GENERAL SYSTEM DYNAMICS SIMULATION SOFTWARE SYSTEM -- ZU-DYNAMO

Wenhua Wu, Chingrui Xu, Shaozhong Jiang
Dept. of Industrial Engineering
Zhejiang University
Hangzhou, P. R. C.

ABSTRACT

System dynamics modeling has been applied in a wide variety of areas. However, as a means of simulating models in computers, there is no any general DYNAMO compiler system that can be used in various types of computer. The purpose of this paper is to deal with a general compiler software system ZU-DYNAMO, which is used to simulate models in various types of computer with outputs in English or Chinese. Being different from traditional method, a new idea suggested in this paper is the selection of C language instead of assemble language as objective code. The aim of such selection is to make ZU-DYNAMO independent on a particular computer. The overall structure and design principle of the system are presented. The algorithms and techniques used in the system, and the structure of objective code are designed and analysed. The description of extensions of Arrays, FOR card, etc. and the ways to implement are also given.

I. INTRODUCTION

With the further development of System Dynamics, it becomes more and more important to develop a general DYNAMO software system that is independent on a particular computer. The usual DYNAMO compiler system being dependent on one type of computer, has set a limit to wide applications of DYNAMO modeling. This paper intends to introduce a General System Dynamics Simulation Software System -- ZU-DYNAMO (ZU is abbreviated from "Zhejiang University") developed by authors, which is a translator and

compiler for translating, compiling and running continuous models with outputs in English, or Chinese.

In the development of DYNAMO system, traditional method is constructing a compiler to translate DYNAMO cards into assemble language code or machine instructions. Because these low-level languages are dependent on one type of computer, this method make it difficult or impossible to transfer DYNAMO system from one computer to other. Being different from this traditional method, a new idea was suggested in the development of ZU-DYNAMO system, which is developing a translator to translate ZU-DYNAMO cards into an equivalent C language program, and then referring the C compiler installed on the computer to compile this C program, finally generating the running code to simulate models. The illustration of this design idea is shown in Fig. 1.

ZU-DYNAMO was coded in structured language C that is called " system design language " and possesses the features of short but strong and capable. Consequently, the running code of the system is very efficient and runs fast. In order to make the ZU-DYNAMO useful and helpful in simulation of big models, the system provides not only the fundamental functions of DYNAMO II, but also the extensions of Arrays, FOR card, WHILE card, IF card and Macros. The algorithms and techniques to implement these functions will be described in the following. But in the next section, we will first introduce the overall structure and design principle of the system

II. THE OVERALL STRUCTURE AND DESIGN PRINCIPLE OF THE SYSTEM

The whole system is constituted of five large parts :Syntax and Semantic Checking Program, Equation Ordering Program,Automatical Translating Program, Compiling & Linking Program and Running & Output Producing Program. The whole system is controlled by Master Control Routine. The overall structure of the system is given in Figure 2.

