

ACSIM 2 : A SIMULATION SYSTEM

by

O. Levent Mollamustafaoğlu

B.S. in I.E., Boğaziçi University, 1982

B.S. in Math., Boğaziçi University, 1984

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of

Master of Science

in

Industrial Engineering

Bogazici University Library



39001100315392

14

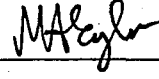
Boğaziçi University

1984

ACSIM 2: A SIMULATION SYSTEM

APPROVED BY

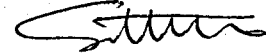
Y.Doç. Dr. M. Akif EYLER
(Thesis Supervisor)



Y.Doç. Dr. Ali Riza KAYLAN

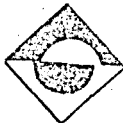


Y.Doç. Dr. Selahattin KURU



DATE OF APPROVAL

July 19, 1984



ACKNOWLEDGEMENTS

I would like to thank numerous friends and colleagues who have contributed to the development of the ACSIM simulation system and this thesis, including Mrs. Yasemin Birgil, Mr. İstemihan Örgüven and Mr. Mehmet Erentöz, who were on the original design team and carried out the initial design phase; Mr. Güven Mercanköşk, Mr. Ömer Ayzan, Mr. Turgay Aytaç and Mr. Hakan Saraoğlu, who helped me in overcoming problems special to Apple microcomputers; and Miss Belgin Turgut who drew most of the figures.

Special thanks are due to my thesis advisor, Dr. Akif Eyler, who, besides determining the initial structure and principles of ACSIM, spent an effort at least equal to mine throughout the year, and bore with me patiently, guiding me with his timely suggestions.

ABSTRACT

In this thesis, ACSIM 2, the much-improved new version of the simulation language ACSIM, is being reported. ACSIM 2 has some characteristics which can not be shown in classical activity cycle diagrams, but which are indispensable for simulation studies. Creating and discarding entities, choosing different paths depending on various conditions, collecting consecutive activities within a single activity, are some of these features.

Program and data has been separated and it has been made possible to save the final state of the system, thus continuing the simulation from the same state. It is also possible to give any initial state to the program, thus the possibility of different scenarios.

One other novelty is a display facility which renders it possible to follow the simulation on the screen and interfere with it if necessary.

KISA ÖZET

Bu tezde, faaliyet çizgelerini temel alan bir benzetim sistemi olan ACSIM'in önemli ölçüde geliştirilerek yeniden yazılan şekli tanıtılmaktadır. ACSIM 2, faaliyet çizgelerinde gösterilemeyen, ancak benzetim çalışmalarında vazgeçilemeyecek bazı özelliklere sahiptir. Nesnelerin ortaya çıkışı ve yok olmaları, faaliyetlerde karmaşık koşullara göre birkaç yoldan birinin seçilmesi, biri diğerinin devamı olan faaliyetlerin tek bir faaliyet bünyesi altında toplanmaları, bu özelliklerin en önemlileridir. Program/veri ayrımı kesin çizgilerle yapılarak, bir ACSIM 2 programının benzer birçok veri üzerinde çalışması sağlanmıştır. Dizgenin son durumunun diskte saklanarak daha sonra ilk durum yerine kullanılması ve böylece benzetime kalınan yerden devam edilmesi de olanaklı kılınmıştır. Programa bir gösterim olanağı da eklenerek benzetimin ekranda izlenmesi sağlanmıştır.

TABLE OF CONTENTS

	Page
ACKNOWLEDGEMENTS	ii
ABSTRACT	iii
ÖZET	iv
LIST OF FIGURES AND TABLES	vi
I. INTRODUCTION	1
II. STRUCTURE OF THE LANGUAGE	6
2.1 SYNTAX	6
2.2 SEMANTICS	15
2.3 DATA STRUCTURE	23
2.4 ALGORITHM	26
2.5 COMPARISON WITH EXISTING LANGUAGES	28
III. EXAMPLES	40
3.1 A MACHINE-SHOP SYSTEM	40
3.2 A BANKING SYSTEM	46
IV. CONCLUSION	51
APPENDIX A - RESERVED WORDS OF ACSIM	52
APPENDIX B - SYNTAX DIAGRAMS	53
APPENDIX C - ACSIM ERROR MESSAGES	61
REFERENCES	62

LIST OF FIGURES AND TABLES

FIGURES

- FIGURE 1 Sample Activity Diagram
- FIGURE 2 Simplified Data Structure of ACSIM
- FIGURE 3 Machine-Shop Cycle Diagram
- FIGURE 4 ACSIM Program Describing the Machine-Shop
- FIGURE 5 Results of the Simulation of the Machine-Shop
- FIGURE 6 Bank Cycle Diagram
- FIGURE 7 ACSIM Program Describing Bank
- FIGURE 8 Results of the Simulation of Bank

TABLES

- TABLE 1 Structure of Memory Assignments - Data Structures
- TABLE 2 Changing the State of the Simulated World
- TABLE 3 Commands to Facilitate Subprogram Execution
- TABLE 4 Programming Features
- TABLE 5 Mechanics of Use

I. INTRODUCTION

Discrete-time simulation systems and languages have been founded on one of the three viewpoints in modelling :

- (a) Event-oriented view
- (b) Activity-oriented view
- (c) Process-oriented view

All of these three approaches have their virtues and vices. In this study, an activity-oriented approach is chosen. This is mainly because Activity Cycle diagrams, which are thought to reflect the structure of complex systems better than anything else, are a means of better modelling and easy restructuring. Since, in activity-oriented special-purpose simulation languages, changes in the model are done through changes in only some of the activities, updating models are considerably easier. [1]

A sample activity cycle diagram can be seen in Fig. 1.

There is only one activity, that is SERVE, in the cycle. Along with the two entities of the system, namely MACHINE and PART, it forms a closed activity cycle. While not involved in the activity SERVE, the two entities are in their passive states, denoted by a "queue". MACHINE has only one passive state, IDLE, and PART has only one passive state, LOADED. The condition that the activity SERVE can be initiated is that

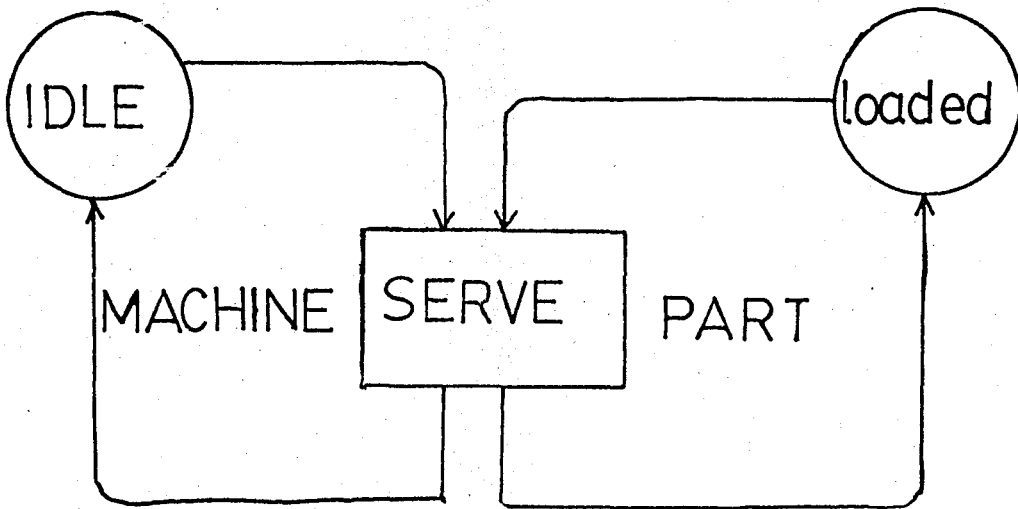


Fig. 1 Sample Activity Diagram

there is at least one MACHINE in queue IDLE and one PART in queue LOADED. Once the two entities are involved in the activity SERVE, they can not be involved in any other activity during SERVE. After SERVE ends, MACHINE and PART go to their respective passive states, namely queues and wait for activation.

ACSIM 2 has been formed on ACSIM [2], which is an activity-oriented special-purpose simulation language, after various modifications and improvements. In doing this, various points have been considered :

(A) The activity-oriented nature of the language must not be changed.

In addition to the reasons revealed above, it can also be noted that activity-oriented simulation languages are closer to Structured Programming concepts and implementations.

(B) The language must be simple, but also powerful enough to model complex systems.

The previous version of ACSIM was limited, in that it only allowed closed-cycle systems to be modelled. This of course made the language simple and easy-to-model with. In order to be able to model complex systems with the new version, some additional features have been supplied. Some of these are complex expressions, creation and discarding of entities, variable conditions, conditional actions, etc.

(C) The language must be implemented on a microcomputer.

This is a feature parallel to the demand on microcomputers in recent years. To accomplish this result under limited resources such as memory and speed, advanced methods of Computer Science have been used. The language has been implemented on an Apple //e (64 KByte) and an

Apple /// microcomputer and results have been satisfactory in terms of speed and model complexity.

(D) The compilation and execution of the language must be short, in terms of computer time.

It is a well-known fact that simulation runs take a lot of time and systems are analyzed for a large amount of system time. This is the reason for speed being a critical factor in the efficiency of a computer language. Since there is no time-sharing in microcomputers, this goal has been satisfactorily reached. In fact, the simulation becomes too quick for eye-inspection when the "display" option is used.

(E) The language must be flexible.

This provides an easy update facility for models. Using an activity oriented approach has helped in this aspect and updates are being made on an activity-basis, not changing the unrelated proportions of the model.

(F) The language must be structured.

The reason for this goal is that structured languages such as Pascal or PL 1 have been commonly used in recent years and their superiority to traditional languages has been ascertained by studies in Computer Science. Structurality has made the writing of the compiler-simulator itself very easy. Another result is that the burden of modelling on the user, and the burden of compilation on the compiler has been made lighter. Syntax errors are easily detected during compilation.

(G) The language must be interactive.

The meaning of "interactive" in this context is that the user can interfere with the simulation, can stop it, can change the system

state and restart the simulation. To accomplish this a "display" facility has been added to the language, though it is optional.

Most of these goals have been achieved though there are others which are still to be realized. In the conclusion, some of these additional goals and ways to reach them are reported.

II. STRUCTURE OF THE LANGUAGE

2.1 SYNTAX

ACSIM has been formed with regard to the general structure of the programming language PASCAL. This characteristic shows itself not only in the "declarations" section, but also in statement separators.

