Military Technical College,
Kobry El-Kobbah,
Cairo, Egypt

9th International Conference
On Aerospace Sciences &
Aviation Technology

Dynamic Modification of Software Systems (Design a Syntax Directed editing)

Mohy El Deen Horani*

ABSTRACT

The distributed computing systems are characterized by their ability to service applications of distributed origin. Due to the rapid development of digital technology and especially in the field of microprocessors it becomes viable to place a microcomputer in the same place of the planet or device needs to be controlled. In this respect we can say that the microcomputer has replaced the traditional electronic circuits.
So, to control a number of machines or devices or plants it is possible to rely upon a number of (micro)computers distributed according to the requirements of the application on to the points needed to be controlled. This organization depends on a communication network that provides -in addition to communication- the inter-tasking activities between computers. It is necessary for a computer system designed to service such organization to take a grate benefit of the features and abilities of the hardware of these distributed systems.
This paper introduces a study of a software system for distributed computer system for industrial applications.
This study suggests a change to the Module Structure (considering the Module as the main programming block of the system) and also introduce a new design for a Syntax Directed Editing activity which is very effective on the mechanism of carrying out the modification.
What is meant by modification here is the expected modification for the functional activities as well as the topology during the system use. This stems from the consideration that in the case of expensive industrial systems we have to accept the possibility of the future modification of the system structure and the routines of its components .

1. INTRODUCTION

As a result of advanced developments in digital technology we have got the microprocessors that replaced the traditional electronics in, almost, all fields. Continuation of developments in this fields in addition to dropping prices were the motives for replacing industrial electronic control systems by microcomputers.
It became feasible for the computer to carry out the control task in the real time domain, which used to be the role of electronic control systems.

* Assistant Prof., Assad Academy for Military Engineering, Aleppo, Syria

On the other hand the computer provided the possibility of information handling and exchanging with other computers within the system. So, it becomes natural to have a number of microcomputers that may cooperate in controlling an industrial establishment according to the real time requirements. Each computer may have its own task which will be part of the whole task. Network communication is essential for such organization.
Any way this study is directed towards only the software side of the application.
Due to the importance of the flexibility of for such systems, this contribution aims at providing a study for system structure that fulfil the possibility of increasing the flexibility of industrial applications. A previous paper [7] introduced the dynamic modification concept and its effect on the system and its execution.
The ultimate aim of this work is to support implementing high degree of dynamic modification which is indirectly proportional to the modified program component . To get such system realized certain changes to the mechanism of program developments should be carried out in addition to some activities that support this new scheme of program developments.
A new view of the Module structure is introduced in this paper, even its implementation issues are not within the scope of this paper, so that it is divided into two parts, the Interface and the architecture. While the interface part represents the interaction between this module and other modules, the architecture part involves the rest of the module. Of course, implementing such structure and the accompanied mechanism require some special activities for program development.
In this respect the syntax directed editing is considered an essential concept that provide a suitable environments for interactive incremental compilation which is necessary for the implementation of dynamic modification. So this contribution is essentially dedicated for the task of syntax directed editing and the most elaborated part is introducing an implementation for the syntax analyzer as an automaton represented in state table.

The second section discusses the importance of the flexibility in programming systems. The third section, on the other hand, discusses the subject of the application from the point of view of the system structure and the language.
Fifth section covers the study of analysis. Finally the section six gives the conclusion of this study.

I APPLICATION ENVIRONMENT

2 Computer systems and flexibility

It is expected for the Industrial establishments and consequently the computer systems that services this establishments to exist for long period. However, these systems, for many reasons, will not stay without any modification, it should accept and evolve the changes of the application environments which come as a result of new technology.
Actually the presence of the computer in this field is viewed as a motive for changes in the application environments and at the end for the services the system can give.
In addition to the changes in the area of application environments these systems should accept or service the processing changes.

So, system components should be physically located as a response either to changes of the human beings or the establishments. Generally, with respect to distributed computer systems It should accept the modification.
That is the system should be flexible enough to be adaptable with the improvement changes as well as the processing changes.

3 System specifications

The system being thought of here is classified as an Integrated system where the system is build to handle the user program from as early as program editing to last stage , the execution., i.e the system should include all the necessary activities for program developments and processing. So, the system specifications should include:

- The necessary programming language specification for such application.
- The hierarchical structure of the system.

It should be emphasized here that the language and its syntax and consequently the program components to be consistent with the system i.e the program should reflect the structure of problem being dealt with.

3-1 The programming language specifications

The language is studied according to the following two points of view:

1. The compilation view

One of the main issues that is considered in language selection is whether it is a compiled or an interpretive one. The main differences between these two are such as:
Each of the simple statements in the interpretive languages expresses a separate action, i.e it does not depend on other statements.
This kind of structure fulfills the possibility of interpreting (executing) each statement separately and independent on other statements. BASIC and LISP are well known interpretive languages.
This structure issue supports using these languages in system applications that are specialized in receiving and processing commands.
On the other hand, the statements in the compiled languages are more structured and has much wide syntax scope, so that languages are classified as structured languages. So, there are dependencies and links among the statements of the program, so that it is not as easy and direct as the interpretive languages to interpret the statements without knowing the relation with other statements and components of the program.
This structure imposed a basic mechanism to deal with such languages called "Compilation", where the source program is translated to get the object program which is then executed. Certainly the compiled program will be faster than the interpreted one during execution.

