On the Conversion of Program Specifications into Pseudo Code Using Jackson Structured Programming[*]

Kenneth Sörensen and Jan Verelst

Department of Operations Research, Logistics and Information Systems, University of Antwerp – RUCA, Belgium

In this paper, we present a technique to automatically translate program specifications into pseudo code. This technique is developed in the context of the well-known programming method Jackson Structured Programming (JSP).

The objective of our research is to investigate to what extent a programming method can be automated. Current CASE tools are only able to automate programming methods to a very limited extent, whereas our technique automates the entire programming cycle by creating pseudo code from program specifications. We show that the JSP programming method can be transformed into a set of formal rules when the scope of the technique is limited to a well-defined area of problems. The rules are implemented in a CASE tool, called `JSPTool`, which is currently operative, although still in a prototyping phase. We believe that the strength of the CASE tool lies in the fact that it is able to automate the programming process completely, although its scope possibly is still rather limited.

In this paper, the technique is explained by solving an example programming problem. The source language that has been developed to enter program specifications is briefly explained. Also, the differences between other JSP CASE tools and `JSPTool` are dealt with. Some additional features of the method are discussed and suggestions for future research are given.

Keywords: Program specifications, Jackson Structured Programming, JSP, pseudo code, automatic programming, CASE tool

1. Introduction

Many programming methods exist that instruct the programmer on the different steps to perform when writing a program. Most of these methods, however, leave a lot of room for subjective interpretation and human intervention. This suggests that human knowledge and experience play a large role in these methods, which in turn results in different correct solutions to the same problem. The influence of knowledge and experience on the programming process has been widely researched [Adelson & Soloway, 1985; Chatel & Détienne, 1996; Guindon, 1990; McKeithen et al., 1981; Rist, 1990]. In this paper, we investigate to what extent the different steps of the programming method JSP can be expressed as a set of rules. It is traditionally assumed that only the later, more deterministic steps of JSP can be automated. The most surprising conclusion of this research is that JSP can be completely automated (i.e. program specifications can be automatically transformed into a pseudo-code program) when the scope is limited to a well-defined area of programming problems. We develop a technique that uses a set of rules to derive a pseudo-code program from a program specification text. The program specification text uses a notation that was especially developed to allow for *complete* automation of the programming process. In other words, when correctly dressed up, the programming specification text contains enough information for a CASE tool to be able to generate a computer program from it. This technique is not a new programming method, rather it is an automation of an existing one.

In this paper, we choose a depth-first approach, limiting the number of programming problems that can be solved by the CASE tool in favour of the depth of the automation. The scope of a CASE tool based on our technique is limited

[*]The authors wish to thank M.A. Jackson for his useful suggestions.

to administrative programming problems that transform sequential input files into sequential output files, using some simple mathematical and aggregation functions. When translated into a program specification text, these problems are solved by the CASE tool without any further intervention from the programmer.

2. Jackson Structured Programming (JSP)

2.1. Introduction

JSP was developed by Michael Jackson and published in 1975 [Jackson, 1975]. JSP builds on the foundations of structured programming, but goes further in that it solves many problems and uncertainties of it [Jansen, 1986: 1]. Structured programming is a generic name for any programming method that divides a program into sequences, iterations and selections. Although there was already a broad awareness that the problem statement should influence the structure of the program [Welsh & McKeag, 1980: 24-25] programming remained a rather subjective activity: programming methods were 'vague' in that they could lead to several different and correct sets of sequences, iterations and selections. Jackson was the first to show exactly when each of these three components had to be used to obtain the most flexible and the most maintainable program structure [Jansen, 1983: 8]. His ideas are simple. As a program does nothing more than transform input into output, input and output should force their structure onto program [Jackson, 1975: 10]. JSP is a 'data first' method (as opposed to 'function first' methods) and is mainly used for administrative programming problems, which are characterised by complex input and output flows, but relatively simple transformation logic. Data is regarded to be the more stable part of an information system and, therefore, an information system is modelled around the structure of the data. In later years, Jackson, together with John R. Cameron, developed JSD (Jackson System Development), an extension of JSP which covers most of the software lifecycle [Cameron, 1986; Cameron, 1983; Jackson, 1983].

2.2. Why JSP?

The aim of our research is to investigate to which extent a programming method can be automated and transformed into a set of formal rules, which can serve as the basis for a CASE tool.

