KYBERNETIKA — VOLUME 9 (1973), NUMBER 2

Programming Means for Simulation of Logical Networks III

EVŽEN KINDLER

The present paper is the last part of a serie, containing the information which could lead the non-computer-oriented investigators in logics and neurophysiology to simulate their subjects on computers. This part is oriented to the third generation computers: it presents how one can introduce the conceptions of neurons and of the logical network into the third generation languages and how the users can call the interpretation of such conceptions for the pourpose of simulation. The examples are formulated in SIMULA 67. Beside that information the present part contains the conclusions corresponding to all the three parts.

5. INTRODUCING OF FACILITIES FOR SIMULATION OF LOGICAL NETWORKS INTO SOFTWARE OF THIRD GENERATION COMPUTERS

5.1. The computers of the third generation form the last type of computers which can be seriously considered in the time when this paper is written. During the development of the computers in the last decade of years certain influences (new demands and new discoveries) have caused that the type of the second generation has become unmodern and unprogressive. Although the third generation of the computers exists for a time which is relatively small (about six years against 10 years for the second generation), one can consider the type of the third generation already nowadays: a certain property of the third generation computers that they solve the contradiction between the facilities of the second generation computers and the advantages which were present in the computers of the first generation but have dissapear in the second generation (see further), seems to assure the type of the third generation to be actual still a few future years.

The third generation of computers have the following typical properies: the electronical circuits of the computers are based on integrated elements; the memory is organized in several levels, regarding the time of writing or reading an information (access time): there is a small memory with access time of nanoseconds, then there

is an internal memory with access time of about one microsecond (realized usually as a monolitical memory), there is an external memory "with random access" and a slow external memory. The random access memory is usually realized as a magnetic disc unit (the access time for one case does not depend on the preceding access), while the slow external memory is formed by several magnetic tape as in the second generation (it is not a random access one because every access time depends on the preceding access). The capacity of the monolitical memory is about ten times greater that of the core memory in the second generation (almost 100 000 words). The superfast memory working in nanosecond time can store about 200 words. The random access external memory can contain millions of words while the magnetic tape one has its capacity depending on the length of the tape and on the number of units (ordinarily its capacity is more than 2 mililons of words). The memory is organized (structured) also in logical aspects differently as in the preceding generations: the word has about 50 bits but it can be divided in two half-words, each of which can be devided in two quarter; every quarter can be divided in bytes which correspond to alphanumerical symbols and can be divided into bits. The words can be composed to pages, the pages can be grouped to chapters. The computers have wired in their hardware the facilities for operation with all the terms of the described hierarchy.

The third generation computers are facilitated by time shearing and by multiprogramming. It permits parallel work of independent programs in the same computer. If one program is interrupted (e.g. as it must halt until the magnetic disc memory gives it an information) another program which is prepared runs in the fast elecronic units so that these units – the most expansive parts of the computer – are well used in every microsecond. If a program is prepared to go on never a situation becomes that it might halt for a long time if another program works without interruption: the time shearing circuits cause that every program can access its run in a short time (e.g. 20 milliseconds) if it signalizes that it is prepared.

The time shearing admits that the computer can have a great number (hundreds) of input/output units. The typical unit of that type for the third generation is the *terminal*. It is a simple tool which can accept the information from a human and send them to the computer and which can receive information from the computer and display them to be acceptable for humans. Thus the terminals use to be electrical typewriters or electronic displays with a keyboard, eventually with a joined simple reader of a paper medium (tape, cards). The connection between the terminals and the computers can be thousands km long and it is often realized through standard telefone networks. The operating systems of the computers of the third generation have facilities for contacts between the terminals and the computer, specially those which admit storing of various informations in the disc memory even for many days.

5.2. The programming languages of the third generation are designed to be *universal*. Of course a certain universality of programming languages has existed in the first generation algorithmic languages: in an algorithmic language one can

describe any computing process which can be realized in an automatic computer. The concept of universality of the third generation languages is rather more general: they would admit to introduce any consistent expression means by which we should like to describe the computing processes. The solution of that target has been solved by various languages which use diverse technical means (see e.g. [23], [24] and [25], eventually by the discussion in [12] concerning the subjects of the last two mentioned publications), but the main principle is always the same. The translator from such a third generation programming language is very complicated but if it has been realized introduction of any class of expression means (for example of those for simulation of certain systems) need no special translator. Moreover their introduction can be done by means of other suitable means introducted before even in case that these have not been realized in the original third generation language.

