ABSTRACT .................................................................................................................................................. 1

1. INTRODUCTION .................................................................................................................................. 2

2. MODIFIED E-NETS ............................................................................................................................... 4
   2.1 Definition and general features ....................................................................................................... 4
   2.2 Elementary net of type T .............................................................................................................. 8
   2.3 Elementary net of type Y .............................................................................................................. 9
   2.4 Elementary net of type X .............................................................................................................. 9
   2.5 Elementary net of type G ........................................................................................................... 10
   2.6 Elementary net of type I ........................................................................................................... 10

3. MODEL DESCRIPTION LANGUAGE ................................................................................................. 12
   3.1 General features of MDL ............................................................................................................. 12
   3.2 Variables and reserved words ...................................................................................................... 13
   3.3 Statements for the description of a segment .................................................................................. 14
   3.4 Referencing transitions, places, token attributes, and SNAs ..................................................... 18
   3.5 Statements ATTACH and LINK for multi-segment models ..................................................... 21
   3.6 Pascal section in an MDL segment ............................................................................................... 22

4. MODELING CONTROL LANGUAGE .................................................................................................. 25
   4.1 Statement FOR ......................................................................................................................... 25
   4.2 Statement MARK .................................................................................................................. 26
   4.3 Statement DEMARK ............................................................................................................ 27
   4.4 Statement SET ....................................................................................................................... 28
   4.5 Statement STATISTICS .......................................................................................................... 29
   4.6 Statement HISTO .................................................................................................................... 30
   4.7 Statement TRACE ................................................................................................................... 32
   4.8 Statement STOP ....................................................................................................................... 34

5. METHODOLOGY OF MODELING AND SIMULATION WITH E-NETS ........................................... 35

6. EXAMPLES OF SIMPLE E-NET MODELS ......................................................................................... 39

7. CONSTRAINTS IN THE IMPLEMENTATION OF MDL AND MCL ................................................... 59

8. INSTALLATION AND USE .................................................................................................................... 61
   8.1 Installation .............................................................................................................................. 61
   8.2 Steps in the use of WINSIM ..................................................................................................... 62

APPENDICES ............................................................................................................................................. 69

APPENDIX 1 ........................................................................................................................................... 70
   SYNTAX OF THE MODEL DESCRIPTION LANGUAGE ................................................................ 70
APPENDIX 2 ........................................................................................................................................... 75
   SYNTAX OF MODELING CONTROL LANGUAGE ........................................................................ 75
APPENDIX 3 ........................................................................................................................................... 79
   EMBEDDED FUNCTIONS OF MDL ................................................................................................. 79
ABSTRACT

A simulation system Winsim based on the use of modified E-nets is described. E-nets are a specific class of extended Petri nets. Possessing the computing power of a Turing machine, E-nets, at the same time, have all the power of original Petri nets to represent the parallelism of events and processes in modeled systems. The document presents the basic features and formalism of extended E-nets. It includes the model description language (MDL) which is used to describe simulation models in terms of extended Petri nets. The modeling control language (MCL) is also described which is used at the stage of simulation to set an initial state of the model and to watch and control simulation runs. A large number of simple models illustrate in detail the features of MDL and explain the use of MCL statements. The manual outlines the constraints in the implementation of MDL and MCL, explains the main steps in the use of the system, and gives, in Appendices, the detailed syntactical description of MDL and MCL. One of the Appendices describes a detailed model of the CSMA/CD-based LAN, with the results of simulation and their comparison with the known analytical modeling.
1. INTRODUCTION

In order to model and analyze concurrent and distributed systems, many different techniques and tools are used today. One of the most popular techniques are Petri nets. Petri nets are a graphical and mathematical tool for the formal description and modeling of the systems whose dynamics is characterized by the inherent concurrency of events and processes, with complex temporal patterns of interprocess interaction, which is typical for many data processing systems, especially for concurrent and distributed ones [1, 2].

A number of software packages were developed for using Petri nets as a mathematical tool for the investigation of structural and behavioral properties of systems being modeled. Among these packages are GreatSPN, DEMON, SPAN, NECON, NET and others. A good survey of such packages up until 1986 is given in [3]. The research in this area still continues, and new sophisticated software tools for the investigation of structural and behavioral properties of Petri net based models were designed during last ten years, including MOBY [4], PROD [5], Product Net [6], QPN-Tool [7], GreatSPN [8], DSPNexpress [9], UltraSAN [10], XsimNet [11], to mention only some.

Being quite powerful as a mathematical tool, the original Petri nets are not suitable for the simulation purpose which is too often necessary for performance evaluation of data processing systems. The reason is that Petri nets are not an algorithmic system (i.e. they do not allow to represent the Turing machine) [1].

There are modifications and extensions of Petri nets which are supposed to overcome some shortcomings of the original Petri nets as a simulation tool. One of the directions for the modification of Petri nets is related to the notion of time which is absent from their definition. But for Petri nets to be used as powerful models of real data processing systems, timing information might provide a new important feature, even though this addition is not consistent with the basic philosophy of the original Petri nets.

One class of extended Petri nets is evaluation nets, or E-nets. The initial form of these nets, proposed first by G.Nutt [12], included not only timing information, but also the concepts of attributed (or valued) tokens, control functions, and transformation of token attributes. These features allow to express in terms of E-nets the four fundamental processes in information systems - information transfer, time (delays), data transformation and control - and make them very powerful and effective apparatus for the formalized description and simulation of systems. At the same time, E-nets have all the power of original Petri nets to represent the parallelism of events and processes in modeled systems and can be used for formal analysis of systems with some known techniques of Petri net theory. According to J.Noë, "one usually needs some mix of modeling levels that allows one to concentrate upon details in a given subsection while still maintaining grasp of the interaction between this subsection and the remainder of the system" [13]. Actually, E-nets meet this need in a very balanced form.

More elaborate form of E-nets had the macro capabilities which simplified the description of models [14]. In subsequent works of J.Noë and other researchers, the general
aspects of the model building and using E-nets for the formalized description and simulation of a number data processing systems were investigated, including transaction-oriented systems, operating systems and hardware components. But software tools, developed since then for the application of E-nets, included mainly simple graphic editors for the preparation and printing of E-net models [15 - 16].

Our research in the area of E-nets began in 1977. Starting with E-nets proposed by G.Nutt, we developed a new form called modified E-nets. In comparison with the initial form of E-nets proposed by G.Nutt, in which five types of the elementary nets were allowed, with strictly limited number of input and output places, we generalized the set of the basic types of the elementary nets and allowed for them an unlimited number of input and output places. Second, in addition to simple places, we provided also so called queue places (places which may hold an arbitrary number of tokens) which could be used freely everywhere in models instead of simple places. And third, we introduced a completely new type of an elementary net, the interruptible net, which is very essential for the modeling and simulation of many systems, especially concurrent and distributed ones. All these features provide the increased efficiency of the modified E-nets for simulation purposes and make them a convenient and easy-to-use modeling tool which combines the advantages of the original Petri nets with those of high level programming languages.

With the use of modified E-nets, the first interactive simulation system DSIM was implemented in 1980 on the IBM System/360 computer [17]. DSIM had an original E-net based language for model representation and operated in an interpretative mode. DSIM was extensively used for investigation of a number of E-net models of systems such as a multiuser interactive system, multiprocessor systems with a common bus, a data flow system, a job dispatcher in a distributed system, communications protocols, a multiprocessor system for numerical control, local area networks and so on.

On the base of the subsequent investigations of the formal aspects of the modified E-nets [18, 19], the second version of the E-net based simulation system Microsim was designed and implemented on DEC’s PDP and then on IBM PC computers [29]. The system provided the direct execution of a precompiled code of simulation programs. Recently, the system has been adapted and extended under the name Winsim for running in environments of MS Windows 95/98/NT/2000/XP operating systems.

This document has three goals. First, it explains the basic formalism of the modified E-nets as a class of extended Petri nets. Second, it presents the model description language (MDL) and the modeling control language (MCL). And third, it demonstrates the application of the modified E-nets for simulation by examples which help understand all steps in creation of models of information systems.
2. MODIFIED E-NETS

2.1 Definition and general features

The minimal structural elements in modified E-nets, as in Petri nets, are places, transitions and directed arcs which connect places and transitions in accordance with the rules of a bipartite directed graph. From the structural point of view, E-nets may be considered as marked graphs, with the relaxed condition that each place may have one input transition or one output transition or both. A good survey of the marked graphs is given in [2].

The number of arcs which may be incident to a transition is theoretically unlimited, with one exception which will be considered later in this section.

There are two types of places in modified E-nets - simple places and queue places. A simple place can hold at most one token at a time. A queue place has theoretically unlimited capacity and hence it can hold any number of tokens at any moment of time. In a graphical representation, simple places and queue places are depicted by circles and ovals accordingly. In this paper, we will call a simple place by S-place and a queue place by Q-place.

As in Petri nets, the behavior of any E-net is expressed as firing of transitions and subsequent moving of tokens from input places to output places of the fired transitions. In order to consider the underlying processes, we will now introduce the notion of an elementary net as a minimal, functionally complete, structural component of E-nets.

A graph consisting of a transition and associated input and output places is an elementary net pertaining to the given transition. Formally, we define an elementary net $E(t)$ of a transition $t$ as

$$E(t) = < C, P_1, P_2, r_1, r_2, d, m >,$$

where $C$ is a necessary (but generally not sufficient) condition to fire the transition $t$; $P_1$ and $P_2$ are finite sets of input and output places for $t$, with $P_1 \cap P_2 = \emptyset$ and $P_1 \cup P_2 \neq \emptyset$; $r_1$ and $r_2$ are functions of input selection and output selection respectively; $d$ is a delay function; $m$ is a data, or memory, transformation function. Consider all the components of the definition in more detail.

The condition $C$ can be represented by a logical expression with components denoting the presence or absence of tokens in input and output places of the transition $t$. For example, if an elementary net has two input places $x_1$ and $x_2$ and one output place $y$ then we might demand that, for this elementary net,

$$C = B(x_1) \land B(x_2) \land \neg B(y),$$

where $B(x)$ is a predicate which has the true value if the place $x$ contains at least one token. Here '$\land$' and '$\neg$' are logical operators of conjunction and negation.
A prerequisite to fire a transition $t$ is the true value of $C$. In our example, this corresponds to the presence of a token in places $x_1$ and $x_2$ and the absence of a token in place $y$. It is supposed that firing a transition will generally require some nonzero time which we call the activity interval of the transition.

The full definition of sets $P_1$ and $P_2$ implies an enumeration of corresponding places, denoting the type (S or Q) and dimensionality for each place. The place dimensionality is the number of numerical attributes a token may have in the place. Therefore, tokens in E-nets are considered as attributed entities which is very important for E-nets to be not only an apparatus for the formalized system description but also a powerful simulation language. We will consider this aspect of E-nets in more detail later in this section.

The function $r_1$ defines the subset of $P_1$ of those input places from which, at the end of activity interval of a fired transition $t$, a token will be removed. The function $r_2$ defines the subset of $P_2$ of those output places into which, at the end of activity interval of a fired transition $t$, a token will be added.

The delay function $d$ may be formally defined as $d: T \rightarrow Z$, where $Z$ is a set of nonnegative real numbers. This function calculates the length of the activity interval of the transition $t \in T$ when it fires. At the end of this interval, tokens will be removed from some input places and added into some output places of $t$ according to the values of the functions $r_1$ and $r_2$.

At last, the function $m$ is defined as $m: \Delta \rightarrow \Delta$, where $\Delta$ is the power set of a set of data elements (or memory slots). This function provides a data transformation that has to be carried out after finishing the activity interval of the fired transition and moving tokens from some input places $s$ to some output places. The transformation may involve token attributes. In our E-net model description language which will be described in the next section, we have provided for the function $m$ the possibility to operate not only on the token attributes, but also on memory variables in order to have all the power of an ordinary algorithmic language to process data.

Now one can see, that the four fundamental system processes - information transfer, control, time, and data transformation - are clearly expressed in an elementary net by token "movement" from the input places to the output ones after firing the transition, functions $r_1$ and $r_2$ (for the control), $d$, and $m$ respectively.

There are many possible structures of elementary nets. It was proved in [19] that, to model any data processing system in terms of E-nets, it is sufficient to have the basic set of types of elementary nets given in Fig. 2.1. In the proof, it was shown that the known fundamental constructs which are sufficient for representing any algorithm [20], can be expressed in terms of the proposed basic set of elementary nets. It was shown also that all the constructs of the structured programming [21] and parallel programming [22] are expressible in terms of this set. Therefore, without going into theoretical details, we may state that, for the area of data processing systems, the proposed basic set is functionally complete, even though it is not minimal, as we will see later.
In Fig. 2.1, for each type of an elementary net, its name (T, Y, X, G, or I) and the graphical representation (scheme) are given. For the definiteness, input and output places for all types of elementary nets are shown as S-places. Each S-place in the figure may be

<table>
<thead>
<tr>
<th>Type of net</th>
<th>Graphical representation of net</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong></td>
<td><img src="image" alt="Graph T" /></td>
</tr>
<tr>
<td></td>
<td>$x_1$</td>
</tr>
<tr>
<td></td>
<td>$x_2$</td>
</tr>
<tr>
<td></td>
<td>$x_m$</td>
</tr>
<tr>
<td><strong>Y</strong></td>
<td><img src="image" alt="Graph Y" /></td>
</tr>
<tr>
<td></td>
<td>$x_1$</td>
</tr>
<tr>
<td></td>
<td>$x_2$</td>
</tr>
<tr>
<td></td>
<td>$x_m$</td>
</tr>
<tr>
<td><strong>X</strong></td>
<td><img src="image" alt="Graph X" /></td>
</tr>
<tr>
<td></td>
<td>$x_1$</td>
</tr>
<tr>
<td></td>
<td>$x_2$</td>
</tr>
<tr>
<td></td>
<td>$x_m$</td>
</tr>
<tr>
<td><strong>G</strong></td>
<td><img src="image" alt="Graph G" /></td>
</tr>
<tr>
<td></td>
<td>$x_1$</td>
</tr>
<tr>
<td></td>
<td>$x_2$</td>
</tr>
<tr>
<td></td>
<td>$x_m$</td>
</tr>
<tr>
<td><strong>I</strong></td>
<td><img src="image" alt="Graph I" /></td>
</tr>
<tr>
<td></td>
<td>$x_1$</td>
</tr>
<tr>
<td></td>
<td>$x_2$</td>
</tr>
</tbody>
</table>

Fig. 2.1. The basic set of types of elementary nets.
replaced by a Q-place but in this case the firing condition \( C \) for the corresponding elementary net can change. This change affects only the output places: if such a place is a Q-place, then its state (i.e. the number of tokens in the place) does not influence the condition \( C \) of the corresponding transition and must be excluded from the expression for \( C \).

Connecting elementary nets from the proposed basic set with each other, we can build E-nets of any size and complexity. The points to connect elementary nets are places. When we connect two elementary nets, at least one output place of the first net has to be united with some input place of the second (or even the same) net. Both places must be of the same type (S or Q), and the type of the united place coincides with the type of the places to be united.

Two nets may be connected to each other through many pairs of places, and an elementary net may be connected with many other elementary nets via many pairs of places. In particular, as it was mentioned, output places of an elementary net may be united with some input places of the same net. The result is a non-elementary net with the decreased number of places.

Let us discuss in more detail some aspects of token attributes. Tokens exist only in places, and the values of token attributes are stored in the place which is occupied by the corresponding token. For this purpose, each place in an E-net may be assigned a number of memory slots with the same or different data types. For a place \( x \), holding some token, the current contents of the slots may be conveniently represented as a finite ordered set of values \( V(x) = [v_1(x), v_2(x), ..., v_s(x)] \), where \( v_i(x) \) is the value of the \( i \)-th attribute of the token, and \( s = s(x) \) is the number of slots, or the dimensionality, of this place. It must be stressed that the values \( v_i(x) \) may be of different types, so that \( V(x) \) generally represents a record associated with the place \( x \).

Each S-place may have only one set of memory slots \( V(x) \) for the attributes of one token in that place, whereas for a Q-place the number of sets of slots (i.e. the number of records \( V(x) \)) is theoretically unlimited.

It is quite possible to copy token attributes from one place (source) to another place (destination). It is assumed, of course, that both places hold a token. Actually this is the only way for tokens to carry their attributes when moving in an E-net. There is a tricky point in it. Namely, if the dimensionality of the source place is greater than the dimensionality of the target place, then the redundant attributes of the token from the source place should be discarded (or cut) in the destination place. If, on the other hand, the dimensionality of the destination place is greater than the one of the source place, then the redundant slots of the destination place should be initialized to some predefined values (for example, to zeroes for numerical attributes). We will see in Section 5 that, in our E-net based system, all places of the same model (or, to be more exact, of the same segment of a model) must have the same dimensionality. The reason for this is the desire to simplify the system implementation.

Consider in informal way the semantics of firing a transition in E-nets. As in classical Petri nets, firing a transition in an E-net models some event, and the distribution of tokens among places represents a state of the net. In E-nets, all transitions are timed, even though the activity interval may be zero for some firing transitions. The duration of the activity interval of a transition is allowed to be deterministic or random, with some probability
distribution. For this purpose we provide, in our system, a collection of random number generators.

Before a transition in an elementary net fires, two actions related to this net are generally required. The first action is checking the condition $C$ for the elementary net. We shall see later that this condition is different for the different types of the elementary nets. The second action involves the evaluation of the function $r_1$, or $r_2$, or both, depending on the type of the elementary net. The result (or results) of the evaluation must be valid and correspond to the current marking of the elementary net. This will be made clear when we consider the particular types of the elementary nets. If both actions are successful, the value of the activity interval is calculated by the function $d$, and the transition starts firing. For the elementary nets of types $T$, $X$, $Y$, and $G$, the transition continues firing during the calculated time interval. In the net of type $I$, the activity interval of the firing transition may be interrupted. At the moment when the transition stops firing, new tokens are created in the output places of the transition according to the value of $r_2$, data transformation takes place according to the function $m$, and some tokens are removed from the input places of the transition according to $r_1$. This finishes all the operations related to the activity of the fired transition. Only after that, the transition may be considered for new firing. This means, that the activities of a transition in E-nets are strictly serial, which corresponds to the single-server firing semantics as it defined in [23].

There are defaults for all functions in the definition of an elementary net. For the function $d$, its default calculates always the zero value. For the other functions, the defaults depend on the type of the elementary net and will be presented in the description of the nets. The defaults are used if the corresponding functions are not given explicitly for the elementary net. This possibility simplifies the creation of E-net models.

Now we are ready to give brief descriptions of all types of the elementary nets shown in Fig. 2.1. It is assumed that all input and output places of the elementary net are S-places. As was mentioned, each S-place of the net may be replaced by a Q-place if necessary. But in this case the condition $C$ becomes more simple in that it should not include the output Q-places. The presence of input Q-places in the elementary net does not change the condition $C$.

For all elementary nets, the default data transformation involves only assigning values to the attributes of new tokens in output places of the net.

### 2.2 Elementary net of type T

This type is intended for joining and/or forking of tokens, along with the data transformation of token attributes and, possibly, other data items. If the net has no input or output places then it can serve as a token source or sink respectively. The necessary condition $C$ to fire a transition in this type of elementary net is the presence of a token in all input places and the absence of tokens in all output S-places (the emptiness is not required for Q-places). This is also the sufficient condition for firing the transition. For this type of an elementary net, the functions $r_1$ and $r_2$ may be only default ones (and have constant values). In accordance with the default functions $r_1$ and $r_2$, at the end of the activity interval of the fired transition, one token is removed from each input place and one token is added into each output place.
The length of the activity interval is determined by the function \( d \) (default or explicitly specified). This is true for all types of elementary nets, with some peculiarity for a net of type I.

The default function \( m \) copies the attribute values of a token from the place \( x_1 \) (see Fig. 1) and assigns these values to corresponding attributes of new tokens in all output places.