2. The view of real time requirements

As the real- time is the area of the application it is necessary to mention the relation between the language and the application area. The characteristics of Real-time languages was studied previously in[5] which are given here in brief:

- Security.
- Readability.
- Flexibility.
- Simplicity.
- Portability.
- Efficiency.

Implementation of these characteristics needs the program to be checked thoroughly before the run-time. This means the necessity of using the compilation rather than interpretation.
Additional requirements should be realized according to the following points:

a) Syntactically

The language should provide some ways to divide the program into smaller parts (MODULES), each of which involves some of the program concepts. The Module should have some programming structures such as, Functions, Monitors, Classes or processes. Also it should have the ability for building the variety of concept types that provide more control on system complexity.

b) Semantically

- It is necessary for the S.W structure to view the system as a group of nodes interconnected by some kind of Network connection.
- It should be possible to build the code within the node (i.e within the target computer that form the node) as a group of concurrent processes that are interconnected with each other through some communication mechanism (within the node).
- The necessity of providing some inter-communication mechanism between nodes.

2. Related developments

There are a variety of Real-time applications each of which was designed according to different view. Also there are a number of distributed systems applications. In this respect we mention Nil [13] and MOD[12], and the ADA programming language which provided the means for a number of distributed system applications.
On the other hand Dynamic modification was supported by a number of programming languages, even not all of these languages are Real-time languages.

- PASCAL language applied the concept of Heap to enable the possibility of dynamic modification for certain type of data structure (Pointers), which is considered as dynamic data type.
- In C language in spite of that the Heap concept is not available, it is possible for the programmer to create what is look like dynamic data type.
- Visual Basic supports the concept of dynamic array that enables the user with the possibility of changing the dimensions of the arrays in the period of run-time.

- On the other hand the CONIC system (developed at Imperial College [1], [2]) - which aimed at supporting the development of distributed dynamic re-locatable control system- provided a system configuration language where the program is composed of a number of Modules. That is possible to add a whole Module to the system. The CONIC Language depends on its structure on PASCAL (a modified version).

3-2 The system structure

The system configuration was discussed in a previous paper [7], where it was argued the necessity of the system to be composed of two parts :

- The Host computer.
- Target computer(s), which should be located close to controlled plants. (Fig 1).

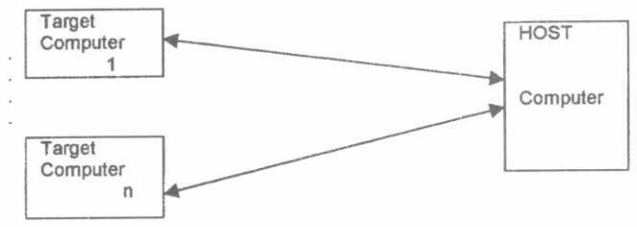


Fig.1 The system configuration

According to this configuration the program development (Editing, Compilation, and producing the object code) should be carried out on the Host computer.
The object code is down loaded into the Target computer (s) that are located closely to the plants.
So, taking into consideration the proposed system configuration as well as the characteristics of the Real time languages we will find that a Language such as ADA or CONIC (which is derived from PASCAL) would be suitable for this kind of application. Conic language can be considered as an example of the languages that may be adopted for this kind of applications.
We aim in this paper at studying the main issues to service the dynamic modification for a distributed computer system, through introducing the syntax analyzer as a Finite State machine and exploiting that in the design of syntax directed editing, we do not specify any specific language as the language of the intended application. Instead we depend on general language items, which is just used for clarifications.

4. The modifications study

It was given in a previous study [3] that it is not easy or direct to carry out the dynamic modifications for any of the program components. However, the CONIC

system provided an application for the dynamic modification in a way to modify the Module as a whole (add or delete).
In this paper we argue that it should be possible to implement the dynamic modification for components of the module. We put the following proposals :

- A structure change of the Module so that the it consists of two parts. The first is the code block of the program which is called "Module Architecture". The other part is the "interface" that defines the information exchange between this Module and other Modules. The following Fig.2 clarifies this sugessted structure.

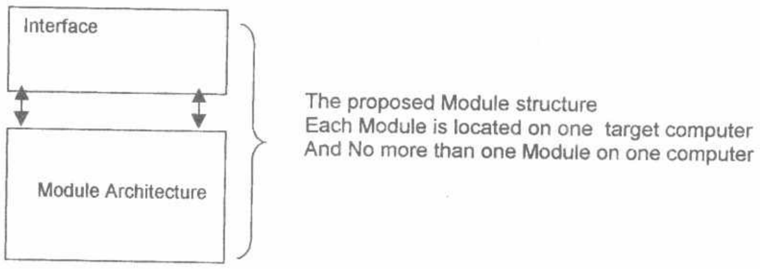

Interface
Module Architecture
The proposed Module structure
Each Module is located on one target computer
And No more than one Module on one computer

Fig.2 A proposed structure for the Module

This newly suggested Module structure is a try for each of the two components to have a (relatively) separate identity so that modifications of either the Module architecture or the Interface can be carried out in the same sense as the CONIC system. So by adopting this new structure the part that can be modified will be smaller than the whole Module.

- The other additional approach for modifying components of the Module is to specify some of the Simple statements which may have least possible relations with the rest of the program. Adding a simple statement that might be of least side effect. All these, of course, within the architecture part.

```
NumberOfKeys:=NumberOfKeys+m;        ←——        additional statement

For I = 1  To  NumberOfKeys  ⎫
Begin                        ⎪
……………………                       ⎬            Old    statement
                             ⎪
End;                         ⎭
```

The execution of this LOOP with this modification will be changed from the point of view of number of times according to the new value of the variable NumberOfKeys.