We chose to base our research on JSP because of several reasons. Firstly, JSP tells the user what to do in almost every possible situation, except in some special cases like data structure clashes and program inversion problems. For the latter cases, Jackson provides specific guidelines. Other methods leave a much greater part of the decisions to be made to the user, making them less suitable for automation. On the other hand, JSP is not a completely formal method like VDM or Z, allowing us to reach some interesting results. Secondly, JSP is a rather simple method, which is also a great advantage when implementing it in a CASE tool. Thirdly, JSP (and JSD) have been thoroughly examined academically and many methods were developed to test programs written using JSP [Hughes, 1979; Roper & Smith, 1987]. Fourthly, recent studies show that JSP and JSD are still widely used and rate very good on comparative tests [Hoorelbeke, 1993: 731; Song & Osterweil, 1994: 377].

2.3. CASE Tools for JSP

There are several other CASE tools based on the JSP programming method. `JSP-COBOL` is a commercial program, developed by Michael Jackson Systems Ltd. It is able to generate a program from a program structure and provides a variety of testing aids and some interesting programming aids [Triance, 1979: 198]. `MAJIC`, developed at the University of Manchester, has much the same functionality as `JSP-COBOL` [Sutcliffe & Davies, 1987: 122-123]. In the development process of this tool, some research has been done on which changes can be made to program structures without having to go through the method all over again [Davies, 1990: 175-192]. Of a more recent date is `JSP-editor`, which allows the user to draw program structures. It also generates Pascal and C code from these structures. The program is written in Java and can be evaluated and used online [`http://www.ida.his.se/ida/~henrike/JSP/`].

2.4. The Jackson Structured Programming Method Adapted for Automatic Conversion of Program Specifications Into Pseudo Code

Our technique slightly modifies the order in which some of the steps of JSP are performed to allow for a more efficient implementation. The technique consists of the following steps:

From the program specification text:

1. Make a separate data structure for each data flow.
2. For each data flow, make a list of input or output instructions and allocate them to the corresponding data structure.
3. Join the input structures to form the combined input structure.
4. For each iteration component, allocate an iteration condition to the combined input structure.
5. Join the combined input structure with the output structure(s), forming the program structure.
6. Make a list of all logical instructions and allocate them to the program structure.
7. Add selection conditions to the selection components of the program structure.
8. Transform the program from the program structure representation to a code or a pseudo code representation.

The proposed changes do not affect the essential mechanisms of the JSP method. The most important change is that input and output instructions are allocated to their respective data structures independently. This means that for every flow (either input or output), a separate list of input or output instructions is made and that these instructions are then allocated to the proper data structures. Some instructions however, cannot be allocated before the structures are merged, because they are neither input nor output instructions. These logical instructions are allocated to the program structure (which is, as stated earlier, the union of all data structures). Also, iteration conditions are attached to the combined input structure (the union of all input structures) instead of the program structure. New iterations are never created in an output structure, so this does not change the outcome of the JSP process.

This approach to JSP is equivalent to the original in that the program structure with allocated instructions (and accordingly the program itself) found by the altered JSP method is identical to the one found by the original JSP method. This is due to the fact that only the order of some of the steps of JSP is slightly altered. The advantage of the modified method is that a CASE tool which is based on the altered version of JSP, can disregard other data structures when allocating the input and output instructions of a certain file. We found that this greatly simplifies the rules for our CASE tool.

3. Automatically Converting Program Specifications Into Pseudo Code Using JSP

3.1. Introduction

In this section, we discuss how a pseudo code program can be derived automatically from program specifications using the steps mentioned in the previous paragraphs. This is done by solving a simple example problem. The rules that allow for an automatic conversion of program specifications into pseudo code are often trivial and would take up too much space. We will therefore skip the description of the greater part of these rules.

Program specifications serve as input to the CASE tool by means of a program specification text. The source language for entering program specifications is explained briefly. Because of the limited amount of space available, some features are not discussed. These features include program inversion and data structure clashes. For these problems, the source language offers specific tools (pseudo-variables, sub-files, etc.) that are beyond the scope of this article.