The facilities of the third generation language are not commonly known as e.g. those of the algorithmical ones. Thus it would not be suitable to write about a general language of the third generation computers in the next text, as the conclusions would be very unclear and vague. Therefore we have determined to express all the considerations in the language SIMULA 67 which is a typical one for the third generation family of the programming languages; moreover it has been implemented for various computers and it is an enlargement of the commonly known algorithmic language ALGOL 60. Our considerations can be translated for other third generation language simply according to their meaning.

5.3. To realize the described process enriching the programming language needs in the language SIMULA 67 (see [26]) two levels. In the first level the computer reflects definition of a new concept; in the second one the computer reflects an interpretation of the concept in a certain subject. We see the levels correspond to similar levels in logics and noetics but in the third generation languages there are all the affairs more simple because the computer as well the language are not able of the reflection of themselves: the definitions and the interpretations thus do not concern themselves and the obstacles with diagonal processes etc. because always there is a metalanguage in which the basic programming language is described, unaccessible by the same programming language. The definition of a new concept is called the *declaration of a new class* in the terminology of SIMULA 67 because it is understood so that any interpretations of the concept form a class described by certain programming facilities (see further). But there is no list or set of the interpretations of the concept: in the third generation languages there are the facilities to handle with the sets but it is not dependent at the class notion (see further). To use the name class has only meaning for the imagination, regarding to the implemention in the computer it would be better called concept. The class can be defined „from nothing" or by a specialization of a class declared before. The first type of declaration is a limit case of the second one as it can be considered that the declared class is a specialialization of a class of all the subjects (it reflects the concept of the entity

with no properties – as there is no possibility to do a diagonal process it leads to no contradictions). If there is a class A declared before and if B is a new class risen as a specialization of A we say that B is a subclass of A. The declaration of B has the following form in SIMULA 67:

A **class** *B parameters; the description of the parameters;*
begin *program pattern* **end**

The further details about all the facilities of such declarations are presented in [26] but also in [25] and [27].

5.4. If the programmer wish to form an „interpretation" of a „concept" introduced by a class declaration, he must use an expression **new** A where A is the name of the concept (the same, introduced after the word class in the class declaration defining the same concept. The mentioned expression can occur in situations which are similar to those of occuring of arithmetic expressions (see more detailly also [25], [26], [27]). Specially we can let the mentioned expression assign (more exactly: the reference value of the mentioned expression) to a reference variable (which is e.g. B); the same process can be explicated that we can assign a new name B to the "created" interpretation of the concept A. Thus we must introduce before the convention that B means a name for an interpretation of A by a declaration **ref** (A) B which is analogue to any type declaration of ALGOL 60 (see e.g. [6]) and the new interpretation of the concept A with its name B is performed by a statement $B :- :-$ **new** A where the sign $:-$ is analogue to the assignment sign $:=$ of ALGOL 60. By the declaration **ref** (A) B no interpretation is created!

5.5. There are introduced a priori certain classes which meet the programmer often. Thus SIMULA 67 posess the classes for handling with sets and with processes. The means are introduced exactly in [26] for both the types of handling while in [25] and [27] there are recognizable certain precessors of the introduced classes in the development of SIMULA 67. We need not to explain the introduced facilities while the intuitive meaning is clear from the phrases which their application forms and the exact description is presented in the mentioned publications (it is too long as to be presented in this paper). Let us only mention that the space has no meaning and no importance in SIMULA 67 while the space as a delimiter between two words is in the same language replaced by a point.

5.6. In order to introduce a neuron and a logical network we must introduce their classes. But it is suitable to declare before it also the class of signals which go through the logical network. It is done by the following declaration:

link **class** *signal*; **begin integer** *information* **end**;

where the class *link* is the class of objects which can be elements of sets. The attribute *information* denotes the proper value which can be carried by the instants (inter-

pretations) of the declared concept. If A is a name of such an instant we can identify its information by an expression *A.information* where the point is to be read as a space or a saxon genitive postposition. Now we can define the class of neurons:

```
process class neuron (delay); boolean delay;
  begin ref (signal) output, auxiliary output; ref (head) input;
          input :— new head; output :— new signal;
          auxiliary output :— new signal;
    passivate; cycle: inner; if ¬ delay then
        output. information := auxiliary output. information;
        hold (1); output. information := auxiliary output. information;
        go to cycle
    end neuron;
```

