SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 19(4), 393–409 (APRIL 1989)

Generation of File Processing Programs based on JSP

ROLF ENGMANN AND FRANS VAN HOEVE
Department of Informatics, University of Twente, Postbus 217, NL-7500 AE Enschede, Netherlands

SUMMARY

This paper describes the generation of file processing programmes within the TUBA environment. Program structures are derived from data structures according to the JSP method. Expressions describing output data are specified in user–system dialogues. The program specifications are stored in the dictionary. Complete executable programs can be generated from these specifications.

KEY WORDS Application generation Program generation JSP Interactive program development File processing Business data processing

INTRODUCTION

One of the main goals of software engineering is upgrading the software production process in order to bridge the gap between hardware and software costs. At present an increasing number of tools are available to support the production of software. A well-known example of such tools is the class of non-procedural languages, such as SQL. Non-procedural languages are characterized by great expressive power in combination with a rather simple syntax. However, the programmer still has to cope with syntactic constraints, which prevent him focusing his full attention on the real problem. A class of tools that avoid this disadvantage is that of the so-called program generators.

A program generator[1] accepts, in an interactive dialogue with the user, specifications of a particular application program, and transforms such specifications automatically to the source code of the application program. Usually the generated program is written in a third-generation multi-purpose language, e.g. COBOL or Fortran. Program generators should provide user-friendly interfaces and timely error detection and should check consistency and completeness of the entered specifications. It would be conceivable to construct program generators for a wide range of programs. However, too much freedom in specifying application programs prevents the generator exercising sufficient control on correctness and completeness of the user input. For this reason it is desirable that a generator system be specific to a limited range of applications.

The main objective of the TUBA project[2–5] is the design and implementation of user-friendly tools for the generation of business-oriented application programs. (TUBA

0038–0644/89/040393–18$09.00

Received 7 June 1988
Revised 5 November 1988

is an acronym for Tools for User-friendly Business Applications). For technical reasons and from the user's viewpoint application programs are classified into the following types:

1. *Batch* programs or file processing programs, which have no user interaction.
2. *Report* programs, a type of batch programs dedicated to generating reports from one or more input files.
3. *Interactive* programs, which maintain a dialogue with the user during execution.

The TUBA system is intended to provide generators for these different types of programs. The TUBA report program generator has been described in Reference 2. The present article describes principles and techniques for the generation of batch application programs. Generation of this class of programs with a non-procedural language interface has been discussed by several authors.[6–8] The TUBA batch program generator is based on an interactive interface and is especially designed for the generation of programs that process one or more files; sequentially reading one of the input files triggers operations, such as access and write operations on the other relevant files. Practical examples are programs for order processing, invoicing and stock control.

One of the main techniques used in the TUBA environment is Jackson Structured Programming[9–12] (JSP). JSP represents a systematic method for designing correct, well-structured and maintainable programs. It is based on the structural similarity between streams of data (i.e. sequentially processed files) and the programs that operate on the data, because a program as well as a data stream can always be arranged in sequences, selections and iterations. In the present system the structure of a batch program is derived from the structure of the input and output data streams according to the JSP technique. Expressions specifying the values of output items are entered interactively; in this way complete programs can be generated without a programming language interface and without the insertion of any manually-written code.

In the TUBA environment generation of a batch program consists of the following steps:

1. Entry of specifications of files into the dictionary; generation of so-called file modules containing declarations of data structures in these files. This step is only required for new files.
2. Entry of global specifications (program name, connected files) of the batch program into the dictionary.
3. Specification of types of transactions (*input*, *output*, *delete*, *update*) on the connected files and the correspondences between data structures.
4. Specification of expressions describing output items.
5. Generation of the source program.

This paper describes the following aspects of the TUBA batch program generator system:

(a) the modular composition of batch application programs
(b) the specification of file structures including their group structure
(c) the derivation of a program structure from the data structures of files
(d) the types of the programs that can be generated by the present system
(e) the role of the dictionary in the generation process and the technical aspects of the generation of programs.

THE TUBA ENVIRONMENT

Programs in the TUBA environment are composed of software modules sharing only very restricted interfaces. Modules may contain both data and procedures; they enable the construction of software systems according to principles of object-oriented programming.[13] The implementation language at present used in the TUBA project is Simula.[14]* Simula provides the possibility to specify modules as classes, which can be compiled separately. A module is implemented as an object (instance) of a class or as a program block. Modules can be mutually connected either by prefixing or by external referencing.

In the TUBA environment the following types of modules are distinguished:

1. *System modules* are used as a mechanism to cluster a set of procedures and data, which implement specific functions, such as file organization, file manipulation, screen handling and report writing. System modules are generally available in the TUBA environment and may be linked to any program as they are application-independent.
2. *Generated modules* are created by TUBA code generators for specific applications. Examples of generated modules are file modules, which contain the description (meta data) of a specific file type and procedure calls for the creation of objects representing items, records and groups.
3. *Application modules* contain the statements and local data declarations specific for an application program.