3.2. Scope of the Technique

While other tools exist that automate JSP, the scope of the technique described here differs from all of them in that its purpose is to automate JSP completely. This is a basically different option. Most tools are able to assist the user in any programming problem, however, they do not completely automate the task of programming, i.e. certain programming tasks are not supported (e.g. the conversion of program specifications into data structures). The programs that our tool can create are those that convert one or more sequential input files into one or more sequential output files, performing some simple calculations. Although this is a rather limited scope, the programming problems that are supported are solved completely automatically, i.e. without any intervention of the user after providing the program specification.

3.3. An Example: Article Mutations in a Warehouse

The example is adapted from Jackson [1975, p.59].

In a warehouse, several products are stored with different article numbers. The mutations (in or out) of the articles in the warehouse over the past period are stored in a file called `mutations`. This file contains one line per mutation.

Article number	Mutation code	Amount
152652	In	500
251365	Out	1200
251365	Out	570
...	...	...
999999	ZZZZ	0

The file is sorted by article number (`artnr`). The mutation code (`mutcode`) can take values `In` or `Out` (except for the last `mutcode`, where it is put equal to `ZZZZ`). The file ends with article number equal to `999999`. The objective of the program to be written is to transform this file into a file, which contains:

- a heading (with titles `Article number` and `Resulting mutation`)
- a line per article containing the resulting mutation (i.e. the sum of all amounts for ingoing mutations minus the sum of all amounts for outgoing mutations)
- a line containing the number of active articles (i.e. articles with at least one mutation).

3.4. The Program Specification Text and the Source Language

The program specification text, which allows the CASE tool to derive the example program, is:

```
INPUT: mutations
= (artnr * (mutcode, amount));
mutcode = 'In' OR 'Out';
LAST (artnr) = '999999';
OUTPUT: results
= heading (artnr, resmut) artcounter;
resmut = totIN - totOUT;
totIN = SUM (amount, mutcode = 'In');
totOUT = SUM (amount, mutcode = 'Out');
artcounter = COUNT (artnr);
```

This source text contains the following parts:

1. A description of all input files: all input files are described (`INPUT: filename = description`, where `description` is instantiated by a succession of all variables in the file). If the variables are repeated more than once, they are put between round brackets. Variables that always appear on the same level in the file, are separated by a comma. The asterisk behind `artnr` signifies that it is repeated for each mutation. `artnr` is, however, not between the same brackets as `mutcode` and `amount` because it repeats (or iterates) on a different level, i.e. there are multiple mutations in the file for each article. Using brackets, the user can indicate the different iteration levels of the input file.
2. An enumeration of the possible values for enumerated types in the input files: it is possible to indicate the finite set of values that some variables can take on by separating these values by `OR`. In this case, the variable `mutcode` can take on only two values: `In` and `Out`. It is also allowed to use `ELSE` as a value. The values (including `ELSE`) can be used in selections later on in the program specification text.
3. End-of-file conditions for input files: an end-of-file condition can be specified, using the reserved word `LAST`.
4. A description of all output files: The syntax is identical to the syntax for the description of input files.

5. The formulas that calculate variables in the output file(s): some variables in the output file(s) are the results of calculations using variables in the input file(s) and temporary variables. Arithmetic operators are provided, as well as `SUM` and `COUNT` functions. The `SUM` functions used in this example are conditional, e.g. `totIN` is incremented with `amount` only if `mutcode` equals `In`.

3.5. Transformation of the Program Specification Text Into a Pseudo Code Program

In the following paragraphs, we closely examine the program specification text and the way to convert it automatically into its pseudo code equivalent, i.e. the program that converts the input file into the output file.

Step 1: Make a data structure for input and output flows

The input flow is a file called `mutations`. The output flow is the file to be created by the program. We have called this file `results`. For each file, a data structure has to be made. Data structures (sometimes called Jackson structures)