The main building blocks of ACSIM are "activities" and, as a subgroup of these, "statements". The starting position of a statement in an ACSIM line is irrelevant, just like Pascal. Indentation is recommended, but is not compulsory.

The symbols used are reserved words, identifiers, arithmetical symbols, relational symbols, separators and numbers. Reserved words of ACSIM are given in Appendix A. Words are separated from each other by spaces, but there is no need for a separator between words and other symbols. Upper or lower case letters can be used in words, since the compiler converts everything to lower case letters. All reserved words will be shown in upper case letters within the context of this report.

There is a limited freedom in the usage of separators, but they must generally follow the system syntax.

The general structure of an ACSIM program can be summarized as follows :

Descriptions (Declarations)

Activities

- Conditions
- Time increment
- Actions

Statistics and reports

Parallel to this structure, the general syntax of ACSIM, along with the syntax of the program's individual components is given in Appendix B.

(A) Declarations

Among the things to be defined or declared in this section, there are the following :

Name of the system, random seed, classes of entities, attributes of entities if they exist, queues these entities might be in, global variables.

The definition of the simple system in Fig. 1 can be done in ACSIM as follows :

<u>DEFINITION</u>	<u>ACSIM STATEMENTS</u>
Program name	machineshop
Random seed	3567
Classes and queues	CLASS part : loaded ; machine : idle
Variables	VAR completed END

We must note that an entity might be allowed to enter various queues, whereas a queue may contain only entities of a certain class. The CLASS or VAR declaration may be missing, but not both of them at once.

(B) Activities

The activity is the main building block of a ACSIM program, so updates must be thought on an activity basis. If the conditions of an activity are satisfied, then the related entities are taken out of their respective queues and activated. After a certain time, which is given in the time increment statement(s), the entities are deactivated and they go to their queues, though it may be the case that they go to a different queue, any other required actions are done and activity scanning continues.

(a) Conditions

In addition to conditions checking whether there are entities in a specified queue, optionally with a certain attribute value; complex conditions are also allowed. Since the definition of an "expression" is highly complex and expressions can be used in conditions, this supplies a degree of complexity to the conditions themselves, thus rendering it possible to build complicated models.

There are basically two kinds of conditions which are denoted as "queue conditions" and "variable conditions". Queue conditions check whether there is at least one entity in a given queue, optionally with an attribute value satisfying a condition. Namely, in addition to an equality check on an attribute, there is the possibility of checking whether the value of the attribute is in a certain relation (such as "greater than" or "less than or equal to") with a given expression. Variable conditions may be thought to be "boolean variables" since they consist of two expressions linked by a relation. If that relation exists between the two expressions, then the condition is satisfied.

The quantities that could be used in a expression are :

- Numbers
- Queue contents (Called by the queue name)
- Variable values (Called by the variable name)
- Random functions (Called by the function name)
- Attribute values (Called by attribute and class name)
- Other expressions

An example for each type of expression :

30/56

2xwaiting+3 ("waiting" is a queue name)

x-y+1

2x UNI(10,20)

weight OF part+3 ("weight" is the attribute
of class "part")

UNI(x+y,x+2xy)

2x(x+y+z-weight OF part)

(b) Time increment

Time increment can contain any expression. Thus it may be

- Constant AFTER 5:
- Variable AFTER time OF part + 3:
- Random AFTER XPO(20):

In case the value is zero, the AFTER statement can be skipped.

(c) Actions

Actions are done at the end of the time increment, if the conditions of the activity are satisfied. Again, the only action allowed in the previous version was to put an entity to one of its queues. This

was enough, considering the fact that only closed activity cycles were modelled. With the addition of options such as open activity cycles, global variables, creation and destruction of entities, etc., new forms of actions have been added to the language. These can be summarized as follows :

- | | |
|----------------------------|---|
| - Creation of an entity | NEW part loaded |
| - Entity entering a queue | machine idle |
| - Variable action | y + 7 |
| - Conditional action | IF weight OF part > 150
THEN part out
ELSE part loaded
ENDIF |
| - Printing | PRINT (x,y,z) |
| - Destruction of an entity | DISCARD part |
| - End of simulation | STOP |

An entity is created and put in a given queue, denoted in the NEW statement. If wanted, its attribute can be initialized to a value different from zero by the WITH clause. The initial value is given by an expression.

An existing entity is deactivated and enters a queue, by means of stating its name and the name of the queue involved. No change occurs in the attribute value.

The value of a variable is modified in a variable action statement. We may add or subtract the value of an expression from the value of the variable, or we may equate the variable to an expression. Although there are no syntactical limitations in this operation, as far as a

correct expression is given, some semantical limitations might exist. Semantical matters will be discussed in Section 3.

The conditional action is an IF statement. If the condition of the action(s), which is given in the form of two expressions linked by a relation, is satisfied then the actions following the THEN clause, until an ELSE or ENDIF is encountered, are done. If the condition is not satisfied, and there is an ELSE clause, then the actions following the ELSE clause, until ENDIF is encountered, are done. ELSE is optional. Infinite nesting in IF statements is possible, though ENDIF must be supplied for each IF. ENDIF has been used to avoid use of BEGIN and END as in Pascal or PL1. An AFTER clause might be added after the THEN or ELSE clauses, so that the conditional action may be started with a time lag.

The values of variables can be printed at any time during the simulation. To do this a PRINT clause, with the names of the wanted variables as parameters, is used. When this action is done, the name of the variable, and its value after an equals sign, is printed on the output file (or the screen).

In contrary to the creation, entities can also be destroyed, namely they can be thrown out of the system, by a DISCARD clause. The class name of the entity to be discarded is given in this statement.

The end-of-simulation mark, or the STOP clause, can be put to anywhere in the program (provided that it is among the actions of a particular activity). So, complex stopping conditions can be utilised in this way. More simply, an activity consisting only of the stopping conditions and as the only action, STOP, can be put at the end of the activities.

Further details on actions can be got by analyzing the syntax diagram for them.

(C) Statistics and reports

At the end of simulation, some standard information is given as a report. This includes the final state of the system and the number of activation of each activity. The final state is given in terms of the queue contents, variable values and events waiting to "happen". Besides this information, the user might want to get some statistical measures such as waiting times and queue lengths, as an option. These figures can be calculated by the program if a COLLECT clause is used. This clause collects statistics about given queues and prints them at the end. The statistical measures calculated for waiting times and queue lengths are mean, standard deviation, maximum and minimum values. Waiting times are demanded by the DELAY clause, whereas queue lengths are demanded by the LENGTH clause. More than one of these clauses can be used, but it is also possible to give all queue names that are required in one clause.

(D) Activities with no conditions

There may be some activities which need no condition to be activated, which either start at definite time points or start at random time points. An example would be the arrival of trains to a station, assuming they always come on time. Another would be an arrival process with a random arrival function, which would need no conditions to start.

To implement this characteristic to the language, the EVERY clause has been used. An activity beginning with EVERY happens after a defined amount of times passes. This amount is calculated from the expression following the EVERY clause. For example,

```
arrival EVERY XPO(5):
```

```
    NEW customer waiting;
```

defines a Poisson arrival process with the interarrival times exponentially distributed with mean 5.

```
arrival EVERY 30:
```

```
    NEW train ready;
```

defines deterministic arrivals.

The second component of a model built in ACSIM is the "data". The separation between program and data has been accomplished in ACSIM, and this has brought the concept of static and dynamic elements of the model. The program represents the static element, since, once written, the program describes a particular real-life or hypothetical system, and unless changed by a change in the program, this representation is the same throughout subsequent simulation runs. What changes is the "system state". As the simulation proceeds, the system state changes with the actions done, or it may stay the same if no significant action is done or the executed actions have no real effect on the system, besides increasing the number of times the particular activity is activated. By separating program and data, we are able to simulate the same system with different initial states, or different sets of data. No change is required on the program in such a case.

The general structure of a set of ACSIM data can be summarized as follows :

Data name (State name)

Variable values

Queue contents

Waiting events

Variables can be initialized by giving the name, putting an equals sign and writing the value. Note that only numbers are allowed

here, instead of expressions such as in the variable actions syntax. This is because, first, the data signifies a snapshot of the system at a particular moment, and thus all values are known and need not be calculated from an expression, and second, data exists independent of the program and thus the expression can not relate to any of the components of the program itself, as it would frequently be needed in a complex expression.

Queue contents are given by a number and the name of the queue. Class name can also be included before queue name, to increase readability. At the beginning of the simulation, or the "initialization" phase, the denoted number of entities will be created and put to the denoted queue. To initialize the attributes of these queues, attribute values may also be supplied. If not given, all attributes are considered to be zero.

Waiting events are defined by giving the event time, name of the activity creating this event, the sequence number of the action to be done when this event occurs, entities involved in this event and their attributes if different from zero. An event is created at initialization and it occurs when its time comes.

If a variable is not initialized, its value is taken to be zero, similarly, if a queue content is not initialized, it is assumed to contain no entities.

The data can be defined by the user, or it may be obtained at the end of a simulation run by taking a snapshot of the system. When the user gives the data, it is generally to give an initial state to the system. When automatically generated system state is recorded onto disk and used later, it is generally to record the system state at a particular moment and to continue with the simulation with the same state, some time later.

2.2 SEMANTICS

There are lots of semantical checks in ACSIM, to facilitate the usage of the language with "correct" models, correct meaning containing no logical errors. Some of these errors are detected during compilation and some during the actual simulation. Currently error-correcting schemes are not used and the simulation or the compilation is stopped in case of an error, syntactic or semantic.

Actually, there is little or no need for error-correction, since the system will be used interactively and any detected error will be corrected by the user himself with the help of an intelligent editor program. So, the compiler becomes simpler to implement and quick edit-compile sequences are made possible.

The controls for semantical checking can be grouped into three, depending on the phase in which the control is done.

- Parsing (Compilation) phase
- Initialization phase
- Simulation phase

Controls done in the parsing phase are usually complementary to syntactical checking. An error number is given for a detected error and the compilation is stopped.

When initialization information, or initial state denoted as "data" is read, only syntactical check is done, since the program is not known at the particular moment and semantical errors can be detected only after both the program and the data are parsed. Similarly, an error number is given and the parsing process is stopped.

Errors occurring during simulation are related to the model itself, and more complex in nature, in regard to the fact that they have

not been detected during syntactical and semantical checking. They might be named "execution errors" or semantical errors, since there are errors of both kinds that can be listed under the group name.

