factory: grammar definition, defining a generic transformation environment, transformation specification, etc. Mark van den Brand and Jeroen Scheerder (CWI) for their work on the ASF + SDF Meta-environment, and Eggie van Buiten (also University of Amsterdam) for his work on the prettyprinter used in the COBOL factory.

References