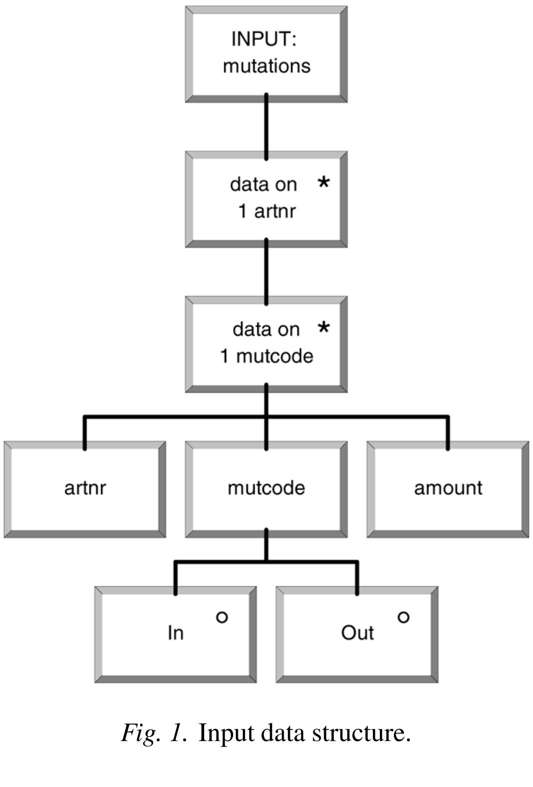

Fig. 1. Input data structure.

combine sequence, iteration and selection components. Data structures carefully reflect the structure of the file they represent. The data structure for the example input file is:

In Jackson structures, iteration components are represented by an asterisk in the upper right corner. Selection components are represented by a small circle. The information supplied to the tool should reflect the structure of the file. In the source language, variables that take part in a sequence are separated by commas, iterations are put between round brackets. The asterisk behind `artnr` (article number) in the program specification text means that the article number is repeated for each mutation

If this is not the case, the asterisk should be omitted. The third line tells us that `mutcode` can only take values `In` or `Out`.

The data structure for the output file `results` is:

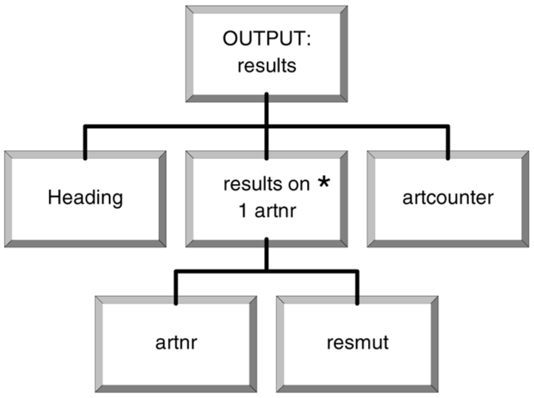

Fig. 2. Output data structure.

The rules for deriving the input and output structures from the program specification text are very simple, as the user is forced to indicate the structure of these files by using brackets and other structure indicators. We will not describe these rules in detail.

Step 2: Make a list of input and output instructions and allocate them to the corresponding structure

The input and output instructions are easy to find. All files have to be opened and closed. Variables in an input file have to be read from this file and variables in an output file have to be written to this output file. Variables that iterate at the same level and are not separated by a lower-level iteration can be read or written together. The input instructions are:

```
1. Open file: mutations
2. Close file: mutations
3. Read (mutations): artnr, mutcode,
   amount
```

It can be easily seen how this reasoning can be extrapolated to other problems. The allocation of the input instructions is the next step. Files are always opened at the beginning of the program and closed at the end. The first instruction is allocated at the beginning of component `INPUT: mutations`. Similarly, the second instruction is allocated at the end of this component. The third instruction is allocated twice, because a read-ahead is needed to evaluate the iteration condition. The result of the allocation of the input instructions is:

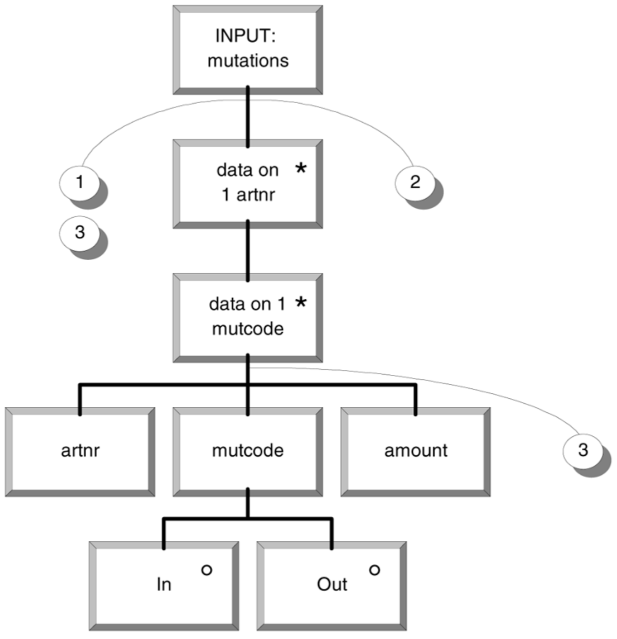

Fig. 3. Input structure with allocated input instructions.