TUBA application programs may contain several system modules and generated modules; in all cases there is one application module.

Meta data describing files, reports, screens and programs are stored in the dictionary. Generated modules are generated by code generators, which use data and parameters to be retrieved from the dictionary. In this way modules may be created representing specific objects, such as file structures, records, items, reports, screens, etc.

SPECIFICATION OF DATA STREAMS AND PROGRAMS

In JSP a program is modelled corresponding to the input and output data streams. Data streams originate from reading or writing data sequentially. The structure of a data stream may be represented by a JSP data structure diagram, containing elementary components, sequences, iterations and selections. The program structure diagram is derived from the corresponding data structure diagrams.

In the TUBA system the description of file structures is stored in the dictionary. A file description contains specifications of record types and item types. For the specification of the layout of reports a facility to add one or more group structures to a file description has been implemented.[2, 5] In a group description the contents of a file is divided into groups of consecutive records. Each of these groups may be subdivided into subgroups, etc. In the TUBA environment each particular type of group is defined

* Simula is a registered trademark of Simula a.s., Oslo.

by its starting condition. Depending upon the number of record types that may occur in the related file two types of starting conditions may be distinguished:

(i) a particular field, a so-called control field of a specific record changes its value compared with its value in the previous record
(ii) a particular record type becomes current; this condition is only meaningful when the file contains more than one record type.

During report generation the occurrence of a group in the input file may cause specific print lines in the report, e.g. heading lines and detail lines of an invoice.

As these group structures are functionally equivalent to the JSP description of data streams this facility was applied to the generation of programs from the structure of the data. The report writer environment of TUBA provides facilities for processing data and evaluation of arithmetic functions over groups of records. These facilities have also been employed in the implementation of batch application programs.

Global specifications of a batch application program contain the specification of the files to be processed by the program, including a description in terms of group structures relevant for the application. As an example we describe an invoice file with the record types header, invoice line and trailer. The file consists of an iteration of invoice groups, each invoice group consists of one header record, followed by an iteration of invoice line records and is terminated by one trailer record. The corresponding JSP data structure is shown in Figure 1.

In the TUBA environment a program is composed of a set of modules. Figure 2 shows the modular configuration of a batch application program in a simplified schema. The application module is connected to each of the files through a so-called file module. A file module contains declarations of records, items and the relevant group structures and statements for the creation of objects representing these records, items and groups. File modules are generated from the file specifications, which are stored in the dictionary. The VARDAT module is common for all TUBA application programs. The structure and functions of the modules in TUBA applications have been described in greater detail in Reference 5.

As has been explained in References 3 and 5, originally the application module had to be coded manually. However, in the batch program generator system the application module can be generated automatically for several types of batch application programs. The implementation of the generation of this type of application modules is the main

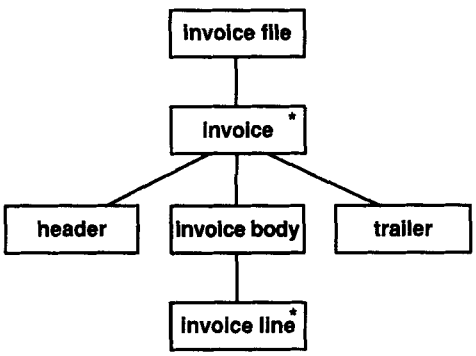

Figure 1. JSP structure of an invoice file

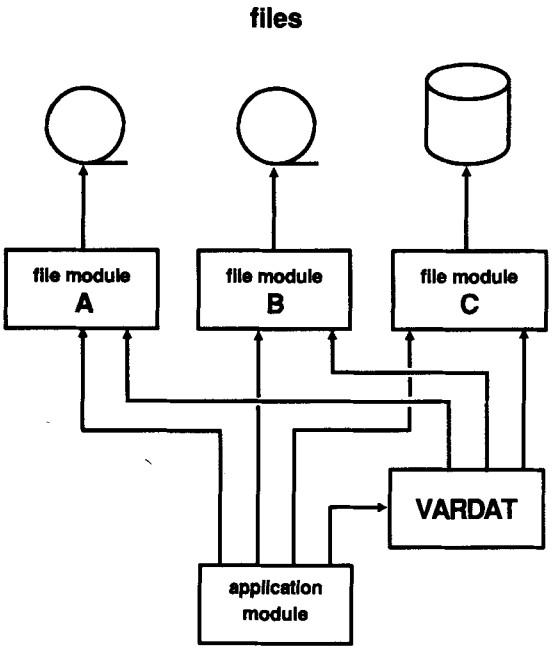

Figure 2. Modular configuration of a batch application in TUBA

subject of the present paper. In the following we refer to the Simula program code of the application module produced by the generation process as the *source program*.