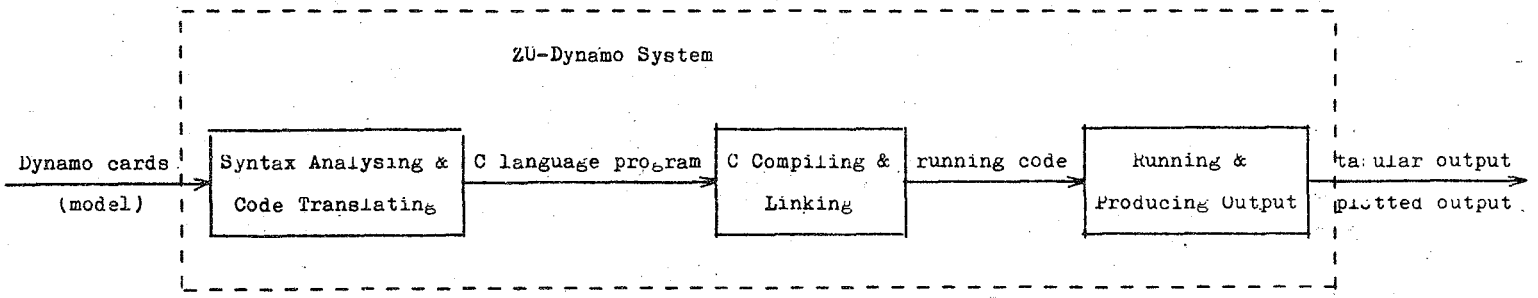

Figure 1: The Illustration of Design Idea of the System

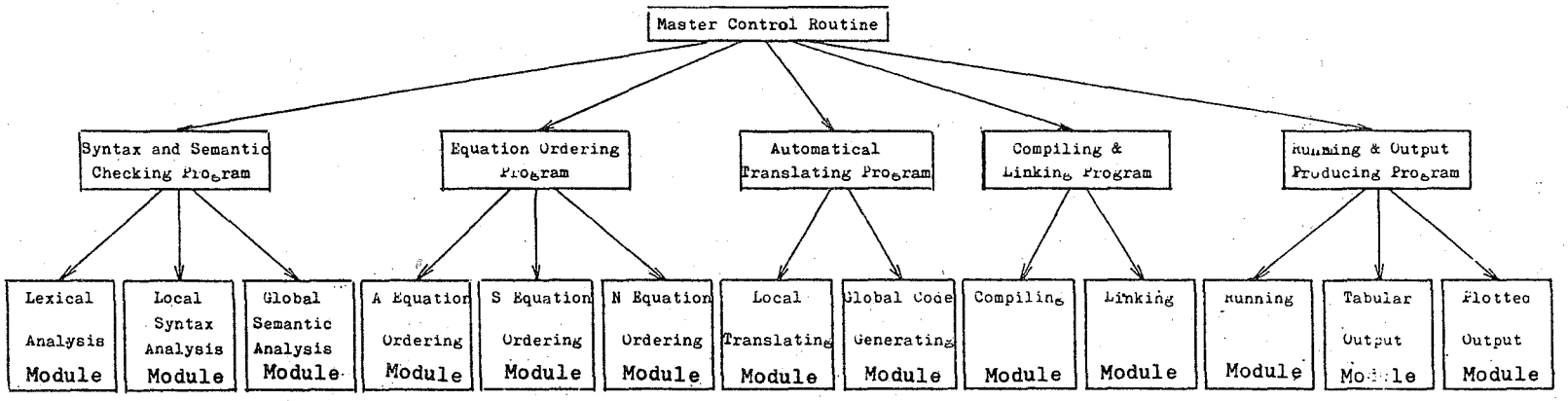

Figure 2: The Overall Structure of the ZU-DYNAMO System

The Syntax and Semantic Checking Program was developed to scan the model and to check the syntax and semantic of DYNAMO cards in the model. This program is constituted of Lexical Analysis Module, Local Syntax Analysis Module and Global Semantic Analysis Module. The Lexical Analysis Module called the Lexical Analyzer, or Scanner, separates characters of DYNAMO cards in the model into groups that logically belong together; these groups are called tokens. The usual tokens are key words, such as PLOT or PRINT, variables, such L.K or R.KL, operator symbols, such as + or *, and punctuation symbols such as commas or parentheses. The functions of the Local Syntax Analysis Module are to check the syntax in each card, including the checks for the legality of variables, for the correctness of expressions, for the completion of function references and for the exactitude of time subscripts. In the mean time, the variable name defined in each equation was recorded to the table of variable names. The Global Semantic Analysis Module, and then, checks whether variables used in the right of equation are defined, and whether each level variable has initial value defined by N equation. If the errors were detected, the error messages are displyed and the locations where the errors were detected are approximately pointed out. Only when there is no any error in cards, can the DYNAMO cards be sorted and reordered, and translated to the C language program.

The Equation Ordering Program arranges in the proper order of computation among equations. Unlike the level and rate equation, the Auxiliary equations or Supplement equations, or Initial equations cannot be computed in arbitrary order. Some A equations could be components of others, and must be computed in the proper order so that one can be used by the next. The Priority Computation Variable Set is used to order these equations. When this program discovers a group of equations in which none can be computed without first knowing the value of one or more others in the group, the simulataneous equations is reported. In the matter of algorithm how to order equations, we will describle it in the next section. Only when the proper

order has been arranged by this program, can the correct object code (C language code) be generated.

