JKAU: Eng. Sci., **vol. 10 no. 1**, pp. 67-86 (1418 A.H. / 1998 A.D.)

Design and Development of a Program Generator with Natural Language Interface

M. HAMED and M.A. AJIZ
Dept. of Computer Science, College of Science,
University of Bahrain, Isa Town, Bahrain

ABSTRACT. Program Generators are considered a basic tool for the efficient use of computers. In this paper, a design of a program generator with a natural language interface is presented. The main contribution of the paper is to provide the user with a natural language interface for feeding program specifications. Moreover, it gives a design for a meta language to which the program specifications are converted. It also gives the possibility of translating the meta program to one of several target languages (target language is one of the common programming languages).

1. Introduction

Until recently, users without programming skills have to make use of standard packages to do some specific task on computer. This way of doing jobs by nonprogrammers is not the most appropriate way for several reasons. First, packages are usually designed to serve several users (of different requirements), and consequently none of the users will find that the package caters for all his needs. Second, the execution time of a program in the package (as a general-purpose program) will be much more than that of a tailor-made program that fufills the same task. Third packages are usually inflexible in the sense that the user can exercise little control over what they do. Fourth, a nonprogrammer has to learn a lot about how to use the package (usually there is no friendly interface). Program generators with friendly interface[1] seem to be the most appropriate way for nonprogrammers to do tasks on computer. These generators would enable the user to generate programs that solve his own problems in a reasonable execution time. The ultimate goal of applying artificial intelligence techniques to software engineering is automatic programming. In the limit, automatic programming would allow the user to say what is wanted and have a program produced automatically.

Automatic programming can be viewed as consisting of three successive stages : Specification acquisition, Interpretation of specifications and Code generation. Usually, a research project in the field of automatic programming[2] cannot be done for all stages of all domains of knowledge at the same time. Consequently, the problem is cut down either horizontally (working in one stage for broader domains) or vertically (working in all stages for a narrow domain). The first stage is concerned mainly with improving the user interface. The user interface in most currently available program generators is composed of a dialogue module[1] that requires the user to select answers from a menu or respond in simple Yes/No fashion. Horowitz *et al.*[3] give a proposal for an application generator that includes a non-procedural language which is used for definition of data and describing the actions required. This was considered an improvement over the systems mentioned in the paper, like: NOMAD2, FOCUS, DBASE-III. Cheng *et al.*[4] give a description of an experiment done on a prototype system 'MODEL' where a nonprogrammer developed a commercial application subsystem using the available facilities of a program generator with a Very High Level Language (VHLL) interface. Still, recent application generators like DBASE-IV[5] provide the user with a non-procedural VHLL for writing the necessary commands for specific application.

The second stage is concerned mainly with interpreting the user specifications to tangible algorithms to solve the problem.

In the third stage the generated algorithms are translated to a language acceptable by the computer. Naturally, when the user supplies the system with commands (e.g., in a non-procedural VHLL, stages two and three may be combined together in one stage whose output is the generated program).

It should be noted that the current trend in designing the user interface[1,6,7] is to use a natural language dialogue for feeding the specifications.

In this paper, a design of a program generator (PG) with a natural language interface is given. The program specifications are fed to the system through a natural language dialogue. The program specifications obtained from the dialogue are stored internally in a symbolic language known as program specifications language. The program specifications are then translated into a metalanguage[8] which is used specifically for writing the necessary algorithms. Finally, the metaprogram is translated into one of the available target languages (BASIC, PASCAL, FORTRAN or COBOL).

The proposed design includes many elegant features. Some of the features concerning the natural language dialogue are: (1) Helping the user to correct mis-typed key words, (2) Trapping and pointing out inconsistencies, (3) Enabling the user to ask for help at any point in the specification phase, (4) Ability of backtracking in order to correct any previous response, (5) Ensuring complete independence between the various components of the program generator, (6) Enabling the user to select one of several target languages.

In Section 2 of the paper, a general outline of the components and their functions is given. In Sections 2.2 and 2.3 the program specifications and algorithm description languages are described.

In Section 3 the general design principles are presented. This includes a description of the natural language processor and other features.

In Section 4, a case study is given to illustrate the mechanism of the system starting from the first stage of feeding the specifications.

In Section 5, a brief description of the current developments status is given.

2. Overview of The System

In this section, the components of the program generator are described together with both the program specification language and the algorithm description language.

2.1 Components of the Program Generator

The Program Generator consists of three main components: User Interface, Algorithm Formulation, and Code Generation.

2.1.1 User Interface

The user interface is composed of the following units: Interface Controller, Dialogue Module (Natural Language Processor, Menu-Driven Analyzer, Commands Analyzer), Inconsistency Check and an Editor. The input of the user interface is the user requirements, and the output is a program specification file. The editor can be used to change any information of a previously stored specification. The structure of the user interface is given in Fig. 1. The Help Text gives information to the user about various topics such as the syntax of commands, meaning of options of a menu, syntax and semantics of natural language sentences. Cases of inconsistency check are explained later in Section 3.2.1. Other subcomponents of the user interface are described below.

I. Interface Controller. The interface controller is the first part that is invoked. It starts by interrogating for the existence of an old program specification to be edited. It then asks the user to select one of the three available modes of conversation (Natural Language Dialogue, Menu Driven Choices, and Typing Commands), and accordingly the unit corresponding to the user respond is invoked. Return to the interface controller from the dialogue module is done in three cases: End of specification, check for inconsistency, and occurrence of some types of errors.