2.3 Elementary net of type Y

The net of this type makes the conditional selection of an input place to remove a token from it and adds a token into each output place. To fire a transition, it is necessary (but generally not sufficient) that there is a token in at least one of input places and that all output S-places are empty (i.e. not marked). This corresponds to the true value of the condition \( C \). If the condition \( C \) is true then the function of input selection \( r_1 \) is calculated which returns as a result the identifier of some nonempty input place or undefined. Assume that the function is defined, and its result is a nonempty input place \( x_b \), where \( b \in \{1, 2, ..., m\} \). In this case the transition fires. At the end of its activity interval, a token is removed from the place \( x_b \) and one token is added into each output place. The default transformation function \( m \) assigns the attribute values of the token from \( x_b \) to the corresponding attributes of all new tokens in output places.

If, on the other hand, the function \( r_1 \) is undefined (this happens, in particular, when the selected place \( x_b \) is empty) then the transition does not fire. When this elementary net is a component of some E-net then the function \( r_1 \) is recalculated every time after any other transition fires in the E-net until it becomes defined.

The function \( r_2 \) for this net has always constant value.

Default functions \( r_1 \) and \( r_2 \) are described in Section 3.3.

2.4 Elementary net of type X

The net of this type provides the conditional routing of a token to some output place. The necessary (but generally not sufficient) condition \( C \) for the transition to fire requires that each input place of the net has a token and at least one output place allows a token to be added into it (this may an empty S-place or any Q-place). If the condition \( C \) is true then the function of output selection \( r_2 \) is calculated which returns as a result the identifier of some nonempty output S-place (or any output Q-place) or undefined. Assume that the function is defined, and its result is an output place \( y_b \), where \( b \in \{1, 2, ..., n\} \). In this case the transition fires. At the end of its activity interval, a token is removed from each input place and one token is added into the place \( y_b \). The default transformation function \( m \) assigns the attribute values of the token from \( x_1 \) to the corresponding attributes of the new token in the output place \( y_b \).

If the function \( r_2 \) is undefined (this happens, in particular, when the selected place \( y_b \) is a nonempty S-place) then the transition does not fire. When this elementary net is a component of some E-net then the function \( r_2 \) is recalculated every time after any other
transition fires in the E-net until the result is a legal output place (i.e. an empty S-place or any Q-place).

The function \( r_1 \) for this net has always constant value.

2.5 Elementary net of type G

The net of this type combines the properties of the elementary nets of types Y and X. It permits to simplify the E-net representation of a system in cases when two elementary nets of types Y and X must be connected to each other in such a way that all output places of the net X are all input places of the net X. Clearly, the net of type G represents a switch with many inputs and many outputs.

2.6 Elementary net of type I

The net of this type provides very important possibility to interrupt the activity of its firing transition. In the other types of elementary nets, if the transition started firing then it continues firing during the complete time interval defined by the pre-calculated value of the function \( d \), and no event may stop the activity of the firing transition. The possibility to interrupt the activity interval of a firing transition by an external event is necessary for modeling of interruptible time-outs which are used quite often in communication protocols and in queuing systems with the absolute priorities, to mention only two cases. Without this possibility, many published Petri net models of communication protocols are complicated or unrealistic at all. Our experience shows, that the use of this type is very natural for such tasks.

The net has two input and two output places. The roles of the two input places are different. The place \( x_1 \) is the main input place and the place \( x_2 \) is the interrupting place. To distinguish graphically the two input places, we use an arrow symbol on the transition bar directed to the arc which connects the main input place with the transition of the E-net (see Fig. 2.1).

There are a few cases for firing the transition in this elementary net. First, if a token appears in the main input place and the interrupting place is empty at this moment and continues to be empty then this elementary net behaves like a simple elementary net of type T with the input place \( x_1 \) and the output place \( y_1 \) and with the delay defined by the function \( d \). Second, if a token enters the interrupting place and the main place is empty at this moment, then the net behaves as an elementary net of type T with the input place \( x_2 \) and output place \( y_2 \), with the zero delay. Therefore, in these the two cases, the complete elementary net may be considered as a pair of independent subnets of type T - the main subnet with the places \( x_1, y_1 \), and the interrupting subnet with the places \( x_2, y_2 \). Third, if, during the activity interval of the transition in the main subnet, a token enters the interrupting place and the transition in the interrupting subnet is capable to fire also (it possible only when \( y_2 \) is an empty S-place or just Q-place) then the activity interval of the transition in the main subnet instantly finishes, a token is removed from both input places \( x_1 \) and \( x_2 \), and a token is added to both output places \( y_1 \) and \( y_2 \). This is the interrupting case - the main functional case for this type of elementary nets. In this case the
default transformation function \( m \) assigns the attribute values of a token from \( x_1 \) to the attribute values of the new token in \( y_1 \), and the attribute values of a token from \( x_2 \) to the attribute values of the new token in \( y_2 \).

It must be emphasized that, when the transition in the interrupting subnet fires, its activity interval is always zero. This corresponds to the assumption that the interrupt in this net type is an instant event (this is also true for real systems which can react to an unmasked interrupt signal during one machine instruction). If it is necessary to simulate the time for the interrupt handling, it can be done, in particular, outside of this elementary net. A nonzero activity interval may be associated, in this net, only with the main subnet.

We feel that a few examples of the use of the elementary nets might help in more deep understanding of these nets. But it is more informative to present such examples using the model description language (MDL) which is the topic of the next section. Therefore, we will delay presenting the examples and give them in the next section.
3. MODEL DESCRIPTION LANGUAGE

Model Description Language (MDL) and Modeling Control Language (MCL) are the main tool for interaction of a user with the simulation system during the creation and running of models, represented in the form of extended Petri nets, or E-nets.

MDL gives the possibility to describe an E-net based simulation model and its components as a sequence of statements. The minimal model unit, that can be expressed in MDL and then compiled, is called a segment. In a general case, an E-net model can consist of a number of interconnected segments.

MCL is intended for setting and changing of parameters and data in the compiled model that is ready for execution. It is important to note, that the changes done by MCL statements do not require re-compiling and re-linking of the model. This capability essentially decreases time for carrying out simulation experiments with the compiled model.

Therefore, MDL is used on the stage of creation of E-net models, while MCL is necessary on the stage of running of compiled models.

3.1 General features of MDL

The purpose of the model description language (MDL) in Winsim is to express any E-net model as a set of interconnected segments and to input them into a computer. The concept of a segment in MDL is similar to the concept of a module in an ordinary programming language.

Generally, a complete E-net model consists of one, two or more segments. Each segment is a complete unit of work for the MDL compiler. The information links between segments in an E-net model are provided only by attributed tokens moving from one segment into another via external places chosen for linking the corresponding segments to build the complete model.

If a model consists of a number of segments then they are linked together on the stage of preparation of the executable model. Any global variables and common memory locations for different segments in an E-net model are not provided in MDL. This is consistent with the contemporary view on the organization of open systems and, in particular, with the ideas of the object-oriented design according to which the communication between different modules, or objects, should be carried out only by the use of messages.

MDL has means to describe all the structural aspects of an E-net (elementary nets and their links with each other in a segment, connections between different segments) and the procedural aspects of each segment in an E-net model (all the non-default functions for elementary nets in a segment). Also MDL allows to use local variables and arrays in any segment and to set initial values for them at the compilation stage. Declared variables and arrays in a segment may be used by any function in that segment. Since all links between elementary nets in a segment are specified explicitly, there is no need to order statements in the segment even though this ordering is desirable for better understanding the segment and from the aesthetical point of view.
MDL is implemented as an extension of Object Pascal language and hence has all the power of Pascal to process data which is important for simulation purposes. Elements of the extension are declarations of token attributes and so-called net variables, the descriptions of elementary nets (with all explicitly defined functions in the definition of the corresponding elementary nets), statements for attaching and linking segments.

In each segment, an ordered sequence of named attributes for tokens must be specified by ATTRIBUTE statement. The sequence of attributes is the same for all tokens which can exist in a given segment, but the values of attributes may be different for different tokens (in different places of the net in the segment). Using the same sequence of attributes in a segment simplifies the description of the segment and decreases the time necessary for allocation and de-allocation of memory used by tokens during simulation run.

In different segments of the same E-net model, the sequences of attributes may be different. However, since segments in a model are linked via places, there are some peculiarities in “pumping” of attribute values from one segment into another, linked segment via common places. These peculiarities will be described in Section 3.5.

The attributes of tokens may have integer and real values. All attributes of a token can be considered as components of a homogeneous record associated with the place which is occupied by this token. Any reference to a token attribute contains the name of the place and, after the dot, the desired attribute name. The complete reference begins with the symbol "%" to distinguish it from a reference to an ordinary record component in Pascal. For example, %S100.TIM is the reference to the attribute TIM of a token in the S-place S100.

Before running a ready E-net model, it is quite usual to assign initial values to tokens in some places of some segments of the model dependent on the required initial marking. This is done by the use of the modeling control language (MCL) which is the topic of Section 4.

### 3.2 Variables and reserved words

There are two classes of variables in MDL. The first class consists of usual Pascal variables, which obey the syntactical rules of the Pascal language.

The second class includes so-called net variables. The net variables have the special status. First of all, they may be assigned initial values at the compilation stage which is not allowed for Pascal variables. More important, the names of all net variables in a segment are saved after the compilation and may be used during simulation run of the E-net model to watch and, if necessary, modify the values of these variables. A net variable can be a simple variable or one- or two-dimensional array of integer or real type.

All net variables are declared in a segment by the non-Pascal statement DATA. To reference a net variable, one must prefix the symbol "%" to the identifier of that net variable. In all other respects, the net variables are like usual variables and may be used freely everywhere in functions and procedures of the segment in which they are declared.

Net variables are similar to static variables of the C language for the given segment. Therefore, if a net variable is assigned to a value in one procedure, then this value is available for using in another procedure.

There are a number reserved variables and words in MDL which may be used in each segment. The most often used variables are %DELAY, %IN, and %OUT.
The variable %DELAY is assigned the value equal to the desired duration of the activity interval of the firing transition in an elementary net. This value should be of real type.

The variables %IN and %OUT are assigned the values which specify an input or output route of a token when transition fires. In the basic set of types of elementary nets, %IN and %OUT may be specified only for the types X, Y, and G. Values of variables %IN, %OUT, and %DELAY may be read and written. The use of these variables will be illustrated below.

Identifiers of transitions and places in a segment are not reserved words. For this reason, one may define, in a segment, Pascal variables and net variables with names like Sxxxxx, Txxxxx, or Qxxxxx, where xxxx are decimal digits. However, these names must be unique in the segment. Referencing places and transitions is described in Section 3.4.

### 3.3 Statements for the description of a segment

The description of an E-net segment in MDL begins with the statement SEGMENT and ends with the statement SEGEND.

Between the statements SEGMENT and SEGEND, all the other types of statements may be placed. These are ATTRIBUTES, DATA, NET, TIME, CONTROL, TRANSFORM, ATTACH, LINK and comment statements. The comment statement may also precede the SEGMENT statement.

Statements SEGMENT and SEGEND serve to indicate, for the MDL compiler, the start and end of the segment text. As parameters, SEGMENT must specify the segment name and unit of simulation time. The reserved words for units of time are PSEC, NSEC, MCSEC, MSEC, SEC, MINUTE, HOUR and DAY. For example, the statement

```
SEGMENT DSYS, TICK = MSEC;
```

specifies the segment name DSYS and a simulation time unit equal to a millisecond.

The segment ends with statement SEGEND, with a dot after this statement.

If a model consists of a few segments, the time units may be different in different segments.

**Statement ATTRIBUTES** is used for the declaration of attributes of tokens for given segment. Only integer and real attributes may be used in MDL. For example the statement

```
ATTRIBUTES
    MTYP: INTEGER;
    TIM: REAL;
    PID: INTEGER;
```

declares two attributes MTYP and PID of integer type and an attribute TIM of real type.

Note that attributes are not allowed to be arrays.
Statement **DATA** is intended for declaration of net variables of integer and real type in the segment. The variables may be scalar or one/two dimensional arrays. If necessary, initial values may be assigned to the declared net variables. For example, the statement

```
DATA
  FLAG : INTEGER;
  QMAX /50/: INTEGER;
  QUE: ARRAY [120, 8] OF REAL;
  AMIN /0.2, 0.5/: ARRAY [2] OF REAL;
  PRIZ /5, 2, 1, 10/: ARRAY [4] OF INTEGER;
```

declares a non-initialized scalar integer net variable FLAG, initialized integer net variable QMAX, real-type arrays QUE and ADMIN, and integer-type array PRIZ, with the initial values QMAX = 50, AMIN[1] = 0.2, AMIN[2] = 0.5, PRIZ[1] = 5,… , PRIZ[4] = 10, respectively.

Statement **NET** is used to specify structure of an elementary net. Recall that an elementary net consists of a transition and its input and output places. The statement includes an identifier of the elementary net and identifiers of input and output places. Depending on the type of the elementary net, its identifier is an alpha-numeric string of form \(ad\), where \(a \in \{T, X, Y, G, I\}\) and \(d\) is a nonnegative 16-bit integer. For example, the statement

```
NET X23 : S5, Q2 / S15,  S7,  S123;
```

specifies an elementary net of type X with the identifier X23, input places S5 and Q2, and output places S15, S7 and S123. Here S5, S7 and S123 are simple places, and Q2 is a queue place.

Note that, for different types of transitions and places, the numeric part of their identifiers are allowed to be the same.

Statement **CONTROL** (or CONTR) is used to select an input or output route, from which a token will be taken or through which a token will be passed as a result of firing of the transition. A route value is the position number of a place in a list of input or output places. This statement may be used only for elementary nets of type X, Y, and G. For example, statement

```
CONTR X130: if (%S15.NUM = 5)
        then %OUT := 2
        else %OUT := 1;
```

requires that, for elementary net X130, output route of a token must be 1 or 2, depending on the value of attribute NUM of a token place S15. Here %OUT is the reserved word which denotes the output route for elementary nets of type X and G. The corresponding reserved word for an input route is %IN.
Note that place \( S_{15} \) may be any place in given segment, not only a place incident to transition \( X_{130} \). Before testing \( \text{NUM} \) attribute in \( S_{15} \), it is generally necessary to test that \( S_{15} \) contains a token at all. This test can be done as

\[
\text{IF } \%S_{15} = 1 \\
\text{THEN …}
\]

Attempting to reference (read, write, or test) an attribute value in a place which does not hold a token at this moment will produce a diagnostic error and will cause a halt of simulation run.

If, for elementary net of types \( X \), \( Y \), and \( G \), the statement \( \text{CONTR} \) is not given, then the elementary net behaves according to the default rule. For elementary net of type \( Y \) in this case, input route will correspond to the first non-empty input place in the list of input places. For elementary nets of types \( X \), \( G \), output route will correspond to the first output place which can accept a token. This place is a queue place or an empty simple place.

If \( \%\text{IN} \) is assigned zero value, then this will prohibit elementary net of type \( Y \) or \( G \) from firing even if it has tokens in some of its input places. This prohibition will be in effect until \( \%\text{IN} \) is assigned a (nonzero) route number corresponding to some its nonempty input place.

If \( \%\text{OUT} \) is assigned zero value, this will prohibit elementary net of type \( X \) or \( G \) from firing, until \( \%\text{OUT} \) is assigned a (nonzero) route number corresponding to its output place of \( Q \) type or to an empty place of \( S \) type.

An example:

\[
\text{NET} \quad Y_{201} : \quad S_{3}, S_{12} \quad / \quad S_{54} \\
\text{CONTR} \quad Y_{201} : \quad \text{if } \quad \%Q_{23} > 0 \\
\quad \text{then } \%\text{IN} := 1 \\
\quad \text{else } \%\text{IN} := 0;
\]

For elementary net of type \( Y \), this statement tests the current number of tokens in some queue place \( Q_{23} \) (which is not required to be an incident place of \( Y_{201} \)). If this place is not empty then the first route is assigned to \( \%\text{IN} \) to take a token from input place \( S_{3} \) if any. If place \( Q_{23} \) is empty, then input route \( \%\text{IN} \) is assigned zero value which prohibits transition \( Y_{201} \) from firing, even if \( S_{3} \) contains a token. Note that \( \text{CONTR} \ Y_{201} \) will be activated with each event in the segment until \( \%\text{IN} \) is assigned value 1 and \( Y_{201} \) fires. Thus, this mechanism can be conveniently used to postpone a transition from firing (even if the current configuration of tokens allows its firing), until a desired test becomes true.

\[
\text{Statement TIME} \quad \text{is necessary to specify the time delay for an elementary net. This statement is not required if the time delay is zero. For example, the statement}
\]

\[
\text{TIME} \quad T_{200}: \quad \%\text{DELAY} := \%\text{TIM};
\]
specifies, for elementary net T200, the time delay equal to the current value of net variable TIM. Here %DELAY is a reserved word of the language; it is assigned the length of the firing interval for the corresponding transition.

Statement TRANSFORM (or TRANS) expresses data transformation which must be performed after the corresponding elementary net ends its firing. This happens after elapsing the time delay specified by the statement TIME for this net. Note that, data operations may be carried out with the use of token attributes for a token in any place of the segment and of Pascal and net variables. When using token attributes, generally a check should be done initially to be sure that the referenced place contains a token. An example of this statement is:

TRANS Y11: if %S13 = 1 
    then %S15.CHAN := %S13.CHAN - 1
    else %S15.CHAN := %CAT;

It is assumed here, that place S15 holds a token at the time of performing this statement. If place S13 contains a token then attributes CHAN of the token in S15 is assigned the decremented value of this attribute of token from place S13. Otherwise, attribute CHAN in S15 is assigned the current value of net variable CAT.

Procedures in the statement CONTROL, TIME and TRANSFORM will be written according to Pascal language. These procedures may use the Pascal variables and net variables, they also may reference imbedded functions of MDL. Moreover, these procedures may reference transitions, places and token attributes, in any place of the segment. As was stated before, procedures of the statement CONTROL must use the reserved word %IN (in the elementary net of type Y), %OUT (in the net of type X), or both %IN and %OUT (in the net of type G). One more reserved word %DELAY must be used in the procedure of the statement TIME.

The reserved words %IN, %OUT, and %DELAY may be used by procedures in expressions as well.

It should be taken into account that statements CONTROL, TIME and TRANSFORM mutually exclude each other. This means that each of these statements can be considered as atomic. That is, when some of these statements is being executed, no other statement from the segment is being executed concurrently.

It should be also taken into account that, even for the same elementary net, the statements CONTROL, TIME and TRANSFORM are performed at different moments of time. As a result, a variable of the segment can have different values as seen by different statements. For this reason, a user should avoid to associate a non-zero time delay with an elementary net of types X, Y, and G. Consider the following example:

NET X125: S23/ S12, S57; 
TIME X125: %DELAY := 68.0; 
CONTR X125: if %FLAG = 4 
    then %OUT := 1
    else %OUT := 2;
In this example, the decision which output route to use (1 or 2) is done by CONTR X125 before transition X125 starts firing. If, during the firing interval of 68.0 time units, the value of FLAG is changed (by some other firing transition) then the decision can become false.

If necessary, data transformation may be done not only by statement TRANS, but also by statements CONTR and TIME.

Two more statements which can be used in the description of a segment are ATTACH and LINK. They are used only for creation of a multi-segment model and are described in section 3.5.

3.4 Referencing transitions, places, token attributes, and SNAs

MDL allows to reference transitions, places, token attributes, input and output route and delay of transitions, and so called standard numerical attributes (SNA). Result of a reference can be used by procedures in statements TRANS (TRANSFORM), CONTR (CONTROL), and TIME.

To reference a transition or a place, it is necessary to use identifier of the desired transition or a place with the prefix ‘%’. The result of such a reference is the number of tokens in the referenced place or the state of the referenced transition. It is essential that references to transitions and places may be only in the right part of an assignment statement. That is, the state of a place or a transition may be read, but not changed.

For example, %S25 and %T4 are references to simple place S25 and transition T4. %S25 is 0 if S25 is empty or 1 if S25 holds a token at the moment of referencing. Similarly, %T4 is 0 if transition T4 is not active or 1 if T4 is active at the moment of referencing. The reference %Q78 gives the current number of tokens in a queue place Q78.

As was explained earlier, to reference an attribute of a token in some place, it is necessary to use identifier of the place and, after a dot, name of the desired attribute. The reference must be prefixed by ‘%’. For example, %S25.NUM is a reference to attribute NUM of a token in place S25. As a rule, before referencing an attribute of a token in a place, one must be sure that this place really holds a token.