the identifier *delay* means a logical value which is either **true** or **false** according to the determination whether the neuron is with delay or without it. The action determining the real function of the neuron is covered in this general declaration by the statement **inner**. Every neuron has its attributes *output* (i.e. its output signal), *input* (i.e. a set of input signals) and — for the further specializations — the attribute *auxiliary output*, with which the user does not meet. Let us introduce special neurons:

```
neuron class negation;
        inspect input. first when neuron signal then
        auxiliary output. information := 1—information;
neuron class identical; inspect input. first when neuron signal
  then auxiliary output. information := information;
neuron class constant 1;
        auxiliary output. information := 1
neuron class constant 0;
        auxiliary output. information := 0;
neuron class input neuron;
        begin < instructions for reading the value which is
        assigned to the auxiliary output. information > end;
neuron class binary neuron;
        begin inspect input. first when neuron signal
                  do X1 := information;
              inspect input. first. suc when neuron signal
                  do X2 := information;
        end;
binary neuron class conjunction;
        begin auxiliary output. information := X1 * X2 end;
binary neuron class disjunction;
        begin auxiliary output. information := X1 + X2 — X1 * X2 end;
```

the declarations of other types of neurons differ only by other operations in the assignment (for auxiliary output. information) see the paragraph 2.6. One can introduce also the printing units which are considered to be joined to outputs from certain neurons; they can be declared in the following form:

```
neuron class print;
  begin < instructions for printing of the input > end;
```

In order to establish the whole logical network we must declare the *clock* of the network as a process which waits the described time (T) and then stops the work of the network:

```
process class clock(T); real T; begin hold(T); stop end;
```

To introduce the general concept of *logical networks* we can do the following declaration:

```
head class logical network; virtual: procedure reorder;
  begin ref (neuron) P; inner; reorder;
  for P :— first, P. suc while P none do activate P; end;
```

The procedure *reorder* can be redeclared according to eventual demands for the ordering of the simulated neurons. It is described in the following part, while for our considerations we can assume the procedure as an empty one. The statement **inner** represents the forming of the simulated system. Thus we can present the description of the logical network mentioned in the paragraph 2.11 (see fig. 3) as an example:

```
logical network class example;
begin ref (neuron) G8, D1, E2, E3, D4, D6, E5, E7; P;
G8 :— new input neuron (false); E7 :— new negation (false);
D4. output. into (E7. input); D1 :— new conjunction (true);
G8. output. into (D1. input); E7. output. into (D1. input);
E5 :— new negation (false); D6. output. into (E5. input);
D6 :— new identical (true); D1. output. into (D6. input);
E2 :— new negation (false); D1. output. into (E2. input);
E3 :— new disjunction (false); E2. output. into (E3. input);
E5. output. into (E3. input); D4 :— new conjunction mult (true);
G8. output. into (D4. input); E3. output. into (D4. input);
E5. output. into (D4. input);
for P :— G8, D1, E2, E3, D4, D6, E5, E7, new print do P. into;
P :— this example. first;
inspect last when neuron do
for P :— P, P. suc while P =/= this neuron do P. output. into (input);
new clock (T). into (this example)
end example;
```

Let us mention that the class *conjunction mult* which performs the conjuction of more input informations can be declared in the following way without previous determination of the number of terms in the conjunction:

```
neuron class conjunction mult;
begin auxiliary output. information := 1;
for K :— input. first, K. suc while K =/= none do inspect K
when neuron signal do this conjunction mult. auxiliary
```

output. information := **this** *conjuction mult. auxiliary*
output. information × *information* **end**;

Concerning the types of the ordering we can only state that one can respect both the types without any distinguishing.

6. APPENDICES

6.1. There are various possibilities of other ordering of the printing regarding to the other paragrapgs of the description of the models. It is possible to introduce general rules for the situations which can be logically meaningful in the ordering of the paragraphs. We have not introduced it in the present paper as the fine details concerning the results are without importance for the non-computer-oriented users; moreover the fine details can be expressed only for the second generation languages: in the first generation ones they degenerate into a trivial analyzis of the sequence of the body of the simulation (the instruction is performed before another one iff it is written before it), while in the third generation languages the description needs to know profoundly the exact definitions of the class SIMULATION of the language SIMULA 67, which implies that the conclusions cannot be generalized for the other third generation languages.