The Automatical Translating Program is the kernel part of system. It translate the correct DYNAMO arranged in order into an equivalent C program. The Automatical Translating Program is constituted of Local Translating Module and Global Code Generating Module. The Local Translating Module translates each equation card into an equivalent C statement in terms of the syntax of C language. Because the equation in DYNAMO is similar to the assignment in C language, the equation cards can be easily converted to the C statements with few modifications. On the base of these C statements, the Global Code Generating Module generates some global C statements such as the declarative statements to specify allvariables, the repetitive statement "for" to control the simulation cycle, the data store statement to store the values of variables plotted or printted in data file, and the assignments to assign the values of K or KL variables to J or JK variables for the simulation of next time interval.

The Compiling & Linking Program is composed of Compiling Module and Linking Module. The Compiling Module invokes the C compiler installed on the computer to compile the C language program just converted. And then, the Linking Module invokes the linker to link it with library of DYNAMO functions to implement in C and the library of C language to generate the running code. The Compiling Module uses the system calling statement provided in C language : system("Compiling Command in the computer") to let the computer ompile the C language program. Simularly, the Linking Module also invokes this system calling : system(" Linking Command in the computer") to let computer link them. Because the compiling and linking commands are different in various types of computer, the two system calling should be modified in terms of the formats of commands in the computer.

The Running & Output Producing Program is constituted of the Running Module, the Tabular Output Module and the Plotted

Output Module. The Running Module also refers the statement of system calling : system("Running Command in the computer") to let computer execute the running code and simulate the model. The Tabular Output Module is very simple, here is not discussed. The Plotted Output Module is very complicated. First, it cacu-lates the scales of variables. And then, for each time interval , a set of values of variables are read from data file and are sorted. Following these, the Plotted Output Module begin to plot from the left to the right, in terms of the ascending values just sorted. This process is repeated until the results of simulation in all time intervals have been plotted.

According to the structure of the system and function of each program in the system presented above, we can give the design principle of the system. The illustration of the design prin-ciple of the system is given in Figure 3. First, the Syntax and Semantic Checking Program parses the DYNAMO cards inputted. If no error was detected in the cards, the Equation Ordering Program begins to order the DYNAMO cards with no error. And then, the Automatical Translating Program translates these DYNAMO cards ordered with no error into an equivalent C language program. The Compiling & Linking Program compiles this C language program, and links it with the library of DYNAMO functions and the library of C language to generate the running code. Finally, the Running & Output Producing Program executes the running code, simulates the model and produce the tabular output or plotted output.

III. THE ALGORITHMS AND TECHNIQUES DESIGNED IN THE SYSTEM

In this section, we will describle the important algorithms designed in the system. These algorithms have been programmed in the ZU-DYNAMO and run very well.

Operator Precedence Parsing Algorithm

In a DYNAMO model, most of all cards are equations, and the basic format of these equations is