A reference to an attribute of a token in a queue place implies that the head token is used. For example, %Q45.TIM is a reference to a TIM attribute of the first (head) token in place Q45. As for simple places, before referencing an attribute of a token in a queue place, one should check that the place is not empty.

References to attributes may be in the left and right parts of an assignment statement. That is, one can read and change the value of an attribute of a token in a place.

Example 1.

This example contains references to attribute ATR1 of a token in places S2 and S12. Checking whether S12 really holds a token is done before referencing. Since S2 always holds a token when procedure TRANS of T11 is executed, there is no need to check the presence of a token in S2 before referencing %S2.ATR1:

NET T11: S1/S2;
TRANS T11: if %S12 > 0 then %S2.ATR1 := %S12.ATR1;

To reference an input route, output route and delay of a transition in its CONTR, TIME, and TRANS statements, it is necessary to use reserved words IN, OUT, and DELAY respectively. The reference must be prefixed by ‘%’. All these references can be used in all procedures (TRANS, TIME, CONTR) of a transition in the right part of any assignment statement. In the left part, %DELAY may be used only in TIME procedure, and %IN and %OUT may be used in CONTR procedure.

Any transition can reference, in its CONTR, TIME, and TRANS statements, an input route, output route, and delay of another (remote) transition. For this reason, each of the reserved words IN, OUT, and DELAY must be used together with the identifier of the desired remote transition. For example, %G1.IN is a reference to the input route of transition G1. References to remote transitions may be used only in the right part of an assignment statement.

Therefore, reserved variables IN, OUT, and DELAY can be referenced directly (with prefix %) from any procedure associated with given transition, and in this case these references will correspond to the variables of this transition. Or, for a remote transition, the reference to its variables IN, OUT, DELAY can be done from a procedure associated with some other transition. In this case, the remote transition must be initially tested for firing. For example, in some procedure associated with transition T10, we can reference variable DELAY of another transition T25:

TRANS T10: if %T25 = 1 (* If transition T25 is firing at this moment *)
then %ABC := %T25.DELAY;

This statement assigns the duration of firing of transition T25 (if it is really firing at the moment of testing) to the net variable ABC inside procedure TRANS of transition T10.

To reference a standard numerical attribute (SNA) of a transition or place, it is necessary to use identifier of the transition or place together with the name of a desired SNA. Table 3.1 gives the list of SNAs for referencing in MDL statements, for a transition, simple place, and queue place. For example, %Q1.SNA5 is a reference to SNA5 of queue place Q1. At the moment of referencing, %Q1.SNA5 is assigned the maximum number of tokens stored in Q1 during the elapsed simulation interval, starting from the beginning of the simulation run. Correspondingly, %T3.SNA4 is a reference to the average firing time of transition T3 calculated for the duration of the elapsed simulation interval. Below is an example of referencing of a SNA, reserved variable OUT, and a transition.

Example 2.

NET X11: Q1/S1, S2;
CONTR X11: if %T1.SNA3 <= 10000
then %OUT := 2
else if %I2 = 1
then %OUT := 1
else %OUT := 0;
In this example, by procedure CONTR of transition X11, a remote transition T1 is tested for 10000 firings, and another remote transition I2 is tested for firing at the moment of referencing. Depending on the result of the tests, the reserved variable OUT (it represents the output route of transition X11) is assigned one of three integer values. Note that assigned value 0 prohibits the firing of transition X11.

Table 3.1 Standard Numerical Attributes for Referencing in MDL

<table>
<thead>
<tr>
<th>SNA Name</th>
<th>Meaning of SNA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>For a transition</td>
</tr>
<tr>
<td>SNA1</td>
<td>State: 1 – is firing, 0 – not active INTEGER</td>
</tr>
<tr>
<td>SNA2</td>
<td>Current utilization REAL</td>
</tr>
<tr>
<td>SNA3</td>
<td>Number of firings INTEGER</td>
</tr>
<tr>
<td>SNA4</td>
<td>Average firing time REAL</td>
</tr>
<tr>
<td>SNA5</td>
<td>Time left to the end of current firing REAL</td>
</tr>
</tbody>
</table>

Note: When using SNAs in arithmetical expressions, take into account the following data types for SNAs: INTEGER for SNA1 and SNA3, REAL for SNA4, REAL for SNA5 for a transition, and INTEGER for SNA5 for a queue place as is indicated in the table.
3.5 Statements ATTACH and LINK for multi-segment models

These two statements are required to create a large model from a number of segments. Statement ATTACH is used to attach one or more segments to a model, without specifying any information links between segments. A segment may have more than one statements ATTACH.

The segment which contains ATTACH statement is considered logically as a root for its attached segments, each of which may have its own statement ATTACH. Therefore, the complete multi-segment model structurally represents a rooted tree, with one head segment as the root. A model consisting of one segment is a particular case of the tree-like structure.

In general, statement ATTACH can specify the original name of attached segment and the names of a few copies of this segment. In this case, attached are copies of the original segment. This feature is very important if the model is created as a collection of many separate units which have the same structure and behavior, like a model of a multiprocessor system, with processors designed as separate units.

For example, the following statement

ATTACH RICART /RIC01, RIC02, RIC03, RIC04, RIC05/;

attaches five copies RIC01, RIC02, …, RIC05 of original segment RICART to the segment which contains this statement. Naturally, all copies have exactly same identifiers of transitions, places, attributes, and variables as in the original segment, and differ only in the names of copies.

Logically, attaching of segments is like putting the attached segments in one box, together with the head segment on top. The head segment and the attached ones have no information links and will behave completely independently if this is desired.

The same segment may be attached in different attaching segments. However, it is not allowed for a segment to attach itself or to make attachments which create a loop in the model structure. A multi-segment structure of the model created by ATTACH statements may be only a rooted tree, with the only, head segment as the root. The system will warn the user if he violates this scheme.

Statement LINK is used to establish information links between this segment (i.e. the segment which contains the LINK statement) and segments attached to this segment, or between the segments attached to this segment. Thus, generally the existence of statement ATTACH in the segment implies the use of one or more statements LINK in this segment. However, if necessary, the attaching and attached segments may be left without any linking at all. In this case they will have no information links between each other and will behave completely independent of each other.

Each statement LINK specifies a pair of names of segments and a list of pairs of places to be linked (or connected like electrical contacts) in the pair of segments. For example, statement

LINK NETW3, PROC1: S101, S1/S10, S1000;
specifies two links between segments NETW3 and PROC1. One of these two segments (for example, PROC1) could be attached to the second segment (NETW3, in this example), or both segments could be the attached segments. The first link is created by connecting of place S101 in NETW3 with place S1 in PROC1. The second link is created by connecting of places S10 and S1000 in segments NETW3 and PROC1, respectively.

When linking segments, it is essential to remember what happens in the connected places during simulation. First of all, the correctness of linking is checked by the system during the assembling of the complete model. For the correct linking, the following conditions must hold:

1. In each pair of connected places from two segments, one place must be an output place, and the second one must be an input place. In the pair, their identifiers may be written in any order. It is allowed for both places to belong to the same segment.
2. In each pair of connected places from two segments, the places must be of the same type (that is, both of them must be simple places or queue places).
3. The number and names of attributes in the linked segments are allowed to be different. However, in the segment with smaller number of attributes, its attributes must be of the same type as the corresponding first attributes in the second segment (which has larger number of attributes).
4. If an output place belongs to a segment with a larger number of attributes than in the second segment (with an input place), then the extra attributes of a token are cut when the token moves from the first segment to the second one.
5. If, on the other hand, an output place belongs to the segment with a smaller number of attributes than in the second segment, then the token moving from the first segment to the second one is appended by the extra attributes of the second segment, with undefined values.

The difference in the number of attributes in different segments can be necessitated by the desire to have, in addition to attributes common for all involved segments, some attributes specific for the segment. In this case, the modeler should write the common attributes, in all segments, at the beginning of the attribute list (in statement ATTRIBUTES), in the same order of types in all segments. Additional specific attributes, if any, should follow the common attributes and may be different, in number and types, in different segments.

### 3.6 Pascal section in an MDL segment

Since MDL is based on Object Pascal language (and represents its extension), an MDL segment may contain a Pascal section. The declared objects of this section (labels, constants, types, variables, functions and procedures) can be referenced by any procedure associated with transitions of the segment. This will decrease the size of the segment if different procedures associated with transitions use the same code.

According to the syntax of an MDL segment, its Pascal section (if any) should be placed after statement DATA and have the following structure:
INTERFACE
IMPLEMENTATION
<Pascal declarations and definitions >
BEGIN
<Initialization statements (if any)>
END.

If initialization statements are absent then statements BEGIN and END are not mandatory. An example:

INTERFACE
IMPLEMENTATION
CONST  creq = 1;
       Ndat = 12;
VAR       work : integer;
          proba: real;
END.

In this example, statement BEGIN is absent (since there are no initialization statements). Statement END may be absent as well.

If the Pascal statement WRITELN is used in any transition procedure or independent function of the Pascal section, then the corresponding output is directed to the file <name of the first run>.LOG created automatically by the system in the same directory which contains the model. If the model does not have any statements WRITELN, then the corresponding file <model run name>.LOG is empty. Using the statement WRITELN can be especially useful for debugging purposes. The use of this statement is illustrated in Example 8 of Section 6. Note that the LOG file is closed only at the end of application.

Procedures associated with elementary nets may not use any Pascal labels. The reason is that MDL does not allow to declare labels (constants and variables as well) locally in separate procedures of elementary nets since the MDL compiler automatically generates the word BEGIN before the text of any such procedure. However, labels may be declared and used in independent procedures and functions in the Pascal section. The use of independent function is demonstrated by Example 8 in Section 6.

MDL does not provide global, for all segments of a model, objects. Nevertheless, the objects of the Pascal section can be used by different copies of the same original segment as global objects if necessary since the procedural component of all segment copies exists in the model as one common Pascal unit. Segments which are not copies of the same original segment may also share global variables. For this purpose, a Pascal variable or array intended for shared use by all segments should be declared in the INTERFACE section of some segment X. Such a variable now can be referenced in any other segment which has the Pascal statement USE X in its Pascal section. However, this style of using of MDL is not recommended since it violates the principle of inter-segment communication only via the places by which the segments in the model are linked with each other.

The user should take into account that Pascal syntactical errors are not diagnosed by the MDL compiler. These errors, if any, will be displayed later, at the stage of model making, when Pascal units of segments are Pascal-compiled and collected in the model, before
producing its executable file. In case of Pascal errors, the user should return back to the MDL source text of the corresponding segment and correct the errors. Then the corrected segment must be MDL-compiled.
4. MODELING CONTROL LANGUAGE

The modeling control language (MCL) is used at the stage immediately before starting a ready model or during the run. Its purpose is to manage different kinds of user’s interaction with the ready model, such as setting an initial state of the model before the start of simulation, watching and controlling of the simulation run. MCL consists of the following statements: FOR, MARK, DEMARK, SET, STATISTICS, HISTO, TRACE, and STOP. Each of these statements generally requires some set of parameters.

All MCL statements can be input in an interactive command line mode or in a batch mode. In the last case, a user has to prepare the desired sequence of MCL statements in a text file with JZP extension and later to supply the name of this file before starting the simulation run.

The possibility to make parametric changes in a ready E-net model considerably simplifies the work of users. Quite often, when an E-net model is ready, many subsequent changes in that model require only those operations that are supported by MCL. Using MCL you need not recompile modified segments and carry out other activities to prepare the complete E-net model. This saves a lot of time and effort for a modeler during the simulation.

Below are informal descriptions of MCL statements, with examples of their using. The syntax of MCL statements is described in Appendix 2.

4.1 Statement FOR

In almost all MCL statements, it is necessary to specify (as the first parameter) a segment to which this MCL directive should be applied. To make the MCL text more short, it is possible to specify, by a separate statement FOR, the segment to which all subsequent MCL statements will be applied until the next FOR statement or unless the statement specifies a segment explicitly.

If, after applying of a number of MCL statements to some segment, it is necessary to start applying new MCL statement to a new segment, then statement FOR, with the name of this new segment, must be placed which will be followed by desired MCL statements.

Thus, MCL statements having no segment identifier as the first parameter will be applied to the segment named in the last FOR statement or, if there is no FOR statement at all, to the head segment of the model, on default.

The same statement FOR can contain a few segment identifiers. In this case, the subsequent MCL statements will be applied to all segments listed in statement FOR.

Example 1

FOR SEGMENTS SEGA, SEGB;
FOR SEGMENT SEGA, SEGB;

These two statements are equivalent; they differ only in the form of writing of the two reserved words SEGMENT or SEGMENTS.
Example 2

FOR SEGMENTS SEG1.B.C;
FOR SEGMENT SEG1.B.C;

In these two (equivalent) statements, a compound segment name is specified. It is assumed here, that the statements will be applied to segment C, which is attached to segment B, which is, in its turn, attached to segment SEG1. Thus, the combined name represents a path name of segment C.

If all segments in a model have different names, then it is sufficient to use, in statement FOR, only simple (non-compound) names of segments. However, the compound names must be used if the same segment is attached to different segments. (See an example in Section 5)

Different MCL statements (with the exception of FOR) may be used, in an intermixed way, with or without segment identifier.

All MCL statements with a segment identifier will be applied to the referenced segment, while MCL segments without a segment identifier will be applied to a segment (or segments) given in the last segment FOR.

4.2 Statement MARK

This statement is used to create tokens in the desired places of a ready model and to assign initial values to attributes of the created token. For a queue place (a place whose identifier starts with letter ‘Q’), statement MARK can be used also to specify the ordered number of a token for which one wants to set initial attribute values. The same statement MARK can specify a few places to be marked.

It is allowed to mark a place without specifying attribute values. In this case tokens in the marked places will have an undefined initial attribute values. It is allowed also to specify values for part of attributes.

Example 1

MARK S1[PNUM/1/], S2[PNUM/2/];
MARK S2[PNUM/2/], S1[PNUM/1/];

These two equivalent statements mark simple places S1 and S2, and attribute PNUM of a token in S1 is assigned value 1 and that of a token in S2 value 2. The segment to which S1 and S2 belong is the one which was specified in the last FOR. If FOR is absent at all, the head segment is assumed.
Example 2

FOR SEGMENTS A.C;
MARK S1[A1/1/, A3/2.5/];

With these two MCL statements, place S1 in segment C attached to segment A is marked by a token, and attributes A1 and A3 of the token are assigned the integer value 1 and real value 2.5, respectively. If the token has other attributes, then they remain undefined.

Example 3

MARK S1[A1/0/], S2;

By this statement, places S1 and S2 are marked, and attribute A1 of token in S1 is assigned integer zero value. Attributes of the token in S2 are undefined.

Example 4

MARK S1[[NUM/4.0/], Q2(1..5), Q17[A1/0/,A2/3.4/];

By this statement, simple place S1 and the queue places Q2 and Q17 are marked. Attribute NUM of a token in S1 is assigned a real value 4.0, attributes A1 and A2 of a single token in Q17 are assigned integer value 0 and the real value 3.4, respectively. Besides, place Q2 is marked with 5 tokens having numbers 1,2,3,4, and 5, without assigning any values to attributes of these tokens.

4.3 Statement DEMARK

This statement can be used to remove tokens from some places of a segment in a ready model. This can be done before the start of simulation (if tokens were created in some places by statement MARK earlier) or during the simulation run, when model is suspended.

It is not recommended to apply MARK or DEMARK statements during a simulation run (for example, after a suspension of simulation), since this can result in the inconsistency of the model.

Example 1

DEMARK SEGMENT SEGA: S1, Q25;

As a result of this statement, tokens are remove (if any) from places S1 and Q25 in segment SEGA.
DEMARK ALL;

The statement removes tokens from all places in the segment specified by the last statement FOR or, if there is no statement FOR, in the head segment of the model.

### 4.4 Statement SET

The statement is used to set or modify values of the net variables and net arrays. Recall that net variables and net arrays are those elements of the segment which are defined by MDL statement DATA. For net arrays, their indices start always with 1. Recall also that the names of all net variables and net arrays of each segment are saved after compiling of the segment and are accessible by a user before the simulation run and at the stage of simulation (for example, when the simulation is suspended).

With statement SET, the user can set and modify elements of the net arrays. Moreover, SET can be used to assign a desired value to many elements of a net array in the segment. Remember that, in MDL, net arrays can be only one or two-dimensional. A two-dimensional net array is stored in a row major order.

If a value of some net variable is specified in its declarative MDL statement DATA or set with the use of MCL statement SET, then this value can be changed later, if necessary, by a new statement SET. Thus, statement SET may be used for the same net variable a few times.

**Example 1**

SET L1/10.0/, MA/100.0, 200.0/;

This statement assigns real value 10.0 to net variable L1 and real values 100.0 and 200.0 to the first two elements of one-dimensional net array MA.

**Example 2**

SET M18A/5/, MASS/(8)2, (7)0/;

First, the statement assigns integer value 5 to the net variable M18A. Then values of $8+7 = 15$ elements of the integer one-dimensional net array MASS are set: the first 8 elements are assigned value 2, and the next 7 elements are set to zero value. If the array MASS has more than 15 elements, then the rest of the elements are not affected by this statement.

**Example 3**

SET MDAT [1, 5]/12.0/;
The statement sets the real value 12.0 for element MDAT[1,5] of two-dimensional array MDAT.

Example 4

SET B23 /4.0, 12.0, (5) 3.0, 7.0, (4) 9.0/;


It is essential that the type of the value assigned to a net variable coincides with the declared (in MDL section DATA) type of the variable. According to MDL, net variables can be of integer or real type only.

4.5 Statement STATISTICS

The statement is used to assign or cancel the collection of some statistics related to places and the transitions of a segment. The type of collected statistics depends on the type of a related element in the segment.

For a simple place, the following items are calculated: utilization (ratio of the complete time during which the place contained a token to the simulation time), number of tokens passed via the place (including the token at the end of simulation, if any), average token time in the place and state of the place at the moment of termination of simulation run (0 or 1 depending on whether the place is empty or holds a token).

For a queue place, the calculated items are number of passed tokens, average number of tokens during the simulation interval, average time of a token in the place, maximum number of tokens during the simulation and the number of tokens at the moment of simulation termination or suspension.

For a transition, the calculated items are utilization (the ratio of the complete duration of the activity intervals of the transition to the simulation interval), number of firings, average firing time, and the state at the moment of simulation termination or suspension (1 or 0 depending on whether the transition is active or not active, respectively). Below are examples of using the statement.

Example 1

STATISTICS T0, T1, T2, T3, T9, S1, Q5;

This statement requires collecting of statistics for transitions T0, T1, T2, T3 and T9 and places S1, Q5 in the segment specified in the last FOR statement or in the head segment, if there is no preceding FOR statement.

Example 2

STATISTICS EXCEPT Y3;
The statement requires collecting of statistics for all places and transitions, in the corresponding segment, with the exception of transition Y3.

Example 3

STATISTICS SEGMENT ABC: ALL;

This will output statistics for all places and transitions in the segment ABC.

Example 4

STATISTICS STOFF;

The statement cancels the collection of statistics for all places and transitions of the corresponding segment.

Example 5

STATISTICS STOFF S1, Q5, T5, X7;

The statement cancels the collection of statistics for places S1, Q5 and transition T5 and X7 of the corresponding segment.

### 4.6 Statement HISTO

This statement is used to specify a histogram for tabulation of some variable in a segment. The histogram, created at the end of simulation, will contain average value of the tabulated object, its standard deviation and the histogram itself.

For the same segment, one can specify more than one histogram. Each histogram should have a unique (for this segment) integer identifier, which is written immediately after the word “HISTO”.

If the statement HISTO is repeated with the same identifier of a previous statement but without parameters, then the new statement just deletes the previous statement.

Parameters of HISTO are the right limit of the first interval of the tabulated argument, the width of one interval, the number of intervals and, for a special argument NFI, time interval. In addition, places and transitions should be specified with which this histogram is associated.

Argument of a histogram can be a net variable (i.e. a variable declared in section DATA of a segment), a token attribute, a standard numerical attribute (SNA) or special modes NFI or IAI.