The output instructions are obtained in the same way. The rules for allocation of the instructions are identical to those used for input instructions. There are, however, several write instructions. The instructions are:

```
4. Open file: results
5. Close file: results
6. Write (results): heading
7. Write (results): artnr, resmut
8. Write (results): artcounter
```

The automatic allocation of these instructions to their correct components yields Figure 4.

Step 3: Join the input structures, forming the combined input structure.

As we only have one input structure here, this step does not apply. However, the joining of input structures happens exactly the same way as the joining of input with output structure(s) in step 5.

Step 4: For each iteration component, allocate an iteration condition to the combined input structure.

The input structure has two iterations: `data on 1 artnr` and `data on 1 mutcode`. The iteration for the former one repeats until the end of the file has been reached. In this case, the end of the file is marked by the article number `999999`. The iteration condition that is added to the combined input structure is:

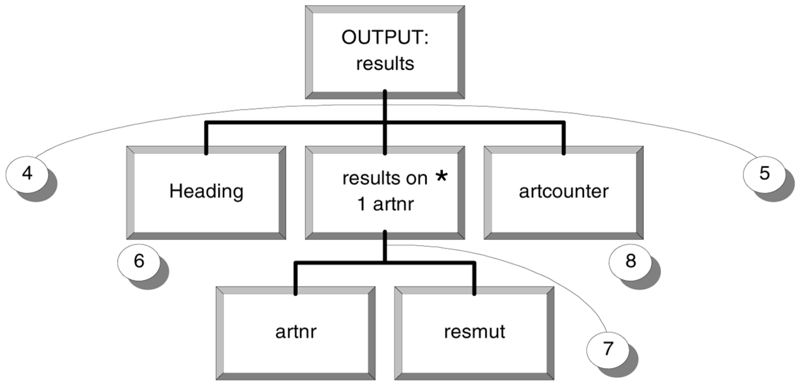

Fig. 4. Output structure with allocated output instructions.

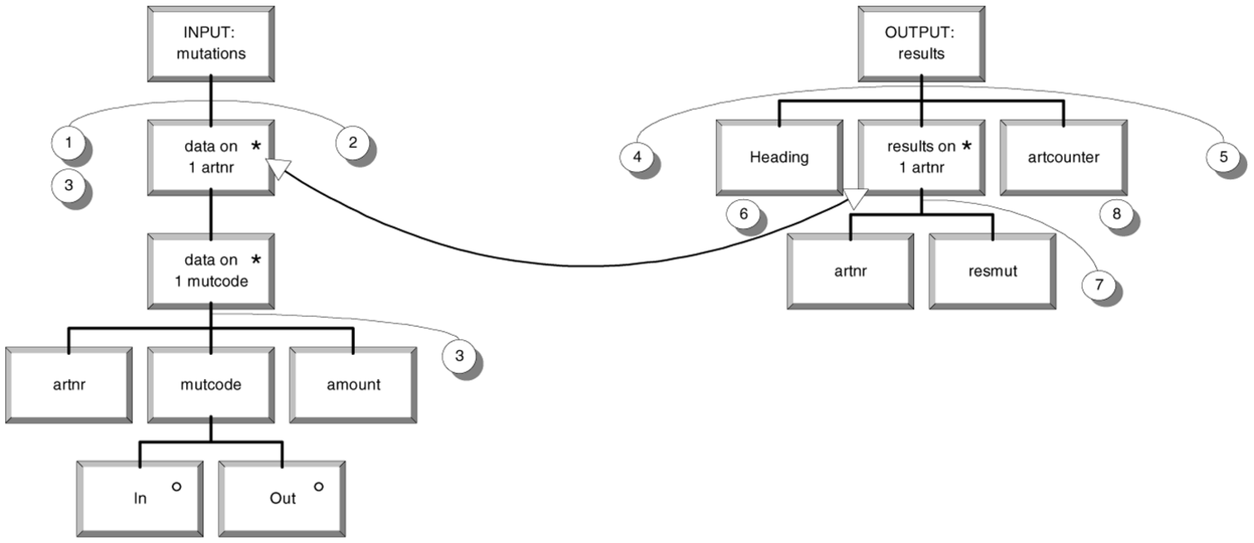

Fig. 5. Merging input and output structures.

```
While artnr <> 999999
```

The next iteration condition deals with the `data on 1 mutcode` component. This iteration condition checks whether the program is still dealing with the same article. If the `artnr` read is the same as the previous one, this is the case. Therefore, the program must compare the current article number with the previous one, which we call the reference article number `artnrREF`. The iteration condition is:

```
While artnr = artnrREF
```