As it seems, the implementation of dynamic modification of the program at this level will increase the flexibility degree due to the possibility of modifying smaller components than the Whole Module as in CONIC [3].

The CONIC system used to carry out the dynamic modification on the level of Module, i.e to replace a Module with another Module. This includes all the links of information exchange with other Modules (each of which is located on separate computer).
However, with respect to the requirements and mechanism of the suggested dynamic modification it is out of concern this paper. The rest of this paper will concentrate on the design issues of the first stage of program building. What can be said now is that a concept of reserved program memory and reserved data memory would support the implementation of HEAP and STACK which could be used in the process of dynamic modifications. That is the modified program part will be carried out on the form of connecting of memory parts currently not in use (which will hold the object code of the modified part) to the original program memory.

II PROGRAM BUILDING STUDY

5. Program Development Mechanism

The process of program development passes through a number of stages starts by the Editing and ends at execution. However, this process gets more importance in the system that is supposed to accept modification for its data structures and | or for smaller programming components dynamically.
The diagram in Fig.3 introduce the proposed structure view of the early stages of the system. According to this diagram the program file is prepared by a Syntax Directed Editor which should give a syntax error free program.

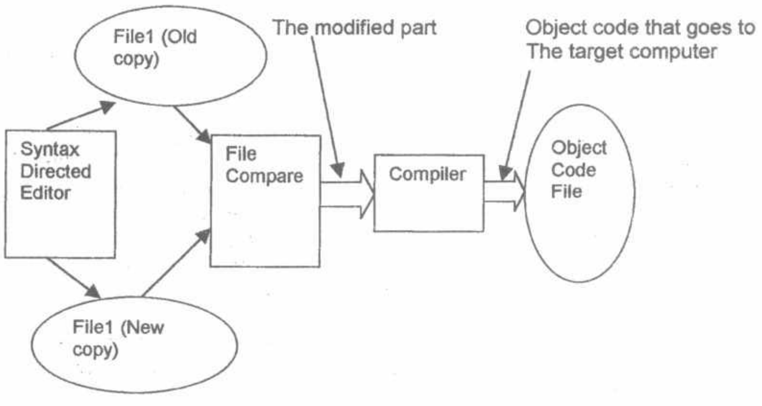


Fig.3 Structure of program developed process

This kind of editors is characterized by its ability to build a file for one language only, i.e it is not a general editor. So, it is possible for this editor to be built according to the syntax of the adopted language. In addition this Editor should be syntax directed because it is necessary for the program parts that will be sent to the compiler to syntactically error free so that the compiler can give the object code. That is we are in need for a certain mechanism that may interact directly with the user. This scheme

of organization supports debugging the program errors as well as responding to their existence.
The developed program file is moved to the Compiler to get the object code which is then down loaded to the Target computer. However the importance of this scheme of organization increases in the case of modification (which is the ultimate aim of our study). In this case the compilation could be carried out for only some part of the code (which was modified).
For this reason the compilation should be designed to be incremental. So the modified part of the code which is going to be compiled is sent to the compiler in a form that specify its location within the program structure (that is in addition that this program should be syntactically error free). So, the modified part of the program has to arrive to the compiler distinctively in order not to re-compile the whole program once more.
For this task to be carried out the two copies of the program (the old one and the new one) are compared by FileCOMPARE utility to get the modified parts of the program. To get benefit of this comparison it is necessary for the compared copies to be syntactically error free. So it is highly important to study the design of a mechanism to support building a syntax directed editor.

5-1 Syntax analysis

In a previous paper [14] we introduced a design of Lexical analyzer according to the theory of Automata. The automaton get the input characters and through its state transitions builds the lexical items which them selves are the components of the program statements.
Here we adopt the same idea in the design study of the syntax analysis mechanism that will be the main part of the syntax directed editor.
The syntax analyzer which is the subject of this study is characterized by:

- Its immediate response for the received tokens.
- Its ability to correct the code.
- So it has to be built to handle the tokens while they are coming.

The syntax analyzer depends for its task on the lexical analyzer. The Lexical analyzer has to be called repeatedly by the syntax analyzer, and in each time it gets a new item which is Lexically error free from the item's grammar points of view. This item is handed to the syntax analyzer in addition to a special value for the item (Lexval) which can be used as an alternative of the token itself. So by specifying a set of values for the lexical items (as a proposition) we can continue the discussion of this design method.
Table.1 gives some examples about the Lexval of some items which the syntax analyzer obtains from the Lexical analyzer. These values will be used in the place of the item it self in our discussion.
A statement is processed by firstly calling the Lexical analyzer for the input character. The Lexical token produced by the lexical analyzer is checked by the syntax analyzer from the point of the grammar rules and according the result of this checking it may be used in building the statement.
Table.3 is a state table for an automaton represents the mechanism of the syntax analyzer for the part of the program block (statements). As the grammar is built according to the golden rule "One symbol look ahead and No back tracking", this