Note that the identifier of a net variable (if used as an argument) is written without the prefix “%”. An attribute is specified just by its name as declared in section ATTRIBUTES.
As SNAs one can use fixed names SNA1, SNA2… SNA5. All SNAs are related only to places and transitions of the segment. Table 4.1 below gives a short description of these SNAs, for a simple place, a queue place and a transition.

<table>
<thead>
<tr>
<th>SNA Name</th>
<th>Meanings for a transition</th>
<th>Meanings for a simple place</th>
<th>Meanings for a queue place</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNA1</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>SNA2</td>
<td>Utilization</td>
<td>Utilization Number of tokens</td>
<td></td>
</tr>
<tr>
<td>SNA3</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>SNA4</td>
<td>Firing time</td>
<td>Time of a token in the place</td>
<td>Not applicable</td>
</tr>
<tr>
<td>IAI</td>
<td>Time interval between firings</td>
<td>Time interval between arrivals of tokens</td>
<td>Time interval between arrivals of tokens</td>
</tr>
<tr>
<td>NFI</td>
<td>Number of firings during a sliding window</td>
<td>Number of token entries during a sliding window</td>
<td>Number of token entries during a sliding window</td>
</tr>
</tbody>
</table>

Note: The value of a SNA corresponds to the moment of referencing of this SNA. For a transition, this moment corresponds to the end of firing. For a place, this is the moment when its input transition finishes firing.

There are two special modes IAI and NFI of using of statement HISTO. In the mode IAI (Inter-Arrival Interval) time intervals between arrivals of tokens in a place or between firings of a transition are tabulated. For this purpose, a moment of time of the first event of this type is registered (for example, the moment of arriving of the first token in the desired place). After that, when the next event happens (for example, the next token arrives), a time difference between this event and the previous event is tabulated.

Example 1

HISTO 1 (IAI, 10.0, 20.0, 15) S5;

The statement tabulates inter-arrival intervals of tokens for place S5 in a segment. The mode NFI (Number of events For Interval) is used to tabulate the rate of tokens arrivals in a place or the rate of transition firings during a defined NFI interval. The duration of this interval is specified as the last parameter of HISTO. This mode operates in
the following way. A counter is created, corresponding to this HISTO, which will be used to count the number of events (the number of token arrivals for a place or the number of transition firings for a transition). After elapsing of the NFI interval, the contents of the counter is tabulated and the counter is cleared. After that, a new NFI interval is started, and the contents of the counter at the end of this interval is tabulated. Thus, the entire simulation time is divided into equal NFI intervals, and the counter value at the end of each interval is tabulated.

Example 2

HISTO 5 (NFI, 5.0, 10.0, 15, 200.0) T8;

This statement tabulates number of firings of transition T8 during NFI interval of 200 time units.

Note that the same histogram can be used for tabulation of the desired argument associated not only with one place or one transition, but with a list of places or a list of transitions or even with a mixed list of transitions and places.

Example 3

HISTO 2 (DEL, 30.0, 10.0, 12) S3, S7;

This statement tabulates the argument DEL (which can be a net variable in DATA section of a segment or an attribute in ATTRIBUTE statement) every time, when a token arrives in place S3 or S7. If DEL is an attribute, then attribute values of tokens arriving in S3 and S7 will be tabulated.

Example 4

HISTO 3 (SNA2, -1.0, 1.0, 10) T3, T4;

The statement requires tabulating, in the histogram 3, the standard numerical attribute SNA2, for transitions T3 and T4.

4.7 Statement TRACE

This statement is used to set the tracing of firings of transitions in a segment. In a statement, one can specify a list of transitions for which the tracing is desired. During the simulation, for each of the specified transitions, a trace of firings is output to the screen and, concurrently, to *.LOG file for possible off-line analysis. For each fired transition (a trace point), the following information is output: segment name, transition name, start time of transition firing, finish time of transition firing, delay time (firing interval), input (for a net of type Y), output (for a net of type X0, or input/output (for a net of type G). If delay is zero, then only one time is output. To fire the next transition, the user should just press “TRACE” button after each output.
It should be noted that, for a transition of I type, start time of firing will be output in the form of “????????” if the firing of the transition has been interrupted.

The file *.LOG is the same to which the results of statements WRITELN are output. Thus, for the user’s convenience, the tracing information and results of WRITELN are kept in the same file.

The statement can be used also to cancel the tracing (for example, when the simulation has been suspended).

The tracing can be specified for different or even the same transitions, by the use of a sequence of statements TRACE.

Example 1

TRACE ALL;

The statement requires the tracing of all transitions of a segment. It is assumed that the segment name has been specified in the previous statement FOR or, if no FOR statement was used, the head segment is assumed.

Example 2

TRACE SEGMENT ABC: ALL;

Here the tracing is set explicitly for all the transitions of segment ABC.

Example 3

TRACE T4, T2;

The tracing is required for transitions T4 and T2 only, in the head segment or in the segment specified by the previous FOR statement.

Example 4

TRACE SEGMENT RICAR: EXCEPT T1, T2, T5;

The tracing is required for all transitions in segment RICAR, with the exception of T1, T2, and T5.

Example 5

TRACE STOFF T3, X2;

The statement cancels the tracing of transitions T3 and X2.

Example 6
TRACE STOFF;

The statement cancels the tracing in a segment for all transitions.

4.8 Statement STOP

The statement is used to specify a condition for stopping the simulation before elapsing of the complete simulation time. The statement can be used also to cancel the previously specified stopping condition.

A stopping condition is specified in a form of a list of places with which this condition is associated. Let \( a_1, a_2, \ldots, a_n \) be a list of places listed in a STOP statement. The simulation will stop as soon as there will be at least one token in each of these places. In the list of places, any combination of simple and queue places is allowed.

A stopping condition can be specified by a few statements STOP. In this case, the simulation will stop as soon as a condition specified by any of stop statement is satisfied.

Example 1

STOP SEGMENT DSYSM: S1, S3, Q15;

According to this statement, the simulation will stop as soon as there is one token in places S1, S3 and at least one token in place Q15 of segment DSYSM.

Example 2

STOP S1, S3;
STOP S3, S15, Q8;

Here are two stopping conditions specified. The first condition is the appearance of tokens in places S1 and S3, and the second one is the appearance of tokens in S3, S15 and Q8. The simulation will stop if any of these two conditions become true. Thus, stopping conditions specified by two or more statements STOP are combined by the logical operator “OR”.

Example 3

STOP STOFF;

The statement cancels all stopping conditions that could be specified by previous statements STOP.
5. METHODOLOGY OF MODELING AND SIMULATION WITH E-NETS

Initial data for the creation of an E-net model of a system are the result of a conceptual description of the system. This description generally will cover such aspects as the purpose and the tasks of the system, the set of the system components and their functions, the flows and the character of requests, the points of possible time delays, with their general specification (fixed, random, varying), the features of requests entering the system from its environment and the disciplines of handling these requests. For a concurrent or distributed system, it is very important to specify, in the conceptual description, the character and extent of parallel and concurrent data processing, the characteristics of a communication subsystem for message passing between the concurrently functioning components, and the access mode of the communicating components to a media. It is useful to postulate, in the conceptual description, also the desired granularity of events and processes to express in the model, taking into account that the excessive granularity results in the additional complexity of the model and too rough granularity can make difficult or even impossible to get, with the model, the answers which are essential for the designer. The conceptual description may include formal outlines, which specify or clarify the behavior of some system components or the system in the whole. An approximate analytical model, if it exists or can be easily derived, is a very desirable part of the conceptual description. The analytical model is helpful not only for more deep understanding the system being modeled but also for the verification of the simulation model.

The next stage in the creation of a model is the strict specification of basic assumptions underlying the model. The reasonable and clear set of the assumptions not only fixes the conditions, under which the model is valid, but also helps in using the model by other researches for comparison purposes.

Having the conceptual description of the system and the basic assumptions for modeling, a researcher is now ready to design an E-net scheme of the model and then implement it. To this end, the following steps should be carried out.

1. Design of a segment structure of the model. At this step, with the known set and functions of the components, segments of the model, their tasks and links between the segments are specified. A reasonable separation of the model into a number of segments simplifies the creation of the complete model and provides better conditions for its scalability. For example, in a model of a LAN, each station might be represented by a separate segment, and one additional segment would be required to represent a communication subsystem.
In Winsim, segment structures of models are hierarchical. An example of the structure is shown in Fig. 5.1. The model consists of three explicitly defined segments - A, B, and C, with the segment A as a root. The segments A and B contain the MDL statements `ATTACH` for including additional segments or their copies. In particular, the segment A includes two copies of the segment B (with the names B1 and B2) and a segment C. The segment B (and each of its copies) includes a segment D. The statements `LINK` in the segment A (Fig. 5.1, a) specify the links between the segments. Each segment in a statement `LINK` is referenced by its full path name in the hierarchical structure of the model. Note that the segment D is included...
2. Development of an E-net scheme of each segment of the model. This is carried out by the use of the basic types of elementary nets and E-net schemes of typical components of data processing systems, such as job generators, task schedulers, interruptible processors, etc. An E-net scheme of a segment should include all structural elements of the segment - transitions, places, and arcs - and show the supposed concurrency of events and processes. The places chosen for the connection of the segment with other segments should be clearly defined. A desirable initial marking of some places of the segment may be fixed at this step. Functionally, this step can be viewed like drawing an electrical circuit with the use of standard electrical elements but without their detailed specification.

3. Specification of token attributes and net variables in each segment. Note that in different segments of the same model the number of token attributes may be different but, for any pair of linked segments, their attributes must be declared with the same ordering of data types. This is illustrated by Fig. 5.1, in which the segments A and B have the same number of attributes, but only two attributes are declared in the segment C. As Fig. 5.1, a and c show, tokens from the root segment A can bring into each copy of the segment B (B1 and B2) the values of three attributes A1, A2, and A3. On the other hand, tokens from both copies of the segment B can bring into the segment C the values of only two first attributes A11 and A12 because only two attributes are declared in C.

4. Determination of probability distributions for all random values in each segment. The random values can describe arrival patterns of external requests, service patterns, addressing random destinations, delays, time-outs, etc. As it was mentioned in Section 3, MDL has a few random number generators to model different probability distributions. If necessary, additional random number generators may be implemented by the use of the facilities of the basic language.

5. Determination of all elementary nets in each segments for which it is necessary to explicitly specify the associated functions \( r_1 \) and/or \( r_2, d, \) and \( m \). If possible, it is quite desirable to use the defaults for these functions. The default functions need not be explicitly specified which considerably simplifies the model.

6. Specification of all explicit functions, determined at the preceding step. The specification, at this step, may be done informally, not necessarily in MDL.

7. Complete description of each segment in MDL. A name of a segment will be a part of the description. The result of this step will be source files of segments in MDL.

8. Compilation of source files of all segments, using the MDL compiler. This will produce two separate files for each compiled segment of model. One of these two files
contains the structural information about the segment. The second file is a Pascal unit which contains all functional and procedural components of the segment.

9. Pascal compilation of units of all segments and linking the compiled Pascal units with the system object modules to get the executive subsystem, containing the created simulation model.

10. Starting the executive subsystem. At this step, statements of MCL are used to initialize and then run the model. The most important operation in initializing the model is marking the selected places in its segments, possibly with some initial values of token attributes. A user specifies, at this step, also the modeling interval, the identifier of the simulation run, and, if necessary, the initial time period during which statistics should not be gathered (to eliminate the initial bias).

11. Launching the simulation. A user watches the progress of the modeling time. He/she may stop the simulation at any moment to check the current state of the model or change the values of some net variables. On the base of this information, the user may return to the creative subsystem for correction or modification of the model.

During the simulation, different statistics is gathered and evaluated in Winsim. This statistics includes measures on transitions (number of firings, mean firing times, intervals between firings, utilization) and places (number of passed tokens, mean token waiting times, mean and maximum numbers of tokens in Q-places), and histograms for token attributes in selected places, standard numerical attributes and net variables. The results are represented in the form of tables with fixed formats. Interpretation of these results will be done by the modeler.

12. After elapsing the simulation time, a user may output the results of the simulation run on the screen, print them or save in a file for subsequent analysis. This analysis includes, first of all, an interpretation of evaluated measures on transitions and places according to the features of the system being modeled. The researcher may perform some technique of verification of the results and/or apply different techniques of statistical analysis and planning of experiments.
6. EXAMPLES OF SIMPLE E-NET MODELS

This section presents a number of examples of E-net fragments and simple models which serve to clarify the basic types of the elementary nets and the features of MDL. Each example includes the corresponding description in MDL, and a brief explanation of the model functionality.

Example 1.

This example represents an E-net fragment which models a job generator. The E-net scheme and the description of the generator in MDL are given in Fig. 6.1. The scheme consists of two transitions T20 and T30, two simple places S20 and S30, and one Q-place Q40 which is intended as a queue for generated tokens, or jobs. In MDL, the first symbol in a transition identifier is one of the letters G, I, T, X, or Y, depending on the type of the underlying elementary net. In the example, the model is composed from two connected elementary nets of type T each. In the initial state, the place S20 contains a token and other places are not marked. This provides the possibility for the transitions T20 and T30 to fire alternately, when the model starts running. With each firing of the transition T30 a new token is added into the place Q40. It is supposed that the tokens can be removed from this place as a result of the activity of a net to which the place Q40 might be connected.

ATTRIBUTES
JBid: INTEGER; (* Id of the job *)
Jtm : REAL; (* Time to fulfill the job *)

DATA
Tgen /800.0/: REAL; (* Mean time to generate a job *)
Twrk /500.0/: REAL; (* Mean time to fulfill a job *)

NET
T20: S20/S30;
TIME
T20: %delay := exp (1, %tgen);
TRANS
T20: %s30.jbid := %s30.jbid + 1; (* Id of a new job *)
%S30.JTM := EXPON (1, %TWRK);
NET
T30: S30/Q40, S20;

Fig. 6.1. The E-net model of a job generator (the scheme and the description in MDL).
The text of the description of the fragment in MDL shows, that each token in it has two attributes - the integer attribute JBID and the real attribute JTM. The purpose of both attributes is clear from the comments. By the statement DATA, two net variables TGEN and TWRK are declared and initialized. The statement NET T20 defines the structure of the elementary net related to the transition T20, and the statement TIME T20 defines the explicit function \( d \) for the elementary net. According to this function, the activity interval of the transition T20 is calculated (and assigned to the reserved variable \%DELAY) as a realization of the exponentially distributed random value (see the function EXPON) with the mean value TGEN. The first parameter of the function EXPON is the number of the basic random number generator. There are a few other random number functions in the MDL, including FRANDOM, UNIFORM, IUNIFR, BINOM, NORMAL, POISSN, and FRASP.

The statement TRANS T20 defines the explicit data transformation performed after firing the transition T20. According to this transformation, the attribute JBID of a token in the place S30 is incremented, and the attribute JTM is assigned the exponentially distributed random value, with the mean TWRK.

The statement NET T30 defines the structure of the second elementary net. In this net, the defaults are used for the function \( d \) (the activity interval for T30 is supposed to be zero) and for data transformation (the attribute values of a token in S30 are copied and assigned to the corresponding attributes of new tokens in Q40 and S20.

To put an initial token into S20, with the zero initial value of the attribute JBID, the appropriate statement of the modeling control language should be used:

```
MARK S20 [JBID/0/];
```

Note that the simplest job generator could be modeled using only one elementary net of type T, without any input places. But in this case the numerical identifier of the previously generated job should be saved in an additional net variable.

Example 2.

The example illustrates the use of the elementary nets of types X and Y in a model of a round-robin task scheduler. This is a very popular scheduler in multitasking operating systems. The E-net scheme of this scheduler and the description of its model in MDL are given in Fig. 6.2. The model includes the elementary nets of types Y, T, and X. The first elementary net, with the transition Y1, is used for collecting request tokens from the input places S1, S2, ..., Sn and S300 and including them into the output place Q1, which models a FIFO input queue of a task processor.

Note that there are no explicit functions for this elementary net, so that its specification consists of only the statement NET Y1 defining the structure of the net. Therefore, only defaults are used for the functions \( r_1, d, m \) in this elementary net. According to these functions, when the transition Y1 fires, its activity interval is equal to zero. Then a token is included into the output place Q1, the attribute values of this token are assigned the values of the corresponding attributes of the token from the first nonempty input place (counting
in the order of enumeration of the input places in the statement NET Y1), and the token from the selected input place is removed.

The elementary net with the transition T2 models the processing of the next request, represented by the first, or head, token in Q1. For this net, two explicit functions \( d \) and \( m \) are specified. Namely, the statement TIME T2 implements the function \( d \), calculating the activity interval of the transition T2 with the condition that this interval does not exceed the fixed quantum value QNT. The statement TRANS T2 corresponds to a data transformation function, according to which the value of the attribute TIM of a token in the place S100 is decreased by the value of QNT. It is supposed that when the request token enters one of the places S1, S2, ..., Sn, the value of its attribute TIM is equal to the required processing time.

![E-net model of a round-robin task scheduling](image)

**ATTRIBUTES**

\[
\begin{align*}
\text{TIM} & : \text{REAL}; & (* \text{Complete time to process the task} *) \\
\text{DATA} & \\
\text{QNT} & /20.0/: \text{REAL}; & (* \text{Time quantum} *) \\
\text{NET} & \text{ Y1: S1, S2, ..., Sn, S300/Q1; } \\
\text{NET} & \text{ T2: Q1/S100; } \\
\text{TIME} & \text{ T2: if } %\text{Q1.TIM} > %\text{QNT} \\
& \quad \text{then } %\text{DELAY} := %\text{QNT} \\
& \quad \text{else } %\text{DELAY} := %\text{Q1.TIM}; } \\
\text{TRANS} & \text{ T2: } %\text{S100.TIM} := %\text{S100.TIM} - %\text{QNT}; & (* \text{Time left} *) \\
\text{NET} & \text{ X3: S100/S200, S300; } \\
\text{CONTR} & \text{ X3: if } %\text{S100.TIM} > 0 \\
& \quad \text{then } %\text{OUT} := 2 & (* \text{Decision} *) \\
& \quad \text{else } %\text{OUT} := 1; \\
\end{align*}
\]

Fig. 6.2. The E-net model of a round-robin task scheduling.

At last, the elementary net of type X, with the transition X3, models the decision, concerning the request after each visit to the processor. For the statement X3, the explicit function \( r_2 \) is specified by the statement CONTR X3. If the value of the token attribute TIM in place S100 is greater than zero, then the processing of the request is not completed, and the token must be routed along the second output arc to the place S300. From the place
S300, after firing the transition Y1, this token will be included into the place Q1 for the new quantum of processing.

On the other hand, if the value of the token attribute TIM in the place S100 is not greater than zero, then the processing of the corresponding request is completed, and the token will be routed along the first output arc to the place S200.

It is supposed that the scheduler is a part of a more large E-net model, which supplies tokens into the places S1, S2, ..., Sn and removes them from the place S200. It should be noted also that the functions of the elementary net with the transition T2 in this example might be delegated to the elementary net X3. In this case the model would consist of only two elementary nets of types X and Y, and for the elementary net with the transition X3 all three functions \( r_2 \), \( d \) and \( m \) had to be explicitly specified by the statements CONTR X3, TIME X3, and TRANS X3 respectively.

Example 3.

The example illustrates one possible use of the interruptible elementary net (type I). Suppose that a processor executes main requests, and that during processing a request the processor can be interrupted by a signal. After the interruption, the processing of the request must be postponed, and the processor will be busy handling the interruption during some time dependent on the interruption type. After completing the interruption handling the processing of the postponed request continues (and may be interrupted again).

Assume for the simplicity, that during the interruption handling a new interruption is impossible. Suppose also, that the time which is necessary to process the main request or interruption is given as an attribute value of the corresponding token.

The E-net scheme of the interruptible processor, with its description in MDL, is given in Fig. 6.3. The model consists of three elementary nets of types X, Y, and I, with the transitions X24, Y23 and I24 respectively. The purpose of the Y-type net is to collect input requests for the processing. The place S21 is for the main requests to be processed. The place S23 is for the postponed main requests. At last, the place S27 is necessary to input the interruption handling request. This request is the result of the interruption which is modeled by the transition I24. To fix the current model time before starting processing a request, represented by a token in S211, an imbedded function CLOCK is used in the statement TRANS Y23 corresponding to a data transformation function \( m \) of the elementary net.