This condition cannot be evaluated yet, because the reference article number is not yet assigned. A new instruction needs to be created:

```
9. ArtnrREF = artnr
```

With this instruction added, the program is able to check whether the `artnr` read is still the same as the previous one. Because the reference article number changes when a new article number is read, this instruction is allocated at the beginning of the `data on 1 artnr` iteration component. This reasoning can be easily generalised. Each iteration component (except for the highest level iteration) needs a reference variable to evaluate whether the iteration should be stopped or continued. The instruction changing the value of this variable is always the same. It is always allocated at the beginning of the iteration component of this variable.

Step 5: Join the combined input structure with the output structure(s), forming the program structure.

Searching for correspondences between structures is very easy because data structure components are labelled by variable names. Iteration components that have the same name (except for the `data on 1` or `results on 1` part) are connected. The result is Figure 5.

Merging the structures into the program structure is done by taking the union of both structures, thereby merging two corresponding components into one. The rules for this operation

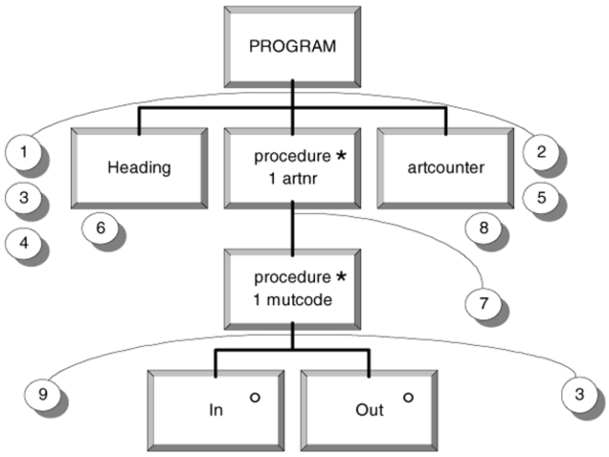

Fig. 6. Program structure with allocated input and output instructions.

are not discussed here. For more complex correspondences between program structures, involving e.g. structure clashes, there are specific rules. These rules are beyond the scope of this article. The result of joining the input and output structures is given in the figure 6.

Note that the lowest level sequence is left out to save space and instruction 9 is added, i.e. the instruction that updates the reference article number. Also, the `data on 1` and `results on 1` prefixes are changed into `procedure 1` to make clear that the iteration components are no longer input or output components, but program components.

Step 6: Make a list of logical instructions and allocate them to the program structure.

The logical instructions serve to calculate the resulting mutation for each article and the number of active articles. The first variable, `resmut`, is calculated as the sum of all mutations `In`, minus the sum of all mutations `Out` for each article. The following line is added to the program specification text:

```
Resmut = totIN - totOUT;
```

Two new variables: `totIN` and `totOUT` have to be declared. To do this, we define the conditional `SUM` function. The lines in the program specification text are:

```
totIN = SUM (amount, mutcode = 'In');
totOUT = SUM (amount, mutcode = 'Out');
```

Given these lines, the tool has sufficient information to make a list of all instructions needed to calculate `resmut` and allocate them to the program structure. The list of instructions:

```
10. Resmut = totIN - totOUT
11. TotIN = 0
12. TotIN = totIN + amount
13. TotOUT = 0
14. TotOUT = totOUT + amount
```

The first instruction is copied without change from the program specification text. In general, all calculations without a `SUM` function (or a `COUNT` function, see later) can be copied literally from the program specification text. Instructions 11 and 13 initialise respectively the variables `totIN` and `totOUT`. This is necessary because instructions 12 and 14 use these variables both in the right and left hand side of the equation. These instructions can be allocated automatically without any extra information added to the program specification text. The tool has to check at which level the variables iterate. As a general rule, a variable on the right hand side of an equation iterates at the same level as the variable on the left hand side. So, `totIN` and `totOUT` iterate at the same level as `resmut`. As `resmut` and `artnr` are between the same pair of brackets in the program specification, both variables iterate at the same level. The initialisation instructions are allocated to the beginning of the iteration component that iterates at the same level (i.e., instructions 11 and 13 are allocated to the beginning of the `artnr` iteration component). Instructions 12 and 14 are allocated to the selection components `mutcode = In` and `mutcode = Out` respectively. This reasoning applies to each instruction of this type, derived from the conditional `SUM` function. The only variable left to be calculated is `artcounter`. The calculation of this variable is very similar to that of `resmut`. We will not go into any further detail. The final program structures, with all instructions allocated, is given in the following figure:

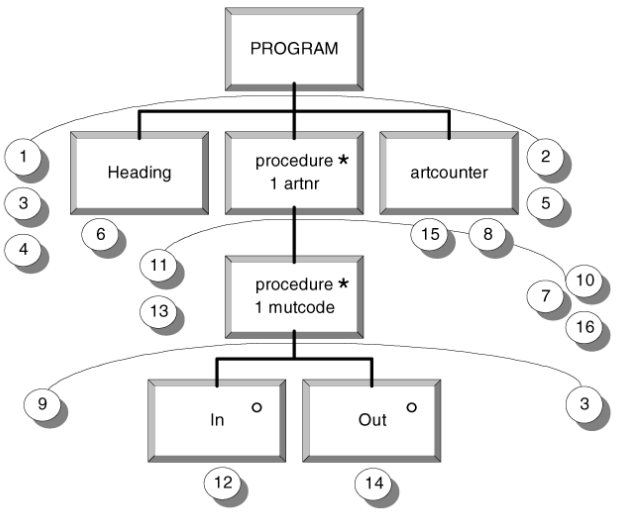

Fig. 7. Program structure with all instructions allocated.

Step 7: Add selection conditions to the selection components of the program structure.

In the example, only two selection components exist. Their conditions are respectively: `mutcode = In` and `mutcode = Out`.

Step 8: Transform the program structure representation into a code or pseudo code representation.

The final step in JSP is to read the program structure counter clockwise and write down any instruction, selection and/or iteration condition encountered. The result is a pseudo-code program that transforms the input file into the output file using the transformation rules described in the program specification text:

```
Open file: mutations
Open file: results
Read (mutations): artnr, mutcode, amount
artcounter = 0
Write (results): heading
Itr while artnr <> '999999'
  totIN = 0
  totOUT = 0
  artcounter = artcounter + 1
  artnrREF = artnr
  Itr while artnrREF = artnr
    Sel
      if mutcode = 'In'
        totIN = totIN + amount
      if mutcode = 'Out'
        totOUT = totOUT + amount
    Endsel
    Read (mutations): artnr, mutcode,
                      amount
  Enditr
  resmut = totIN - totOUT
  write (results): artnr, resmut
Enditr
Write (results): artcounter
Close file: mutations
Close file: results
```

3.6. Additional Features of the Technique

The example solved here uses only part of the possibilities of the technique we developed. As mentioned before, for programming problems involving more complex input and output structures, other constructs are available in the source language. This allows the user to deal with problems involving program inversion and program structure clashes.

3.7. JSPTool

Based on the technique described in this paper, a program has been developed which follows the rules presented earlier. This program is called `JSPTool`. It is conceived as a simple command-line compiler, which generates a target file from a source file. The source file contains the program specification text in the source language described in this paper. `JSPTool` works like a compiler. It parses the input file sequentially to find tokens (meaningful units). Depending on the token found, different actions are taken. Most of these actions update internal data structures, which are used to perform the steps of the modified JSP algorithm. We will not go into the technical details of the implementation of these data structures.

We have succeeded in solving a large number of problems within a limited problem class and believe that creating a source file is a lot easier and less error-prone than traversing the entire JSP process. At this stage, the scope of the CASE tool is probably too limited to be of much use in commercial projects. We believe, however, that similar techniques and CASE tools can be devised that are able to solve a much broader class of problems.

4. Conclusion and Further Research

In this paper, the development of a technique for automatically transforming program specifications into pseudo code using JSP is described. This technique is explained by means of an example. We have illustrated how a CASE tool based on this technique can convert a program specification text, describing input and output files, into a pseudo-code program using JSP as a programming method. Based on our technique, a prototype program has been developed, `JSPTool`, which implements the technique in practice.

From the beginning, our objective has been to completely automate a programming method.