Table.1 The tokens and Lexval

For	1	To	12	:=	23
If	2	Value	13	=	24
Begin	3	Enter	14	)	25
Module	4	Procedure	15	(	26
And	5	Use	16	/*	27
While	6	var	17	Number	28
Then	7	Type	18	String	29
Identifier	8	Constant	19	Array	30
Architecture	9	.	20	else	31
End	10	Do	21	Record	32
Space	11	;	22	Function	33

state table should implement this rule bearing in mind that the symbol here is the whole token.
The input to this table is the lexical items of the lexical analyzer such as: For, And, To, := ,, . Begin End.....etc. In this table the lexval replaced the item itself .
On the other hand the State of the table indicates the stages of receiving statements of the program. Due to the huge size of the real table of the syntax analyzer we will give just part of it as a sample to serve the clarification of this study.
The programming representation of this table is done as a huge two dimensional array. Each of the entrances (cells) of this array is itself a record of two items, as follows:

```
Struct Rec {

     Int    stateNo;// Next state's number
    Char  ind[2];// status indicator for receiving the lexical token and a code of
                          // action that should be carried out.
}
struct Rec syntaxary[n,m]; // the Automaton is represented as a state table
```

The first row of the table represents input symbols which are the tokens came as output of the Lexical analyzer . These tokens (items) are replaced here in this table by a coded value (Lexval), as it was mentioned before. The left most column is the index of the state number. The information within each entrance (cell of the array) represents a guide line for the next step processing. Two components are coded in each cell as shown in the declaration of " Struct Rec " above. The first , which is an integer value, tells about the next state the automaton should go to next in the process of receiving this current statement. On the other hand the second component , which is a string, indicates the additional necessary processing at this stage (actually this string activates certain function). Table.2 gives descriptive information for the values this string may take.
Any way the most illustrative tool for this discussion is the state diagram of receiving the items of the statement.

The following diagram of Fig.4 is a part of whole state diagram. It is the part of the FOR-statement. It should be noted here that the program is written on a statement per line bases.
This state diagram explains the steps of receiving and recognizing of the FOR-statement. This process starts as early as the recognition of the FOR item in state (1) and passes to state (2) if the coming token was the item FOR, with the code of action "0" which is a sign of a proper process of reception . In this case this token will be added to the statement line structure , and the Lexical analyzer is requested to get a new token. This token should be (for this specific example) space (also its possible to have n-space with the effect of one space only as expressed through a local state-transition at state (2)).
In state (3) where the required token is Identifier it is necessary to check the validity of this identifier which is carried out through accessing the symbol table. The code of action is given in the diagram as "00". After the execution of the proper task a state transition is carried out to the sate specified within the cell.
The presence of other tokens which are not specified in the diagram would be considered as wrong inputs. The code of action in this case will be an error code F's. This is given in the table.3 where we can see the cells that contains the code for F1,F0 or F3 (as indication of error).

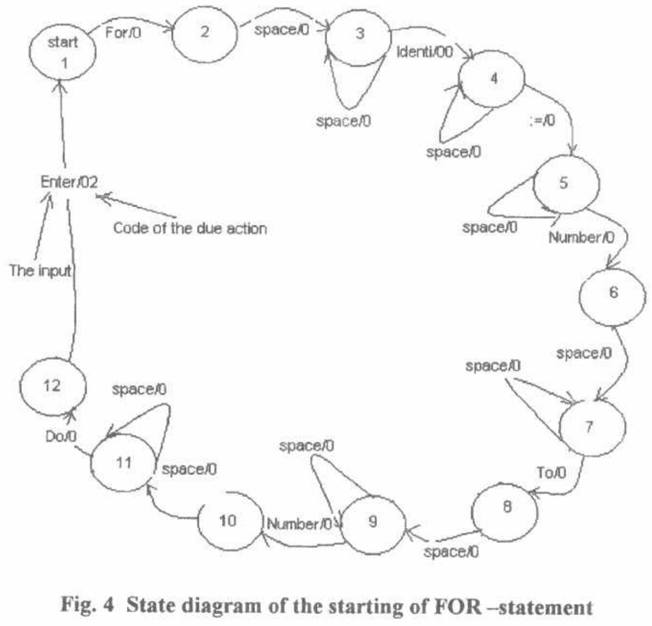


Fig. 4 State diagram of the starting of FOR –statement

This operation will continue to receive the other items of the For-statement . This goes through the states (2,3,4,5,6,7,8,9,10)and ends at the recognition of ENTER item which actually is (Cr+Lf). As we see this ends at the state number (10).
The reception ends (for this example) with the value (02) for the code of action, which indicates that the state of reception is the start of compound statement.

At the ends of successful reception of each statement we will get certain code of action to guide the next operation.
The same mechanism of processing will be included within the state table of Table.4 which in turn an Automaton expressing the activities of syntax analyzer but for the part of declaration of the data items.
Handling of the reception of declarations will be carried out in the same way as the reception of statements, of course the actions that might be executed will be different. This table involves a number of examples (Constant, variables ....).
Symbol table should be updated with each newly introduced data item, and for this to be carried out after checking if the coming data item has not been declared before.
As each of the tables 3 and 4 support the task of checking and connecting the tokens of the statements the designed task of editing is surly supported too.
The program stored copy will be similar to the source copy, in addition to a special field at the beginning of each statement line which will indicate the state number of this statement line.

Table.2 The code of action with each state transition

Code	The explanation
0	Continuation of the process of receiving and all this is going OK.
00	The token itself is true, according to the position in the statement, and still it is necessary to check the validity from the point of view of symbol table. The symbol table should be updated by the line number of this token .
10	Starting the program, end of program heading. So, the function declaration decl() will be Called.
01	The received line is OK, the case of receiving declaration line. Update the Symbol table with the new identifier, add the line to program body.
11	The line is received OK, a line of the program body. This line is to added to the program.
02	End of starting a program block such as FOR, WHIL, IF...etc. So call for a new reception function, i.e recursively .
03	End of the program. Check matching the Begin-end pairs.
F0	Unwanted I/p do not store it.
F1	Wrong Input (Token), so remove it and get a new one at the same state.
F3	Wrong input (Token) , remove it and get back to previous state.
B1	It is the case of receiving of the Begin Key word. This induce a call for the reception function recursively.
E1	The end of a block, adjust the levels of nesting.