6.2. One can imaginate that beside the declarations of neurons in the third generation (or beside the rules for program patterns in the first generation and beside the rules for the simulation language and its translation in the second generation) we can formulate the declarations (the rules) which enable to simulate the systems composed not only of neurons but also of elements joined with their outputs but performing simple statistical processing of the comming information. The principle would be the same only the arithmetical operations would be other, giving the result not as zero or one but as a number (real or integer).

6.3. The initial conditions in the simulated logical networks can be introduced in the section before the main simulation cycle (in the third generation languages: after generating the neurons, i.e. before or after generating the clock). We have formulate no general rules for them because from the view point of applications the most interesting simulations concern the system which establish themselves the initial conditions by a certain phasis of the simulation when uniform values at the input come.

6.4. We have introduced the concepts of the first and the second type of ordering (see 2.4 and 2.5). Though the rules of them can be satisfied manually without complications it would be possible to program so that the description of the system would be given without satisfying any rules of ordering and the computer would order the simulated system itself. In the second generation languages it means that the description must be translated into another description before translating into an algorithmic language. In the third generation the ordering can be performed

as a component of the initial actions of the simulation (see the procedure *reorder* in 5.6). In the first generation languages it is difficult to present any method if the program for the simulation is not specially labeled so that it has formally a form of the second generation one. We shall present one algorithm for reordering the set of neurons so that they satisfy the rules of the first type of ordering (and thus automatically those of the first type of ordering). The algorithm is written in SIMULA 67 because we can have use of the concepts introduced in the preceding part (namely the attributes of the neurons, the ordering of the neuron in the logical network), because we can have use of the facilities for ordered sets handling, which have been built in the SIMULA classes, and because the described algorithm is prepared as the body of the mentioned procedure *reorder*.


```
begin ref (neuron) P, Q; ref (signal) S;
P :- first;
L: if P. delay then
   begin if P. suc =/= none then
            for Q :- P. suc, Q. suc while Q =/= none do
               if ¬ Q. input. empty then
                  for S :- Q. input. first, S. suc while S =/= none do
                     if S == P. output then
                        begin Q. precede (P); P :- Q; go to L end
   end
   else if  P. prec  =/= none then
            for Q :- P. prec, Q. prec while Q =/= none do
               if ¬ Q. input. empty then
                  for S :- Q. input. first, S. suc while S =/= none do
                     if S == P. output then
                        begin P. precede (Q); go to L end;
P :- P. suc; if P =/= none then go to L;
end reorder;
```

Let us mention that the procedure *reorder* is related to a certain instance of a logical network which is a subclass of the class *head* of sets. To this instance must be related the procedures *first*, *suc* and *prec* if they are presented in the algorithm in connection with *P*.

6.5. The concepts introduced in this paper can serve to describe exactly a certain facility of the programming system COSMO. This facility behaves for the user so that if he is not sure whether the simulation might not have numerical errors he changes the position of a certain key at the computer desc and lets the computer repeat the simulation. If the original results do not rather differ from the last ones the simulation can be accepted as exact (see [28]). It is possible to be realized as the simulated processes are continuous but the substance of the realization cannot be simply described by means of compartmental system theory and by means of the programming languages as well. In our terminology we can describe it exactly and simply by the following way: one position of the mentioned key causes that the compartments are processed as automata ordered as in the first type of ordering.

The second position of the key causes that the compartments are processed as automata with delay ordered in the second type of ordering. This formulation illustrate a certain practical value of the formulated relations.

6.6. In the paragraph 2.10 we have mentioned the possibility of using procedure and function facilities in the first generation languages when applying them to simulation of logical networks. If we transfer the results presented there for the second generation simulation languages we can formulate the following general rules: if the algorithmic language to which the compiler translates has facilities of procedures but does not admit the parameters the possibility of composed program pattern is more suitable (see par. 3.8), while if the algorithmic language admits the procedures with parameters the simple program patterns are more suitable. In the last case the program pattern form the line

2E/5 (A3, A4)

is translated into the following program pattern

DISJUNCTION (A3, A4, A2)

and the procedure *DISJUNCTION* is declared in the following form:

PROCEDURE DISJUNCTION X, Y, Z
$Z = X + Y - X . Y$

while in the case of procedures without parameters the program pattern generated from the same line is

$X2 = A3$
$X3 = A4$
DISJUNCTION
$A2 = X1$

and the procedure *DISJUNCTION* is declared in the following form:

PROCEDURE DISJUNCTION
$X1 = X2 + X3 - X2.X3$

In the third generation languages the use of procedures has no importance because it is better to declare an element for simulation than an element of programming.