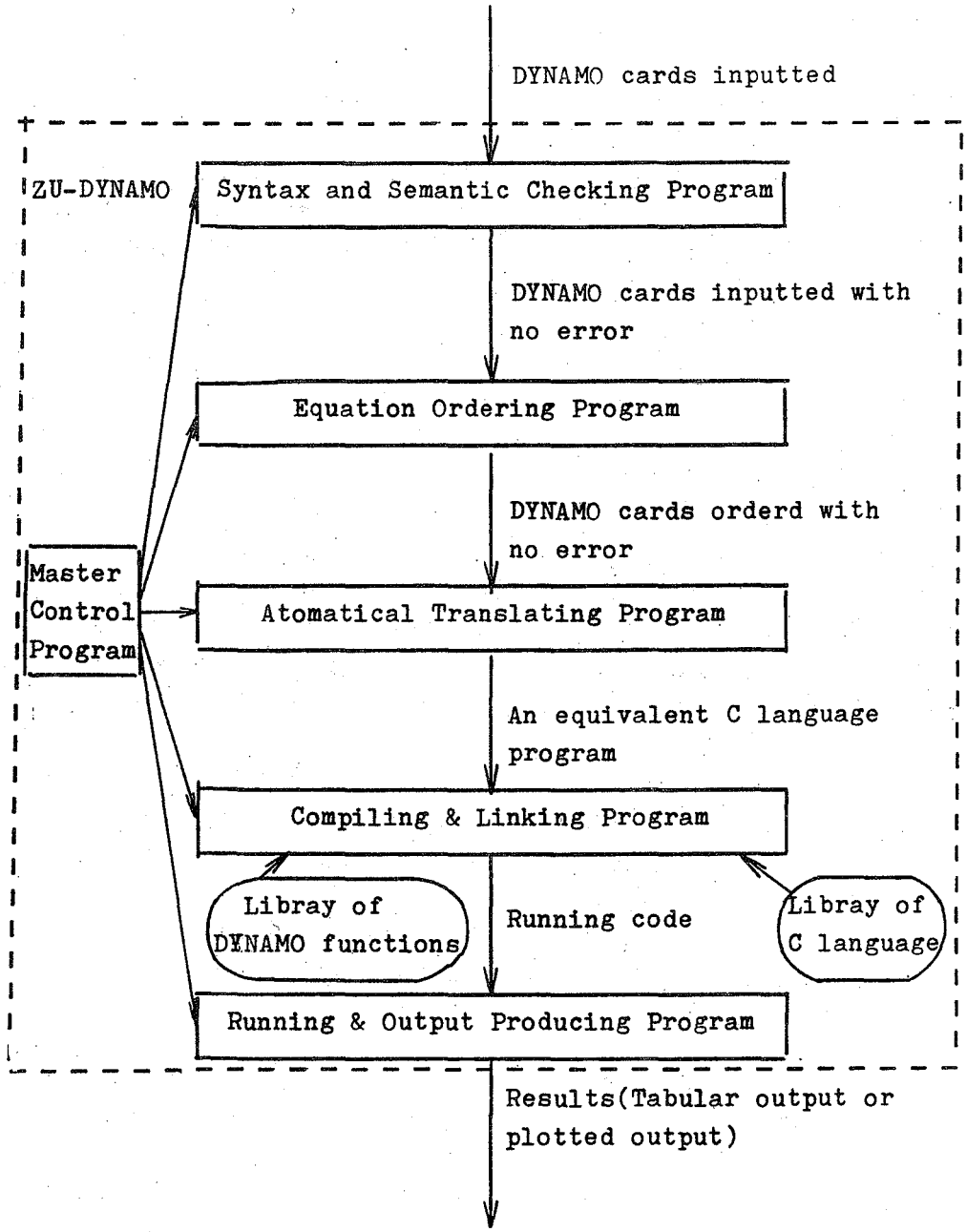

Figure 3: The Design Principle of ZU-DYNAMO System

quantity-name ± expression

So the key of syntax checking is to parse the expression. In the ZU-DYNAMO system, Operator Precedence Parsing Algorithms in the Syntax and Semantic Checking Program has been designed to parse the expression in the equation cards.

In the DYNAMO language, the expression may be anything from simply a number of quantity to a very complicated combination of factors and terms involving functions , quantity names and numerical values. The operations of addition, substraction, multiplication, division and exponentiation are indicated respectively by +, -, *, /, **. According to the definitions of expression above and the rules of the notation called BNF (Backus Naur-Form), we can give the operator grammar which have no two adjacent nonterminals :

$$E \longrightarrow E+E \mid E-E \mid E*E \mid E/E \mid E**E \mid -E \mid (E) \mid id \qquad (1)$$

where E is an abbreviation for expression called nonterminal symbol, id represents variable or numerical value or function reference called terminal symbol, the remaining symbol are terminals.

In operator-precedence parsing, we use three disjoint precedence relations, $\lessdot$, $\doteq$ and $\gtrdot$, between certain pairs of terminals. These precedence relations guide the selection of handles. If $a \lessdot b$, we say a "yields precedence to" b; if $a \doteq b$, a " has the same precedence relations as" b; if $a \gtrdot b$, a "takes precedence over" b. Although these relations may appear similar to the arithemetic relation "less than", "equal to", and "grater than", the precedence relations have quite different properties. For example, $a \gtrdot b$ does not imply $b \lessdot a$.