N.B The flow charts at the end of this study give clarification of the mechanism that represents the analysis.

5-2 The Editing Mechanism

The syntax directed editing is characterized by a fact that the program viewed as a collection (according to syntax rules) of syntactically true statements.
So, it is one of the editor tasks is to build the lines of the program and connect them in some kind of structure that will express the view of a program.
This kind of editor should be composed of the activities of building the program as well as the activities of checking and correcting and modifying the tokens of the program. So, the syntax analyzer should be a component of the editor (here), at the same time it has the lexical analyzer as a subtask.

Table.3 State table of the program block (statements)

prog	For	If	begin	ident	end	' '	To	\n	Do	;	:=	Num	=	then	proc			
	2/0	13/0																
/1/F3	1/F3	1/F3	1/F3	4/00	1/F3	3/0	1/F3	....	...	..								1/F3 →
/←F3					1/F3	4/0	1F3	→		/1F3	5/0	/1F3						→
←					1/F3	5/0	1/F3		→		1/F3	6/0	1/F3					→
←					1/F3	6/0	1/F3											→
	←				1/F3	7/0	8/0	1/F3									→	
←					1/F3	9/0	1/F3										→	
←					1/F3	9/0	1/F3		→		1/F3	10/0	1/F3					→
←					1/F3	11/0	1/F3	1/F3	12/0	1/F3							→	
←							1/F3	1/02	1/F3									→
←					1/F3	14/0	1/F3											→
←			1/F3	15/00	1/F3													→
												1/F3	16/0	1/F3				
←			1/F3	17/00	1/F3												→	→
	←				1/F3	18/0	1/F3									→		
←														19/0	1/F3			→
←							1/F3	1/02	1/F3									→

Table 4. State Table of receiving declaration

Use	var	type	const	;	.	:	[	]	Rec	arry	\n	' '	//	ident	integr	real	char	numb
/0	/0	/0	2/0	1/F1	1/F1	1/F1	1/F1	1/F1	1/F1	1/F1	1/F0	1/F1	1/9	4/20	1/F1	1/F1	1/F1	1/F1
←					0/F3	3/0	0/F3										→	
←										0/F3	1/con	0/F3						→
←					0/F3	5/0	←			0/F3	4/0	4/0	0/F3					→
								0/F3	36/0	18/0	0/F3		→		8/0	11/0	15/0	6/0
←	←		0/F3	7/0	7/0	0/F3												→
				0/F3	8/0	0/F3			→		1/22	0/F3						→
←	←				0/F3	9/0	0/F3											→
←			0/F3	10/0	0/F3													→
←										0/F3	1/22	0/F3					→	

The program itself is composed of a group of statements that is in addition to the data entities.

The process of handling these component to build the program depends on a point that each statement, declaration of some data or the program heading is arranged in a separate line. These lines are connected with each other to form the program. The dynamic data structure, The POINTER, is used to implement the proposed way of connection. So the Line structure will be as given in the following diagram.

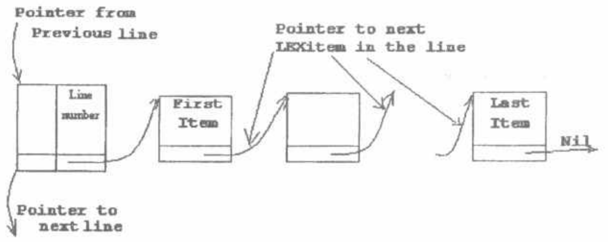

Pointer from Previous line
Pointer to next LEXitem in the line
Line number
First Item
Last Item
Nil
Pointer to next line

Fig.5 Structure of the statement and its link with the line

Building a line is an distinctive process. When the line that is being received is proved to be error free and the end of line character is also received, it will be connected with the other lines that form the program.
The statement it self is defined in the program according to the view of Fig.6 and has got a structure as:

```
Struct Theline {
 Int   stateno;
 String thetoken;
 Struct Theline   *currentline;
     }
```

Editing activities should be carried out on an interactive basis, so that suitable actions could be taken in the proper time.
Error handling should be carried out on more than one level. With respect to the lexical token the lexical analyzer it self will do the necessary handling on on-line basis.

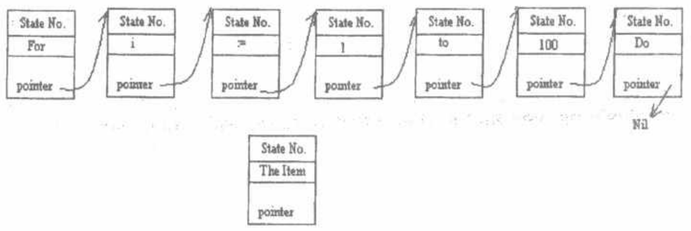


Fig.6 Structure of the statement itself

The file, as we indicated before, is a collection of lines. So, the linking of these lines is defined according to the following programming structure of the program lines which is given in Fig.5:

```
Struct lines {
  Int lineno;
  Struct Theline    *thisline;
  Struct  lines  *Next;
  Int   stateno;//
}
```

7- Conclusion

In view of the importance of dynamic modification this paper has introduced:

- A proposition for dividing the Module into two parts, Interface and Architecture, as a way to increase the level of dynamic modifications.
- Suggested a scheme for program development (syntax directed editing) that can take into account handling the modified code from the point of view of editing as well as the syntax analysis up to the full compilation.