II. Natural Language Processor. The Natural Language Processor consists of the following modules: Dialogue Manager, Natural Language Analyzer, Translator, and Natural Language Generator. The unit operates according to the following mechanism:

a) A Sentence is entered by the user.

b) The Natural Language Analyzer checks the sentence both syntactically and semantically (Semantics are used to guide the syntactic parser).

c) The Translator translates the sentence to an internal form.

d) The sentence is checked to be consistent with the available information.

e) According to the inference rules, the Natural Language Generator produces an output sentence.

It should be pointed out that the described mechanism is controlled by the dialogue manger.

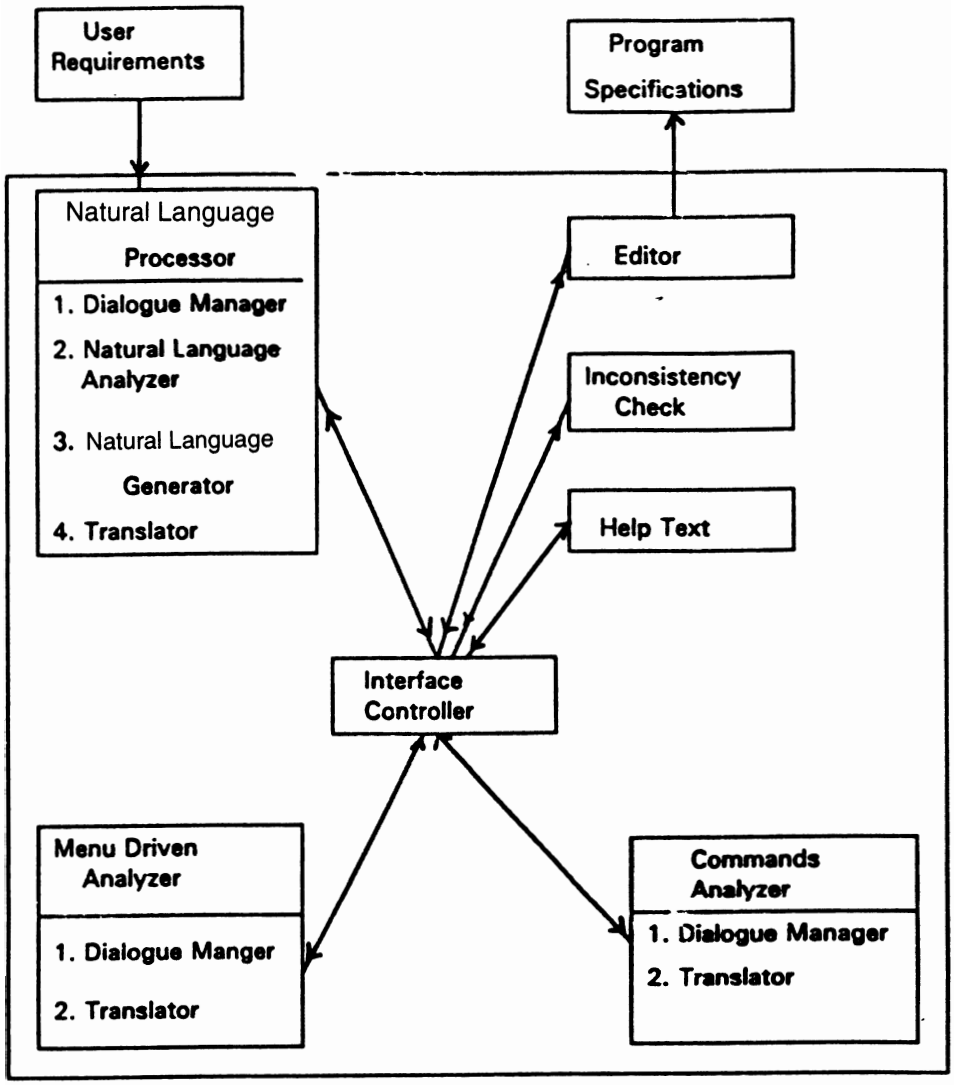

FIG. 1. Structure of the user interface.

III. Menu-Driven Analyzer. The Menu-Driven Analyzer consists of two modules: Dialogue Manager and Translator. Menus (or Yes/No questions) are output to the user in a prescribed sequence and the user response is checked to be one of the possible answers and to be consistent with the available information. Then it is translated to an internal form.

IV. Commands Analyzer. The Commands Analyzer consists also of Dialogue Manager and Translator. As a command is input, it is analyzed syntactically and checked for inconsistency. Errors in erroneous commands are pointed out while correct commands are translated into an internal form.

The output of the user interface component is a file of program specifications (in an internal form) together with the original user's dialogue. The program specifications should include:

a) Description of data structures, files, records, storage units, intermediate areas, ... etc.

b) Shapes of reports and screens.

c) Processing statements.

2.1.2. Algorithm Formulation

The Structure of the Algorithm Formulation component is given in Fig. 2. It accepts the program specifications as input and produces a program in metalanguage as output. It consists of two units: Algorithms Description unit and a Translator. Algorithms descriptions are used to expand some program specification instructions to several metalanguage instructions. Algorithms obtained from the Algorithms Description unit contain some comments which are modified (if necessary) to suit the problem in hand. Algorithms in this unit are usually supplied in the form of procedures that use dummy names (variables, files, ... etc.). Dummy names are replaced by actual names at time of preparation. Examples of such procedures are: A procedure to sort an array of n elements, a procedure to match two files (sequential access), ... etc. Under certain conditions, any procedure may be invoked at any point in the main program or inside another procedure. Arithmetic operations and conditions are interpreted directly from user requirements. More algorithms can be fed to the unit whenever appropriate.