In the third generation languages and in a lot of the languages of the preceding generations there are boolean operations and types. We have not has use of these facilities in order to present properties of programming which would be the most general ones, adequately to the size of that paper. To transform the presented considerations so that we apply the logical or the boolean operations corresponds only to great simplifying of certain special cases of logical (boolean) operations considered in the present paper.

(Received February 24, 1972.)

[1] E. Kindler: Basic facilities of application of mathematical modelling in life sciences. Acta Univ. Carol. medica. *15* (1969).
[2] V. Dupač, J. Hájek: Pravděpodobnost ve vědě a technice. NČSAV, Praha 1962.
[3] S. C. Kleene: Representation of events in neuron networks and finite automata. Automata studies, Princeton 1956. (Russian translation: Avtomaty, Moscow 1956.)
[4] A. T. Bharucha - Reid: Elements of the theory of Markov processes and their applications. New York, Toronto 1960.
[5] Stroje na zpracování informací (Information processing machines), No. 1, Prague 1953.
[6] J. W. Backus a kol.: Programování v jazyku ALGOL. Praha SNTL 1963.
[7] J. Szczepkowicz: Autokód MOST 1. Kancelářské stroje, Hradec Králové 1966.
[8] Autokód pro samočinný počítač NE 803. Kancelářské stroje, Praha 1963.
[9] O.-J. Dahl, K. Nygaard: SIMULA, a language for programming and description of discrete event systems. Introduction and user's manual. 5th edition. Oslo 1967.
[10] V. Černý, J. Půr: Příručka programátora samočinného počítače ODRA 1013. Kancelářské stroje, Hradec Králové 1967.
[11] O.-J. Dahl: Discrete event simulation languages. Norsk Regnesentralan, Oslo 1967.
[12] Simulation programming languages. Proceedings of the IFIP working conference on simulation programming languages, Oslo 1967, North—Holland Publishing Co., Amsterdam 1968.
[13] R. D. Brennan: Continuous system modeling programs state-of-the-art and prospectus for development. In [12], 371—394.
[14] System/360 continuous system modeling program User's Manual. IBM 1967.
[15] E. Kindler: COSMO (Compartmental system modelling), description of a programming system. In [12], 402—424.
[16] E. Kindler: Simulation System COSMO — description of its language and compiler. Kybernetika *5* (1969), 4, 287—312.
[17] E. Kindler: Computer software for modelling of compartmental systems. In: Computers in radiology — proceedings of the international meeting, Brussels 1969. S. Karger, Basel, München, Paris, New York 1970, pp. 444—447.
[18] E. Kindler: Styk lékaře s počítačem. Lékař a technika *1* (1970), 1, 10—11.
[19] E. Kindler: Použití kompartmentových systémů k modelování živých organismů. Biologické listy *35* (1970), *3*, 161—166.
[20] E. Kindler: Automatic modelling of compartmental systems. In: Information processing 68. North—Holland Publishing Co., Amsterdam 1969, pp. 1502—1505.
[21] C. W. Sheppard: Basic principles of tracer method. J. Wiley, New York, London 1962.
[22] A. Resceigno, G. Segre: La cinetica dei farmaci e dei traccianti radioattivi. Edit. Univ. Boringheri, Torino 1961.
[23] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Coster: Draft report on the algorithmic language ALGOL 68. Mathematisch centrum, Amsterdam 1968.
[24] J. V. Garwick: Do we need all these languages? In [12], pp. 143—155.
[25] O.-J. Dahl, K. Nygaard: Class and subclass declaration. In [12], pp. 158—171.
[26] O.-J. Dahl, Bjorn Myhrhang, K. Nygaard: SIMULA 67 Common base language. Norsk Regnecentralen, Oslo 1968.
[27] O.-J. Dahl, K. Nygaard: SIMULA 67 common base definition. Norsk Regnecentralen, Oslo 1967.
[28] E. Kindler: MINICOSMO — universal generator of steady state compartmental system model. Acta Univ. Carol. medica *16* (1970), 3/4, 281—294.

PhDr. RNDr. Evžen Kindler, CSc., Biofyzikální ústav Fakulty všeobecného lékařství Karlovy university (Biophysical Institute, Faculty of General Medicine, Charles University), Salmovská 3, 120 00 Praha 2. Czechoslovakia.