Now we show how to compute precedence relations of the grammar. Let G be an ε-free operator grammar. For each two terminals a and b, we say : (P, Q, R is nonterminal)

i) $a \doteq b$ if ther is a production of the form $P \longrightarrow \ldots ab \ldots$ or $P \longrightarrow \ldots aQb \ldots$ where Q is nonterminal. That is, $a \doteq b$ if a

appears immediately to the left of b in a right side, or if they appear seperated by one nonterminal. For example, the production of $E \longrightarrow (E)$ implies that $(\doteq)$.

ii) $a \lessdot b$ if there is a production of the form $P \rightarrow \ldots aR \ldots$ and $R \Rightarrow^{+} b \ldots$ or $R \Rightarrow Qb \ldots$. That is, $a \lessdot b$ if a nonterminal P apperas immediately to the right of a and derives a string in which b is the first terminal symbol. For example, in grammar (1), there is $E \longrightarrow E+E$ and $E \Rightarrow (\ldots$, so $+ \lessdot (\ldots$

iii) $a \gtrdot b$ if there is a production of the form $P \longrightarrow \ldots Rb \ldots$ and $R \Rightarrow^{+} \ldots a$ or $R \Rightarrow^{+} \ldots aQ$. That is, $a \gtrdot b$ if a nonterminal appearing immediately to the left of b derives a string where last terminal is a.

If the precedence relation $\lessdot$, $\doteq$ and $\gtrdot$ constructed as above are disjoint in operator grammar G, that is, for any pair of terminals a and b, never more than one of the relations $a \lessdot b$, $a \doteq b$, and $a \gtrdot b$ is true, the operator grammar is called the operator precedence grammar. It is evident that Grammar (1) is not an operator precedence grammar because two precedence relation $+ \gtrdot +$ and $+ \lessdot +$ hold between + and +.

In terms of traditional associativity and precedence of the operators, grammar (1) can be transformed into an equivalent grammar that is both operator-precedence and unambiguous.It is

$$E \longrightarrow E+T \mid T \qquad (2)$$
$$E \longrightarrow E-T \mid T \qquad (3)$$
$$T \longrightarrow T*F \mid F \qquad (4)$$
$$T \longrightarrow T/F \mid F \qquad (5)$$
$$F \longrightarrow P**F \mid P \qquad (6)$$
$$P \longrightarrow (E) \mid id \qquad (7)$$
$$(8)$$

According to the method of computing precedence relations and the productions (2)--(8), the operator precedence relations of above grammar can be constructed, and shown in Figure 4. (Blanks denote error entries, # is a special symbole which

marks the ends of string checked.)

	+	-	*	/	**	id	(	)	#
+	>	>	<	<	<	<	<	>	>
-	>	>	<	<	<	<	<	>	>
*	>	>	>	>	<	<	<	>	>
/	>	>	>	>	<	<	<	>	>
**	>	>	>	>	<	<	<	>	>
id	>	>	>	>	>			<	<
(	<	<	<	<	<	<	<	=	
)	>	>	>	>	>			>	>
#	<	<	<	<	<	<	<		

Figure 4. Operator Precedence Relations of Grammar of Expression in DYNAMO

Now let us explore how a Operator Precedence parsing algorithm is built from precedence relations that one constructed from an Operator-Precedence Grammar((2)--(8) above) in DYNAMO language. The stack is used to store the terminals and nonterminals. The input of this algorithm is the precedence relations and an input string of terminals (i.e. the expression in an DYNAMO equation). Let the input string be $a_1 a_2 \ldots a_n \#$. Initially, the stack contain #. If only # is on the stack and # is on the input, the input string is correct and accepted. The algorithm in detail is shown in Figure 5.

The Equation Ordering Algorithm

Because the ordering algorithm among N equations or S equations is similar to the algorithm among A equations. Now we only consider the Auxiliary equation ordering algorithm among A equations.

```
Repeat Forever
  if only # is on the stack and only # is on the input
     then
          accept and break     /* the string is correct */
     else
         {
          let a be the topmost terminal symbol on the stack
             and let b be the current input symbol;
          if a ⋖ b or a ≐ b then
                                shift b onto the stack
             else
                  if a ⋗ b then                /* reduce */
                     Repeat
                            pop the stack
                     Until  the top stack terminal is
                            related by ⋖ to the terminal
                            most recently poped
                     else
                            call the error correcting routine
         }
```

Figure 5. Operator Precedence Parsing Algorithm

We assume, there are m Auxiliary equations in DYNAMO cards in the model inputted by user.In order to deal with conveniently, we may give a number to each Auxiliary variables from 1 to m according to the order of inputs. Priority Computation Variable Set PCVS S[i] for some variable A_i ($1 \leqslant i \leqslant m$) is introduced, which is defined as the set of the numbers of variables, which are in the right side of ith Auxiliary equation. that is to say, only if all variables in Priority Computation Variable Set S[i] for ith Auxiliary variable have been computed, the ith Auxiliary Variable then can be computed.