The place S211 is the main input place for I24, and S212 is the interrupting one. When the activity interval of the transition I24, caused by the main request, is interrupted by a signal, represented by a token in the place S212, then the token from S211 moves into S213 and the token from S212 enters the place S27 as a request for the interruption handling. From S27, after firing the transition Y23, the interruption handling request moves into the place S211, and the interruption handling starts.
The activity interval of the transition I24 is calculated, in the statement TIME I24, as the value of the attribute WORK of the token in S211. This value is the time to process the main request or the interruption. The statement TRANS I24, corresponding to the data transformation function $m$ of the elementary net, provides the calculation of the processing time (see the attribute WORK of a token in S213) which is left after firing I24. With our assumptions, this time is always zero for an interruption, but may be nonzero for the main request (if the processing of the request was interrupted). Note that, in the statement TRANS I24, the function CLOCK is used to fix the current model time.

The elementary net, associated with the transition X24, performs a decision operation. The statement CONTR X24 implements checking the current value of the attribute WORK of a token in the place S213. If this value is greater than zero (this is possible, with our assumptions, only for an interrupted main request), then the token is routed, along the first output arc, to the place S23 for the continuation of processing. On the other hand, if the
value of the attribute WORK is not greater than zero (this means that the token in S213 represents the completed request or interruption), then the token is routed, along the second output arc, to the place S28 to move to another part of a system.

Example 4.

This is another example of a model with interruptions (Fig. 6.4). It is assumed that there are two inputs to the modeled system. One input is for some requests, and the second input is for interruptions. The specified attributes are not used in the model. The model does not need any initial marking since tokens are generated by transitions T1 and T3. To collect the overall statistics, it is necessary to apply, before starting a simulation run, only one MCL statement STATISTICS ALL;

```
SEGMENT GEN3,TICK=MSEC;
ATTRIBUTES      A1,A2:  INTEGER;
                A3:     REAL;
DATA  DEL1/10/,DEL2/3/,DEL3/3/: INTEGER;
NET T1: /S1;
NET I2: S1,S2/S3,S4;
NET T3: /S2;
NET T4: S3/;
NET T5: S4/;
TIME T1: %DELAY := %DEL1;
TIME I2: %DELAY := %DEL3;
TIME T3: %DELAY := %DEL2;
SEGEND.
```

![Fig. 6.4. One more model with interruptions.](image)

Example 5.

This is a model of a two-processor system with a common bus and a common (shared) memory (Fig. 6.5). Moreover, each processor has its own, local memory. Each processor alternately accesses its local memory and the common memory. Conflicts are possible if both processors try to access the common memory at the same time (which is not allowed). Intervals of using the local and common memory by a processor are random, with exponential probability distribution and mean values L1 and L2 (for local memories) and M[I] (for the use of the common memory by processor I).

The goal of simulation can be to evaluate utilization of the common bus (common memory) and the slow-down of processors due to conflicts. This simple system can be investigated also analytically.
In the model, the default values for L1, L2, M[1], and M[2] are set to 40.0 time units. These can be changed by a MCL statement as shown below.

```plaintext
SEGMENT TWOPRC, TICK=MSEC;
ATTRIBUTES
   PNUM:   INTEGER;   (* Processor ID *)
DATA
   L1/40.:/ REAL;
   L2/40.:/ REAL;
   M/40.,40.:/ ARRAY[2] OF REAL;
NET  T1: S1/S3;
NET  T2: S2/S4;
NET  Y3: S3,S4/S5;
NET  X4: S5/S1,S2;
TIME T1: %DELAY := EXPON(1,%L1);
TIME T2: %DELAY := EXPON(1,%L2);
TIME X4: %DELAY := EXPON(1,%M[S5.PNUM]);
CONTROL X4: %OUT := %S5.PNUM;
SEGEND.
```

Fig. 6.5. A model of a two-processor system.

Before starting a simulation run, the following MCL statements can be applied to specify the initial marking and change the default values of parameters:

```plaintext
MARK S1[PNUM/1/],S2[PNUM/2/];
SET L1/10./,L2/80./,M/100.,100./;
STATISTICS EXCEPT Y3;
```

Example 6.

The example illustrates a multi-terminal (for two users) system (Fig. 6.6). It is assumed that the system contains a CPU, two independent disk devices, and two terminals. CPU and disks serve a request during a random exponentially distributed interval, with mean values TCPU, DSK1, and DSK2 respectively.
Each user prepares its request at the terminal during a random exponentially distributed interval with mean value THNK. After handling a request by the CPU, the request is routed to disk 1, or disk 2, or returns as a response to the user with probabilities 0.6, 0.35, and
0.05 respectively. After handling the request by any disk, it returns back to the CPU. The example can easily be extended for any number of terminals.

The corresponding MCL statements are as follows. The statement HISTO is used to tabulate the response time of the system.

```
MARK S6[NUM/1/],S8[NUM/2/];
STATISTICS ALL;
HISTO 1 (TIM,100.,1000.,15) S5;
```

Example 7.

The example illustrates the use of a multi-segment model. It implements the scheme presented in Fig. 5.1 of Section 5.

```
SEGMENT A,TICK = MSEC;
ATTRIBUTES      A1:     REAL;
                A2:     INTEGER;
                A3:     REAL;
DATA    DEL1/10/,DEL2/10/: INTEGER;
        A4  /0/          : INTEGER;
NET   T2: /S2;
NET   T1: S2/S1;
TIME  T2: %DELAY := %DEL1;
TRANS T2: %S2.A1 := %A4;
        %S2.A2 := %A4 + 1;
        %S2.A3 := %A4 + 2;
        %A4    := %A4 + 3;
ATTACH B/B1,B2/,C;
LINK A,B1:S1, S1;
LINK A.B1,B2:S2, S1;
LINK A.B2,A.C:S2, S1;
SEGEND.

SEGMENT B,TICK = MSEC;
ATTRIBUTES      A11:     REAL;
                A12:     INTEGER;
                A13:     REAL;
DATA DEL1/10/,DEL2/10/: INTEGER;
NET T1: S1/S2;
TIME T1: %DELAY := %DEL1;
SEGEND.

SEGMENT C, TICK = MSEC;
ATTRIBUTES      A21:     REAL;
                A22:     INTEGER;
DATA DEL1/10/,DEL2/10/: INTEGER;
FC       : ARRAY [4] OF REAL;
NET T1: S1/Q1;
TRANS   T1: %FC[1] := %S1.A1;
```
Example 8.

The example illustrates the use of a Pascal section which contains two independent functions to be referenced in procedures associated with transitions of the model. The model corresponds to a three-processor system with a common bus (and common memory, see Fig. 6.7). Each processor has also its own local memory (Example 5).

In the example, the function TimeDel() uses only one parameter passed on value. The function computes the delay time. The actual parameter is the net variable TL. Note that the parameter and returned value are of REAL type of MDL.

The second function DatFun() illustrates the use of parameters passed on reference. The actual parameters are net variables TL, DT1, and DT2. In the function definition, real-valued parameters are declared of type REAL, and the integer parameter must be declared of type INTEGER. The function modifies two parameters. The effect of statements WRITELN can be seen from the file MULT.LOG that is automatically created during simulation. To check the model, it is sufficient to specify the simulation time 1000 ms.

```pascal
(* Multiprocessor system with a common bus *)
(* Author: A.E. Kostin *)
(* Attributes: *)
(* *)
(* NUM - Id of a processor; *)
(* TIM - Mean time of using common memory by a processor. *)
(* *)
(* Net variables: *)
(* *)
(* TL - Mean time of using local memory by a processor. *)
(* *)

SEGMENT MULT, TICK = MCSEC; (* Segment title *)

(* Attributes declarations *)
ATTRIBUTES NUM : INTEGER;
TIM : REAL;

(* Net variables *)
DATA TL /50./ : REAL;
DT1 /10/ : INTEGER;
DT2 /20.0/:real;

INTERFACE

IMPLEMENTATION

}```
(* TimeDel function calculates random time *)
(* Input parameter: Tm - base time. *)
(* The function returns calculated time. *)

FUNCTION TimeDel (Tm: REAL): REAL;
begind
    if FRANDOM(3) <= 0.5 then
        TimeDel := EXPON(1, Tm)
    else
        TimeDel := UNIFRM(1, 0, Tm);
end;

Function Datfun(var DL:REAL; var DT1: integer; var DT2:real):REAL;
begind
    DT1:= 5;
    DT2:= 7.4;
    writeln ('In Datfun:', DL, DT1:3, DT2);
end;

(* Body of E-net model *)
NET T1 : S1/S201;
TIME T1 : %DELAY := TimeDel(%TL);
TRANS T1 : Datfun(%TL, %DT1, %DT2);
    writeln('In TRANS T1:', %DT1:3, %DT2);

NET T2 : S2/S202;
TIME T2 : %DELAY := TimeDel(%TL);
NET T3 : S3/S203;
TIME T3 : %DELAY := TimeDel(%TL);
NET Y200 : S201,S202,S203/Q300;
NET T300 : Q300/S100;
NET X100 : S100/S1,S2,S3;
    (* Using common memory *)
TIME X100 : %DELAY := TimeDel(%S100.TIM);
    writeln('Processor Id: ', %S100.NUM,
        ' Time of using common memory: ', %DELAY);
CONTROL X100 : %OUT=%S100.NUM;
    (* Routing a token *)
SEGEND.

The corresponding MCL statements are:

MARK S1[NUM/1/,TIM/20./];
MARK S2[NUM/2/,TIM/20./];
MARK S3[NUM/3/,TIM/20./];
STATISTICS ALL;
Example 9.

This model (Fig. 6.8) can be used to better understand the behavior of an elementary net of I type. Transition I1 fires periodically and it is interrupted each second firing interval. Simulation data show that because of interruptions, average firing time of I1 is 750 ms. If there were no interruptions then, according to the statement TIME I1, average firing time would be 1000 ms.

Fig. 6.7. Model of a three-processor system with a common bus.
Below are simulation results for this model. The results correspond to simulation time of $1e+7$ ms. Note that the simulation has been done with random number generator which was later modified, so that the results may be slightly different with the new random number generator.
Simulation run name: INET
Date and time: 12 June 2002, 15:57:59
Head segment: INET (6/12/2002 15:54)
File of MCL statements: C:\Winsim\MODELS\INETTEST\INET.JZP
Simulation time: 1.000000E+007 ms
Starting time to collect statistics: 0.000000E+000

Example 10.

This example illustrates the use of different servicing modes of a queue (Fig. 6.9). The first 4 tokens produced by elementary net T1 are put in queue place Q1, all subsequent tokens from T1 are absorbed by T2. The 4 tokens are stored in Q1 in the order of increasing value of attribute NUM: 4, 3, 2, 1, so that the token with attribute NUM = 1 is at the head of the queue. Elementary net T4 activates every time when T3 fires. In the text of the model, T4 uses the mode HIGH(NUM) to extract and service tokens from Q1, in the order of decreasing of value of attribute NUM. If one performs a simulation run (with an MCL statement STATISTICS ALL; and the duration of simulation 1000 ms, for example), then the corresponding LOG file will contain four lines

100 4
This indicates that at times 100, 200, 300, and 400 ms the serviced tokens extracted from Q1 had attribute value 4, 3, 2, and 1, respectively. Other servicing modes are FIFO (default), LIFO, RAND, and LOW(NUM). For this example, mode LIFO will give the same result as mode HIGH(NUM). Note that, for RAND mode, tokens will be extracted from Q1 in a random order (for example, 2, 4, 1, 3 of attribute NUM).

Example 11.

This is the model of a queuing system M/M/c (the queuing system with c = 5 identical parallel servers, with exponential probability distribution for interarrival times of customers and service times of a server, infinite queue capacity and infinite population of customers) (Fig. 6.10). Control procedure of elementary net X1 is used to distribute customers randomly and uniformly among servers. The model will work also without this control procedure, however in this case the servers will be loaded non-uniformly. With mean service time 500 time units (see net variable TSRV) and mean interarrival time of customers TGEN = 200 time units, the theoretical load of a server is 0.5. In the model, the
average loads of servers are represented by average firing times of transitions $T_1 - T_5$, average number of waiting customers and average waiting time are represented by the average length and the average token time of queue place $Q_1$, respectively.

To run the simulation, one need to apply the following MCL statements (for $TGEN = 200$):

```
SET  TGEN [200];
STATISTICS ALL;
```

The simulation time can be specified, for example, $1e7$ ms.

```
(* *********************************************** *)
(* A queuing system M/M/c                        *)
(* with c = 5 identical servers                  *)
(*                                                *)
(* File: MMC5.JOM       Date: 5.6.2002           *)
(* *********************************************** *)
SEGMENT MMC5, TICK = MSEC;

ATTRIBUTES
  WRK: INTEGER;                          (* Not used *)

DATA
  TGEN /0.0/ : REAL;                    (* Mean interarrival time *)
  TSRV /500.0/: REAL;                   (* Mean service time *)

INTERFACE
IMPLEMENTATION
VAR index: integer;
  inputs: array [1..5] of integer;
END.

(* Generator of arrivals, exponential interarrival times *)
NET  T1000: /Q1;
TIME  T1000: %DELAY := EXPON(1,%TGEN);

(* Taking next arrival from the queue to a random free server *)
NET  X1: Q1/ S1, S2, S3, S4, S5;
CONTR X1: %OUT := 0;
  if (%S1 = 0) OR (%S2 = 0) OR (%S3 = 0) OR (%S4 = 0)
    OR (%S5 = 0)
    then begin                          (* Servers are chosen randomly, *)
      inputs[1]:= %S1;                  (* with the same probability *)
      inputs[2]:= %S2;
      inputs[3]:= %S3;
      inputs[4]:= %S4;
      inputs[5]:= %S5;
      while (%OUT = 0) do begin
        index := IUNIFR(7, 1, 5);
        if (inputs[index] = 0)
          then %OUT := index;
      end
    end;
```
Example 12.

This is the model of a finite-population queuing system M/M/1/k, with one server and k = 6 customers (Fig. 6.11). The only attribute of tokens in the model is the identifier of a customer. Net variables TGEN and TSRV are desired mean “thinking” time of a customer and mean service time. Before starting a simulation run, the following MCL statements should be applied:

```mcl
SET TGEN /5200.0/;
MARK S1[ ID/1/], S2[ ID/2/], S3[ ID/3/], S4[ ID/4/], S5[ ID/5/], S6[ ID/6/];
STATISTICS ALL;
```

The first MCL statement specifies TGEN = 5200 units of time which, with k = 6 and TSRV = 500 units of time, corresponds to theoretical server’s load 0.5. The second MCL statement marks places S1 – S6 and assigns to attribute ID in these places the values 1 – 6, respectively.

Simulated load of the server is represented by utilization of transition X1. Average number of customers in the system and average response time are represented by the average length and by average token time of queue place Q1. Theoretical values of these two performance measures (with given parameters) are approximately 0.8 and 805 time units.

Control procedure of X1 is used to route each customer, after servicing, back to its thinking mode which is represented by one of elementary nets T1 – T6.
(****************************************************************
(*          A finite-population queuing system M/M/1/k          *)
(*                with k = 6 identical clients                  *)
(*                                                              *)
(*               File: MM1k.JOM         Date: 5.6.2002          *)
(****************************************************************)

SEGMENT MM1K, TICK = MSEC;

ATTRIBUTES
   ID: INTEGER;                           (* ID of client *)

DATA
   TGEN /0.0/ : REAL;        (* Mean thinking time of a client *)
   TSRV /500.0/: REAL;                     (* Mean service time *)

(* Client 1 *)
NET   T1: S1/S11;
TIME  T1: %DELAY := EXPON (1, %TGEN);

(* Client 2 *)
NET   T2: S2/S12;
TIME  T2: %DELAY := EXPON (2, %TGEN);

(* Client 3 *)
NET   T3: S3/S13;
TIME  T3: %DELAY := EXPON (3, %TGEN);

(* Client 4 *)
NET   T4: S4/S14;
TIME  T4: %DELAY := EXPON (4, %TGEN);

(* Client 5 *)
NET   T5: S5/S15;
TIME  T5: %DELAY := EXPON (5, %TGEN);

(* Client 6 *)
NET   T6: S6/S16;
TIME  T6: %DELAY := EXPON (6, %TGEN);

(* Collecting of clients' requests in the server's queue *)
NET   Y1: S11, S12, S13, S14, S15, S16/Q1;

(* Server *)
NET   X1: Q1/S1, S2, S3, S4, S5, S6;
CONTR X1: %OUT := %Q1.ID;            (* Response to the client *)
TIME  X1: %DELAY := EXPON (7, %TSRV);

SEGEND.
Example 13.

This example illustrates the use, in an independent function, of a two-dimensional integer array declared in DATA section of the segment. Note that, in the independent function Sum, the corresponding array should be declared as OF INTEGER ABSOLUTE QUEUE.

```
SEGMENT DIMEN2, TICK = MSEC;

ATTRIBUTES
  MTYP: INTEGER;

DATA
  QU/1,2,3,4,5,6,7,8,9,10,11,12/ : ARRAY [4,3] of Integer;
  AA : Integer;