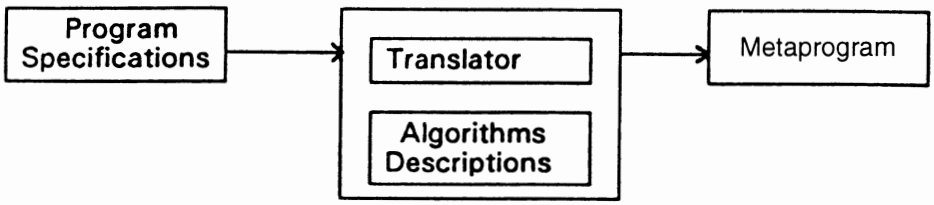

FIG. 2. Structure of algorithm formulation.

2.1.3. Code Generation

The structure of the Code Generation component is given in Fig. 3. It accepts the metaprogram as input and produces a target program and a user guide as output. The translator operates on the basis of the syntactic rules of the target language and features of the machine. Several target languages are made available to the user. The features of the target language (BASIC, COBOL, FORTRAN, PASCAL) may differ from one machine to the other according to the amount of deviation from the standard specifications of the language. The techniques applied in translation are similar to the source-to-source transformation techniques found in Reference [9]. The user guide includes a natural language description of the algorithms, which is obtained from the algorithms description unit, (that is why it is added in Fig. 3) and a copy of the metalanguage program.

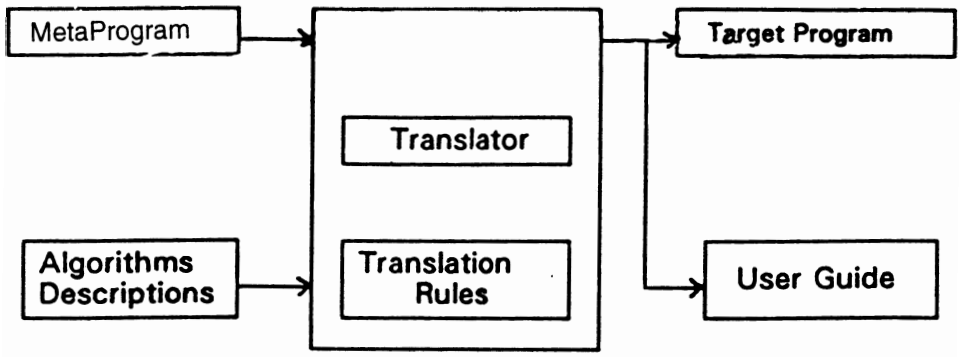

FIG. 3. Structure of code generation.

2.2 The Program Specification Language

The program specification language is used to represent the program specifications in an internal form. This is a kind of symbolic language that includes key words and predefined symbols to which the result of the dialogue is interpreted. The syntax of this language is given in Appendix A.1. The program specifications (as given in this language) consists of the following parts:

i) Function Information

This part includes a combination of key words, where each key word has a specific meaning. The set of key words includes: { PRINT, UPDATE, CREATE, MAINTAIN, COMPUTE, GENERATE, PRODUCE, CALCULATE, SORT, MERGE, INPUT, READ, SEARCH, DELETE, CHANGE, MODIFY, STORE, KEEP. OUTPUT, WRITE, REWRITE, SAVE, DISPLAY, FIND, MATCH }. If we have n key words, then there are 2^n different combinations.

ii) Files Information

This part includes information about files used in this program. Examples of such information are: File name, operations to be done on the file, name of file description segment, ... etc.

iii) Report Information

This part includes information about the format of the required report, for example titles to be printed, fields to be printed, totals, ... etc.

iv) Screen Information

This part includes information about the format of screen which is used in the program. This may involve: Titles, fields, Mode (Input/Output), ... etc.

v) Calculations and Extra Variables

This part includes information about the calculations to be added at the specific points of the program. The execution of a calculations block may depend on the satis-

faction of some condition. The names and other attributes of extra variables are quoted and kept for later use.

An example of a typical program specification is given in Section 4.2.

2.3) The Algorithm Description Language

This language[8] is designed mainly for writing algorithms. The features of the language are similar to those of common high-level programming languages. The syntax of the language is given in Appendix A.2. Its statements are selected such that there is almostly one-to-one correspondence with the statements of the target languages.

An example of a typical algorithm is given in Section 4.3.

3. Design Principles

This section includes the principles upon which the design is based. These principles are related to: Selection of natural language sentences, completing the specifications, and functionality of the components.

3.1. Natural Language Sentences

In the early stages of Natural Language Processing (NLP) mechanisms, no concern was made as to the meaning in a phrase. Key words or word sequences were used to extract information from the phrase to be analyzed. In a subsequent stage, knowledge about the domain were explicitly encoded. It is asserted that information in texts[10] can be recovered through natural language analysis by building and reasoning on a model of the situation described, when both linguistic and detailed world knowledge are provided to the system.

Natural language understanding[11] is characterized as a constraint-based process which constructs both syntactic and semantic interpretations in parallel. Syntactic interpretations can be represented in the form of a parse tree, while semantic interpretations are represented in the form of network consistency graph[11-13], which is a form of semantic network (known also as knowledge graph). Transition networks[14,15] are used as a tool for syntactic analysis. The sentence generator[16] is divided into several modules (e.g., planning and realization), with control and information passing between the modules during the generation process. To generate natural language sentences[16], many kinds of decisions should be made. Researchers are searching for a proper order of these decisions.