(A) Parsing errors

Only some of these errors will be reported here, choosing the most important.

- Number too large.

When the syntax diagram calls for a number, there is a limitation on the length or magnitude of the largest number that can be accepted. Currently this value is 32767, because of the limitation of Apple Pascal, the environment in which the language is implemented. Only integers are accepted as numbers. Since real numbers use up too much storage space, they are not used in ACSIM. Another reason is that there is always a "truncation error" factor in real operations, which could add up to significant figures. Furthermore, real numbers can be represented by large integers by throwing out decimal places and calculations can be adjusted accordingly.

- Undefined name.

Any identifier read by the parser is checked for its kind, namely it must have been declared in the declarations section, and its kind must be the one that is required at the particular statement within the program. Duplicate names are not allowed but this is got rid of by the syntactical check.

- Incompatible class-queue or attribute-class.

Although passing from syntactical checks and the semantical checks above, the class and queue or the attribute and class may not comply with the original definition.

- Program will not stop.

Since the only way the simulation stops is a STOP clause passing in any activity, this is a matter to be checked. If there is no STOP statement in the program, an error message is given and the simulation can not start. Even if this check is made, there is still a possibility of an infinite loop, if the STOP clause is in a conditional action, or an improbable or highly improbable condition is given. Since it can not be known at compilation time whether the condition will ever be satisfied during the simulation, the compiler can not give an error in case of a conditional STOP. It is left to the user to put a terminating condition. One error that could frequently be encountered is a condition involving the system clock. A condition like

```
CLOCK = 1000
```

may not be satisfied at all, since time is not incremented in unit increments. In such cases, something like

```
CLOCK > 1000
```

must be used.

- Nonmatching condition-action.

In order not to lose any entity during the simulation, each entity used in the conditions must also be used in the actions. When an entity is used in a condition, namely the queue in which the entity is currently present is used, the entity is activated to be used in the activity. After the activity ends, it must be put in one of its passive states, namely queues. If this is not done by the user, then there will be a dangling entity not belonging to any queue and this entity will not

be activated by any other activation request. The following activity will be marked as erroneous, by the compiler :

```
serve part loaded, machine idle?
```

```
AFTER XPO(20):
```

```
part loaded;
```

Here, "machine" is not used in any action, thus causing an error.

Similarly, any entity used in actions must have passed in the conditions. If this is not done, then a nonexistent entity will be attempted to be used in an action. In the same example; the following would be an erroneous usage :

```
serve part loaded?
```

```
AFTER XPO(20):
```

```
part loaded, machine idle;
```

Here, "machine" has not been activated by a satisfied condition, so it can not be used in an action.

- Undefined attribute usage.

In any action, if any attribute value is used in an expression, the entity with that attribute value must be active, namely it must not have entered in one of its passive states. Similarly in the conditions, an attribute name can not be used before its corresponding class name is used. In the examples above, if weight of the part is used as an attribute, then the following program pieces will be erroneous :

```
serve part loaded, machine idle?
```

```
AFTER XPO(20):
```

```
part loaded,sum+weight OF part,machine idle;
```

```

serve capacity > weight OF part,part loaded,machine idle?
  AFTER XPO(20):
    part loaded,machine idle;

```

In the first example, the weights of the parts are accumulated in a global variable, "sum". But, after the action "part loaded" is done, there will be no part active, to the expression "weight OF part" can not be calculated. Once the entity enters a queue, it is lost, or opaque to the eyes of an observer looking from the dynamical level, or the simulation itself. The correct version of the activity must be as follows :

```

serve part loaded,machine idle?
  AFTER XPO(20):
    sum+weight OF part,part loaded,machine idle;

```

In the second example, the entity to be used in the actions must have been chosen before "weight OF part" can be calculated. That is only possible after the name of the entity is used. The correct version of the activity must be as follows :

```

serve part loaded,capacity > weight OF part,machine idle?
  AFTER XPO(20):
    part loaded,machine idle;

```

or

```

serve part loaded WITH weight < capacity,machine idle?
  AFTER XPO(20):
    part loaded,machine idle;

```

- Illegal time increment.

System time is shown by the predefined system variable CLOCK. The value of CLOCK can be reached by the user and can be used in any

expression, but he is not allowed to change it, since it is automatically changed by the program when the need arises. So, in variable action statements, the value of CLOCK can not appear on the left-hand side.

(B) Initialization errors

Number too large, undefined name, incompatible class-queue errors are detected in initialization also. Additionally, in waiting events, an illegal statement number may exist.

(C) Execution errors

- Illegal time increment.

Since it is possible to use expressions in time increment statements, it is also possible to get a negative time increment value. This fact can not be detected during compilation, since it is not known beforehand what value the expression will take at the particular moment. During execution, if a negative value comes from such an expression, an error message is printed and the simulation is stopped.

- Illegal parameter.

While random functions are being evaluated, there are some factors to be considered. First of all, the number of parameters for each type of function must be correct. This is controlled by the compiler. Another factor is that the value of these parameters must be logical. This can not be known in advance, since parameters are expressions themselves. Control is done during execution, after the expression is evaluated. The controls to be made are as follows :

Uniform distribution :

UNI(exp1,exp2)

The value of exp1 must be less than the value of exp2.

Exponential distribution :

XPO(exp1)

The value of exp1 must be greater than zero.

Normal distribution :

NOR(exp1,exp2)

The value of exp2, namely the variance value. must not be negative. (It is also meaningless to choose it zero, since it would then be a constant distribution.)

- Nonmatching condition-action.

Some of these errors can be detected by the compiler and compilation can stop, but some can not be detected at compilation time and are left to the execution. For example. if an entity class is used twice in actions but only once in conditions, the compiler can not detect this error. While execution proceeds, there will be no entity to participate in the second action and an error will occur. Similarly, if an entity class is used twice in conditions and only once in actions, then one of the entities will not be used in any action and an error will occur. An example for each case is given below :

Excess action :

act part loaded,machine idle?

AFTER 20:

part loaded,machine idle,machine broken;

Excess condition :

act part loaded, machine idle,machine broken?

AFTER 20:

part loaded,machine idle;

It must also be noted that the sequence of the conditions need not match that of the actions. As long as the number of conditions and actions match, there is no problem. The following usage is legal :

```
serve part loaded,machine idle?
```

```
AFTER 5:
```

```
machine idle,part loaded;
```

These limitations are not valid if an EVERY clause exists instead of conditions, or actions consist of a STOP statement. The following usages will be valid :

```
valid EVERY XPO(20):
```

```
NEW part loaded;
```

```
valid part loaded,x=35?
```

```
STOP;
```

There are also special cases for NEW and DISCARD statements. NEW can be used for a class not passing in the conditions, since it does not need any activated entities and will create one of its own. When DISCARD is used, condition-action matching is assumed to hold. The following activities are legal in terms of syntax and semantics :

```
legal part loaded,x=35?
```

```
part out,NEW finishedgood inqueue,x=0;
```

```
legal part loaded,x=55?
```

```
DISCARD part,x=0;
```

The first activity emphasizes the fact that identifiers are not limited in length, such as finishedgood.

2.3 DATA STRUCTURE

In order to have the possibility of fitting the compiler-simulator to the limited memory space of a microcomputer, data structure has been formed after a thorough study and a teamwork. Pascal's strength in complex data structures has been the pushing factor and the structure has been polished to a very efficient degree.

The most important advantage of using Pascal is the ability to use totally dynamic memory. That is to say, memory allocation is not done till execution time. This property lifts many barriers in terms of memory usage. There is no *a priori* limitation on the number of entities or events that could be simultaneously present in the system. (Only entities and events cause a major problem, because all other elements of the system are static, namely they do not change form or size after they are declared once.) To give a rough estimate, the program can hold

about 500 entities in a 64 KByte microcomputer and

about 5400 entities in a 128 KByte microcomputer.

In the latter case, what is meant is an additional 64 KBytes that can totally be allocated to the program memory. These figures give us really rough estimates, because the size of the model, lengths of names and similar things are factors determining the amount of free memory space.

The concept of "pointer" is dominant in the data structure. Beside names, no string is held in the memory. All other manipulations are done with pointers. One other aspect of the data structure is that no static arrays are used. Static arrays would put a limit on various elements of the system and it would have been impossible to use dynamic data structures. Everything is in form of "linked lists" and operations

on linked lists are also favorable in terms of execution time. A simplified model of the data structure is given in Fig. 2.

Despite all precautions, the program can fill the available memory during simulation. This is controlled throughout the simulation and in case of a possibility of a full memory, namely when there is an available memory less than some predefined number of bytes (currently 1 KByte), the simulation is stopped, a warning message is given and the user may take a report reflecting that particular moment in simulation time. The limiting memory is kept for the reporting program.

While parsing is done, or simulation is displayed on the screen, names of various elements are needed for printing or comparing. These names, which constitute the only "string" portion of the data, are kept in a binary tree. All searches are done on this tree and are very quick, because of the binary nature of the tree and the speed of pointer operations.

Another precaution is to use records again and again. With the new version of ACSIM, entity creation and destruction is allowed. But, this also means creation and destruction of records corresponding to these entities. Record creation in Pascal is done by the NEW statement. When it is used, it creates a new record of the required type, and so it uses up some of the available memory. The opposite of this is the DISPOSE statement. What it does is to clear the memory area corresponding to that record and make it available for future use. This clearing process is called "garbage collection" in computer science, since it involves shifting of useful data and removal of useless data. However, a real garbage collection, namely squeezing the memory every time a record is destroyed, is not feasible, and is not implemented even in main-frames.