INTERFACE
IMPLEMENTATION
(******************************************************************************)
(* Calculates sum of elements of a two-dimensional array *)
(******************************************************************************)
Function Sum (var QUEUE; N1,N2: integer): integer;
var
  Dbl : array[1..4,1..3] of integer absolute QUEUE;
  i,j : integer;
```
S : integer;
begin
  S := 0;
  for i := 1 to N1 do
    for j := 1 to N2 do
      S := S + Dbl[i, j];
  Sum := S;
end;

NET T1: /S1;
TRANS T1:
  %AA := Sum(%QU, 4, 3);
  WRITELN(%AA);
SEGEND.
7. CONSTRAINTS IN THE IMPLEMENTATION OF MDL AND MCL

Currently, MDL and MCL are implemented with the following constraints:
1. Maximal length of a segment name is 6 symbols.
2. Maximal length of a model name is 6 symbols.
3. Maximal number of fields in the compound name of a segment is 7.
4. Maximal number of symbols in the compound name of a segment is \((6 \times 7) + 6 = 48\), including six delimiting commas.
5. Maximal number of token attributes is 320. At least one attribute must be declared in a segment.
6. Maximal number of symbols in the name of an attribute or of a net variable is 4.
7. Maximal number of input places in a transition is 255.
8. Maximal number of output places in a transition is 255.
9. Maximal number in a numerical part of transition identifier is 65535. Transitions of different types of elementary nets may have the same number. Thus, 65535 is the maximal number of elementary nets of the same type in the segment.
10. Maximal number in a numerical part of place identifier is 65535. Places of different types (S and Q) may have the same number. Thus, a segment may have up to 65535 places of type S and up to 65535 places of type Q provided that the structure of the segment, together with transitions, occupies not more than 64K bytes of memory.
11. Dimensionality of an array in DATA section is 1 or 2.
12. Maximal size of an array in DATA section is 32767.
13. Maximal length of an MDL or MCL statement is 126 symbols (including separating blanks).
14. Size of the table of names in the segment is 200 entries.
15. Size of the table real constants in the segment is 50 entries.
16. Maximal number of Pascal words BEGIN in a segment is 1000.
17. Maximal number of copies in one MDL statement ATTACH is 10.
18. Maximal number of segment names in one MCL statement FOR is 100.
19. Maximal number of linked places in one statement MDL LINK is 20.
20. Maximal number of stopping conditions in one MCL statement STOP is 8.
21. Maximal number of intervals in a MCL statement HISTO is 255.
22. Maximal number of net variables in one MCL statement SET is 100.
23. Maximal number of integer constants in one MCL statement SET is 50.
24. Maximal number of real constants in one MCL statement SET is 100.
25. Maximal number of array elements in one MCL statement SET is 10000.
26. Maximal number of places and transitions in one MCL statement STATISTICS is 50.
27. Maximal number of hierarchical levels of segments in a model is 4.
28. Maximal real-valued model time is MAXREAL = 1.7 E+308 time units.
29. Maximal duration of activity of a transition (maximal delay) is MAXREAL.
30. Maximal number of tokens in a queue place is 255.
31. Maximal size of a record for attributes of a token is 32768 bytes (octets).
32. Maximal sample size to create a histogram is 32767.
33. An integer attribute or an integer net variable has the size of 32 bits, and a real-valued attribute or a real-valued net variable has the size of 64 bits.
34. Maximal value of an attribute or a net variable of REAL type is 1.7 E+308.
35. Maximal value of an attribute or a net variable of INTEGER type is 2 147 483 647.
8. INSTALLATION AND USE

8.1 Installation

The system is available for downloading in the form of a ZIP-compressed file. The size of this file is approximately 2.9MB and can change depending on the version of the system.

The user should create, on his/her disk, the folder WINSIM, put the downloaded ZIP file of the system Winsim.zip in this folder, and unzip this file here. After this installation, no information is written in the Windows registry.

As a result of unzipping, the folder WINSIM will have three new folders BIN, UNITS, and MODELS.

The folder BIN contains the files of the WINSIM executive and the MDL compiler. It is desirable to create a shortcut of WINSIM using the icon from this folder.

The folder UNITS includes precompiled Pascal units of the system to be linked with each model.

The folder MODELS, in the downloaded system, contains only one demo model in the subfolder RICART. This is the model to simulate the often-cited Ricart and Agrawala distributed mutual exclusion algorithm [30]. The model is for the case of 3 processes, but it can be easily extended to a larger number of processes.

The model, in the folder RICART, will be used to describe the steps in simulation with WINSIM. The model includes the following files:

- **RICART.JOM** – is the segment simulating the work of a process in the algorithm.

- **NET03.JOM** – is the head segment which simulates a network to pass messages between processes. The segment corresponds to the case of 3 processes. It attaches the segment RICART.JOM, specifies 3 copies of processes and links between copies of processes and the network segment.

- **NET03.JZP** – is the file of parameters to be applied immediately before starting a simulation run.

- **RICART.PDF** – is the file containing the E-net scheme of a process. This file will not be used in simulation and serves only to simplify, to a user, the understanding of the model.

- **NET03.PDF** – is the file containing the E-net scheme of the network part of the model. This file is not used in simulation.
8.2 Steps in the use of WINSIM

The use of WINSIM for simulation is explained with the example of distributed mutual exclusion algorithm of Ricart and Agrawala. The necessary source texts of the model are given in X:\WINSIM\MODELS\RICART\ where X is the drive on which WINSIM has been installed as was described in the previous section. To process the model’s files and simulate, the user should perform the following actions.

1) Start WINSIM. In the appeared window “Pick” (Fig. 8.1), press “Open” and open the file RICART.JOM from \WINSIM\MODELS\RICART\. The user will see the MDL source text of segment RICART (Fig. 8.2). If necessary, any desired modification of the segment text can be done in this window.

![Fig. 8.1. The window "Pick"](image)

2) In menu “Segment” select “Compile” option. The segment will be compiled by MDL compiler. If there are syntactical errors, then the user will see the corresponding diagnostic messages in a separate window. After the successful MDL compilation, the user can select “Properties” to be sure that two new files RICART.PAS and RICART.NET have been created by the MDL compiler. RICART.PAS is the Pascal unit in which all Pascal procedures and functions of the segment are collected, and RICART.NET contains the structural representation of the segment (that is, all elementary nets, with their interconnections).

3) In “File” menu, open now the head segment NET03.JOM. Perform the MDL compilation of this file (if necessary, after doing any modification of it) as was done for segment RICART.JOM. The result is two new files NET03.PAS and NET03.NET. For an arbitrary model, the user should compile separately each segment. It is recommended to compile the head segment of any model as the last one. In this case, the model will be assigned (on default) the name of the head segment.
4) In menu “Model”, select “New Model”. The user will see the default name of the model to be created and the name of the head segment. If the head segment was compiled as the last one, then the model will have the name of the head segment. In this example, the head segment name is NET03.JOM and the default model name is NET03.EXE, since the model will be created as an executable file (Fig 8.3). Press “OK”. In menu “Model”, the user can use the option “Show” to see components of the model.
5) In menu “Model”, select “Make” to create the executable file of the model. At this stage, *.PAS files of the model will be automatically compiled by the command-line Object Pascal Compiler (which is a component of the system), and Pascal syntactical errors, if any, will be indicated. In case of Pascal errors, the user should return back to open the corresponding segment, to correct it (in procedures and functions used by transitions) and to recompile the segment by MDL compiler. If the corrected segment is not the head segment, then the user is recommended to recompile also the head segment to ensure that the model will have the name of the head segment.

6) In menu “Model”, select now “Run” to prepare the model for a simulation run. As a result, the window “Parameters of Run” will appear (Fig 8.4). The user should, first of all, specify, in this window:
   a. Name of run (or leave the default name, which is NET03 in this example).
   b. Simulation time (for example, 1e+7 for this example is appropriate).
   c. Starting time to collect statistics (it can be left to be zero).
   d. Steps to suspend the simulation run, if necessary, or leave the maximum time as shown in the box.

![Fig. 8.4. The window "Parameters of Run"

7) Now, MCL statements from a *.JZP file should be applied to the model. In this example, the MCL statements are in file NET03.JZP (it is recommended to use, for this file, the name of head segment). Open this file by clicking the symbol on the right side of the box “MCL statements”. The user will see a new window “MCL statements” with the statements of the opened *.JZP file (Fig 8.5). Any modifications and additions can be done in this file if necessary. In this example, parameter TGEN = 635 ms specifies the mean
thinking time of a process. Since the mean time of using the shared resource \(\text{TSR} = 500\) ms (it is fixed in RICART.JOM), this corresponds, for a finite-population queuing system, to the high load (0.9) of the use of shared resource by processes. After possible modifications and/or additions of MCL statements, press “Apply” to apply the MCL statements to the model. Then close this window to return to the window “Parameters of Run”.

8) In the window “Parameters of Run”, press “OK”. The simulation run window having the name of the model will appear. In this example, this name is NET03.EXE (Fig. 8.6). The simulation run time shows simulation time (current time, time of the next stop, and finish time) and has 4 buttons: “Parameters of run”, “Save Model”, “Load Model”, and “START”. The button “Parameters of run” can be used to check the parameters applied to the model at the previous step. The button “Load Model” is used to load the state of a previously saved model. The button “START” initiates the simulation run. Press it to start the simulation run. When simulation proceeds, the user will see, in the box “SIMULATION TIME”, the progress of simulation time. If there are no simulation errors, the end of simulation run will be indicated by message “SIMULATION TERMINATED”.

![MCL Statements](image)

**Fig. 8.5. The window "MCL statements"**
During the simulation, the user can suspend the run and watch the intermediate results of simulation. In addition, by the button “Save Model”, the user can save the current state of the suspended model on disk to be resumed at some time later. It is important to know, that only data structures of the suspended model are saved in this operation. Later, when the user wants to continue the saved model, he/she must initially recreate it and then, without applying any parameters, restore the saved data structures of the model by the button “Load Model” in the simulation run window. This will result in the restoring of the exactly same model which was suspended and saved earlier. In this restoration, the system will check the correspondence of the saved data structures to the recreated model.

![Fig. 8.6. The simulation run window](image)

Fig. 8.6. The simulation run window
After this, in the same window, select menu “Statistics” and the option “Information and Statistics” and press “SHOW”. As a result, results of the simulation run, in the form of standard tables, will appear in the window “Statistics for NET03 run” (Fig 8.7). The user can print and/or save these results for the subsequent off-line analysis. The interpretation of results depends on the concrete model and on the association done by the modeler between the modeled system and the transitions and places of the model. For the example of the Ricart and Agrawala model, the interpretation of the results and some performance measures can be summarized in Table 8.1 (for simulation time 1e+7 ms). These results are derived from raw simulation data of segments NET03.PRC01, NET03.PRC02 and NET03.PRC03, where PRC01, PRC02, and PRC03 are copies of segment RICART.JOM.

From this table, the overall average response time of a process is 1020 ms, and total number of the use of shared resource by all processes is 18135. Moreover, transition Y1 in NET03.JOM fired 54405 times; this represents the total number of messages sent in the network. Thus, the average number of messages per use of shared resource by a process is 54405/18135 = 3, as should be for this model.

9) The user now can remove the window “Statistics for NET03 run” and close the window “Information and statistics”, to return to the simulation run window “NET03.EXE”. At this stage, the user can select “New Run” to make a new simulation run of the model, possibly after making some changes in MCL statements. In the new run, even
if it is done with the unchanged MCL statements, the random number generators of the system are not reset, so that the simulation statistics will be slightly different from the previous run. If the user wants to do a new simulation run with exactly the same sequence of random numbers, then he/she should restart the system.

Table 8.1. Summary of the results for Ricart & Agrawala model  
(for three processes)

<table>
<thead>
<tr>
<th>Process</th>
<th>Average waiting time, ms (Token time in place S20)</th>
<th>Average time of using the shared resource, ms (Token time in place S21)</th>
<th>Average response time, ms (Total token time in S20 and S21)</th>
<th>Number of times the shared resource is accessed (Number of firings of transition T2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>526.8</td>
<td>499.3</td>
<td>1026.1</td>
<td>6024</td>
</tr>
<tr>
<td>2</td>
<td>526.1</td>
<td>488.5</td>
<td>1014.6</td>
<td>6063</td>
</tr>
<tr>
<td>3</td>
<td>531.2</td>
<td>489.4</td>
<td>1020.6</td>
<td>6048</td>
</tr>
</tbody>
</table>

10) If no more runs are necessary, the user should close the simulation run window (NET03.EXE, in this example), and then exit the system from “File” menu.
APPENDICES

SYNTACTICAL DESCRIPTION OF LANGUAGES

The syntax of MDL and MCL is formally described in the extended Bacus-Naur form. In this form, an expression $A ::= B$ specifies a grammatical rule, where $A$ is a specified symbol, $B$ is a string that generally contains terminal and non-terminal symbols of a language, and “::=” is a meta-symbol that is read as “is defined as”.

To distinguish terminal and non-terminal symbols, the latter ones are enclosed in the metalinguistic brackets ‘<’ and ‘>’. To make expressions more short, all grammatical rules with the same left part are written as one combined rule by the use, in the right part of the rule, the metalinguistic symbol ‘|’ that is read as “or”. For example, the rule