Syntax analysis was introduced as an integrated part of the whole task of program development. In this respect this study supports the design of syntax analyzer to be an automaton which can be implemented as state table represents the grammar of the adopted language.

- One of the main features of this scheme of analysis is the large amount of data that should be available.
- The other point to be mentioned here is that the maintainability of the designed system is highly supported due to the possibility of modifying or even changing the utilities that are activated with each state transition independently on the state table.
- This scheme minimizes the amount of code that might be written other wise.
- It is possible to adopt this scheme of syntax analysis in a more general way so as to be used for more that one language.

Even so the proposed editing utilities have not been designed, yet the introduced scheme of analysis as well as the data structures go with all of such utilities. These utilities are going to be the subject of the continuation of this study.

References

[1]. J.Kramer.,J. Magee., M. Soloman., and A. Lister" ،CONIC: An Integrated Approach to Distributed Control System ".Research Report., Imperial College, London.

[2]. J. Magee., J. Kramer" Dynamic System Configuration for Distributed Real-Time Systems. Research Report., Imperial College, London.

[3]. M.E.D. Horani" Towards Dynamic Modification Of a Distributed Computer Systems". A PhD Thesis , University of Kent at Canterbury, England.

[4]. James G. Mitchell, The Design and Construction of Flexible and Efficient Interactive Programming Systems, Garland Publishers, Inc. 1970.

[5]. S.J. Yong, Real-Time Languages Design and Development, Ellis Horwood, 1982

[6]. P.J. Brown, Writing Interactive Compiler and Interpreters, John Wiley 1997

[7]. M.E.D.Horani " Objectives of Dynamic Modifications" , First Arab Symposium on Electronic Engineering, Allepo١٩٩.

[8]. David Gries. Compiler Construction for Digital Computers.

[9]. Richard Bornat. Understanding and Writing Compilers

[10]. Alexander & Hanna. Theory of Automata :an Engineering Approach

[11]. John Dedourek & Uday G. Gujar. Scanner Design

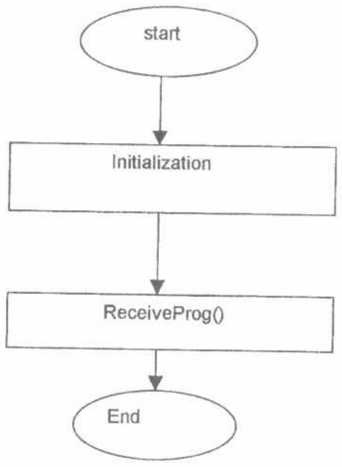
[12]. Robert P.COOK, MOD A Language for Distributed Programming

[13].Robert E Strom,Shaula Yemini , NIL: An Integrated Language and System for Distributed Programming.

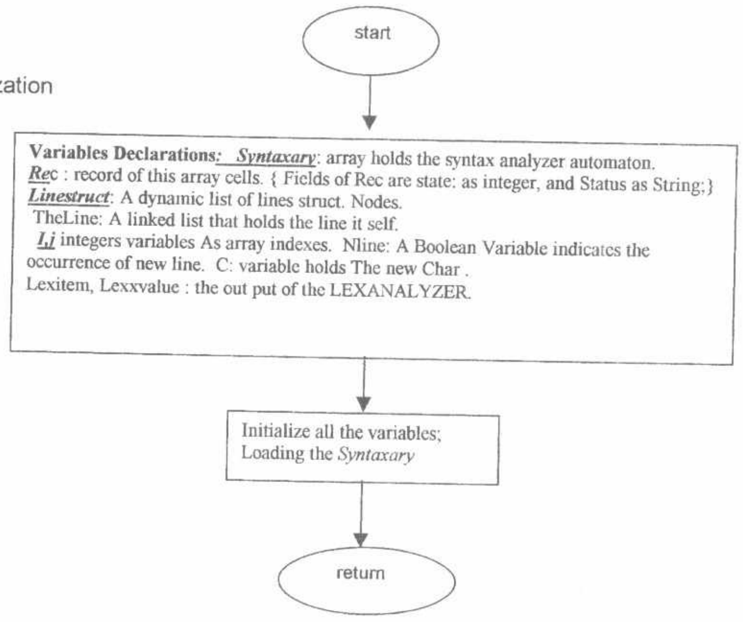
[14]. M. E. D. Horani " Design of a Lexical Analyzer", 38th Science week, 1998.