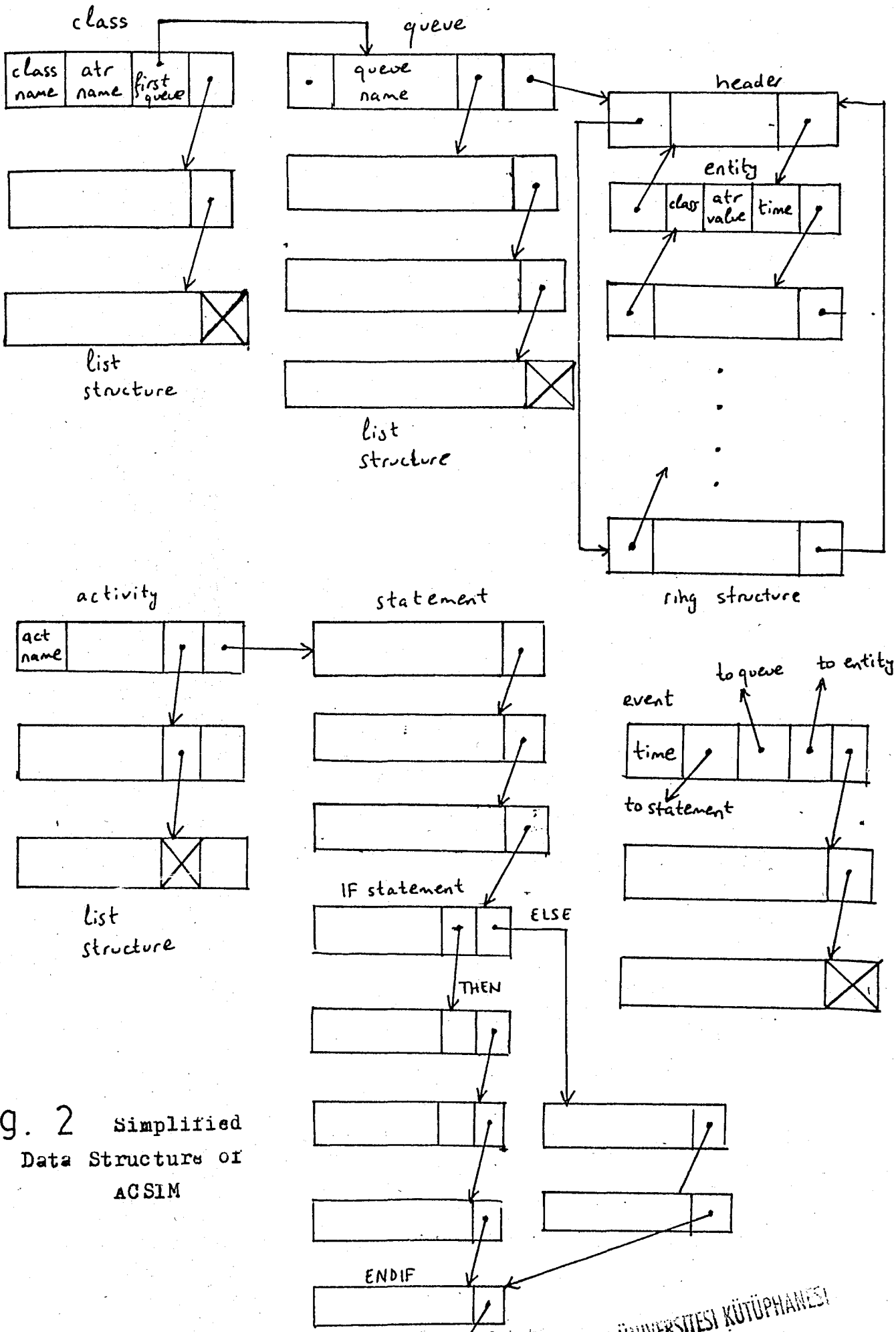


Fig. 2 Simplified Data Structure of ACSIM

To counteract this and to escape from the complications of memory manipulation, a simpler method has been introduced :

When a record is going to be destroyed, it is put into a "free list", namely in a list of records currently not in use. When a new record of the same kind is needed, the first thing to do is to check the free list. If the free list is not empty, then a record from the free list is taken, its contents modified properly and it is put to use. If the free list is empty, then there is no choice but using NEW to use some more memory. By using this method, the total memory needed by records of a particular kind is equal to the maximum memory needed simultaneously, namely to the memory usage of the maximum number of such records that could possibly exist simultaneously.

2.4 ALGORITHM

The method used in the program is activity scanning, but the standard activity scanning algorithm has been slightly modified, to account for the extended form of the language.

The first modification is an event-list, which is generally used in event-oriented simulation languages. This does not change the point-of-view of the general system. It only helps the job of simulation and makes it quicker. Events are sorted with respect to increasing occurrence time. To each event, the entities that are required to realize that event are linked. There is also a pointer showing the first statement of the ACSIM program that is to be executed at the occurrence time of the event. All statements starting with the pointed one, until the first AFTER statement, are executed when the time for

that event comes. Then the value of the expression in the AFTER statement is calculated, added to the current value of the system clock and a new event is created, with its time equaling the value obtained by the addition. This new event points to the first executable statement after the above denoted AFTER statement. This goes on until the activity is totally executed. So, an activity is executed in parts, if there are more than one AFTER statements in it, the AFTER statements denoting the different times at which the parts of the activity are to be executed. Note also that the AFTER statements are cumulative in effect, namely each adds to the value of the system clock and the values of the previous AFTER clauses within the same activity.

The general algorithm of the program can be summarized as follows :

Compile;

Read data;

Initialize;

Repeat

Execute the actions of events with time equaling to
the system clock;

For each activity

If conditions are satisfied

then begin

Take required entities from their queues;

Create a new event;

Link the entities to the event

end;

Advance the system clock to the time of the first event in the event list

Until simulation ends;

Give statistics and state report.

Statistics are collected every time an entity enters or leaves a queue. For queue length figures, collection is done whenever time is advanced.

2.5 COMPARISON WITH EXISTING LANGUAGES

Although there are lots of special purpose simulation languages, there are just a few for microcomputers, and these have not been well-established and tested. Because of this, a comparison can be done between ACSIM and some other languages implemented on main-frames. Doing this, we lose the chance of comparing them on the basis of speed and capacity, since the resources of microcomputers and macrocomputers can not be easily matched.

In tables 1 through 5, various aspects of the simulation languages GPSS II, SIMSCRIPT, GASP and ACSIM are compared. The characteristics for the first three languages have been taken from similar tables given in [3], and the characteristics of ACSIM have been added as a fourth column to the tables. Some characteristics irrelevant to ACSIM have been skipped.

To make the comparison more meaningful, different views in languages have been chosen. GPSS is a process-oriented language, SIMSCRIPT and GASP are event-oriented languages and ACSIM is an activity-oriented language.

TABLE 1 - Structure of Memory Assignments - Data Structures

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
1. Fundamental elements	"Transaction" "Storage" "Queue"	"Individual entity"	"Element"	"Entity" "Queue" "Variable" "Activity"
2. Properties of objects	Transaction : Parameter Priority Mark time Storage : Capacity Max.contents Cur.contents Utilization time integral Total entries Facility : Status Utilization integral Total entries Queue : Max.contents Cur.contents Utilization time integral Total entries	Attribute	Attribute	Entity : Attribute Time Queue : Curr.contents Max.contents Min.contents Total entries Variable : Value Activity : Number of activations
3. Group of objects being simulated	"Transactions" "Storage" "Facilities"	"Entity"	"Element list" "Matrix"	"Class of entities" "Variables" "Queues"

TABLE 1 - (Continued)

	<u>GPSS. II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
4. Data about the environment	"System variables" "Savex" "Function" "Frequency table"	"Permanent system variable"	"System variables"	"Global variables" "System clock"
5. List of names of objects with certain properties	"Events chain" "Assembly set" "Service chain" "Interrupt chain"	"Set"	"Element list" "Queue"	"Event list" "Entity queue"
6. Can records be temporary?	Yes	Yes	Yes	Yes

TABLE 2 - Changing the State of the Simulated World

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
1. Subprogram : agent of change	Block subroutines	Event subroutines	Event subprogram causing activity	Activity scanning routine Event handling routine
2. Who provides subprogram?	GPSS	User	User	ACSIM
3. Time control routine	Main scanning routine	Timing routine	GASP executive	Scanning routine
4. Amount of time advance	To next scheduled future event	To next imminent event	To next scheduled event	To the first event in the events list
5. Exit after time advance	To appropri- ate block subroutine	To appropri- ate event subroutine	To appropri- ate event subroutine	To procedure PERFORM, which performs current events
6. What flows in simula- ted world?	Transaction	Temporary entity	Temporary element	Entity
7. What deter- mines when change occurs?	System status; change of status forces new events. Scan before time advance.	Tests in event subroutine	Tests in event subroutine	Clock change is automatic. Scan before time advance. Other changes user-defined.

TABLE 2 - (Continued)

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
8. Can changes be caused externally?	Exogenous events. "Help" for arbitrary modification	Exogenous event tape	Exogenous events	Simulation can be interrupted and state change can occur

TABLE 3 - Commands to Facilitate Subprogram Execution

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
1. Create temporary records	"ORIGINATE" "GENERATE"	"CREATE"	Temporary elements created by naming, stored in queues, cease to exist upon departure from the last queue	"NEW" Event records created automatically
2. Remove temporary records	"TERMINATE"	"DESTROY"		"DISCARD"
3. Place event on schedule	"PRIORITY" "BUFFER" "ADVANCE" "HELP"	"CAUSE" "CANCEL" Exogenous events	"SCHDL" "REMOVE"	Automatic
4. Change list membership	"SEIZE" "RELEASE" "INTERRUPT" "HOLD" "PREEMPT" "LEAVE" "RETURN" "ENTER" "STORE" "GATE" "LINK" "UNLINK" "QUEUE"	"FILE" "REMOVE" "REMOVE" FIRST"	"FILEM" "FETCHM"	Entity taken out of a queue when activity is activated, it enters a queue when the activity is performed

TABLE 3 - (Continued)

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
5. Sequencing	Current events chain by delay by FIFO; future events chain by departure time by FIFO; service by priority by FIFO	FIFO LIFO Ranked on attribute value	FIFO LIFO High or low ranking or attribute value	FIFO Event list by event time
6. Logical commands and phrases	Selection modes: Both All Pick P FN SIM Gate conditions: NU SE SNF U SNE LS M NM I SF LR Algebraic compare	"FOR EACH" "LOOP" "FIND MAX" "FIND MIN" "REPEAT" "OR" "AND" "WHERE" "IF" "WITH" "IF EMPTY" "GOTO"	FORTRAN	Infinitely nested IF...THEN:...ELSE... ENDIF statements. < < = > > = = < > Number in queue Attribute of entity can be used in expressions

TABLE 4 - Programming Features

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
1. Basic unit of program	Block	Statement Event routine	FORTTRAN statement	Declaration Activity -Statement
1. Programming requirements	None	FORTTRAN	FORTTRAN	None
3. Flowchart symbolism	Yes	No	No	Yes (Activity cycle diagrams)
4. Recursion: Infinite nesting	No	No	No	Yes (In expressions and IF statements)
5. Arithmetic commands	"ASSIGN" "HELP" "TABULATE" "SAVEX"	"LET" "STORE" "COMPUTE" "DO TO"	FORTTRAN	Arithmetic operations, Assignment statement.
6. Commands to collect statistics	"TABULATE" "QUEUE" "SAVEX" "HOLD" "HELP" "STORE" "SEIZE" "RELEASE" "ENTER" "LEAVE"	"ACCUMULATE" "COMPUTE" Number Sum Mean Sumsquares Meansquare Variance Standard deviation	"COLLECT" "HISTOG"	Requested once by COLLECT statement, then collected automatically.