The application module contains statements for the usual file operations such as reading and writing records and operations on individual record items and variables. In JSP the program structure is derived from the data structures, such that the program structure is a superposition of those data structures. The application module should therefore reflect the structure of the connected files, i.e. the group structures of those files. If there is only one file the problem is trivial, because in such a case the program structure is identical to the data structure of the file. When more files are involved the program structure is derived from the data structures of the individual files by a merge process. The merge process is explained in the following section.

MERGING DATA STRUCTURES TO A PROGRAM STRUCTURE

In the following discussion of the process of merging JSP data structures it is presumed that two data structures are involved, namely the input and the output structures. If necessary the merge process can be extended to more files by consecutive merge operations.

For the description and the implementation of the merge algorithm the data structures are represented as tree structures, the so-called data trees. A component of a data structure is represented as a tree node. Tree nodes have an attribute COMPONENT TYPE, which may assume one of the possible values sequence, selection, iteration, terminal. A node of the component type terminal denotes an elementary component; in practical cases such a component represents a single record, because records are the

units of file access operations. A node of the component type sequence has two or more parts occurring once each, in order. A node of the component type iteration has one part, occurring zero or more times. A node of the component type selection has two or more parts, of which one occurs once. This terminology is completely consistent with JSP,[9] but in this paper the type of the components is used more explicitly. Figure 3 shows the different component types.

Data structures can be merged when the mutual correspondences between components in both structures are known. Correspondences should be specified by the user of the system as they reflect the semantics of the problem. In the TUBA implementation the system presents all components in one of the structures one at a time and asks the user to indicate the corresponding component in the second structure. As an example we consider a program that processes an order file and produces an invoice file. The data structures of the files are shown in Figure 4; the correspondences between components as they might have been specified by a user are indicated by double-headed arrows.

During the merge process the TUBA system compares the data structures to be merged. When the system detects structure clashes according to the cases described by Jackson[9] it suggests to the user possible solutions for resolving those clashes. Ordering clashes and multi-threading clashes may be resolved by sort operations; the TUBA system provides utilities for file sorting. Boundary clashes may occur, for instance in the creation of reports; the TUBA report writer can handle this kind of structure clash in a standard way.

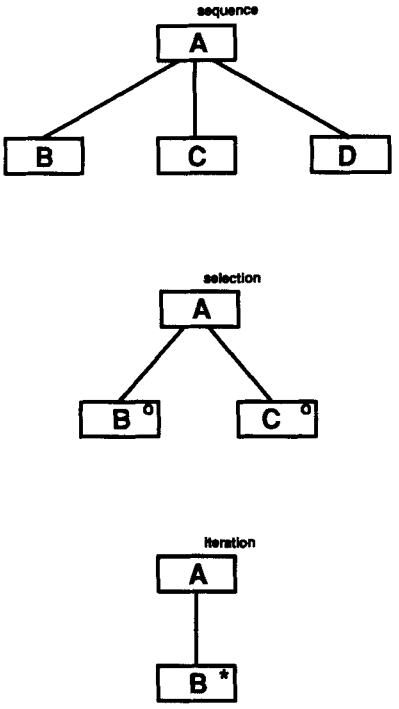

Figure 3. Component types in JSP structures

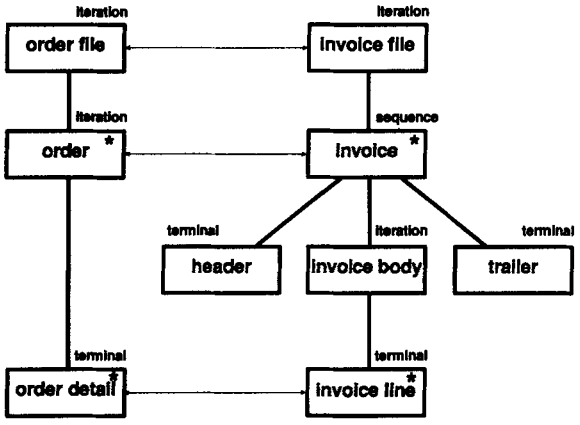

Figure 4. JSP data structures of an order file and an invoice file

For the rather simple case shown in Figure 4 a possible corresponding program structure is shown in Figure 5. For those familiar with JSP techniques, merging data structures into a program structure may not be difficult; however, as we wish to automate the generation of the program from the data structures, the merge operation has to be described more formally. Some aspects of automatically merging data structures have been discussed by Wilson.[15]

The merge operation

The merge operation is applied to two data trees with possible correspondences between nodes as they have been specified by the user. The data trees are represented in internal storage; the nodes of the trees are represented by class objects with attributes