APPENDIX

The Syntax Analysis

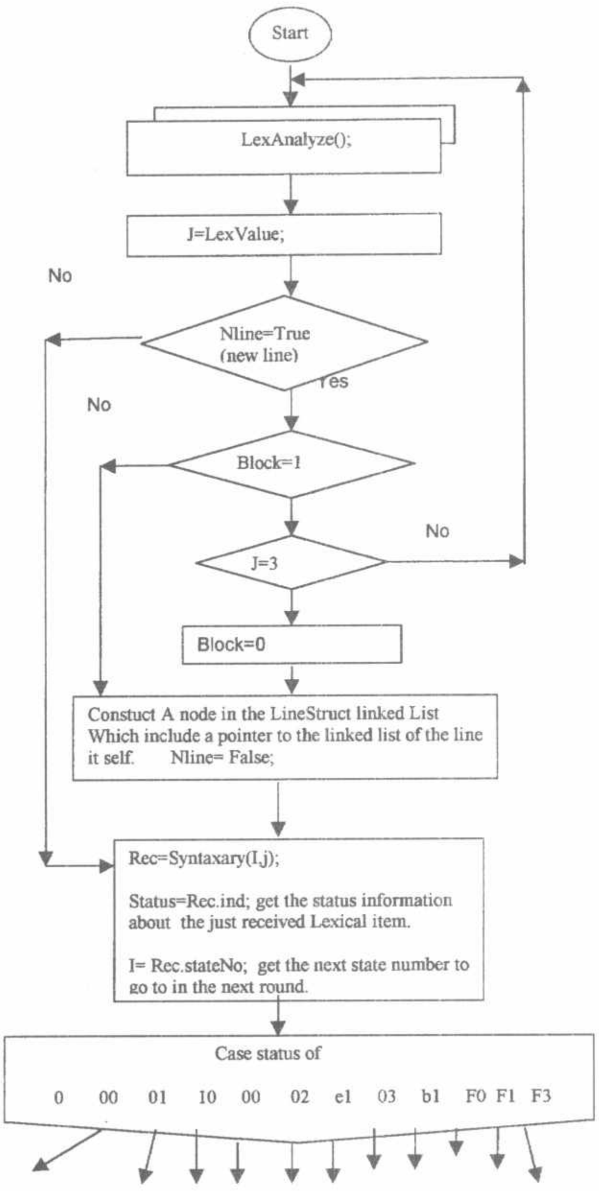


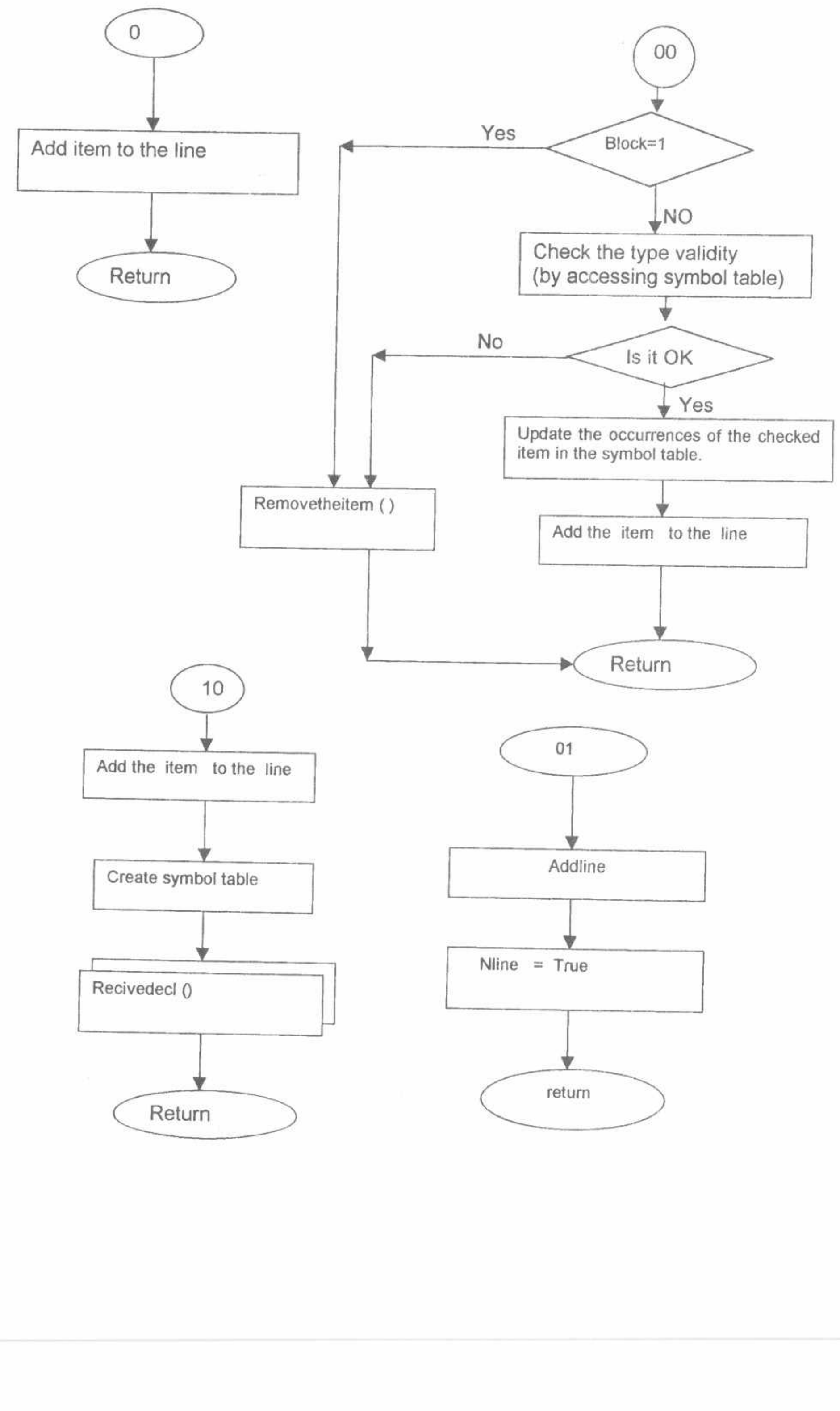
Initialization

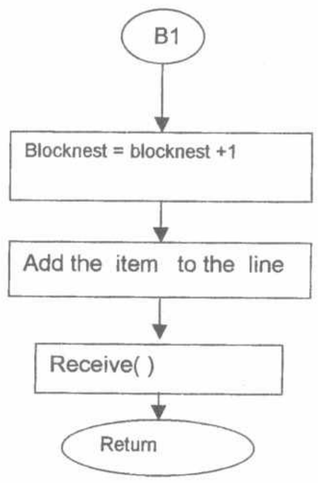


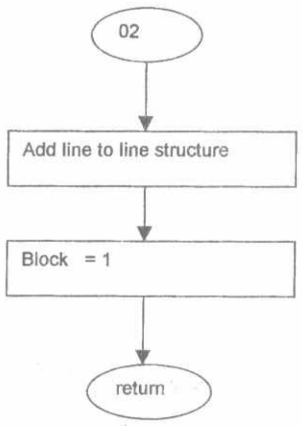
Variables Declarations: ***Syntaxary***: array holds the syntax analyzer automaton.
Rec : record of this array cells. { Fields of Rec are state: as integer, and Status as String;}
Linestruct: A dynamic list of lines struct. Nodes.
TheLine: A linked list that holds the line it self.
I,i integers variables As array indexes. Nline: A Boolean Variable indicates the occurrence of new line. C: variable holds The new Char .
Lexitem, Lexxvalue : the out put of the LEXANALYZER.

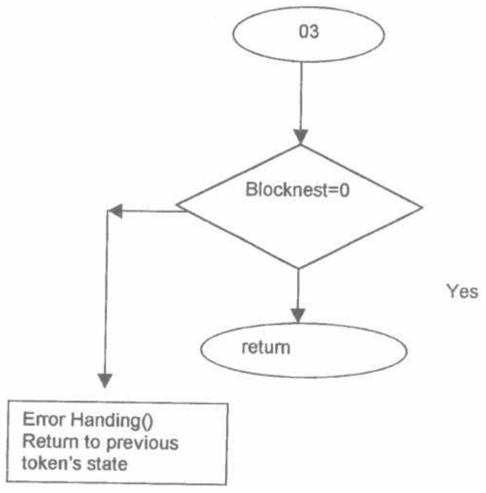
RecieveProg

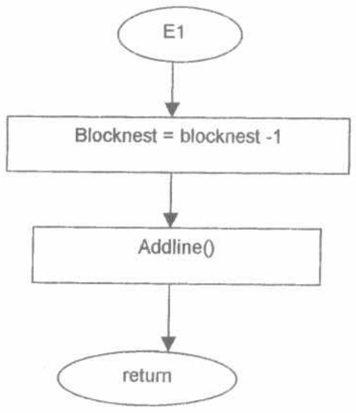

Start
LexAnalyze();
J=LexValue;
No
Nline=True
(new line)
Yes
No
Block=1
No
J=3
Block=0
Constuct A node in the LineStruct linked List Which include a pointer to the linked list of the line it self. Nline= False;
Rec=Syntaxary(I,j);