When designing the specifications of a natural language, many factors should be considered, among which[17] are: Development time and maintainability.

A fairly general type of sentence is considered[18-20] for conducting the dialogue with the user. The structure of the sentence is given in Fig. 4, while the notations are explained in Table 1. For the operation of the module, a set of domain values[10] is selected for each component of the sentence. When a sentence is input, it is analyzed syntactically and semantically, then the inference rules are used to: (1) Decide the output sentence, and (2) Add information to the program specifications.

<S> ::= <VP> <NP> | [<ADV>] <NP> [<AUX>] <VP> [<ADV>]|
<COM> <S>
<NP> ::= [<DET>] [<ADJ>] <N> [<PP>] | [<N>] <S>
<VP> ::= <V> <NP> [<PP>]
<PP> ::= <PRP> <NP>
<Q> ::= <AUX> <NP> [<AUX>] <VP> | <INT> <V> <NP>

FIG. 4. Structure of natural language sentence.

TABLE 1. Notations and their meanings.

Symbol	Meaning
ADJ	Adjective
ADV	Adverb
AUX	Auxiliary
COM	Complementizer
DET	Determiner
INT	Interrogator
N	Noun
NP	Noun Phrase
PP	Prepositional Phrase
PRP	Preposition
Q	Question
S	Sentence
V	Verb
VP	Verb Phrase

3.2 Completing and Correcting the Specifications

To complete the specifications, some checks have to be made. These checks are discussed in Section 3.2.1. Also, correction of key words is discussed in Section 3.2.2.

3.2.1. Check for Inconsistency

Inconsistency may take several forms: (1) Wrong name of stored information on disk, (2) Insufficient information to generate the program specifications, (3) Operations inconsistency, such as a variable being used twice successively in the left hand side of two assignment statements without being used in any right hand side of a statement.

3.2.2. Correcting Key Words

If an extraneous word is detected in the position of a key word in the sentence, its corresponding phonetic code is obtained and compared to possible codes of eligible key words and the user is asked to confirm a choice of the correct word.

3.3 Functionality of the Components

In this section, an explanation of the proper functioning of the components is given. Facilities required for the proper functioning are also explained.

3.3.1. The Natural Language Analyzer

The natural language analyzer work is based on the following built-it components and design principles:

1) A dictionary of nouns, verbs, adjectives, ... etc.

2) Relationships between a word in a category and other words in different categories.

3) Actions to be taken to comprehend a full sentence. The actions involve the process of producing questions about terms related to key words; something which may clarify the meaning of the sentence.

4) A set of attributes to each word. Information should be supplied about these attributes.

5) The help text is used to explain to the user the correct way of responding to a question, specially when the user fails to respond properly after several trials.

6) A facility to differentiate between plural and singular modes.

7) Points about which information is to be gathered, are arranged in the form of a hierarchical structure. A control mechanism is responsible for the transfer from one point to the next, in the same or different levels. It should be pointed out that Augmented Transition Networks (ATN) are used to analyze a sentence. The well-known recursive- descent technique is applied, where routines to analyze the sentence semantically are tied up with those used to analyze the sentence syntactically.

3.3.2. Algorithm Formulation Unit

The work of this unit depends on the information output from the dialogue module as program specifications. According to the given combination of key words (describing the function of the program) and other information about files, the unit selects the main body of the program to be one of the existing internal modules. Selection conditions for records of files and for printing, are added at the appropriate points in the module. Other internal modules are added to the main module at the correct points which are determined by the unit itself. Also, external statements (modules) are added to the main module at the points determined by the unit.

3.4. Limitations

The limitations on the overall design include :

1) Natural language sentences are limited to specific structure and specific domain, 2) The pool of algorithms in the algorithms description units is limited, and consequently some user problems may not be solved until necessary algorithms are added, 3) The set of target languages is limited and future extensions may seem necessary, 4) The metalanguage supports only common features found in most languages, extra features (like pointers and variant records) are not supported.

Other limitations include the definition of a maximum to the following: 1) Number of times for the user to give a syntactically correct sentence, 2) Number of times for the user to give explanation to a given sentence, 3) Number of functions of a program, 4) Number of files included in a program. Also, among other limitations are: 1) No many-to-many relationship between files is allowed, 2) The depth of nesting in complex structures of the algorithm description language is limited.

4. A Case Study

In this section, a case study is considered. This might help in illustrating the way in which the program generator works. The case includes: A portion of a dialogue, program specifications, and the resulting metaprogram. Obviously, it would require a huge space to present the complete case.

The main function of the considered program is to print the transactions of accounts.

4.1. The Dialogue

The dialogue involves the following steps:

1) Interrogate for an old specification to be edited.

2) If an old specification file exists, then use it to display previous responses and ask the user for possible modifications; otherwise continue.

3) Ask the user to select one of the different modes of conversation and consequently invoke the corresponding module. Assuming that the user selects the natural language dialogue mode, then continue.

4) Generate a message to ask for a name to be assigned to the specifications file that will be generated.

5) Ask for the main function of the program (PRINT/UPDATE/CREATE/MAINTAIN/GENERATE/PRODUCE/CALCULATE/COMPUTE/SORT/MERGE/ ... etc.). This will help in defining the necessary algorithms.

6) Ask for data files to be included in the program (NAME, NEW/OLD, DESCRIPTION, READ/WRITE, ACCESS mode, KEY ... etc.).