For example, we assume 5 Auxiliary equations appearing in the model inputted are:

```
A   DFR.K=TABLE( TDFR,IAR.K/RSR.K,4,12,4 )
```

```
A  RSR.K=SMOOTH( RRR.JK,DRR )
A  IDR.K=A1R*RSR.K+DFR.K
A  DDS.K=IDR.K*RRR.JK
A  INSTP.K=STEP(20,1)
```

It is known that IAR.K is level variable. We give a number 1 to DFR.K, 2 to RSR.K, 3 to IDR.K, 4 to DDS.K, 5 to INSTP.K. According to above definition, the Priority Computation Variable sets for 5 Auxiliary variables are:

```
S[1]=[2];
S[2]=[ ];
S[3]=[1,2];
S[4]=[3];
S[5]=[ ];
```

It is evident that variable whose PCVS is empty should be arranged to be computed first, because this variable has not refered any Auxiliary varirble. If there are two or more variables whose sets are empty, it may be arranged in arbitrary order of these variables to be evaluated. These variables arranged don't be considered after. In the above-mentioned example, S[2] and S[5] are empty. So 2th variable should be arranged to be evaluated firstly and 5th variable secondly, or 5th variable firstly and 2th variable secondly.

Now, we try to find which variable should be computed thirdly in the above-mentioned example. We delete the numbers of variables that have be arranged to be computed from the left sets which these numbers of variabies belong to. If there is a variable whose sets is empty, this variable is arranged to be computed and the above process is repeated until the all variables are arranged to be computed. If there is no any variable whose PCVS is empty and no all variables are arranged to be computed, the error occur and the error message " SIMULTANEOUS ACTIVE EQUATIONS IN INVOLVING" should be displayed

The Equation Ordering Algorithm has been given in Figure 6. It is noted that the input of algorithm is the Priority

Computation Variable Sets for all Auxiliary variables which are given numbers from 1 to m. The output of the algorithm is the sequence of computations for all variables. In the above-mentioned example, the proper order of computations is 2, 5, 1, 3, 4, or RSR.K, INSTP.K, DFR.K,IDR.K, DDS.K.

```
# define  FINISH   1
# define  NOFINISH 0
end=NOFINISH;
while ( end==NOFINISH )

  Find all empty sets, in the left sets;
  if ( there is no any empty set )
    {
     print the error message
       "SIMULTANEOUS ACTIVE EQUATION  IN INVOLVING";
    }
  else
      {
       Arrange the variables whose PCVS sets are empty
         to be computed first;
       Count the number of the variables which have
         been arranged;
       Delete the Variable No.s whose sets are empty
         from all left no empty sets, to which these
         Variable No.s belong;
      };
  if (the number of the variables arranged are equal
      to the number of all variables)
     {
      end=FINISH
     }
```

Figure 6. The Equation Ordering Algorithm

The Translation Method and The Structure of Objective Code

The translations may be devided into two parts of local translation, or called the preprocess of the translation and code generation. The first work of preprocessing is to delete all type symbols N, C, T, L, A, R, S from each card, because these symbols are no useful and can not be recognized in C language. The next work is to delete all point "." appearing in time subscripts .J, .K, .JK and .KL in each card, since the point "." can not accepted in the name of variable by C compiler. After these deletions, the DYNAMO equation cards have been changed to the legal assignments in C language, and the variable L.K L.J R.KL R.JK have been changed to LK LJ RKL RJK respectively. So theother work is to concatenate "J" or "JK" witk L, A, R variable in initial equations. In the base of these, the system generates the declarative statement, initial statements , repectitive statement, data store statement,etc. These C statements are organized in the following format shown in Figure 7. In other word, the structure of objective code is given in Figure 7.

```
the declarative statements;
the assignments for computing initial values;
 (LJ=...; AJ=... ; RJK=...)
number=(int)(LENGTH/DT + 1;
for (i=1; i = number; i++)
{
   time= i*dt;
   LK=f1(AJ,LJ,RJK);
          :
   A1K=f2(RJK,A2K,LK);
          :
   RKL=f3(RJK,AK,LK);
          :
   SK=f4(LK,AK,RJK,S2K);  ......
   the statements to store data;
    LJ=LK; ... AJ=AK; ...; RJK=RKL;...
};
the statements to print or plot;
```