We believe the strength of our technique lies in the fact that, for a limited number of problems, the programming work is done completely automatically.

As mentioned before, we believe that a similar technique and similar CASE tools can be developed for more complex programming problems. Such CASE tools would allow for large-scale commercial projects to be performed in less time and with fewer errors.

References

[1] ADELSON, B. & SOLOWAY, E., The role of domain experience in software design, *IEEE Transactions on Software Engineering,* vol. 11, no. 11, 1985, pp. 1351–1360.

[2] CAMERON, J.R., JSP & JSD — The Jackson Approach to Software Development, Silver Spring, *IEEE Computer Society Press,* 1983, vi +257 pp.

[3] CAMERON, J.R., An Overview of JSD, *IEEE Transactions on Software Engineering* vol. 12, no. 2, 1986, pp. 222–240.

[4] CHATEL, S & DÉTIENNE, F., Strategies in object-oriented design, *Acta Psychologica,* vol. 91, 1996, pp. 245–269.

[5] DAVIES, C.G., Problems of Maintenance of JSP Structures, Software Maintenance, *Research and Practice,* vol. 2, 1990, pp. 175–192.

[6] GUINDON, R., Knowledge exploited by experts during software system design, *Int. J. Man-Machine Studies,* vol 33, 1990, pp. 279–304.

[7] HOORELBEKE, G., Gebruik van CASE in België, *Informatie,* vol. 35, 1993, pp. 729–735.

[8] HUGHES, J.W., A formalization and explication of the Michael Jackson method of program design, *Software Practice and Experience,* vol. 9, 1979, pp. 191–202.

[9] JACKSON, M.A., *Principles of Program Design,* London, Academic Press, A.P.I.C. Studies in Data Processing No. 12, 1975, xii +299 pp.

[10] JACKSON, M.A., *System Development,* 1983, London, Prentice Hall, xiv +418 pp.

[11] JANSEN, H., *JSP — Jackson Structureel Programmeren,* Den Haag, Academic Service, 1986, xvi +455 pp.

[12] MCKEITHEN, K.B., REITHMAN, J.S., RUETER, H.H. & HIRTLE, S.C., Knowledge organization and skill differences in computer programmers, *Cognitive Psychology,* vol. 13, 1981, pp. 307–325.

[13] RIST, R.S, Variability in program design: the interaction of process with knowledge, *Int. J. Man-Machine Studies,* vol. 33, 1990, pp. 305–322.

[14] ROPER, M. & SMITH, P., A Structural Testing Method for JSP Designed Programs, *Software Practice and Experience,* vol. 17, no. 2, 1987, pp. 135–157.

[15] SONG, X. & OSTERWEIL, J., Experience with an Approach to Comparing Software Design Methodologies, *IEEE Transactions on Software Engineering,* vol. 20, no. 5, 1994, pp. 364–384.

[16] SUTCLIFFE, A.G., & DAVIES, C.G., MAJIC — an integrated program support environment, *Information and Software Technology,* vol. 29, no. 3, 1987, pp. 122–136.

[17] TRIANCE, J.M., Structured programming in COBOL — the current options, *The Computer Journal,* vol. 23, no. 3, 1979, pp.194–200.

[18] WELSH, J. & MCKEAG, R.M., *Structured System Programming,* London, Prentice Hall, 1980, xii + 324 pp.

Received: February, 1999
Revised: May, 2000
Accepted: June, 2000

Contact address:
Kenneth Sörensen and Jan Verelst
Department of Operations Research
Logistics and Information Systems
University of Antwerp – RUCA
Middelheimlaan 1 – B-2020 Antwerp – Belgium
e-mail: ksorense@ruca.ua.ac.be — verelst@ruca.ua.ac.be
http://tew.ruca.ua.ac.be/orlis

KENNETH SÖRENSEN (1974) graduated as a commercial engineer in information science from the University of Antwerp – RUCA. After spending some time in a software consulting firm, he returned to the University of Antwerp, where he is now working as an assistant teacher/researcher in the department of Operations Research, Logistics and Information Systems. He is interested in automation of programming and CASE tools and is also working in the field of logistics and supply chain management.

JAN VERELST (1970) received the PhD degree from the University of Antwerp – RUCA in 1999. He is currently working as doctor-assistant at the same university. His research interests include program design, conceptual models of information systems, object-orientation and evolvability of information systems.