7) Ask for screen formats to be included in the program (NAMES of FIELDS, TYPES of fields (I/O/I-O), Accompanying TITLES, POSITIONS of fields and titles (row and column) ... etc.).

8) Ask for the kind of report (DETAIL/TOTAL/SUMMARY) and its contents (FIELDS, TITLES, BREAKS, TOTALS, POSITIONS of fields and titles ... etc.

9) Ask for additional conditions and corresponding actions.

10) Ask for additional operations (Computations, Matching, ... etc.).

Figure 5 gives the first portion of a typical dialogue for the above-mentioned problem. It should be noted that a generated statement starting by an asterisk does not follow the prescribed syntax rules, and is considered as a comment. Also, when a definite response (one of different alternatives) is required from the user, the input sentence may not be analyzed according to the prescribed syntax. In Fig. 5, the user response is given in bold face such that it can be easily recognized.

```
PG    * Hello, this conversation is intended for defining your
      * problem. The first step is to decide whether there is
      * any previous specifications to this problem (YES/NO)
User  NO
PG    * Select one of the available modes of conversation (type
      * its number)
        1. Natural Language Dialogue
        2. Menu-Driven Choices
        3. Typing Commands
```

User : 1
PG : Assign a name to the file that keeps the specifications of the problem
* (Maximum 8 characters)
User : ACCTPR
PG : What is the main function of the program ?
User : THE MAIN FUNCTION IS TO PRNT THE TRANSACTIONS OF
ACCOUNTS IN THE CURRENT MONTH.
PG : I think you mean --* 'PRINT'. Confirm (YES/NO)
User : YES
PG : Do you use files in the program? --* If yes, give their names enclosed between quotation marks.
User : The NAMES ARE: 'ACC.MST' AND 'ACC.TRN'.
PG : Are they old files?
User : YES, THEY ARE OLD.
PG : These names do not exist on disk. Type the correct names.
User : 'ACC.MAST' AND 'ACC.TRAN'
PG : Did you store the descriptions of files on disk ? --* If yes, give their names
enclosed between quotation marks.
User : 'ACC.DM' AND 'ACC.DR'
--* In most printing programs, the files are used for READ only. Is this true in your case ?
User : YES, IT IS TRUE.

FIG. 5. A portion of a dialogue.

The layout of the required report, obtained by the dialogue is given in Fig. 6 below.

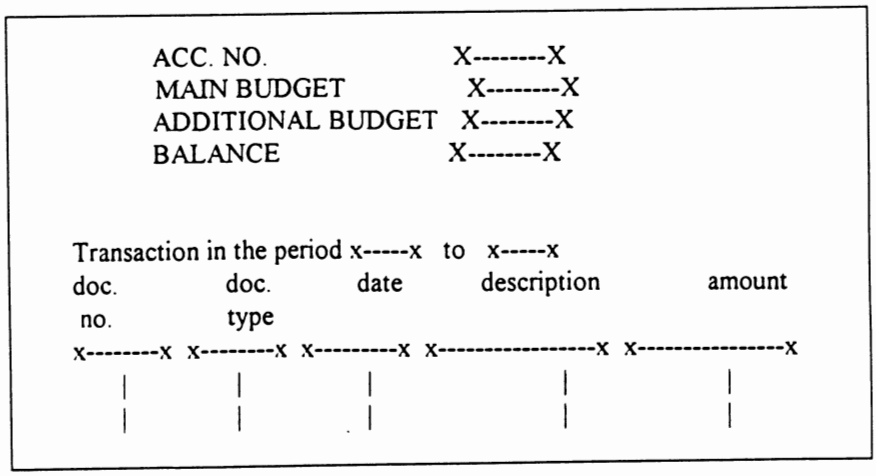

FIG. 6. Layout of the required report.

4.2. The Resulting Program Specifications

In this subsection, the resulting program specifications (in symbolic language) are given. Titles for each part of the program specifications are added to make it clear.

1) Function Information

Function is::=PRINT: MATCH;

2) Files Information

```
Files are ::= ACC.MAST: PRIMARY; ACC.DM: SEQUENTIAL: READ;
              ACC. TRAN: AUXILIARY: ACC. DR: SEQUENTIAL: READ $
              F2.4 > S2.1 and DATE < S2.2;
              ONE-MANY: ACCNO, ACCNO;
```

3) Report Information

```
L1   :    'ACC. NO' $ 5, F1. 1 $ 30: 1: Cl
L2   :    "MAIN BUDGET' $ 5, Fl.2 $ 30: 1
L3   :    'ADDITIONAL BUDGET' $ 5, Fl.3 $ 30: 1
L4   :    "BALANCE' $ 5, Fl. 4 $ 30: 1
L5   :    'Transactions in the period' $10, S2. 1 $ 40, 'to' $ 45, S2.2 $ 55: 1
L6   :    'doc. doc. date description amount' $ 5: 1
L7   :    'no. type' $ 5: 1
L8   :    F2.2 $ 5, F2.3 $ 12, F2.4 $ 20, F2.5 $ 30, F2.6 $ 60 :
M    :    C2
T    :    F2.6: Cl, EOP, EOJ
```

4) Screen Information

```
S1   : 0 : B;
       L1    :'start date' $ 5
       L2    :'end date' $ 5
S2   : I:   B;

       L1    :S2.1 $ 20
       L2    :S2.2 $ 20
```

4.3 The Formulated Algorithms

According to the information obtained from the specifications, an internal module is selected as the body of the program. In our case, the information that guides the selection process can be described in the following terms: The function of the program is to read, match, and print the records of two files. The files are accessed sequentially and matched on a specific field where the relationship between the records of the two files is one-to-many. Some lines are to be printed when a record from either file changes. At the beginning of the program, two variables are to be input from screen. Also, the selection of records from the second file is based upon some given condition.