TABLE 4 - (Continued)

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
7. Functions, distributions, random numbers	Any standard system variable	Uniform Non-uniform continuous or discrete probability distribution	Option-random operation or random decision. Erlang Normal Poisson Uniform Random numbers from probability list, Regression equation	UNIFORM NORMAL EXPONENTIAL
8. Input-output	Built-in fixed I/O SAVES transfers model to tape. READS restores model from tape. WRITE places transactions on tape. JOBTAPE recovers transactions from tape.	"SAVE" "ENDFILE" "READ" "READFROM" "LOAD" "RECORD MEMORY" "WRITE ON" "RESTORE STATUS" "ADVANCE" "BACKSPACE" "REWIND"	Subroutines DATAIN and OUTPUT. Summary report ENDRUN.	Input from disk, output to disk or screen. I/O is not part of the language, but is handled by menus. Program, report or final state may be saved on disk.

TABLE 4 - (Continued)

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
9. Report output	PRINT normal output -Model listing -Clock time -Block counts -Savexes -Facility statistics -Storage statistics -Queue statistics -Frequency tables -Summary statistics -Error conditions	Report generator	GASP summary Contents of all queues, max. and average queue length. Scheduled but unexecuted events.	Value of variables, queue contents, activity activations are standard. Optionally: For waiting times: Mean wai.time St.dev. Max.wai.time Min.wai.time For queue lengths: Mean queue l. St.dev. Max.queue l. Min.queue l.
10. Use for non-simulation purposes?	No	A general purpose language	Imbedded in FORTRAN	No

TABLE 5 - Mechanics of Use

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
1. Compilation and running procedure	Model deck is interpreted.	FORTRAN compiler SIMSCRIPT source is converted to FORTRAN source. Some versions have SIMSCRIPT to machine code conversion	FORTRAN	Main routine does it all
2. Debugging and diagnostics	Dynamic error indications terminate run and print system status and accumulated statistics. Trace allowed, Limited syntactical error checking.	FORTRAN diagnostics	FORTRAN diagnostics Monitor program optional.	Full syntax checking and error reporting at compile-time. Execution errors terminate run and give a complete account of the error. Display of simulation on the screen is allowed.

TABLE 5 - (Continued)

	<u>GPSS II</u>	<u>SIMSCRIPT</u>	<u>GASP</u>	<u>ACSIM</u>
3. Memory	At load time Dynamic for transactions.	Load time Dynamic for temporary records.	As in FORTRAN.	Dynamic
External memory	Tape	Tape	Tape (FORTRAN capability)	None

III. EXAMPLES

3.1 A MACHINE-SHOP SYSTEM

The first system to be analyzed is a machine shop. The entities of the system are parts which are to be processed, workers to load these parts, a material handling system to carry these parts to the machines and the machines which will process these parts. Each part has to be processed twice, before its job is finished. For simplicity, all machines are considered to be of the same kind and all process times, arrival times and moving or loading times are constant, namely deterministic.

Fig. 3 shows the closed activity-cycle diagram describing this system. Part, machine, mhs (material handling system) have cycles of their own, whereas worker has two cycles, since he does two jobs, that of loading and that of unloading. If there is a part in pool and a worker is free, then the loading activity begins and part comes to outq, whereas worker goes back to its free state. If the material handling system is available, it takes the part in outq and carries it to inq. If there is an idle machine, then it processes the part in inq. This processing continues until the part is processed twice. Then a free worker unloads the part from the machine, and part goes to the pool, representing a new part coming to be processed.

This model shows most of the properties of a closed activity cycle. There is no entrance from outside and no entity leaves the system. The crucial point is to put enough amount of parts to pool at the beginning of the simulation. If this is not done, then the arrival process can not be simulated properly and limitation not existing in the original system might occur in the simulated system.

Fig. 4 shows the ACSIM program and data describing this system.

Classes and queues are defined in the declarations section starting with the keyword CLASS. A variable, UNL is defined to store information.

Activity LOAD starts if there is at least one part in the pool and one worker free. After 4 units of time, part goes to outq with its attribute ntp=0. Here, ntp means "number of times processed". It must be known for each entity, so it is declared as an attribute. Worker goes back to its passive state.

Activity MOVE starts if there is at least one part in outq and the material handling system is available. After 7 units of time, part goes to inq, mhs is again available.

Activity PROCESS starts if there is a part in inq with number of times processed less than two and any idle machine. After 15 time units, machine is again idle, part goes to outq with its attribute increased by one.

Activity UNLOAD starts if there is a part in inq and a free worker. Here, there is a trick to simplify the simulation. Actually the condition of UNLOAD must have been in the form

```
part inq WITH ntp > = 2
```

so that only parts which have been processed twice are unloaded. But,

```

machineshop 4256
class part with ntp:pool,inq,outq;
  worker:free;
  machine:idle;
  mhs:available
var unl
end
load part pool,worker free?
  after 4:
  part outq with ntp=0,worker free;
move part outq,mhs available?
  after 7:
  part inq,mhs available;
process part inq with ntp<2,machine idle?
  after 15:
  machine idle,part outq with ntp=ntp of part+1;
unload part inq,worker free?
  after 4:
  part pool,worker free,unl+1;
endact clock > 10000?
  stop
collect delay idle,free,outq;
  length idle,free,outq.

```

Fig. 4 ACSIM program for machine-shop

this condition is got rid of by putting activity UNLOAD below activity PROCESS in the program. If any part has an ntp value less than two, it will be sent to outq by the activity PROCESS, so there can not be a part with ntp 2 in inq when activity UNLOAD is being scanned. Thus no need for the above condition. After unloading is done, namely after 4 time units, part goes to pool, worker becomes free again and the variable un1, which counts the number of parts unloaded is incremented by 1.

Activity ENDACT is put to end simulation. If system clock exceeds 10000, then the simulation is STOPped.

Fig. 5 shows the output obtained at the end of the simulation of this machine-shop system. Simulation has been stopped prematurely, by the intervention of the user. Thus the ending time 4372. The initial state of the system was given by the DATA portion in Fig. 3. Initially there are 15 parts in pool, 1 machine idle, 1 material handling system available, and 3 free workers. It must be noted that, by choosing appropriate names for queues, the program's readability is grossly increased and conditions such as "part outq, mhs available" can be used, to the user's and others' relief.

At the end of simulation, the system state has been changed. Pool and inq are empty, whereas 13 parts are waiting in outq, 2 workers are free, 1 machine is idle, and the mhs is currently in use. System clock is 4372 and 422 parts have been processed and unloaded up to this time.

Number of activations for each activity is also given. The number 422 can be obtained also from the activation number of the activity UNLOAD, but this is only so because the activity is simple. In complex

Simulation started at time 0....

Simulation ended at time 4372

FINAL STATE

Entities in queues

pool : empty
inq : empty
outq : 13 part
free : 2 worker
idle : 1 machine
available : empty

Global variables

clock = 4372
unl = 422

Number of activations

load : 437
move : 624
process : 201
unload : 422
endact : not activated

STATISTICAL ANALYSIS

Queue	Waiting times					Queue lengths			
	MEAN1	MEAN2	ST.DEV.	MAX	MIN	MEAN	ST.DEV	MAX	MIN
outq	85.9	86.0	10.5	90	6	12.5	30.6	13	
free	11.5	11.7	3.0	14	3	2.2	4.7	3	
idle	6.7	6.7	2.1	13	6	0.3	1.5	1	

Fig. 5 Results of the simulation of machine-shop

activities, namely in activities with different actions with time lags between, the activation number will not mean much.

Statistics requests were done by the COLLECT statement in the program. Only three queues are included in the analysis, again by the choice of the user.

When the final state and statistical figures are analyzed, it is clear that the bottleneck of the system is the material handling system. The large waiting time figures for outq supports this result. Of the 15 parts in the system, 12.5 parts on the average are waiting for the mhs at any given moment, and this is a big figure.

3.2 A BANKING SYSTEM

In this model, a single-teller bank is simulated. The bank is open between 9 a.m. and 5 p.m. The server takes a lunch break the first time after noon that he/she stays idle. Lunch break is 30 minutes and the server posts a note telling the return time. Customers arriving during this break check the return time with the current time and decide to wait or leave according to a complex probabilistic analysis. The decision is done as follows :

If the time left (to the return of the server) is less than 10 minutes, then the customer waits or leaves with equal probability. If the time left is greater than 20 minutes, then he stays with probability .25 and leaves with probability .75. If time left is between 10 and 20 minutes, then he stays with .40 probability and leaves with .60 probability. The leaving is called "balking".

After the server returns, he continues serving until 5 p.m. At 5 p.m., doors of the bank are closed. but the server serves all customers waiting there.

Fig. 6 shows the banking system in an activity cycle diagram and Fig. 7 gives the ACSIM program and data corresponding to this system.

The arrival process is given by the activity ARRIVE, and it is a Poisson process with interarrival times exponentially distributed with mean 8. Checks are put to close the door before 9 a.m. and after 5 p.m. Customers start coming after 8 a.m.

Activity LUNCH regulates the lunch-break of the server. If the server is idle anytime after noon, and provided that he has not gone to lunch within that day (checked by the 0-1 variable EATEN) then he goes out for lunch and the return time is set for 30 minutes after.

Activity COMEBACK regulates the server's coming back from lunch. After 30 minutes have passed, server comes back, having eaten.

Activity BALKING is the decision process. A customer who has not given a decision yet (decision = 0), seeing that the server is at lunch (at lunch = 1 is used in order not to activate server, instead of "server at lunch", which would activate him) looks at the time left (return-clock) and by the complex IF statements, decides to stay (customer waiting WITH decision = 1) or to leave (DISCARD customer).

Activity SERVE is the regular service activity, which is done only after 9 a.m. Service time is uniformly distributed between 5 and 10 minutes.

Activity ENDACT ends simulation after checking whether it is after 5 p.m. and nobody is waiting for service (waiting = 0).