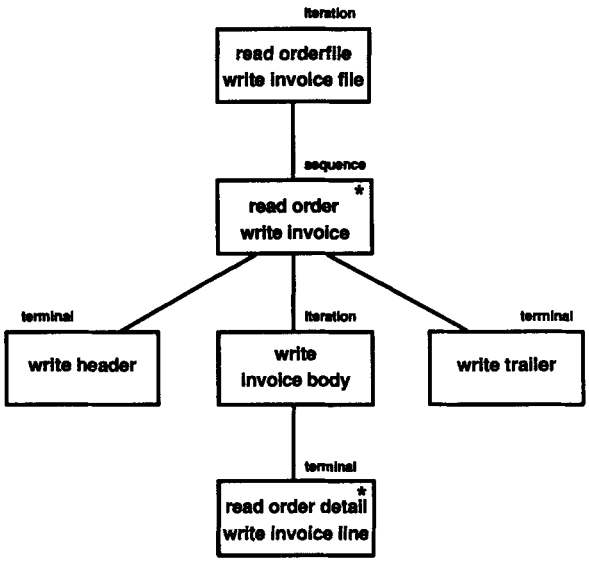

Figure 5. JSP program structure for processing order file and producing invoice file

of the type reference, comparable to Pascal pointers to link the nodes in a tree structure. Component types and correspondences between components in both data trees are also explicitly represented in the nodes by attributes.

The merge operation yields a third tree, the so-called *merge tree*, which represents the final program structure. In the merge tree each node corresponds to either one node in one of the data structure trees or a pair of corresponding nodes, each in one of the original data trees. Initially the resulting merge tree is built up in internal storage, analogous to the representation of the data trees. Eventually the merge tree is stored in the data dictionary.

During the merge operation one of the data trees is traversed in some order, e.g. in post-order. Each visited node is inspected; if it has a correspondence with a node in the other data tree and it has not already been merged, the end nodes of two corresponding branches have been detected, which have to be merged. Such a branch is a path between two nodes in one of the data trees with correspondences to nodes in the other data tree. For short we refer to the nodes at the two ends of the branch as the lower and upper nodes.

For merging two corresponding branches the order of the components in the resulting merge tree should be determined unambiguously from these branches. The following restrictions are therefore adopted:

1. A selection component in the output data structure should correspond to a selection component in the input data structure. This rule ensures that a selection on the output data is controlled by a selection on the input data.
2. An iteration component in the output data structure should correspond to an iteration component in the input data structure. This rule requires that an iteration in the output is controlled by an iteration in the input.
3. Sequence and terminal components in the output should correspond to any type of input components.

An iteration in the output may correspond indirectly to an input iteration through a sequence; the same holds for selections. As an example consider the data trees shown in Figure 4. The iteration invoice body is a part of the sequence invoice; which corresponds to the iteration order in the input tree. Implicitly the components header, invoice body and trailer as direct parts of invoice, also correspond to order, because they occur exactly under the same conditions, in the same ordering and in the same number. This means that effectively invoice body corresponds to order, which satisfies the rule that this iteration should correspond to an iteration in the input. More generally, a correspondence to a sequence or a part of a sequence may be extended to the sequence including its parts.

When the nodes in the output tree are unambiguously related to nodes in the input tree the resulting merge tree can be constructed. If there are not enough correspondences to construct the merge tree there could be a structure clash; in this case the system produces a diagnostic message.

The component type of a node in the merge tree that originates from two corresponding nodes in both data trees (in two corresponding branches this is the case for the lower and upper nodes) is determined by the component types of these nodes. If the component types are equal, the component type of the resulting node is the same. The combination of a sequence with a selection is shown in Figure 6. In interpreting this merge tree it should be mentioned that an occurrence of node Q in the output effectively

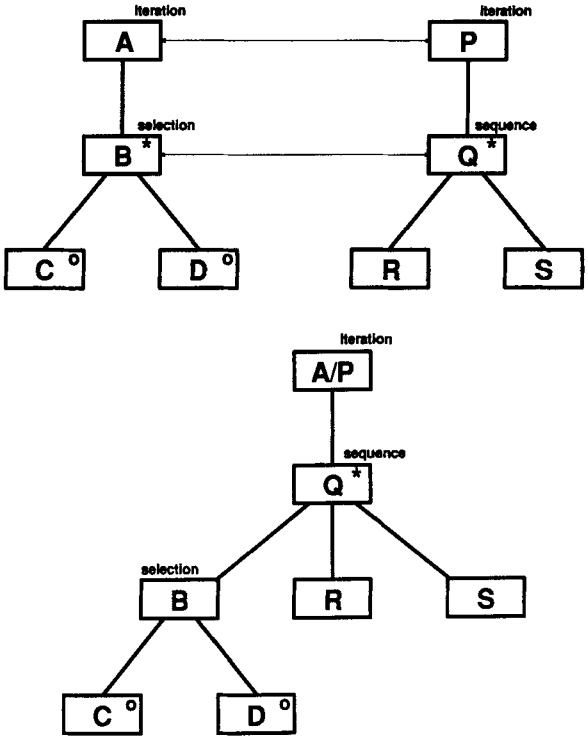

Figure 6. Merge of selection and sequence

is produced by an occurrence of node B in the input. The combination of a sequence with an iteration or a terminal is completely analogous. As ordering and numbers of occurrences of all components remain the same in the merge tree compared to the data trees, obviously the transformations are correct.