Status=Rec.ind; get the status information about the just received Lexical item.
I= Rec.stateNo; get the next state number to go to in the next round.
Case status of
0 00 01 10 00 02 e1 03 b1 F0 F1 F3

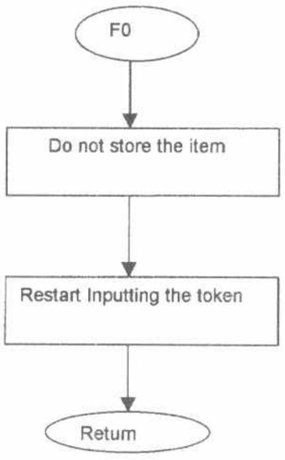

0
Add item to the line
Return
00
Block=1
Yes
NO
Check the type validity (by accessing symbol table)
Is it OK
No
Yes
Update the occurrences of the checked item in the symbol table.
Removetheitem ( )
Add the item to the line
Return
10
Add the item to the line
Create symbol table
Recivedecl ()
Return
01
Addline
Nline = True
return

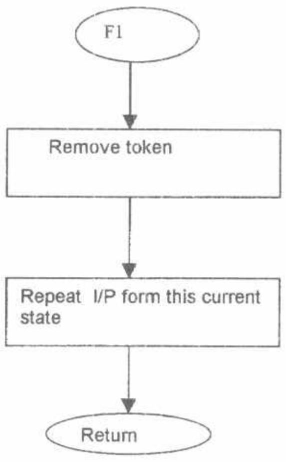

B1
Blocknest = blocknest +1
Add the item to the line
Receive( )
Return

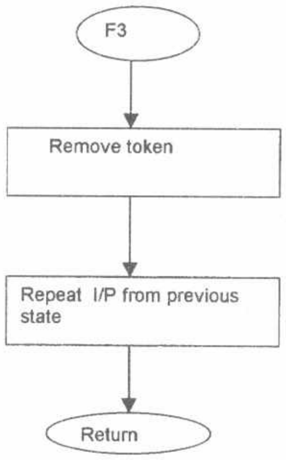

02
Add line to line structure
Block = 1
return


03
Blocknest=0
Yes
return
Error Handing()
Return to previous
token's state


E1
Blocknest = blocknest -1
Addline()
return


F0
Do not store the item
Restart Inputting the token
Return


F1
Remove token
Repeat I/P form this current state
Return


F3
Remove token
Repeat I/P from previous state
Return