Statistics are required for queues waiting and idle.

Fig. 8 gives the resulting report. A total of 66 customers have been served, but none has balked. Server is idle most of the time, as seen from the statistics. Activity ARRIVE has started 137 times, but some of these are before 8 a.m. and have not created customers because of that reason.

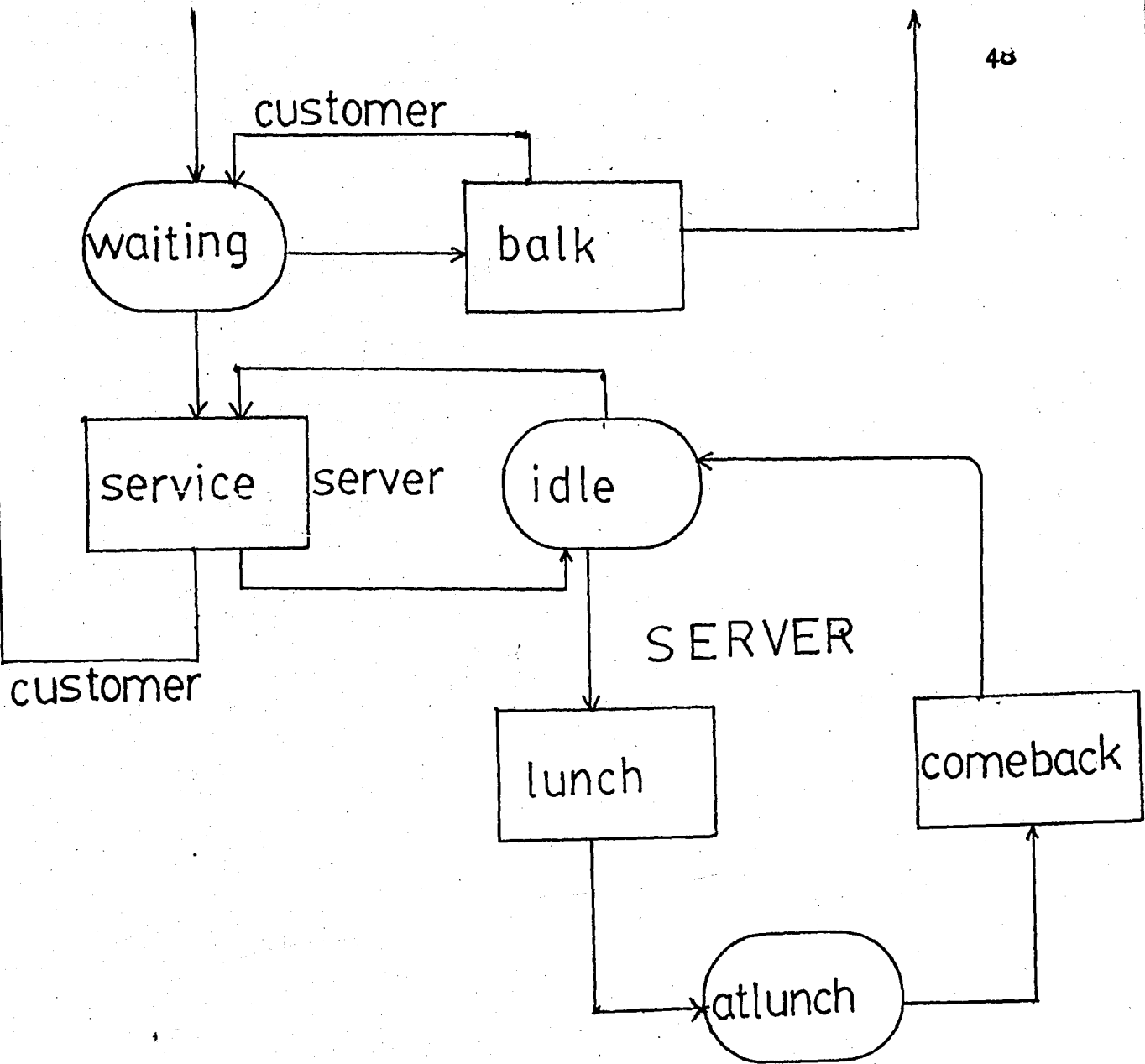


Fig. 6 Bank Cycle Diagram

```

single 3578
class customer with decision:waiting;
    server:idle,atlunch
var    return,timeleft,random,balked,served,eaten
end

arrive every xpo(8):
    if clock > 480 then
        if clock < 1020 then new customer waiting
        endif
    endif;

lunch    server idle,clock>=720,eaten=0?
    server atlunch,return=clock+30;

comeback server atlunch?
    after 60:
        server idle,eaten=1;

balking customer waiting with decision = 0,atlunch=1?,
    timeleft=return-clock,
    random=uni(1,100),
    if timeleft < 10
        then if random <=75 then discard customer,balked+1
            else customer waiting with decision=1
        endif
    else if timeleft > 20
        then if random <=60 then discard customer,balked+1
            else customer waiting with decision=1
        endif
    else if random <=50 then discard customer,balked+1
        else customer waiting with decision=1
    endif
    endif
endif;

service clock >540,server idle,customer waiting?
    after uni(5,10):discard customer,server idle,served+1;

endact clock >1020,waiting=0?
    stop

collect delay waiting,idle;
    length waiting,idle

display.
data 1 idle.

```

Fig. 7 ACSIM program describing bank

Simulation started at time 0...

Simulation ended at time 1163

FINAL STATE

Entities in queues

waiting : empty
idle : empty
atlunch : empty

Global variables

balked = 0
clock = 1163
eaten = 1
random = 0
return = 751
served = 66
timeleft = 0

Number of activations

arrive : 137
lunch : 1
comeback : 1
balking : not activated
service : 67
endact : 1

STATISTICAL ANALYSIS

Queue	Waiting times					Queue lengths			
	MEAN1	MEAN2	ST.DEV.	MAX	MIN	MEAN	ST.DEV	MAX	MIN
waiting	106.5	106.5	21.4	141	63	6.1	25.3	18	1
idle	576.0	--	572.4	576	576	0.5	1.8	1	1

Fig. 8 Results of the simulation of bank

IV. CONCLUSION

ACSIM arises as one of the tools to analyze complex systems, discrete and possibly probabilistic in nature. The present form of the language contains most of the features of previously implemented computer simulation languages. The editing and display facilities increases the user's participation, thus forming the concept of "interactive simulation". It is also an indication of the fact that simulation can be done equally successfully in microcomputers and main-frames. The advantage of the microcomputer is apparent in the fact that there is no time-sharing, which is a hindrance when long simulation runs are considered in main-frames.

The next stage in the development of the language-system is to form an "intelligent" editor, namely one which makes a limited syntax checking at the time of entry. A further improvement may be an interactive program generator which would be even more user-friendly.

The concept of "intervention" will also be extended, in that the user might change the system state or the program itself at any moment he wishes. External stimuli might be given in this way.

The final extension will be the implementation of the results of "Sample Path Analysis", to provide for the first time a simulation language with in-built sensitivity calculation possibility. Details of Sample Path Analysis might be found in [4].