When all corresponding branches have been merged into the merge tree the branches without corresponding nodes are copied to the merge tree. This completes the merge of the data trees into a merge tree.

The merge operation can be illustrated by the derivation of the program structure from the data structures shown in Figure 7. Corresponding branches are A–B–D in the left tree and P–R–T in the right tree. A practical interpretation could be the following: the left tree with root component A represents a file with records describing tests on produced parts; B represents an individual test, C represents a passed part, D represents a failure, E describes the part that failed and F describes the kind of failure. The second tree represents a failure report; Q represents the report heading, T is a report line and S represents the report summary, e.g. the number of parts that failed.

The nodes A and R are indirectly corresponding and can be combined into the compound node A/R. D and T are corresponding lower nodes, which give rise to the node D with component type sequence and parts E, F and T as shown in the merge tree in Figure 8. Finally the remaining nodes, in this case B, are copied to the merge tree. This completes the merge of the branches A–B–D and P–R–T.

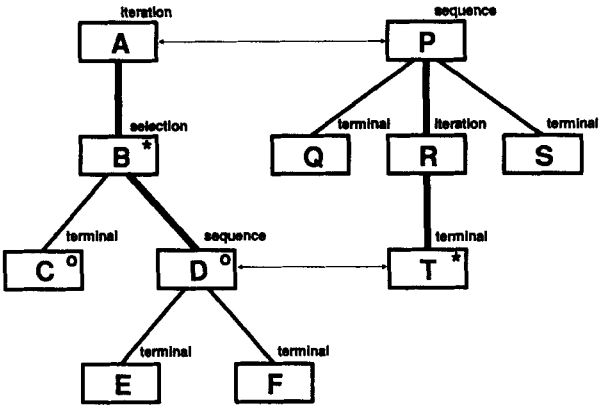

Figure 7. Example of two data structures to be merged

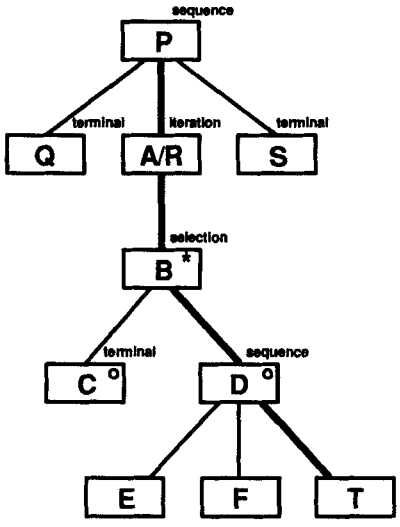

Figure 8. Merge tree resulting from data structures shown in Figure 7

The merge tree shown in Figure 8 is completed by adding the remaining branches without corresponding nodes to the constructed branch P–(A/R)–B–D–T.

After the merge process file access operations (read, write, rewrite, delete) are added to the constructed merge tree. The merge tree is written from internal storage to the dictionary.

SPECIFICATION OF OUTPUT ITEMS

After the specification of the correspondences between data structures and the construction of the program structure the user may enter the expressions determining the values of the output items. In this context output items could be items of output files and items of files that are updated by the program to be generated.

To make the source program simple and concise the facilities offered by the already-existing report writer of the TUBA system are linked to the source program. The

facilities include the evaluation of numeric variables and functions over records and groups of records. Functions over a record are addition, subtraction, multiplication and division of numeric items of the corresponding record. Functions over groups of records are sum, average, minimum and maximum of an item. In this way, for instance, the sum of the values of an item line amount in a group of invoice line records is evaluated and assigned to an item total amount by the procedures in the report writer module. The item line amount may be specified for invoice line records as quantity × article-price. In the source program such evaluations are specified in procedure calls. The execution of the evaluation of the relevant functions is triggered by a call to the report module following each read operation on the input file.

The evaluation of output items is specified as follows. The output items are presented one by one to the user. For an item of type text the user has the choice to enter a constant text or to specify the input item from which the value should be copied.

For a numeric item the user may specify a numeric expression. Such an expression may contain the usual arithmetic operators as well as functions over groups as has been explained previously. The specification of the expression is completely guided by the system. In a step-by-step sequence the system presents menus with relevant operands and operators from which the user has to make a choice. In this way the user is forced to specify a syntactically correct expression. Operands can be constants or items. A constant value has to be specified by the user; items occurring in an expression can be items in the input file. An expression may be conditional of the form:

⟨output item⟩ := IF condition THEN ⟨expression-1⟩ ELSE ⟨expression-2⟩

The expressions are stored in the dictionary together with the other specifications of the relevant source program.