Accordingly, the algorithm formulation component selects an internal routine named 'MATCH-S' to be the body of the algorithm, and adds to it:

a) A routine to accept variables from screen.

b) Necessary routines for printing the information from files, and for summing and printing totals. The generated algorithm is given in Fig. 7.

```
* SW is a boolean variable to indicate the need for reading
* A record from AUXILIARY file
* EOF is a boolean variable to indicate end of file
* TOT-1 to TOT-3 are areas for totals
* EOP is a boolean variable to indicate end of page
Set SW to 0
Read ACC.MAST file record
Repeat While not EOF ( ACC.MAST )
        Call PRINT-1;
        If SW = 0 and not EOF ( ACC.TRAN ) Then
                Read ACC.TRAN file record; Set SW to 1;
        Repeat While not EOF (ACC.TRAN) and MAST.ACNO =TRAN.ACNO
                If DATE > MIN and DATE < MAX Then Call PRINT-2
                and Add AMOUNT to TOT-1, TOT-2, TOT-3
                If EOP Then Call PRINT-3-A
                Read ACC.TRAN file record
        End-Repeat
        If not EOF (ACC.TRAN) or SW = 0 Then Call PRINT-3-A;
        Call PRINT-3-B; Set SW to 1
        Read ACC.MAST file record
End-Repeat
Call PRINT-3-C
Stop.
```

FIG. 7. The generated algorithm.

5. Development Status

The following is a list of modules that are implemented and tested successfully: (1) Dialogue Module (Natural Language Processor, Menu-Driven Analyzer, Commands Analyzer), (2) Algorithm Formulation, (3) Some parts of the code generation module. In this limited space, we will give a brief description about each module.

5.1. Development of Dialogue Module

We will concentrate in this part on the implementation of the natural language processor as the implementations of Menu-Driven Analyzer and Commands Analyzer are direct.

The information about the program to be generated is arranged in the form of a tree. Each node of a tree contains information about a specific topic besides to the information needed for tree traversal. The information about a topic in a node includes: Identification of the topic, indicator to whether information about it has already been

provided or not, set of key words related to topic, which question to be asked to obtain missing information. The tree is traversed in a Depth-First form such that when information about a topic in specific node is complete, we move to its sibling node (if its information is not complete), otherwise we move to parent. Linked allocation was found appropriate for implementing such a tree.

To analyze natural language sentences, we first start by breaking down the sentences to components. Each component is checked to determine its category (Interrogator, Verb, Noun, ... etc.). The structure of sentences and semantics are used to guide the search for the category of the next component. To capture the semantics in the sentence, a dictionary is used. The dictionary is constructed in such a way that each component mainly belongs to one category and may be related to other components of different categories. A relationship of several components (of different categories) provides the system with some specific information.

To generate a natural language sentence, we start by a key word around which the sentence will be built. We use the relationships of this key word with other words of different categories as guide and select the words surrounding the keyword according to the structure of sentence and the required meaning.

5.2. Development of Algorithm Formulation Model

The algorithm description unit has been prepared in such a way to include procedures written in metalanguage. The procedures are selected to cover various situations of file handling, screen operations, printer operations, ... etc. For each procedure, there is a natural language description that describes to the user the functions included in the procedure. Keywords of the program specifications (given in a consize form) are used to decide: (1) The main body of the output program, (2) What are the additional procedures and where to plug them in the main body. The main effort with this module was the selection of procedures for different situations and the coding of these procedures in metalanguage.

5.3. Development of Code Generation Modules

The development of this module can be easily broken down to several parts, each one is concerned with a specific target language. For the target language PASCAL, it is not difficult to translate from metalanguage to it, since there is mainly one-to-one correspondence between statements in the two languages. The only difficulty we found was when there was no one-to-one correspondence between the two languages (e.g., using strings in the metalanguages and its counter part is an array of characters in PASCAL). Also, it should be noted that features which are not common to most languages are avoided (e.g., variant records, pointers, complex data structures).

6. Conclusion

The design of a program generator with natural language interface was presented. The natural language processor was limited to a specific domain which is needed to feed the program specifications. Sentences are analyzed syntactically and semantically. Inference rules are used to decide the appropriate actions when an input sentence is

entered. Other modes of conversation are also made available. A specific metalanguage was used for formulating the program. Several target languages for the program are made available to users.

The given design is realistic, and is characterized by the modularity of the necessary software (divided to parts and subparts). In practice, many modules are developed on VAX-8550 machine and are tested successfully on sample cases. Expansion to the completed software is being carried out in order to cover all features given in the proposed design. This includes the module of translating to target language.

References

[1] **Luker, P.A.** and **Burns, A.,** Program generators and generation software, *Computer J.*, **29**(4): 315-321 (1986).

[2] **Rich, C.** and **Waters, R.C.** (ed.), *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, USA (1986).

[3] **Horowitz, A., Kemper, A.** and **Narasimhan, B.,** A survey of application generators, *IEEE Software*, **11** (January) : 40-54 (1985).

[4] **Cheng, T.T., Lock, E.D.** and **Prewes, N.S.,** Use of very high level languages and program generation by management professionals, *IEEE Trans. on Softw. Eng.*, **SE-10**(5) (September): 552-563 (1984).