Figure 7. The Structure of Objective Code

Following is a example of translation. The DYNAMO cards in the model inputted by user shown in Fig. 7 are translated into an equivalent C program shown in Fig. 8 by ZU-DYNAMO.

```
L    UOR.K=UOR.J+(DT)(RRR.JK-SSR.JK)
L    IAR.K=IAR.J+(DT)(SRR.JK-SSR.JK)
R    SSR.KL=UOR.K/DUR
L    RSR.K=RSR.J+(DT)(1/DRR)(RRR.JK-RSR.J)
A    IDR.K=(AIR)(RSR.K)
A    ISR.K=IDR.K-IAR.K
R    PDR.KL=(ISR.K/DIR)+RSR.K
L    UOD.K=UOD.J+(DT)(PDR.JK-SRR.JK)
R    SRR.KL=UOD.K/DUD
R    RRR.KL=100.0+INSTP.K
A    INSTP.K=STEP(20,1)
N    UOR=(DUR)(100.0)
N    RSR=100.0
N    UOD=(DUD)(100.0)
N    IAR=(AIR)(100.0)
C    DUR=1.0
C    DRR=2.0
C    AIR=3.0
C    DIR=2.0
C    DUD=2.0
PRINT  *,*,SRR(0,2)/IAR(0,2)/*,*,SSR(0,2)/UOR(0,2)/*,*,RRR(0,20)/
X        *,INSTP(0,2)/RSR(0,2)/*,IDR(0,2)/*,ISR(0,2)/*,*,PDR(0,2)/
X        UOD(0,2)
PLOT   RSR=A,SSR=S,PDR=P,SRR=D,RRR=R(0,200)/UOR=B,IAR=I,IDR=T,UOD=U(0,400)
X      /ISR=N(-100,100)
SPEC   DT=0.5/LENGTH=18.5/PRTPER=2.5/PLTPER=0.5
```

Fig. 7. The Dynamo cards in the model inputted

```
#include "stdio.h"
double time,dt;
main()
{
  double length,prtper,pltper;
  double dur;
  double drr;
  double air;
  double dir;
  double dud;
  int i,number;
  double idrj,idrk;
  double isrj,isrk;
  double instpj,instpk;
  double ssrjk,ssrkl;
  double pdrjk,pdrkl;
  double srrjk,srrkl;
  double rrrjk,rrrkl;
  double uorj,uork;
  double iarj,iark;
  double rsrj,rsrk;
  double uodj,uodk;
  time=0.0;
  dt=0.50;
  length=18.5;
  prtper=2.5;
  pltper=0.5;
  dur=1.0;
  drr=2.0;
  air=3.0;
  dir=2.0;
  dud=2.0;
  uorj=dur*100.0;
  rsrj=100.0;
  uodj=dud*100.0;
  iarj=air*100.0;
  idrj=air*rsrj;
  isrj=idrj-iarj;
  instpj=step(20.0,1.0);
  ssrjk=uorj/dur;
  pdrjk=isrj/dir+rsrj;
  srrjk=uodj/dud;
  rrrjk=100.0+instpj;
  number=(int)(length/dt)+1;
  for (i=1;i<=number;i++)
  {
    time=i*dt;
    uork=uorj+dt*(rrrjk-ssrjk);
    iark=iarj+dt*(srrjk-ssrjk);
    rsrk=rsrj+dt*(1/drr)*(rrrjk-rsrj);
    uodk=uodj+dt*(pdrjk-srrjk);
    idrk=air*rsrk;
    isrk=idrk-iark;
    instpk=step(20.0,1.0);
    ssrkl=uork/dur;
    pdrkl=isrk/dir+rsrk;
    srrkl=uodk/dud;
    rrrkl=100.0+instpk;
    if (((time-dt)%prtper)==0)
       storedata(fn1);
    if (((time-dt)%pltper)==0)
       storedata(fn2);
    idrj=idrk;
    isrj=isrk;
    instpj=instpk;
    ssrjk=ssrkl;
    pdrjk=pdrkl;
    srrjk=srrkl;
    rrrjk=rrrkl;
    uorj=uork;
    iarj=iark;
    rsrj=rsrk;
    uodj=uodk;
  }
  output(fn1,fn2);
}
```

Fig. 8. An equivalent C program translated by ZU-DYNAMO

IV. THE EXTENSIONS OF BASIC DYNAMO

The ZU-DYNAMO was developed to process not only the basic functions of DYNAMO, but also the extensions of Arrays, FOR card, WHILE card, IF card, Macro and many other functions. The users may use these cards like other basic cards in basic DYNAMO. It is suitable to simulate the big model.