TYPES OF BATCH PROGRAMS

The types of batch programs that can be generated are characterized by system flow diagrams.[9] In a system flow diagram the connected files and their roles (input, output, direct access) are indicated.

One input file, one output file

The simplest configuration with one sequential input file and one sequential output file is shown in Figure 9. In this and equivalent cases both files may also be organized indexed sequentially, because only the sequential access path is relevant for the processing program. The program structure is the result of the merge of the data structures of the input file and the output file.

One or more directly accessible files

One or more directly-accessible files may be connected to the processing program (see Figure 10). Access to a direct file is by means of a key item providing direct access to the individual records of the direct file. Transactions on direct files can be retrieval, update, insertion and deletion. A special case of this type of program is where there is a direct file and a transaction file as the input file, containing transactions on the direct file. Update operations on a master file have been described by Dwyer.[16]

Figure 9. System flow diagram with one sequential input file and one sequential output file

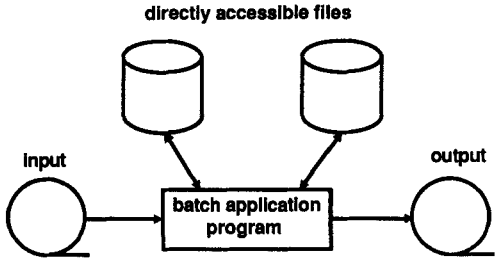

Figure 10. System flow diagram with two direct files

As directly-accessed files do not correspond to a sequential data stream they do not contribute to the program structure derived from the data structures of the sequentially accessed files. Operations on directly-accessed files are determined during the specification of output items.

For each connected direct file the type of transactions (retrieval, insertion, deletion, update) has to be specified; the user also has to specify the item in the input file that yields the value of the access key of the direct file. In the case where several transaction types are possible, the item in the input file indicating the type of the transaction to be performed should be specified.

Two input files

The configuration with two sequentially-accessed input files is relevant for merging two files, e.g. a transaction file with a master file[16] (see Figure 11). This configuration is considered as a special case of the configuration with a transaction file and a direct file. The difference is that for each transaction record the corresponding record in the master file is located by reading the master file sequentially up to the location corresponding to the transaction. Such a transaction could be an insertion, deletion or update. The resulting records are written to the output file, which is the new master

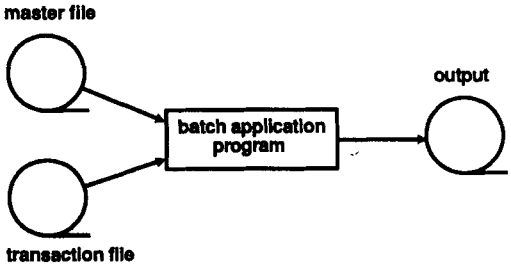

Figure 11. System flow diagram with sequential master file and sequential transaction file

file. Master records that are not deleted or updated are written directly to the output file.

In this configuration the transaction file is considered as the main input file; the transaction records may contain an item indicating the type of the transaction (insertion, deletion, update). The structure of the update program is shown in Figure 12. It is presumed that both files are sorted on the same key item in ascending order of key values. Non-matching master records are records with key values less than the current key value. The component execute transaction depends on the type of the transaction.

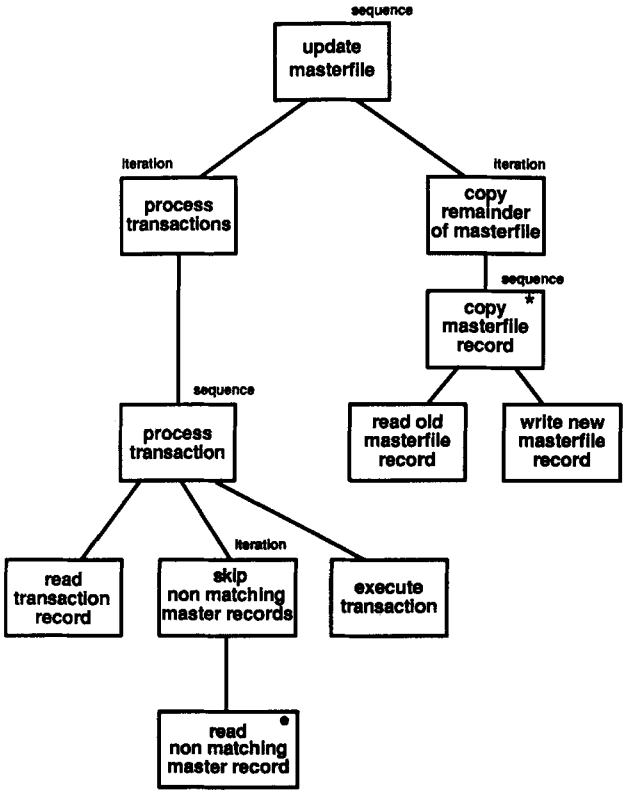

Figure 12. Program structure for updating a sequential file

Two output files

A configuration with two output files is shown in Figure 13. In this case the user has to indicate correspondences between each output data structure and the input data structure. The program data structure is constructed from the correspondences between all connected sequential files.