APPENDIX A

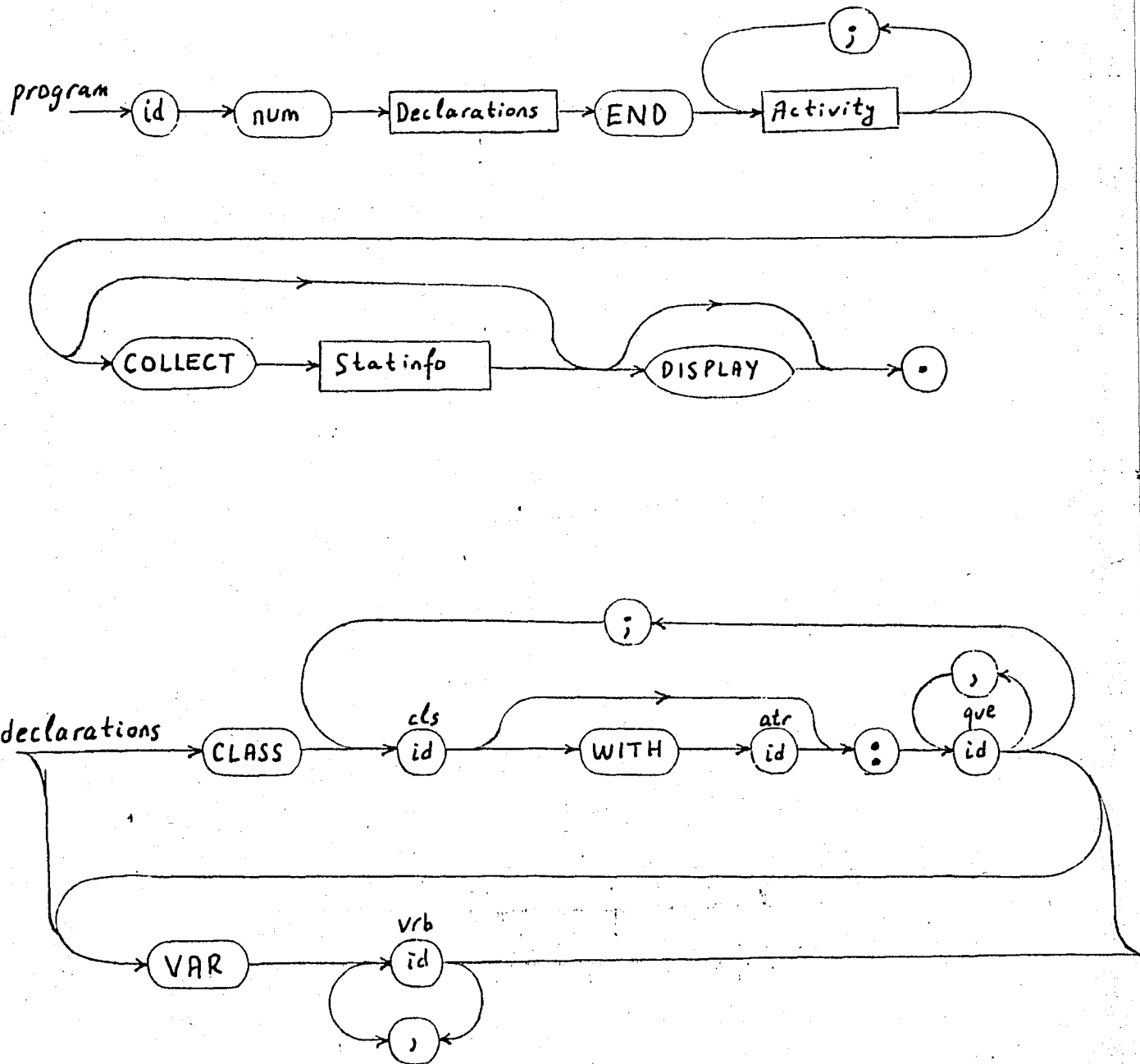
RESERVED WORDS OF ACSIM

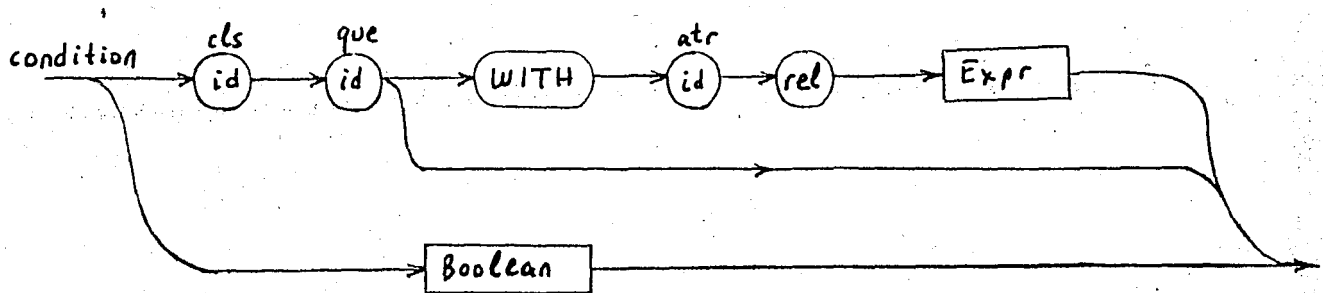
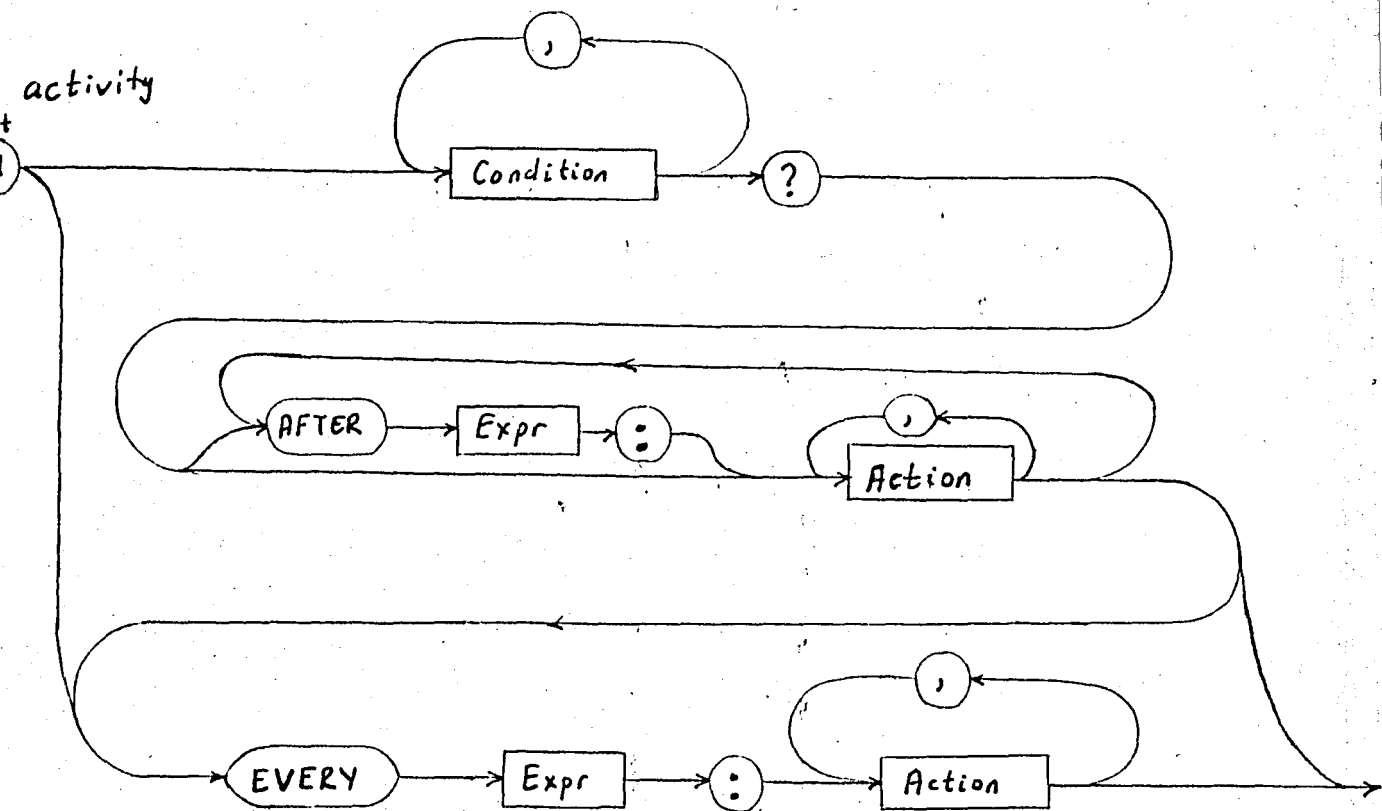
(In alphabetical order)

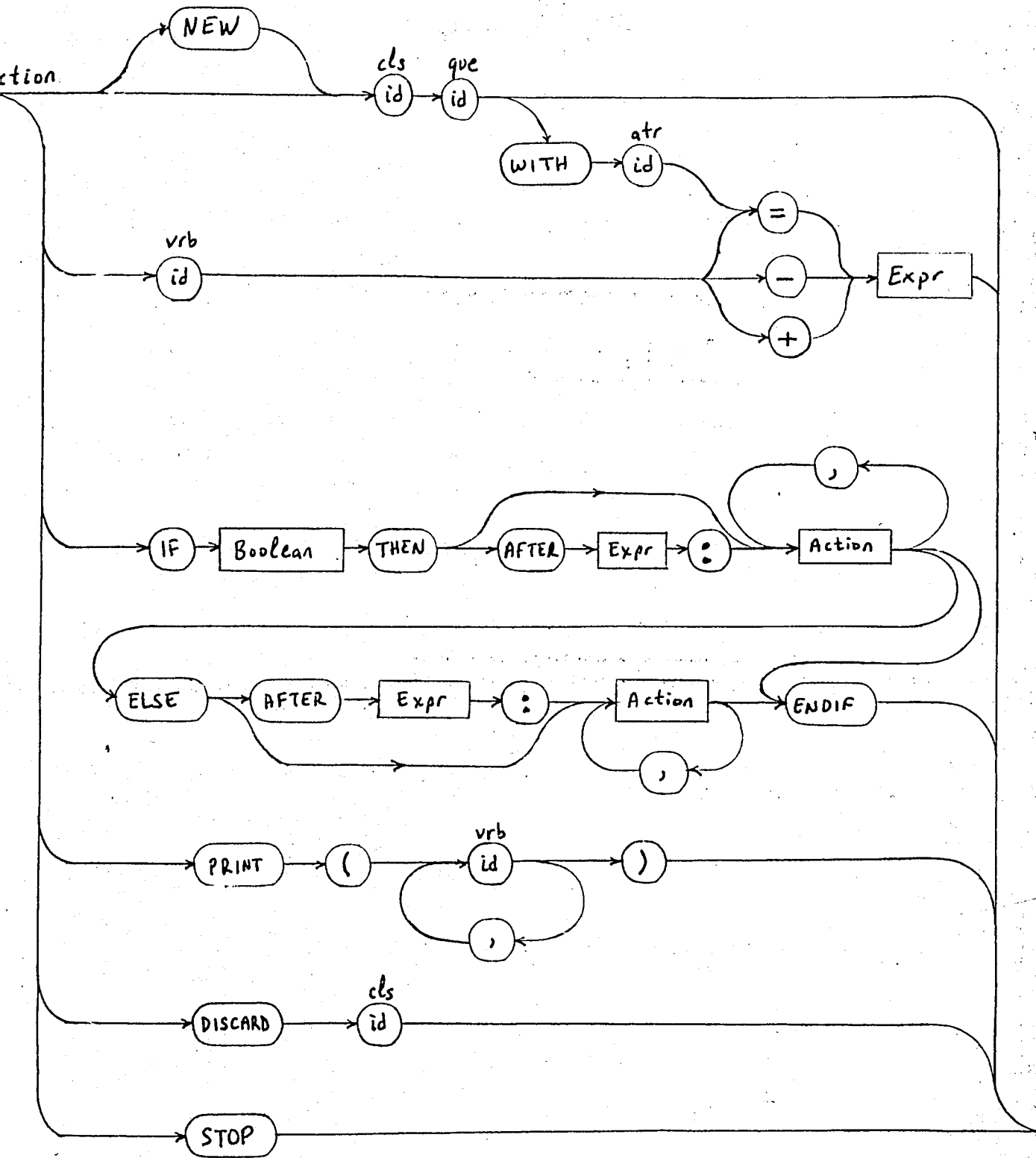
AFTER	LENGTH
AT	NEW
CLASS	NOR
COLLECT	OF
DELAY	PRINT
DISCARD	STOP
DISPLAY	THEN
ELSE	UNI
END	VAR
ENDIF	WITH
EVERY	XPO
IF	