[5] *DBASE-IV: Setting the Data Management Standard*, Reference Manual, Ashton Tate, Los Angeles (1988).

[6] **Toledo, S.W.,** The value of natural language capability in the computer, *Behavioral Science Journal*, **37**(4): 294-309 (1992).

[7] **Brunner, H., Whittemore, G., Ferrara, K.** and **Hsu, J.,** An assessment of written the interactive dialogue for information retrieval applications, *Human-Computer Interactions*, **7**(2): 197-249 (1992).

[8] **Hamed, M.,** *Implementing a Program Generator for Business Applications*, Tech. Report-CS21, Math. Department, University of Bahrain, pp. 26-48 (1989).

[9] **Kelsey, R.** and **Hudak, P.,** Realistic compilation by program transformation, *Proc. of the Sixteenth Annual ACM Conference on Principles of Programming Languages, Austin, TX, USA, 11-13 Jan.*, pp. 281-292 (1989).

[10] **Cavazza, M.** and **Zweigenbaum, P.,** Extracting implicit information from free text technical reports', *Information Processing & Management Journal*, **28**(5): 609-618 (1992).

[11] **Kuttner, E., Havens, W.** and **Cercone, N.,** Processing natural language with schema constraint networks, *Computers & Mathematics with Applications Journal*, **24**(11): 3-10 (1992).

[12] **James, P.,** Knowledge Graphs, *Proceedings of the Conf. on Linguistic Instruments in Knowledge Engineering, Tilburg, Netherlands*, pp. 97-117 (1991).

[13] **Sowa, J.F.** and **Way, E.C.,** Implementing a semantic interpreter using conceptual graphs, *IBM J. Res. Develop.*, **30**(1): 57-69 (1986).

[14] **Bonnet, A.,** Artificial Intelligence: Promise and Performance, Prentice-Hall, USA, pp. 72-96 (1985).

[15] **Kong, P.H., Show, G.Y.** and **Lin, W.K.,** Evaluation of parsing techniques for natural language processing, *Proceedings of the Intern. Conf. on Information Engineering, ICIE '91, 2-5 December (1991), Singapore, pp. 422-431.*

[16] **Inui, K., Tokunaga, T.** and **Tanaka, H.,** Text revision: A model and its implementation, *6th Intern. Conf. on 'Aspects of Automated Natural Language Generation, 5-7 April (1992) Trento, Italy*, pp. 215-230.

[17] **Burton, A.** and **Steward, A.P.,** A natural language interface to management information, *Intern. Conf. on Practical Applications of Prolog, 1-3 April (1992) London, UK*, pp. 1-23.

[18] **Akmajian, A.** and **Heny, F.,** *An Introduction to the Principles of Transformational Syntax*, MIT Press, USA, pp. 154-182 (1978).

[19] **Gaines, B.R.,** The technology of interaction dialogue programming rules, *Intern. J. of Man-Machine Studies*, **14**(1): 133-150 (1981).

[20] **Kaplan, J.S.,** Cooperative response from a portable natural language query system, *Artificial Intelligence*, **19**(2): 165-188 (1982).

Appendix A: Syntax of Proposed Languages

A.1. Syntax of Program Specification Language

< progr spec > :: = < func inf > < ffles inf > < report > < scr inf >
< variables > < calc > < int modules > < ext. modules > < seq >
< func inf > :: = < main func > : < aux >
< main func > :: = < key func >
<key func> :: = PRINT | UPDATE | SORT | .....
< aux > :: = < awc func > | < aux func > , < aux >
< aux func > :: = < key func >
< files inf > :: = < info >; < relation >
<info> ::= <file> | <file> <info>
< file > :: = < file name >: < type >: < descr name >: < access inf >: < file oper >|< nil >
< type > :: = PRIMARY | AUXILIARY
< file name > :: = < word >
< descr name > :: = < word >
< access inf > :: = < access word > [$ < method > $ < hash >]
<accessword> ::= DIRECT | RANDOM | SEQUENTLAL
< method > :: = KEY | RECORD NUMBER
< hash > :: = < calc > | < word >
< file oper > : : = < oper > | < oper >, < file oper >
<oper> ::= <op>$<sel>

<op> ::= READ | WRITE | DELETE | REWRITE
<sel> ::= <cond> | <nil>

< relation > :: = < rel >: < match fld >
<rel> ::= ONE-ONE | ONE-MANY
< match fld > : : = < name >, < name >
< scr inf > :: = < scr no > : < scr type > < scr cond >;: < scr lines >
<scrtype> ::= I | O
< scr lines > :: = < line > | < line > < scr lines >
<line> ::= L <lineno>: <varinf>
<scrcond> ::= B | C <file no>
<scrno> :: = S <integer>
< report > :: = < report inf > < total inf >
< report inf > :: = < line inf > | < line inf > < report inf >
< line inf > :: = < line no > : < var inf > : < rep > : [< new page >]
< var inf > :: = < field inf > | < field inf > , < var inf >

< field inf > :: = < literal > $ < col no > | <field no>$<col no>
< field no > :: = < file no > . < integer > | <integer>
< col no > :: = < integer >
< literal > :: = ' < string > '