A complication may arise when the output record is written conditionally to one of the output files. In principle this may be indicated by a selection in the input data structure. The present system offers the user the possibility to specify a condition during the specification of the correspondences. In this case the correspondence between an input group and an output group may not be as strong as is needed in the JSP

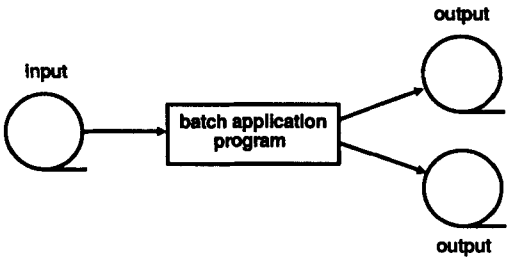

Figure 13. System flow diagram with two sequential output files

method, because the number of output components may be less than the number of input components. The user can specify this as a so-called *weak correspondence*; in such a case the user has to specify the condition controlling the output to a specific file.

GENERATION OF PROGRAMS

The main part of the specifications or meta data of batch application programs is stored in the batch dictionary (BD), which is one of the dictionary files. In this way existing specifications may easily be changed in case of errors or may be adapted to changed user needs.

The data specifying a batch application program consist of the following components:

(a) global specifications of the program (e.g. name, connected files)
(b) the nodes of the program structure, so the program tree representing the program is stored completely in the BD
(c) the transactions and access key items for directly accessed files
(d) the expressions specified for the output items
(e) the attributes of operands (items and constants) used in the expressions.

Figure 14 shows a schematic overview of the programs and dictionary files involved in the generation of batch application programs. It is assumed that the file modules have been generated already. In the schema DD stands for data dictionary file; BD stands for batch dictionary file. The heavy arrows indicate the sequence of operations in the specification and generation process.

For generating a batch application program including (new) file structures the following steps (not shown in Figure 14) should be taken before executing the program EDITBD (see also Reference 5):

1. Enter file structures including group structures into the DD.
2. Generate the file modules.

The proper generation process consists of the following steps.

Operations executed by the program EDITBD:

1. Entry of global specifications of the batch program into the BD and the DD.
2. Generation of a *batch dictionary manipulation program* (BDMP).

Operations executed by a BDMP:

3. Specification of the types of transactions on the connected files and the correspondences between data structures.

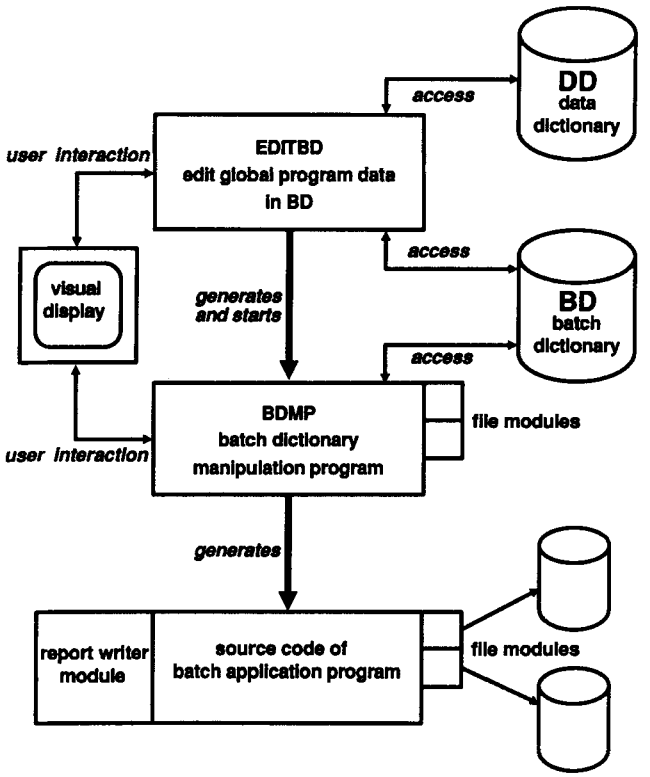

Figure 14. Schema of programs and dictionary files in the generation process

4. Specification of expressions for the derivation of values of output items.
5. Generation of the source program.

The distribution of the different functions in the generation process over several programs (EDITBD and BDMP) was compelled by the size of the programs; a combined program would be unacceptably large. In the following the functions of the programs will be discussed.