$$\text{<binary digit>} ::= 0|1$$

can be read “a binary digit is defined as 0 or 1”.

To denote that some sequence of symbols may be repeated any number of times, including zero number of times, we use curly braces ‘{‘ and ‘}’. For example, the expression \{A\} means $\emptyset|A|AA|AAA|...$

If the maximal number of repetitions of a symbol in curly braces is limited by some constant, then this constant is written after the closing brace. Thus, \{A\}3 means $\emptyset|A|AA|AAA$. 

APPENDIX 1

SYNTAX OF THE MODEL DESCRIPTION LANGUAGE

1. Description of a segment

M1. <SEGMENT> ::= SEGMENT <SEGMENT NAME>, TICK = <TIME UNIT>;  
    <SEGMENT BODY> 
    SEGEND.

M2. <SEGMENT NAME> ::= <IDENTIFIER>

M3. <TIME UNIT> ::= PSEC | NSEC | MCSEC | MSEC | SEC | MINUTE | HOUR | DAY

M4. <SEGMENT BODY> ::= <SECTION OF TOKEN ATTRIBUTES>  
    <SECTION OF NET VARIABLES>  
    <PASCAL SECTION>  
    <SECTION OF ELEMENTARY NETS>  
    <SECTION OF ATTACHED SEGMENTS>  
    <SECTION OF SEGMENT LINKING>

2. Description of attributes and net variables

B1. <SECTION OF TOKEN ATTRIBUTES> ::= ATTRIBUTES  
    <LIST OF ATTRIBUTES OF THE SAME TYPE>  
    {...<LIST OF ATTRIBUTES OF THE SAME TYPE>} 

B2. <LIST OF ATTRIBUTES OF THE SAME TYPE> ::= <ATTRIBUTE NAME>  
    {,<ATTRIBUTE NAME}>:  
    <ARITHMETICAL TYPE>;

B3. <ARITHMETICAL TYPE> ::= INTEGER | REAL

B4. <SECTION OF NET VARIABLES ::=  
    DATA <LIST OF NET VARIABLES OF THE SAME TYPE>;  
    {...<LIST OF NET VARIABLES OF THE SAME TYPE>};

B5. <LIST OF NET VARIABLES OF THE SAME TYPE> ::=  
    <LIST OF SIMPLE VARIABLES> | <LIST OF ARRAYS>
B6. \(<\text{LIST OF SIMPLE VARIABLES}> ::= <\text{SIMPLE VARIABLE}>\)
\{,<\text{SIMPLE VARIABLE}>\}; <\text{ARITHMETICAL TYPE}>

B7. \(<\text{SIMPLE VARIABLE}> ::= <\text{VARIABLE NAME}> | <\text{VARIABLE NAME}> /<\text{INITIAL VALUE}>/

B8. \(<\text{INITIAL VALUE}> ::= <\text{NUMBER}> | (<\text{REPEATER}>)<\text{NUMBER}>

B9. \(<\text{REPEATER}> ::= <\text{UNSIGNED INTEGER}>

B10. \(<\text{LIST OF ARRAYS}> ::= <\text{DESCRIPTOR}> \{,<\text{DESCRIPTOR}>\}; <\text{ARITHMETICAL TYPE}>
\text{ARRAY ['<\text{BOUNDARY INDEX}>\{,<\text{BOUNDARY INDEX}>\}1] OF <\text{ARITHMETICAL TYPE}>

B11. \(<\text{DESCRIPTOR}> ::= <\text{IDENTIFIER}> | <\text{IDENTIFIER}> /<\text{VALUES OF ELEMENTS}>/

B12. \(<\text{VALUES OF ELEMENTS}> ::= <\text{INITIAL VALUE}> \{,<\text{INITIAL VALUE}>\}

B13. \(<\text{BOUNDARY INDEX}> ::= <\text{UNSIGNED INTEGER}>

B14. \(<\text{ATTRIBUTE NAME}> ::= <\text{IDENTIFIER}>

B15. \(<\text{VARIABLE NAME}> ::= <\text{IDENTIFIER}>

3. Section of elementary nets

E1. \(<\text{SECTION OF ELEMENTARY NETS}> ::= \{<\text{ELEMENTARY NET SPECIFIER}>\}

E2. \(<\text{ELEMENTARY NET SPECIFIER}> ::= <\text{NET STRUCTURE}>; | <\text{CONTROL PROCEDURE}>; | <\text{DELAY PROCEDURE}>; | <\text{TRANSFORM PROCEDURE}>;

E3. \(<\text{NET STRUCTURE}> ::= <\text{NET}> <\text{TRANSITION IDENTIFIER}> : <\text{SCHEME}>

E4. \(<\text{TRANSITION IDENTIFIER}> ::= <\text{NET TYPE}> <\text{TRANSITION NUMBER}>

E5. \(<\text{NET TYPE}> ::= T | X | Y | G | I
E6. \(<\text{TRANSITION NUMBER}\> ::= <\text{UNSIGNED INTEGER}>\)

E7. \(<\text{SCHEME}\> ::= <\text{LIST OF INPUT PLACES}> /  
\quad <\text{LIST OF OUTPUT PLACES}> |  
\quad <\text{LIST OF INPUT PLACES}> |  
\quad / <\text{LIST OF OUTPUT PLACES}>\)

E8. \(<\text{LIST OF INPUT PLACES}\> ::= <\text{LIST OF PLACES}>\)

E9. \(<\text{LIST OF OUTPUT PLACES}\> ::= <\text{LIST OF PLACES}>\)

E10. \(<\text{LIST OF PLACES}\> ::= <\text{PLACE}> {,<\text{PLACE}>}\)

E11. \(<\text{PLACE}\> ::= <\text{PLACE IDENTIFIER}> |  
\quad <\text{PLACE IDENTIFIER}> [<\text{SERVICING RULE}>]\)

E12. \(<\text{PLACE IDENTIFIER}\> ::= <\text{PLACE TYPE}> <\text{PLACE NUMBER}>\)

E13. \(<\text{PLACE TYPE}\> ::= S | Q\)

E14. \(<\text{PLACE NUMBER}\> ::= <\text{UNSIGNED INTEGER}>\)

E15. \(<\text{SERVICING RULE}\> ::= \text{FIFO} | \text{LIFO} | \text{RAND} |  
\quad \text{LOW (<ATTRIBUTE NAME>)} | \text{HIGH (<ATTRIBUTE NAME>)}\)

E16. \(<\text{CONTROL PROCEDURE}\> ::= \text{CONTROL}  
\quad <\text{TRANSITION IDENTIFIER}>:: <\text{STATEMENT}>{;<\text{STATEMENT}>}\)

E17. \(<\text{DELAY PROCEDURE}\> ::= \text{TIME}  
\quad <\text{TRANSITION IDENTIFIER}>:: <\text{STATEMENT}>{;<\text{STATEMENT}>}\)

E18. \(<\text{TRANSFORM PROCEDURE}\> ::= \text{TRANSFORM}  
\quad <\text{TRANSITION IDENTIFIER}>:: <\text{STATEMENT}>{;<\text{STATEMENT}>}\)

Note: <\text{STATEMENT}> may contain references to imbedded functions of the MDL (see Appendix 3).

4. Section of attached segments

A1. \(<\text{SECTION OF ATTACHED SEGMENTS}\> ::=  
\quad \text{ATTACH} <\text{LIST OF SEGMENTS}>\)

A2. \(<\text{LIST OF SEGMENTS}\> ::=  
\quad <\text{ORIGINAL SEGMENT}> / <\text{COPY SEGMENT}> {,<\text{COPY SEGMENT}>}\)
A3. `<ORIGINAL SEGMENT>` ::= `<SEGMENT NAME>`

A4. `<COPY SEGMENT>` ::= `<SEGMENT NAME>`

5. Section of linking of segments

L1. `<SECTION OF LINKING OF SEGMENTS>` ::=  
   `LINK <SEGMENT PAIR> : 
   <PLACE PAIR> {/<PLACE PAIR>}`

L2. `<SEGMENT PAIR>` ::= `<COMBINED NAME> , <COMBINED NAME>`

L3. `<COMBINED NAME>` ::= `<SEGMENT NAME> {.<SEGMENT NAME>}`

L4. `<PLACE PAIR>` ::= `<PLACE IDENTIFIER>, <PLACE IDENTIFIER>`

6. Variables and references to net objects

D1. `<Variable>` ::= `<Pascal-variable> | 
   <component Pascal-variable> | 
   <Pascal-pointer> | 
   <REFERENCE TO NET OBJECT>`

D2. `<REFERENCE TO NET OBJECT>` ::= 
   `<REFERENCE TO SIMPLE VARIABLE> | 
   `<REFERENCE TO ARRAY ELEMENT> | 
   `<REFERENCE TO TOKEN ATTRIBUTE> | 
   `<REFERENCE TO TRANSITION> | 
   `<REFERENCE TO PLACE>`

D3. `<REFERENCE TO SIMPLE VARIABLE>` ::= `%<IDENTIFIER>`

D4. `<REFERENCE TO ARRAY ELEMENT>` ::= `%<IDENTIFIER> 
   [<EXPRESSION> 
   {,<EXPRESSION>}]`  

D5. `<REFERENCE TO TOKEN ATTRIBUTE>` ::= 
   `%<PLACE IDENTIFIER>.<ATTRIBUTE NAME>`
D6. \(<\text{REFERENCE TO PLACE}> ::= \%<\text{PLACE IDENTIFIER}>|\%
\<\text{PLACE IDENTIFIER}>.\)
\<\text{STANDARD NUMERICAL ATTRIBUTE}>\n
D7. \(<\text{REFERENCE TO TRANSITION}> ::= \%
\<\text{TRANSITION IDENTIFIER}>|\%
\<\text{TRANSITION IDENTIFIER}>.\)
\<\text{STANDARD NUMERICAL ATTRIBUTE}>|\%
\<\text{TRANSITION IDENTIFIER}>.<\text{PARAMETER}>|\%
\<\text{PARAMETER}>\n
D8. \(<\text{PARAMETER}> ::= \text{IN} | \text{OUT} | \text{DELAY}\n
D9. \(<\text{STANDARD NUMERICAL ATTRIBUTE}> ::= \text{SNA1} | \text{SNA2} | \text{SNA3} | \text{SNA4} | \text{SNA5}\n
7. Pascal section

P1. \(<\text{PASCAL SECTION}> ::= \text{INTERFACE} \\text{IMPLEMENTATION} \text{<Pascal declarations and definitions> BEGIN} \text{<Initialization statements> END.}\n
APPENDIX 2

SYNTAX OF MODELING CONTROL LANGUAGE

1. FILE OF MCL STATEMENTS

<MCL TEXT> ::=  <MCL STATEMENT> {<MCL STATEMENT>}

<MCL STATEMENT> ::= 
  <SPECIFICATION OF SEGMENTS> | 
  <MARKING OF PLACES> | 
  <SETTING OF VALUES> | 
  <SPECIFICATION OF HISTOGRAM> | 
  <COLLECTION OF STATISTICS> | 
  <SPECIFICATION OF TRACING> | 
  <CANCEL OF MARKING> | 
  <SPECIFICATION OF STOP CONDITIONS> | 
  <COMMENT>

2. SPECIFICATION OF SEGMENTS WHICH NEED PARAMETERS
   (“FOR” STATEMENT)

<SPECIFICATION OF SEGMENTS> ::= 
  FOR SEGMENTS <COMBINED NAME> {,<COMBINED NAME>};
<COMBINED NAME> ::= <SEGMENT NAME> {,<SEGMENT NAME>}
<SEGMENT NAME> ::= <IDENTIFIER>
<IDENTIFIER> ::= <ALPHA> {<ALPDIT>}
<ALPDIT> ::= <ALPHA> | <DIGIT>

3. INITIAL MARKING OF PLACES   (“MARK” STATEMENT)

<MARKING OF PLACES> ::= MARK <REFERENCE TO SEGMENT> 
  <REFERENCE TO PLACE> {,<REFERENCE TO PLACE>};
<REFERENCE TO SEGMENT> ::= {SEGMENTS <COMBINED NAME>};
<REFERENCE TO PLACE> ::= <PLACE IDENTIFIER> <TOKEN REFERENCE> 
<TOKEN REFERENCE> ::= [<ATTRIBUTES>] | 
  (<TOKEN NUMBER IN QUEUE> | 
  (<TOKEN NUMBER IN QUEUE>)[<ATTRIBUTES>]

75
<PLACE IDENTIFIER> ::= <PLACE TYPE><PLACE NUMBER>
<PLACE TYPE> ::= S | Q
<PLACE NUMBER> ::= <UNSIGNED INTEGER>
<TOKEN NUMBER IN QUEUE> ::= 
    <UNSIGNED INTEGER> | 
    <UNSIGNED INTEGER> .. <UNSIGNED INTEGER>

<ATTRIBUTES> ::= 
    <ATTRIBUTE NAME> /<NUMBER>/ {,<ATTRIBUTE NAME> /<NUMBER>/}
<ATTRIBUTE NAME> ::= <NAME>
<NAME> ::= <ALPHA> {<ALPDIG>}3

4. SETTING OF VALUES OF NET VARIABLES AND NET ARRAYS
   ("SET" STATEMENT)

<SETTING OF VALUES> ::= SET <REFERENCE TO SEGMENT>
    <VALUE ELEMENT> {,<VALUE ELEMENT>};
VALUE ELEMENT ::= <REFERENCE TO VARIABLE>
    /<LIST OF VALUES>/
REFERENCE TO VARIABLE ::= <NAME> | 
    <NAME> [<INDEX> {,<INDEX>}1]
INDEX ::= <UNSIGNED INTEGER>
LIST OF VALUES ::= <VALUE> {,<VALUE}]
VALUE ::= <NUMBER> | (<REPEATER>) <NUMBER>
REPEATER ::= <UNSIGNED INTEGER>

5. SPECIFICATION OF A HISTOGRAM ("HISTO" STATEMENT)

<SPECIFICATION OF HISTOGRAM> ::= 
    HISTO <REFERENCE TO SEGMENT> <HISOGRAM DESCRIPTION>;
HISTOGRAM DESCRIPTION ::= 
    <HISTOGRAM NUMBER> (<HISTOGRAM PARAMETERS>)
LIST OF NET ELEMENTS
HISTOGRAM NUMBER ::= <UNSIGNED INTEGER>
HISTOGRAM PARAMETERS ::= <ARGUMENT>,
    <UPPER BOUND OF FIRST INTERVAL>,
    <INTERVAL WIDTH>,<NUMBER OF INTERVALS>
    {,<TIME INTERVAL FOR NFI>}1
ARGUMENT ::= <REFERENCE TO A VARIABLE> | 
    <REFERENCE TO AN ATTRIBUTE> | 
    <SPECIAL MODE> | 
    <STANDARD NUMERICAL ATTRIBUTE>
REFERENCE TO AN ATTRIBUTE ::= <NAME>
SPECIAL MODE ::= NFI | IAI
<STANDARD NUMERICAL ATTRIBUTE> ::=  
   SNA1 | SNA2 | SNA3 | SNA4 | SNA5

<UPPER BOUND OF FIRST INTERVAL> ::= <NUMBER>

<INTERVAL WIDTH> ::= <UNSIGNED NUMBER>

.NUMBER OF INTERVALS> ::= <UNSIGNED NUMBER>

<TIME INTERVAL FOR NFI> ::= <UNSIGNED NUMBER>

<LIST OF NET ELEMENTS> ::= <NET ELEMENT> {,<NET ELEMENT>}

<NET ELEMENT> ::= <PLACE IDENTIFIER> | <TRANSITION IDENTIFIER>

<TRANSITION IDENTIFIER> ::=  
   <ELEMENTARY NET><TRANSITION NUMBER>

<ELEMENTARY NET> ::= T | X | Y | G | I

<TRANSITION NUMBER> ::= <UNSIGNED INTEGER>

6. SPECIFICATION OF STATISTICS FOR PLACES AND TRANSITIONS  
("STATISTICS" STATEMENT)

<COLLECTION OF STATISTICS> ::= STATISTICS <REFERENCE TO SEGMENT> <OPERATION>;  
<OPERATION> ::= <SET STATISTICS> | <CANCEL STATISTICS>  
<SET STATISTICS> ::= <LIST OF NET ELEMENTS> |  
   <LIMIT><LIST OF NET ELEMENTS> | ALL  
<CANCEL STATISTICS> ::= CANCEL <CANCELLED ELEMENTS>  
<CANCELLED ELEMENTS> ::= {{LIST OF NET ELEMENTS}}  
<LIMIT> ::= EXCEPT

7. SPECIFICATION OF TRACING FOR A SIMULATION RUN  
("TRACE" STATEMENT)

<SPECIFICATION OF TRACING> ::=  
   TRACE <REFERENCE TO ELEMENT><TRACE OPERATION>;  
<TRACE OPERATION> ::= <SET TRACE> | <CANCEL TRACE>  
<SET TRACE> ::= <TRANSITION LIST> |  
   <LIMIT><TRANSITION LIST> | ALL  
<CANCEL TRACE> ::= STOFF <TRANSITIONS WITHOUT TRACE>  
<TRANSITIONS WITHOUT TRACE> ::= {{TRANSITION LIST}}

8. CANCEL OF PLACE MARKING  
("DEMARK" STATEMENT)

<CANCEL MARKING> ::=  
   DEMARK <REFERENCE TO SEGMENT><LIST OF CANCELLED PLACES>  
<LIST OF CANCELLED PLACES>::= 
9. SPECIFICATION OF STOP CONDITIONS  (“STOP” STATEMENT)

<SPECIFICATION OF STOP CONDITIONS> ::= 
    STOP <REFERENCE TO SEGMENT> <STOP OPERATION>;
<STOP OPERATION>::= <SET STOP>|<CANCEL STOP>
<SET STOP>::=<LIST OF PLACES>
<LIST OF PLACES>::= <PLACE IDENTIFIER>{,<PLACE IDENTIFIER>}
<CANCEL STOP>::= STOFF

10. COMMENTS  <COMMENT> ::= (* <ANY SEQUENCE OF SYMBOLS EXCEPT *)>* ) |
    /* <ANY SEQUENCE OF SYMBOLS EXCEPT */> */
APPENDIX 3

EMBEDDED FUNCTIONS OF MDL

MDL includes the following imbedded functions and procedures: CLOCK, FRANDOM, UNIFRM, IUNIFR, EXPON, NORMAL, BINOM, POISSN, TABUL, FRASP, MARKOV, STATOUT and STATRESET. They can be used in MDL statements TIME, CONTROL and TRANSFORM, and also in independent procedures and functions defined by a user in section of Pascal procedures of the segment. Below, a brief description of these functions and procedures is given.

Function CLOCK. The function returns a real-valued current simulation time expressed in time units of the segment or model. The function is declared in the system as follows:

FUNCTION CLOCK (COD: INTEGER): REAL;

Parameter COD requires that the simulation time should be given in time units of the segment containing this function (if COD = 1) or of the model (if COD = 2).

The following 10 functions implement different random number generators. The first (or the only) parameter in each of these functions is the number of one of 100 streams of random numbers. If this number is given outside the range 1 – 100, then the function uses stream 1.

Other parameters, if present, depend on the function type.

To produce basic random numbers uniformly distributed in (0, 1), the following linear congruential Lehmer’s sequence is used in the system:

\[ X_n = (630360016 \times X_{n-1}) \mod (2^{31} - 1). \]

The quality of this generator has been extensively investigated with 19 different tests and found to be high [27, see URN30, p.128]. It has the period of \(2^{31} - 1\). This generator is used in the SIMSCRIPT simulation language. The generator implemented in Winsim has been adapted from the Fortran program presented in [28, pp. 428 - 429]. However, seeds for the different streams are made 1000000 numbers apart (instead of 100000), with the first seed 1973272912.

Function FRANDOM. It returns a real-valued random number uniformly distributed in (0, 1). The function implements the congruential scheme explained above and is used to create random variates with all other probability distributions. The function is declared as follows:
FUNCTION FRANDOM (N: INTEGER): REAL;

Function UNIFRM. It returns a real-value random number uniformly distributed in the specified range. The function declaration is

FUNCTION UNIFRM (N: INTEGER; RMIN, RMAX: REAL): REAL;

where RMIN and RMAX are desired minimum and maximum values.

Function IUNIFR. It returns an integer-valued random number uniformly distributed in the specified range. The function declaration is:

FUNCTION IUNIFR (N: INTEGER; IMIN, IMAX: INTEGER): INTEGER;

where IMIN and IMAX are desired minimum and maximum values.

Function EXPON. It returns a real-valued random number exponentially distributed with the specified mean value. The function declaration is

FUNCTION EXPON (N: INTEGER; Q: REAL): REAL;

where Q is the desired mean value.

Function NORMAL. It returns a real-valued random number normally distributed with the specified mean value and standard deviation. The function declaration is

FUNCTION NORMAL (N: INTEGER; EX,STD: REAL): REAL;

where EX and STD are desired mean value (expectation) and standard deviation, respectively.

Function BINOM. It returns an integer-valued random number binomially distributed with the specified parameters. The function declaration is

FUNCTION BINOM (N: INTEGER; M: INTEGER; P: REAL): INTEGER;

where M is the number of trials, and P is probability of an event in each of these trials. The returned value is the number of these events in the series of M trials.
**Function POISSN.** It returns an integer-valued random number distributed according to Poisson distribution, with the specified parameter. The function declaration is

\[
\text{FUNCTION POISSN (N: INTEGER; L: REAL): INTEGER;}
\]

where \(L\) is the desired mean value.

**Function TABUL.** It returns an integer-valued random number corresponding to the desired discrete probability distribution. A user should specify two vectors. One is a vector of possible integer values, and the other is a vector of corresponding probabilities. Thus, these two vectors define the desired probability mass function. The function declaration is

\[
\text{FUNCTION TABUL (N: INTEGER; M: INTEGER; VAR PV: ARRAY OF REAL; VAR V: ARRAY OF INTEGER): INTEGER;}
\]

Here \(V\) is a vector of integer values, \(PV\) is a vector of probabilities of these values, and \(M\) is the size of each of these two vectors. As in all previous functions, \(N\) is the stream of random numbers.

Example of the use in a segment is as follows.