< line no > :: = < integer >
<rep> ::= <integer> | M
< variables > : : = < var field > | < var field >, < variables >
<varfield> ::= <serial>$<varname>$<type>$<size>$<dp>
< serial > :: = < integer >
< var name > :: = < word >
<type > :: = c | N | M
< size > :: = < integer >
< dp > :: = < integer >
< total inf > :: = T: < field no >: < levels >
< levels > :: = < level > | < level > , < levels >
<level> ::= C <fileno> | EOP | EOJ
< new page > :: = c < file no >
< calc > :: = < assign > | < assign > < calc >
< assign > :: = < identifier > : = < expression >
< int modules > :: = < name > | < name > : < int modules >
< ext modules > : : = < module > | < module > < ext modules >
< module > :: = < module name > $ < statements >
< statements > :: = < statement > | < statement > $ < statements >
< statement > :: = < if stat > | < assign >
< if stat > :: = IF < cond > THEN < calc >
< seq > :: = < name > | < name > : < seq >
< name > :: = < word>
< word > :: = < letter > { < char > }
< char > :: = < letter > | < digit > | < spec >
< letter> :: = A | B | ... | Z
<digit> ::= 0 | 1 | ·-- | 9
<spec> = + | - | (|) | ...
< string > :: = < char > | < char > < string >
< cond > :: = < simple cond > | < simple cond > < log oper > < cond >
< simple cond > :: = < expr > < rel oper > < expr >
<rel oper := < | > | = | ~ = | > = | < >
<logoper> ::= AND | OR | NOT
< integer > :: = < digit > | < digit > < integer >

A.2. Syntax of Algorithm Description Language

< program > :: = < statement > | < statement > < program >
< statement > :: = < simple-stat > | < simple-stat > < statement >
< simple-stat > :: = [< label > :] { < com-stat >
< com-stat > :: = * < string >
< string > :: = < character > | < character > < string >
< ex-stat > :: = < set-atat > | < read-stat > | < repeat-stat >

< call-stat > | < write-stat > | < if-stat > | < goto-stat >
< set-stat > :: = SEI < identi~ler > { TO } < expression >
<identifier> ::= [<name>.] <name> [(<sub-list>)]
< sub-list > :: = < sub > | < sub > , < sub-list >
< sub > :: = < expression >
< name > :: = < letter > | < letter > { < let-dig > }
< let-dig > :: = < letter > | < digit >
< expression > :: = < exp > | < log-exp >
< log-exp > :: = < exp > < rel-oper > < exp > | < log-var > | [NOT] < log-exp >
< log-exp > < log-oper > < log-exp >
< log-var > :: = < identifier > | < log-func >
< log-func > :: = EOF [(< file-name >)] | EOP
< exp > :: = < arithm-exp > | < char-exp >
< char-exp > :: = < prime > | < prime > + < prime >
< arithm-exp > :: = < sgn-term > | < sgn-tenn > { < add-op > < term >}
< sgn-term > :: = [+ | -] < term >
<term> ::= <factor> | <factor> {<mult-op> <factor>}
< factor > :: = < prime > | < prime > { < exp-op> < prime > }
< prime > :: = < constant > | < identifier > | (< expression >)
< read-stat > :: = READ < variables > | READ < file-par >
< variables > :: = (< id-list >)
< id-list > :: = < identifier > | < identifier > , < id-list >
< file-par > :: = < file-name > [RECORD]
< file-name > :: = < name >
<write-stat> ::= WRITE <variables> | WRITE <file-par> |
WRITE L < line-no >
< line-no > :: = < int >
< repeat-stat > :: = < repeat-until > | < repeat-while >
< repeat-until> ::= REPEAT UNTIL < log-exp > statement > END-REPEAT
< repeat-while > :: = REPEAT WHILE < log-exp > < statement > END-REPEAT
< if-stat > :: = IF < log-exp > THEN < statement > [ELSE < statement >]
ENDIF
< goto-stat > :: = GOTO < label >
< label > :: = < name >
< call-stat > :: = CALL < name > [< arg >]
<arg> ::= <par> | <par>, <arg>
< par > :: = < constant > | < identifier >
< character > :: = < letter > | < digit > | < spec >
< constant > :: = < num > I < char > I < log >
< num > :: = < sgn-int > [. < int >]
< sgn-int > :: = [+ | -] < int >
< int > :: = < digit > | < digit > < int >
< char > :: = " < string > "
< log > :: = .TRUE. | .FALSE.

< letter > :: = A | B | . . . | Z
<digit> ::=0|1 |... |9
<spec> :: = + | - | * | / | & | $ | ·--
<add-op> ::= + | -
<mult-op> ::= * | /
< exp-op > :: ^
<log-oper> ::= ANR | OR
<rel-oper> ::= < | > | = | < = | > = | < >

تصميم وتطـوير مـولد برامـج بواجهـة تستعمـل أحـد اللغـات الحيـة

محمد مصطفى حامد و منظور علي عاجز
قسم علوم الحاسب الآلي ، كلية العلوم ، جامعة البحرين
مدينـة عيسى - البحـرين

المستخلص . تعتبر مولدات البرامج أحد الأدوات الأساسية للاستعمال ذي الكفاءة العالية للحاسبات . يقدم هذا البحث تصميمًا لمولد برامج ذي واجهة تستعمل أحد اللغات الحية . وتعتبر المساهمة الرئيسة للبحث هي إمداد مستعمل الحاسب بواجهة تمكنه من إمداد الحاسب بمواصفات البرامج بلغة حية . كما يقدم البحث تصميم لغة عليا يتم ترجمة مواصفات البرامج إليها . هذا بالإضافة إلى توضيح إمكانية الترجمة من اللغة السابق ذكرها إلى إحدى لغات البرمجة المتداولة .