Global specification of a batch program

Global specifications of batch programs can be entered into the BD and modified by means of the interactive program EDITBD (EDIT Batch Dictionary). Entry of new global specifications is equivalent to the initial definition of a completely new batch program to be generated. The global data consist of:

(a) system name and catalogue name of the batch program
(b) name of the author of the program
(c) a narrative description of the program function
(d) the files connected to the program.

These global data are stored in the BD; additionally the system name and the catalogue name are also stored in the DD, because the DD contains the catalogue of all TUBA programs of a specific user. EDITBD enables the user to insert, modify and delete

global specifications. After insertion or manipulation of global specifications EDITBD may generate a new BDMP.

Internal specifications of batch programs

A Batch Dictionary Manipulation Program (BDMP) is an interactive program, which offers the user the following facilities:

(a) specification of operations on connected files
(b) specification of correspondences between data structures
(c) specification of expressions for output items
(d) modification and deletion of existing specifications
(e) generation of the source program.

A BDMP stores a complete specification of a batch application program in the BD from which the source program may be generated.

As shown in Figure 14 the file modules of the files of the source program are attached to a BDMP. These file modules contain full descriptions of record types, item types and group structures of the files.[5] The coupling of the file modules is specified in the Simula source code of the BDMP, which is the reason that a specific BDMP is generated for each source program. Another solution would be to have one common BDMP, which would retrieve the file specifications from the DD. However, as the relevant data are directly available in a structured way within the file modules the latter solution is less efficient.

The application module of a BDMP is very small as functions that are common to all BDMPs are collected in standard modules. For this reason the generation and compilation of a BDMP are minor operations.

Implementation

The system has been implemented on a DECsystem-20 mainframe computer. As the Simula system is very fast the time needed for operations such as compilation and generation is only a matter of seconds. For this reason the system is very effective in practical circumstances. At present work is being devoted to the implementation of the TUBA system on a SUN workstation.

CONCLUSIONS

The present facilities of the TUBA system enable the user to specify interactively several types of batch application programs. JSP is effectively used in the process of specification and generation of programs, but the user should be familiar with the JSP method. Complete executable programs are generated from the program specifications without manually-written programming code.

ACKNOWLEDGEMENTS

Acknowledgements are due to the students Dwayne Oomen, Martin Luttik, Frank Pijpers, Nico Keeman, Hans Kok and Jan van Kruistum, who assisted in the design and the implementation of the batch program generation system, and to our colleagues Peter Apers, Henk Blanken and Rob van de Weg for their comments on earlier drafts of this paper.

REFERENCES

1. P. A. Luker and A. Burns, 'Program generators and generation software', *Comput. J.*, **29**, (4), 315–321 (1986).
2. F. A. van Hoeve, 'Design and implementation of a report program generator using abstract data types', in H. J. Schneider (ed.), *Proc. Int. Computer Symposium 1983 on Application Systems Development*, Nürnberg, Teubner, Stuttgart, 1983, pp. 382–401.
3. F. A. van Hoeve and R. Engmann, 'Design and implementation of the TUBA application generation system', *Memorandum INF-85-23*, University of Twente, 1985.
4. R. Engmann and F. A. van Hoeve, 'The data dictionary function of the TUBA system', *Memorandum INF-86-32*, University of Twente, 1986.
5. F. A. van Hoeve and R. Engmann, 'An object-oriented approach to application generation', *Software—Practice and Experience,* **17**, (9), 623–645 (1987).
6. B. M. Leavenworth, 'Non-procedural data processing', *Comput. J.*, **20**, (1), 6–9 (1977).
7. M. Hammer, W. G. Howe, V. J. Kruskal and I. Wladawsky, 'A very high level programming language for data processing applications', *Commun. ACM,* **20**, (11), 832–840 (1977).
8. N. S. Prywes, A. Pnueli and S. Shastry, 'Use of a nonprocedural specification language and associated program generator in software development', *ACM Transactions on Programming Languages and Systems,* **1**, (2), 196–217 (1979).
9. M. A. Jackson, *Principles of Program Design*, Academic Press, London, 1975.
10. J. R. Cameron, *JSP & JSD: The Jackson approach to software development*, IEEE Computer Society Press, 1983.
11. R. Storer, *Practical Program Development using JSP*, Blackwell Scientific Publications, Oxford, 1987.
12. B. Sanden, 'Systems programming with JSP: example — a VDU controller', *Commun. ACM,* **28**, (10), 1059–1067 (1985).
13. S. Cook, 'Languages and object-oriented programming', *Software Engineering J.* **1**, (2), 73–80 (1986).
14. G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug and K. Nygaard, *SIMULA BEGIN*, Studentlitteratur, Lund, 1973.
15. A. D. Wilson, 'Programs to process trees, representing program structures and data structures', *Software—Practice and Experience,* **14**, (9), 807–816 (1984).
16. B. Dwyer, 'One more time — how to update a master file', *Commun. ACM,* **24**, (1), 3–8 (1981).