Arrays And FOR Card

The ZU-DYNAMO extensions for arrays are in the style of DYNAMO III. The array features offer convenient notational scheme. The FOR variable is used as the subscript of array. The format of arrays and FOR card are similar to the ones in DYNAMO III. But in the ZU-DYNAMO, the FOREND is used in the end of body of cycle. The format of FOR card is

```
FOR   for1=low1, up1/for2=low2, up2 ...
        ...   cards    ...
FOREND
```

The system translate above FOR card into the following C statement:

```
for  (for1=low1; for1<=up1; for1++)
  {
     statements;  ......
  };
```

While an array is used in the model, ZU-DYNAMO first generates the declarative statements to sepcify the dimensions, sizes and data types of array. And then, the parentheses "(" and ")" in the element of the array A.K(I) are replaced with "[" and "]" respectively, because the element of array in C language reprents AK[I], not AK(I).

WHILE Card

ZU-DYNAMO also provides the WHILE card that DYNAMO III has not

provided. While a number of cards should be repeated to be computed in a certain condition, the WHILE card may be used. The format of the WHILE card is

```
WHILE       condition
        cards
WHILEEND
```

Where condition is the boolean expression that is composed by boolean operator <, <=, >, >=, <> and ==. A.K+B.K>=0 is a example of boolean expression. The above WHILE card can be translated by ZU-DYNAMO into;

```
while(condition)
{
   statements;
};
```

FOR example, the cards

```
WHILE  A.K+B.K>0
A      C.K=A.K/(A.K+B.K)
A      D.K=B.K/(A.K+B.K)
A      A.K=B.K-10
WHILEEND
```

are translated into the C statement by ZU-DYNAMO as following:

```
while(ak+bk>0)
{
    ck=ak/(ak+bk);
    dk=bk/(ak+bk);
    ak=bk-10;

};
```

The ways to implelent other extensions of cards are similar. The other extensions of cards in detail will not be dealt with here owing to the limitation of space.

V. CONCLUSIONS

According to the above-mentioned introduction, it may be seen that general compiler system ZU-DYNAMO has an advantage over usual DYNAMO compiler system. On the one hand, it take shorter time to develop ZU-DYNAMO than to develop usual DYNAMO, because the code generation is easier in ZU-DYNAMO than in usual DYNAMO. In ZU-DYNAMO, the objective code is C language code, and the equation cards are simular to the assignments in C language. Therefor, the translations from DYNAMO equation cards to C assignments, in fact, are few modifications. This is easy. But, in usual DYNAMO, objective code is assemble language, and DYNAMO language is different from assemble language. One equation card may be translated into many instructions in assemble language. As a result, these translations are very difficult. On the other hand, ZU-DYNAMO can be easily installed on various types of computer, but usual DYNAMO can not. In a word, to develop a general-DYNAMO compiler system has become a new important research project in the field of System Dynamics. This paper only deals with some aspects of problems in the development of general DYNAMO compiler system.

Although ZU-DYNAMO is running very well, some functions should be extended in the future, such as the acceptance of diagram input. We intend to provide this extension of diagram input. Finally, we thank Mr. Weiqiong Wang, Mr. Hao Chen and Mr. Baoyi Tong helpful comments and works in the development of the system.

REFERENCES

1. Forrester, Jay W. (1961) Industrial Dynamics, MIT Press.
2. Richardson, G.P. and A.L. Pugh III (1981) Introduction to System Dynamics Modeling with Dynamo, MIT Press.
3. A.L. Pugh III (1976) Dynamo User's Manual. MIT Press.
4. Philip M. Lewis II (1978) Compiler Design Theory, Addison-Wesley Publishing Company.