```plaintext
SEGMENT TABULT, TICK = MSEC;
ATTRIBUTES MTYP: INTEGER;
DATA
  V/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/: ARRAY [10] OF INTEGER; (* Integer values *)
  PV/0.5, 0.1, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05/: ARRAY [10] OF REAL; (* Probabilities of given integer values *)
NET      T1: /S1;
TRANS T1: %S1.MTYP := TABUL(1, 10, %PV, %V);
...
SEGEND.
```

Below is a complete example. This is a modified example 6 from Section 6.

```plaintext
(**********************************************************************)
(*            Interactive system with two terminals                *)
(*                Using embedded TABUL function                  *)
(**********************************************************************)
SEGMENT DSYST, TICK = MSEC;
ATTRIBUTES NUM    : INTEGER;                        (* Terminal ID *)
TIM    : REAL;           (* To tabulate response time *)
DATA
  TCPU/10./    : REAL;   (* Mean time of processing in CPU *)
  THNK/15000./ : REAL;   (* Mean "thinking" time *)
  DSK1/30./    : REAL;   (* Mean servicing time of Disk 1 *)
  DSK2/20./    : REAL;   (* Mean servicing time of Disk 2 *)
  V/1,2,3/ : ARRAY [3] OF INTEGER; (* Array of routes after CPU processing *)
  V/1,2,3/ : ARRAY [3] OF INTEGER; (* The sum of the probabilities must be ONE *)
```
PV/0.35, 0.6, 0.05/ : ARRAY [3] OF REAL;

(* Choosing the terminal *)
NET X1: S5/S6,8;
CONTR X1: %OUT := %S5.NUM;

(* Saving the time of request generation in an attribute *)
NET Y2: S7,9/S1;
TRANS Y2: %S1.TIM := CLOCK(1);

(* Collecting requests and passing them to CPU *)
NET Y3: S1,S2,S3/Q1;

(* CPU *)
NET T4: Q1/S4;
TIME T4: %DELAY := EXPON(1,%TCPU);

(*-------------------------------------------------------------
 Choosing the output route after CPU processing
 in accordance with the TABUL distribution of probabilities
--------------------------------------------------------------*)
NET X5: S4/Q2,Q3,S5;
CONTR X5: %OUT := TABUL
    (1, (* Random number generator # *)
     3, (* Size of vector and array *)
     %PV, (* Vector of probabilities *)
     %V); (* Array of output routes *)
TRANS X5: IF %OUT = 3 (* Calculate the response time *)
    THEN %S5.TIM := CLOCK(1) - %S5.TIM;

(* Disk 1 *)
NET T6: Q2/S2;
TIME T6: %DELAY := EXPON(1,%DSK1);

(* Disk 2 *)
NET T7: Q3/S3;
TIME T7: %DELAY := EXPON(1,%DSK2);

(* Terminal 1 *)
NET T8: S6/S7;
TIME T8: %DELAY := EXPON(1,%THNK);

(* Terminal 2 *)
NET T9: S8/S9;
TIME T9: %DELAY := EXPON(1,%THNK);

SEGEND.

Function FRASP. It returns a real-valued random number corresponding to an approximated cumulative distribution function. A user should specify a vector of desired real values, and a vector of corresponding values of distribution function. The function declaration is
FUNCTION FRASP (N: INTEGER; M: INTEGER; VAR X: ARRAY OF REAL; VAR F: ARRAY OF REAL):REAL;

Here X is the vector of possible values of random variable, F is the vector of approximating values of probability distribution function, and M is the size of these two vectors.

Example.

SEGMENT FRASPT, TICK = MSEC;
ATTRIBUTES MT: REAL;
DATA
V/1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20/: ARRAY [2] OF REAL; (* Real values *)
(* Values of approximated cumulative distribution function *)
PV/0.01, 0.5, 0.55, 0.56, 0.60, 0.61, 0.62, 0.63, 0.70, 0.80, 0.85, 0.86, 0.87, 0.90, 0.91, 0.92, 0.93, 0.94, 0.95, 1.00/: ARRAY [20] OF REAL;

NET      T1: /S1;
TRANS T1: %S1.MT := FRASP(1, 20, %V, %PV);
...
SEGEND.

Function MARKOV. It returns an integer-valued random number corresponding to the next state of a Markov process. A user should specify a square real-valued matrix of transition probabilities and a current state of the underlying Markov process. The function declaration is

FUNCTION MARKOV (N: INTEGER; M: INTEGER; STA: INTEGER;
VAR MATRIX: ARRAY OF REAL): INTEGER;

Here M is size of transition matrix (number of states in the desired Markov process), STA is the number of current state in the range from 1 to M inclusively, and MATRIX is a real-valued transition probability matrix.

Example.

SEGMENT MARKT, TICK = MSEC;
ATTRIBUTES STAT: INTEGER;
DATA
(* Square probability transition matrix *)
PV /0.1, 0.4, 0.3, 0.1, 0.1,
     0.5, 0.05, 0.1, 0.2, 0.15,
0.45, 0.05, 0.1, 0.2, 0.2,
0.2, 0.2, 0.2, 0.2, 0.2,
0.1, 0.1, 0.1, 0.1, 0.6/ : ARRAY [5,5] OF REAL;

... T1: S11/S1;
TRANS T1: %S1.STAT := MARKOV(1, 5, %S1.STAT, %PV); (* New STAT *)
...
SENEG.

The following two procedures STATOUT and STATRESET provide dynamic output
and reset of collected statistics during simulation. and

Procedure STATOUT (not yet implemented). It outputs the collected statistics of a
simulation run into a file. The file is automatically given the identifier <run name>.n or
STAT.n if a run name was not specified at the start of simulation. Here n is number of call
of this procedure. Thus, it is possible to call STATOUT many times in the same or
different segments of the model, with the unique file identifier for each call. The output
statistics is defined by the corresponding statements of MDL given before the start of
simulation run. The procedure does not require any parameters and its declaration is

PROCEDURE STATOUT;

Procedure STATRESET (not yet implemented). It resets (clears) the current statistics
from the simulation run. When resetting, the following actions are performed in the model:
1. In the model descriptor, the current simulation time and the counter of events are
zeroed.
2. In the descriptor of each segment of the model, the time of the next event is set to
zero if this time was not larger than the finish simulation time. Otherwise, the
current simulation time is subtracted from the time of the next event.
3. In the descriptors of all transitions, the number of firings of the transition is set to
zero, and the finish time of firing is also set to zero if this time was not larger than
the current simulation time. Otherwise, the current simulation time is subtracted
from the finish time of transition firing.
4. In the descriptors of all places, the sum of busy intervals and the number of token
arrivals are set to zero, but the state of place (the number of tokens) is not changed.
5. All histograms are cleared (made empty).

The procedure does not require any parameters, its declaration is

PROCEDURE STATRESET;
APPENDIX 4

E-NET MODEL OF CSMA/CD

In this appendix, we demonstrate an application of E-nets and Winsim by means of a complete, rather large example. The primary goal of the example is to demonstrate the expressive features of MDL and the modeling methodology for the preparation and running of E-net models in Winsim.

The example is an E-net model of the CSMA/CD (Carrier Sense Multiple Access/Collision Detection) access method, which is used in a LAN of the Ethernet type. The CSMA/CD is a media access method that allows two or more stations in a LAN to share a common communication bus. A station that wants to transmit a data packet waits until the bus is free. When this happens, the data packet is sent on the common bus. Of course, two or more stations can transmit at the same time. As a result, a collision happens. There is a collision window which is a period of time from the start of transmission to the time the transmitted signal propagates to all stations on the bus.

When a collision occurs because two or more stations are trying to send packets at the same time, each transmitting station intentionally sends a few additional bytes as a buzz, or "jam", to secure that other stations recognize that a collision has taken place. Then each transmitting station remains idle for a random period of time (back-off time) before trying to transmit again. The detailed description of the CSMA/CD and its IEEE 802.3 standard can be found, for example, in [24].

Our motivations in choosing CSMA/CD for E-net modeling are as follows. First, CSMA/CD is a truly distributed protocol, modeling of which exposes clearly the advantages of E-nets as a formal and descriptive apparatus for concurrent and distributed systems. Second, CSMA/CD is a nontrivial protocol, for which rather complicated analytical model exists [25, 26]. Thus it was our aim to show that an E-net model of this protocol can provide insight which is not achievable in approximate analytical models. And third, the existence of an analytical model of CSMA/CD, albeit approximate, provides excellent possibility to compare our E-net model of the protocol with its analytical model.

In our E-net model of the CSMA/CD we have made the following assumptions:

1. The collision event is checked only at the end of the collision window.
2. The back-off time is a random variable with the exponential probability distribution.
3. The transmission time of data packets is a random variable with the uniform probability distribution. We assume, according to the IEEE 802.3 standard, that data packets (frames) have random sizes in the range from 72 up to 1526 bytes.
4. Each station is not limited in the number of its attempts to transmit a data packet.
5. Each station has two FIFO queues. The first queue is for generated data packets which have to be transmitted to some other station on the bus. The second queue is a request queue, it stores the requests to transmit data packets from the data packet queue (one request for each element in the data packet queue). A request activates and departs the request queue whenever the station becomes idle after transmitting the preceding data packet. This request tries to seize the media according to the CSMA/CD protocol and to ensure the transmission of a data packet. The data packet will be removed from the data packet queue after it has been transmitted completely. Thus the time a data packet is
in this queue is the sum of waiting time before the transmission begins and the duration of transmission on the bus.

6. Each station generates data packets randomly according to the Poisson process (i.e. with exponentially distributed interarrival times). Arrival rate of data packet is the same for each station.

7. We use only five active stations in a LAN. The stations are positioned along a relatively short bus segment so that the time for the signal propagation on the bus is supposed to be small (about one microsecond).

8. Each data packet is transmitted to some individual station address (there is no broadcasting).

The first four assumptions are simplifying ones. If necessary, they can be replaced by more realistic assumptions in the refined model. The assumption 5 defines an architectural feature of the protocol implementation. The sixth assumption is important for comparison of our E-net model with an analytical one. The seventh assumption implies that the LAN under investigation is small one. This is a reasonable assumption for a LAN with five stations as in our model. The last, eighth assumption has no special influence on the behaviour of our model. It is quite easy to simulate the broadcasting mode of transmission in the model if it would be necessary.

A fragment of the E-net scheme of the CSMA/CD model is presented in Fig. A4.1. The transitions \( Y_1, X_1, X_2, T_1, T_2 \) and \( X_3 \) simulate the events relating to activities on the bus. In particular, the transition \( X_1 \) simulates the initial period of data packet transmission on the bus during the collision window. The transition \( X_2 \) checks the collision event at the end of the collision window. If a collision happens during the collision window then the transition \( T_1 \) simulates the "jam", or buzz signal transmitted on the bus by the involved station. Otherwise, the transition \( T_2 \) becomes active simulating the remaining period of time necessary to finish the data packet transmission. In any case, after firing the transitions \( T_1 \) or \( T_2 \), a token comes, through the transition \( Y_1 \), into the place \( S_3 \) indicating that the media (the bus) is now free.

If the transition \( T_2 \) fires, a token appears also in the place \( S_8 \). Now the transition \( X_3 \) can fire simulating the localization of the source station for the transmitted packet, so that a token appears in one of the output places of \( X_3 \) depending on the source station (according to the assumption 7, we have, in our model, only five stations). If, for example, a token appears in the output place \( S_{110} \), then the subsequent firing of the transition \( T_{105} \) simulates the removal of the transmitted packet from the corresponding packet queue (the place \( Q_{101} \), in this case) of the source station 1 and setting the source station into the idle state by placing a token into \( S_{103} \).

Now, consider the activity of a transmitting station. For example, for the station 1, its activity is simulated by the transitions \( T_{101}, T_{103}, T_{104}, Y_{101}, X_{101}, X_{102} \) and \( Y_2 \) (\( Y_2 \) is common for all stations).

The transitions \( T_{101} \) and \( T_{102} \) simulate the generation of data packets (which enter the queue \( Q_{101} \)) and the transmission requests (which enter the queue \( Q_{102} \)).

The duration of firing the transition \( T_{101} \) is random, it corresponds, according to the assumption 6, to the exponential probability distribution, with the parameter depending on the arrival rate of packets for one station.

The transition \( T_{103} \) fires when the station is idle (which is indicated by a token in the place \( S_{103} \)) and there is at least one request token in the place \( Q_{102} \).
Fig. A4.1. Fragment of the E-net scheme of the CSMA/CD model.
A token in the place S107 means that the station (the station 1, in this case) waits for the event "the bus is free" to transmit a data packet from the queue Q101. The transition X101 fires when there is a token in the place S107 and the bus is free (i.e. the place S3 also holds a token). Concurrently, the transition X1 fires simulating the transmission on the bus during the collision window.

If another station starts transmitting at the same time or during the collision window, then a collision happens. The collision event is detected first by the transition X2 (this is common detection for all stations) and, after that, by the transition X102 which helps to a station (station 1, in this case) to learn whether some other station (or stations) started to transmit during the collision window.

In case of a collision, a token appears in the place S106 and the transition T104 fires which simulates the random back-off time. Then the process is repeated by firing the transition Y101.

If there is no collision, a token appears in the place S109 (instead of S106) making it possible for the transition Y2 to fire. Remember that at the same time a token is in the place S6.

After firing the transition Y2, the transition T2 can now fire simulating the remaining time of the transmission of a data packet (by the station 1). This time depends on the size of the data packet which is in the range of (72 - 1526) bytes for the Ethernet LAN and, according to the assumption 3, is distributed uniformly.

Note that, in Fig. A4.1, output A goes from this station to a fragment representing the receipt of data packets by destination stations; inputs B and outputs C represent inputs and outputs for other four stations.

The next step in creating an E-net model of the CSMA/CD is to specify its main parameters. With the Ethernet transmission speed of 10 Mbps, as specified by the IEEE 802.3 standard, the range of time required to transmit a data packet on the bus is 57.6 up to 1220.8 microseconds.

We set the collision window equal to 1 microsecond which is consistent with the cable segment length of about 200 meters and the assumption 7. The time to transmit a "jam" signal of 32 bits (as the standard requires) is set equal to 3.2 microseconds. According to the assumption 2, we have specified the exponential probability distribution for the back-off time, with the mean value of 20 microseconds. This value is consistent with the assumption 7 concerning the number of stations in the LAN.

Now we are ready to describe our model in MDL. For simplicity, we organize it as one segment. A fragment of the description of the model in MDL is presented in Fig. A4.2. The fragment shows the declarations of token attributes (ATTRIBUTES statement), net variables for the numerical parameters (DATA statement), and the standard INTERFACE and IMPLEMENTATION Pascal sections (which are optional). The meaning of all net variables, used in the model, is clear from the comments.

********** E-NET MODEL OF CSMA/CD ACCESS METHOD **********
(* Token attributes: *)
(* SRC - Source station address *)
(* DST - Destination station address *)

Parameters:

(* NSTA - Number of stations on the bus. *)
(* GEN - Mean time to generate a data packet by a station *)
(* (exponential distribution) *)
(* WIND - Duration of the collision window (fixed value) *)
(* SND1,SND2 - The minimum and maximum time to transmit a data packet *)
(* on the bus, with the range of data packets 64 up to 1518 bytes *)
(* (uniform distribution) and the transmission speed of 10 Mbps *)
(* SND1 = 64*8/10E7 - WIND, *)
(* SND2 = 1518*8/10E7 - WIND. *)
(* JAM - The time interval to transmit the "jam" after a collision *)
(* (fixed value). *)
(* WAIT - Mean backoff time after the "jam" transmission *)
(* (exponential distribution). *)

Variables:

(* CNT - The counter of concurrent requests from stations to transmit data *)

FILE: CSMA.JOM DATE: 10.11.97

SEGMENT CSMA, TICK=MCSEC;
ATTRIBUTES SRC: INTEGER;
   DST: INTEGER;
(* Declaration of parameters and setting their values *)
(* All times are given in microseconds *)
DATA GEN/5000.0/: REAL;
   JAM/4./: REAL;
   WIND/2./: REAL;
   SND1/50./: REAL; (* 52 - WIND *)
   SND2/1213./:REAL; (* 1215 - WIND *)
   WAIT/100./: REAL;
   NSTA/5/: INTEGER;
INTERFACE (* Pascal specific word *)
VAR CNT: INTEGER;
IMPLEMENTATION (* Pascal Specific word *)
BEGIN
   CNT:=0;
END.
(* Model of the communication media *)
NET Y1:  S1,S2,S3;
NET X1:  S3,S4;    (* The bus is free when S3 holds a token *)
CONTROL X1: %OUT:=0; IF (%S108=1) OR (%S208=1) OR (%S308=1)
    OR (%S408=1) OR (%S508=1)
    THEN %OUT:=1;    (* Are there any requests to transmit? *)
TIME X1: %DELAY:=%WIND;    (* Start transmitting a packet *)
NET X2: S4/S5,S6;
CONTROL X2: CNT:=%S108+%S208+%S308+%S408+%S508;
    IF CNT>1 THEN %OUT:=1    (* There is a collision *)
    ELSE %OUT:=2;    (* No collision... Continue *)
NET T1: S5/S1;
TIME T1: %DELAY:=%JAM;    (* Transmit a jam on the bus *)
NET T2: S7,S6,S2,S8,S9;
TIME T2: %DELAY:=UNIFORM (1,%SND1,%SND2);    (* Transmit a packet *)
NET X3: S8/S110,S210,S310,S410,S510;
CONTROL X3: %OUT:=%S8.SRC;    (* Select the sending station *)
NET X4: S9/S111,S211,S311,S411,S511;
CONTROL X4: %OUT:=%S9.DST;    (* Select the receiving station *)
NET Y3: S111,S211,S311,S411,S511;    (* Remove the received packet *)
NET Y2: S109,S209,S309,S409,S509/S7;

(* Station 1 *)
NET T101: S101/S102;
TIME T101: %DELAY:=EXPON(1,%GEN);    (* Generate a data packet *)
TRANS T101: %S102.DST:=IUNIFR(1,1,%NSTA);    (* Set a destination address *)
NET T102: S102/Q101,Q102,S101;
NET T103: Q102,S103/S104;
NET Y101: S104,S105/S107;
NET T104: S106/S105;
TIME T104: %DELAY:=EXPON(1,%WAIT);    (* Backoff time calculation *)
NET X101: S107/S108;
CONTROL X101: %OUT:=0; IF %S3=1 THEN %OUT:=1;    (* Waiting for a free bus *)
NET X102: S108/S109,S106;
CONTROL X102: %OUT:=0; IF %S6=1 THEN %OUT:=1;    (* No collision *)
    IF %S5=1 THEN %OUT:=2;    (* Collision *)
NET T105: Q101,S110/S103;    (* Remove a data packet from the queue *)

***************************************
(* The descriptions of other stations have to be included here *)
***************************************
SEENG.

Fig. A4.2. MDL text of the model of CSMA/CD access method.
The varied quantity in the experiments is the net variable GEN which had the value 5000 microseconds in one of the experiments. The fragment contains also the descriptions of the communication media (the bus) and one of the stations (station 1) on the media. The descriptions of other stations differ only in the identifiers of transitions and places.

The features of CSMA/CD protocol, which can be derived from its E-net model, are presented in Table A4.1. The table shows clearly the correspondence between the features of the CSMA/CD and the E-net model numerical characteristics. Note, that these features are directly related to the elements of our E-net model.

Table A4.1. The correspondence between the features of the CSMA/CD and the elements of the E-net model

<table>
<thead>
<tr>
<th>The feature of the CSMA/CD</th>
<th>The model element(s) representing the feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>The number of transmitted data packets</td>
<td>The number of firings of the transition T2</td>
</tr>
<tr>
<td>Mean packet transmission time (units of time)</td>
<td>Mean firing time of the transition T2</td>
</tr>
<tr>
<td>Mean time for a data packet to be transferred to a destination from the moment of its generation (units of time)</td>
<td>Mean time tokens wait in the queue places Q101, Q201, ..., Q501 (for stations 1, ..., 5 accordingly)</td>
</tr>
<tr>
<td>The bus utilization</td>
<td>The sum of the utilizations of the transitions T2 and X1</td>
</tr>
<tr>
<td>The number of collisions on the bus</td>
<td>The number of firings of the transition T1</td>
</tr>
</tbody>
</table>

A number of runs have been carried out with this model in Wimsim, for five stations in a network and with the duration of 5000000 microseconds for simulation time in each run, varying the aggregate arrival rate of packets to be transmitted across the network. The main characteristics of CSMA/CD, which were measured directly on our model or calculated, were as follows: bus utilization, number of packets transmitted successfully, number of collisions, and the ratio "mean transfer time / mean packet transmission time" (MTT/MPTT for short). Note, that the packet transmission time depends only on the distribution of packet sizes which is uniform according to our assumption 3. But the
transfer time includes the packet transmission time and the time of waiting in a queue before starting the transmission.

The behavior of the main characteristics of CSMA/CD, with varying aggregate arrival rate of packets, is shown in Table A4.2. From the practical point of view, the most interesting characteristic is MTT/MPTT, albeit other characteristics are also important. For comparison, the behavior of MTT/MPTT for an analytical model is shown in the table. According to the analytical model [25], MTT may be approximated by the following formula:

\[
MTT = E[T_p] + \frac{\lambda\{E[T_p^2] + (4e + 2)\tau E[T_p] + 5\tau^2 + 4e(2e - 1)\tau^2\}}{2(1 - \lambda(E[T_p] + \tau + 2e\tau))},
\]

where \(\lambda\) is the aggregate arrival rate of packets, \(E[T_p]\) and \(E[T_p^2]\) are the first two moments of the packet transmission time \(T_p\) (with \(E[T_p] = MPTT\)), \(\tau\) is the signal propagation delay on the bus, and \(e\) is the base of natural logarithm. We used the complete formula for MTT/MPTT from [25] for calculating the values in the last column of Table A4.2. As one can see from the table, the behavior of MTT/MPTT in our E-net model is very close to the behavior of this characteristic in the analytical model for small and medium values of \(\lambda\), but differs more and more when \(\lambda\) approaches the value 1500 packets/s which corresponds to the bus utilization of about 0.95. Note that for \(\lambda = 1550\) packets/s, the analytical model gives an invalid result, but the E-net model provides quite reasonable value 46.39. Thus for high values of \(\lambda\) the analytical model becomes inaccurate.

We conclude this section with the following observations concerning our E-net model:

1. The model is based on the modified E-nets and represents the CSMA/CD protocol in the natural way, expressing clearly the concurrency of events and processes in the protocol.

2. The model provides the most important characteristics of CSMA/CD - the ratio "mean transfer time/mean packet transmission time", number of transmitted packets, number of collisions, and the bus utilization dependent on the aggregate arrival rate of packets.

3. The model provides insight which is difficult, if possible, to achieve in its analytical counterpart. In particular, important information about the collisions is completely absent in the analytical model. Further, it is not easy, in the analytical model to determine the effect of "jam" signal, which is an essential component of the protocol. But in the E-net model, these and other characteristics are derived quite simply.

4. The model is scalable to represent, if necessary, larger number of stations, without its redesign. To do this, one need to add the E-net descriptions of new stations, increase correspondingly the numbers of input places in the transition Y2 and of output places in X3 (see Fig. A4.1), and set the value of the net variable NSTA equal to the new number of stations (Fig. A4.2). Also the statement CONTROL for the transitions X1 and X2 must be modified to take into account the additional stations.
Table A4.2. Performance characteristics of the CSMA/CD model

<table>
<thead>
<tr>
<th>Aggregate arrival rate of packets, (1/sec)</th>
<th>Bus utilization</th>
<th>Packets transmitted</th>
<th>Collisions occurred</th>
<th>Mean transfer time/ Mean packet transmission time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.062</td>
<td>503</td>
<td>0</td>
<td>1.04</td>
</tr>
<tr>
<td>200</td>
<td>0.122</td>
<td>977</td>
<td>9</td>
<td>1.09</td>
</tr>
<tr>
<td>300</td>
<td>0.187</td>
<td>1465</td>
<td>24</td>
<td>1.15</td>
</tr>
<tr>
<td>400</td>
<td>0.250</td>
<td>1984</td>
<td>72</td>
<td>1.21</td>
</tr>
<tr>
<td>500</td>
<td>0.320</td>
<td>2490</td>
<td>168</td>
<td>1.29</td>
</tr>
<tr>
<td>600</td>
<td>0.376</td>
<td>2966</td>
<td>319</td>
<td>1.41</td>
</tr>
<tr>
<td>700</td>
<td>0.445</td>
<td>3513</td>
<td>541</td>
<td>1.53</td>
</tr>
<tr>
<td>800</td>
<td>0.496</td>
<td>3898</td>
<td>761</td>
<td>1.65</td>
</tr>
<tr>
<td>900</td>
<td>0.562</td>
<td>4389</td>
<td>1087</td>
<td>1.82</td>
</tr>
<tr>
<td>1000</td>
<td>0.627</td>
<td>4910</td>
<td>1750</td>
<td>2.19</td>
</tr>
<tr>
<td>1050</td>
<td>0.674</td>
<td>5336</td>
<td>2090</td>
<td>2.33</td>
</tr>
<tr>
<td>1100</td>
<td>0.703</td>
<td>5484</td>
<td>2520</td>
<td>2.59</td>
</tr>
<tr>
<td>1150</td>
<td>0.731</td>
<td>5732</td>
<td>2804</td>
<td>2.80</td>
</tr>
<tr>
<td>1200</td>
<td>0.759</td>
<td>6034</td>
<td>3384</td>
<td>3.17</td>
</tr>
<tr>
<td>1250</td>
<td>0.790</td>
<td>6203</td>
<td>3944</td>
<td>3.86</td>
</tr>
<tr>
<td>1300</td>
<td>0.831</td>
<td>6475</td>
<td>4546</td>
<td>3.99</td>
</tr>
<tr>
<td>1350</td>
<td>0.849</td>
<td>6665</td>
<td>5270</td>
<td>4.94</td>
</tr>
<tr>
<td>1400</td>
<td>0.878</td>
<td>6924</td>
<td>6056</td>
<td>6.18</td>
</tr>
<tr>
<td>1450</td>
<td>0.928</td>
<td>7256</td>
<td>7467</td>
<td>8.97</td>
</tr>
<tr>
<td>1500</td>
<td>0.948</td>
<td>7484</td>
<td>8710</td>
<td>16.17</td>
</tr>
<tr>
<td>1550</td>
<td>0.973</td>
<td>7636</td>
<td>10144</td>
<td>46.39</td>
</tr>
</tbody>
</table>

Note that the enlarged number of stations results only in the increased aggregate arrival rate of packets. But this can be done more easily by setting the corresponding value of the net variable GEN, without increasing the number of stations. That is why we used a small number of stations to model CSMA/CD.

5. The model may be refined in a number of ways. First, replacing the transition X1 by the interruptible transition, it is possible to check the collision event at the moment it appears. Second, the back-off time can be modeled strictly according to the IEEE 802.3 standard. Third, it is quite straightforward to model limited number of attempts a station performs to send a data packet (up to 16 attempts, according to IEEE 802.3). Fourth, one can model not only Poisson arrival pattern of data packets, but also an arbitrary one. And fifth, the model can cover a LAN with any cable length permissible by IEEE 802.3 (up to
2.5 km). To model this possibility, one needs to increase the collision window (see the net variable WIND in Fig. A4.2) and add time delay to take into account the increased propagation time on the bus.

6. The execution of the model in Winsim is reasonably fast. It takes 2 to 50 seconds to simulate CSMA/CD for one value of the arrival rate of packets, depending on this value. For the simulation, we used a computer of type PC/AT 486.

Even though our CSMA/CD model is only illustrative one, it can be used to investigate the protocol behavior under varying load, with controllable values of such critical parameters as the width of collision window, the time interval for transmitting a "jam", the mean back-off time, and the transmission time of data packets on the media. This model can be used, for example, in a course on computer networks in universities to clarify the peculiarities of the CSMA/CD access method for students. It can be helpful also in developing more realistic models of the CSMA/CD.
REFERENCES