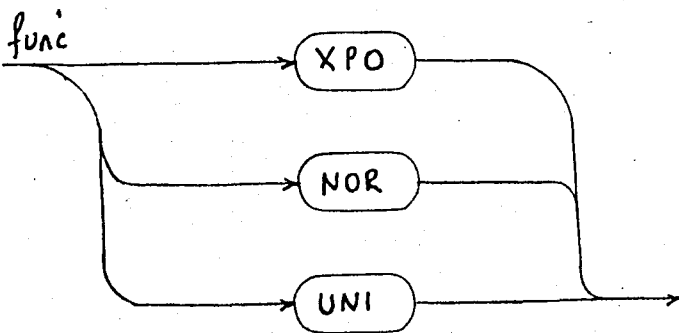
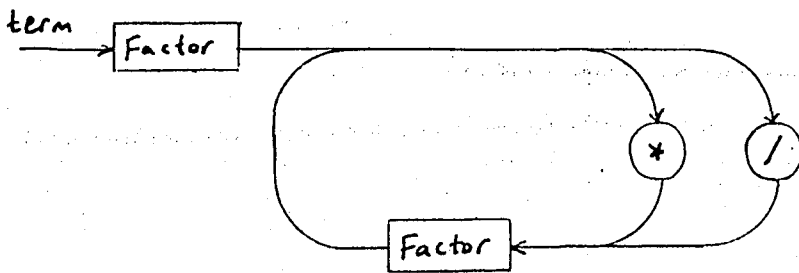
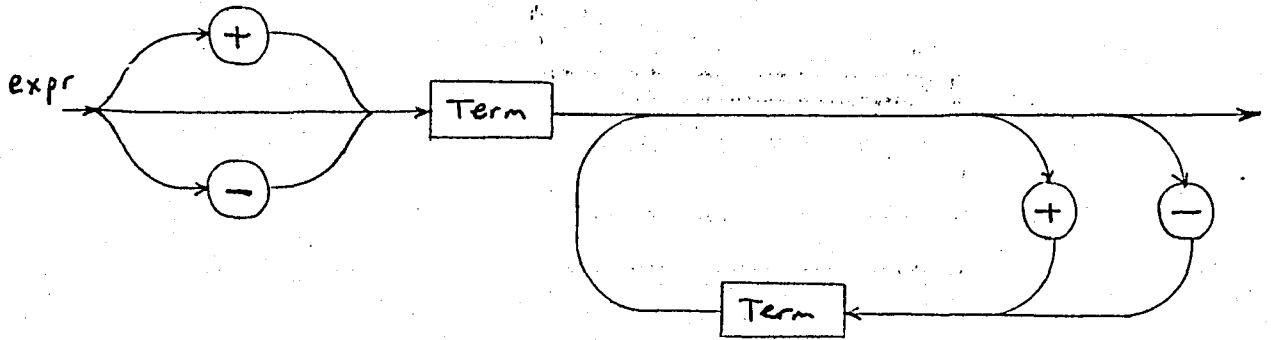
APPENDIX B

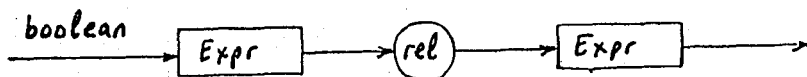
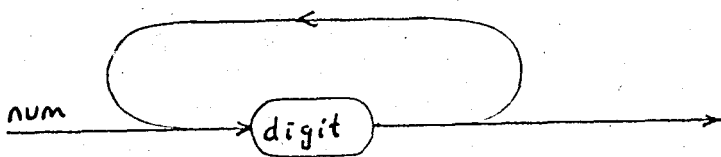
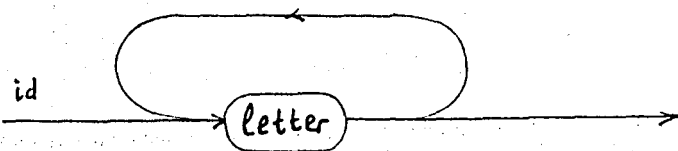
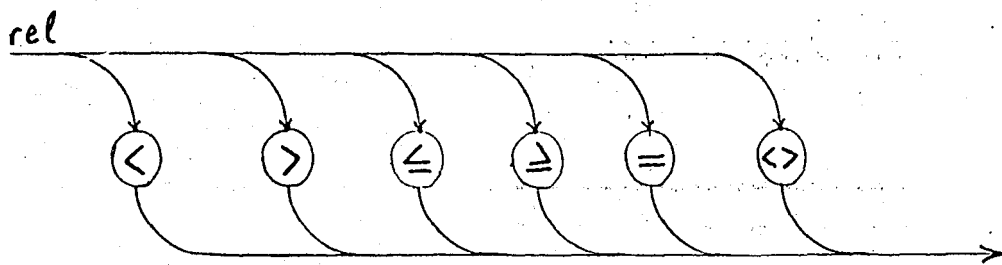
Syntax of ACSIM

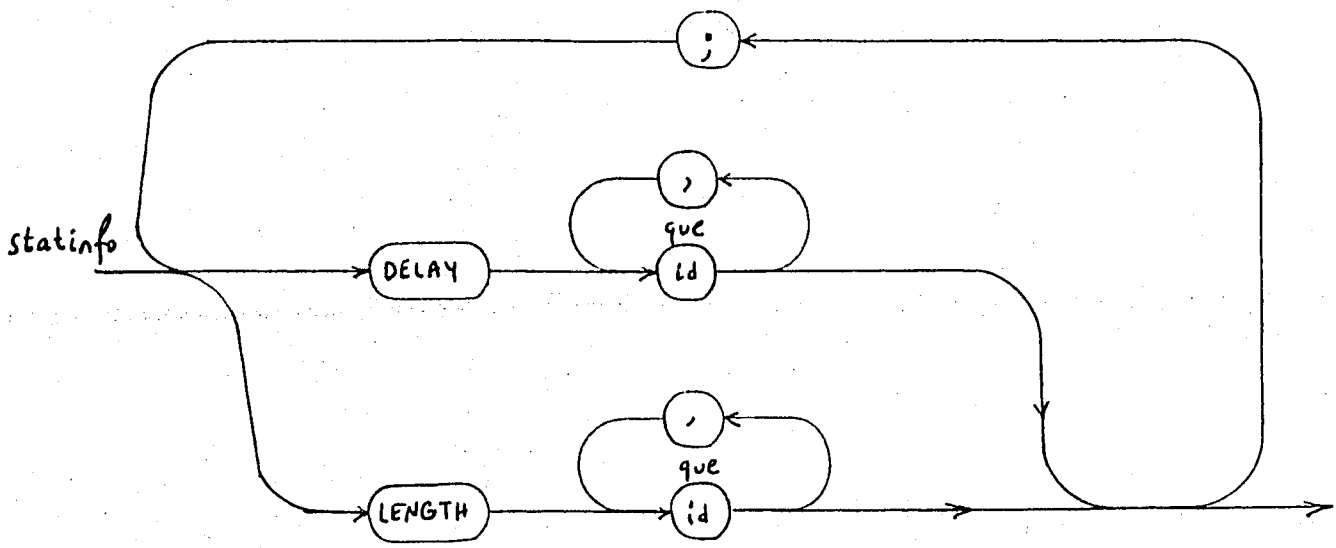


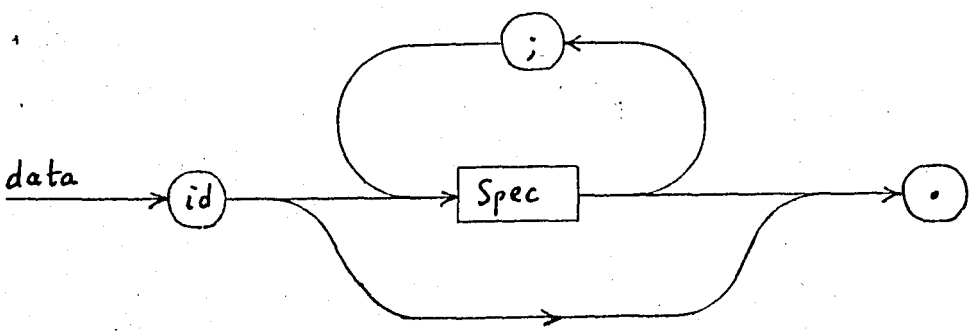
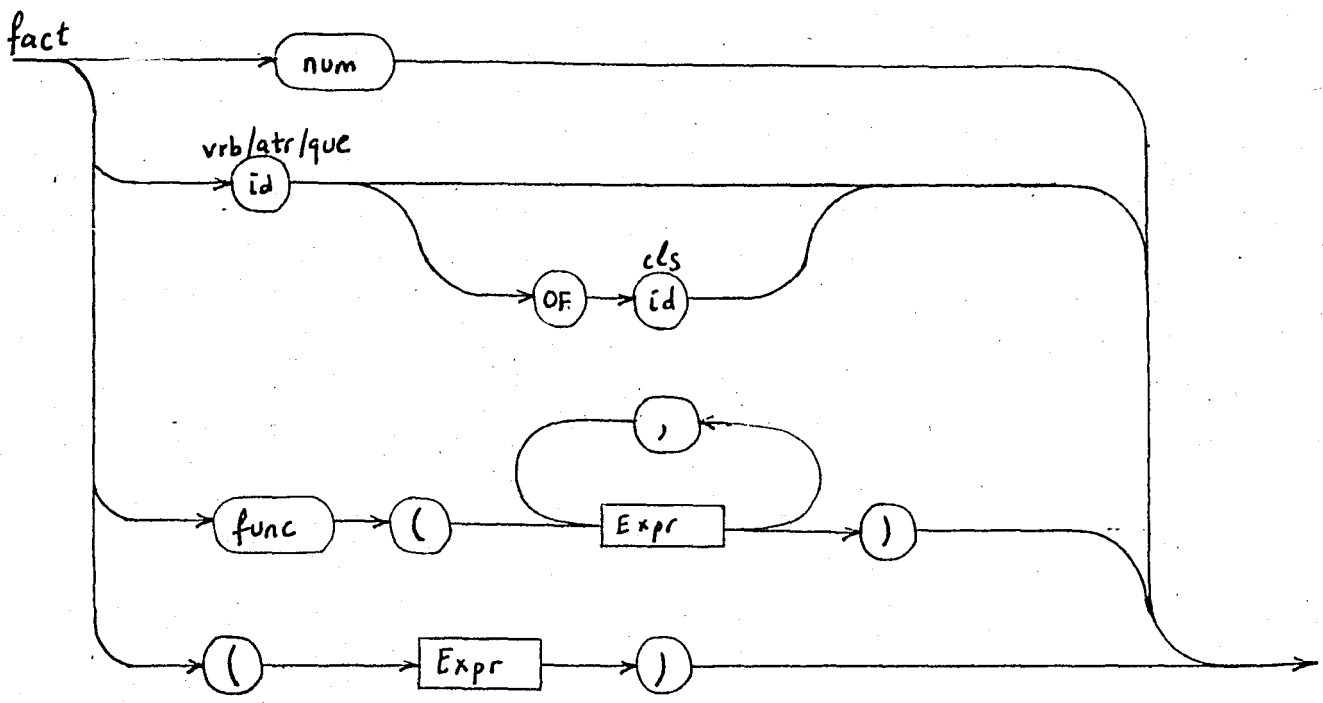


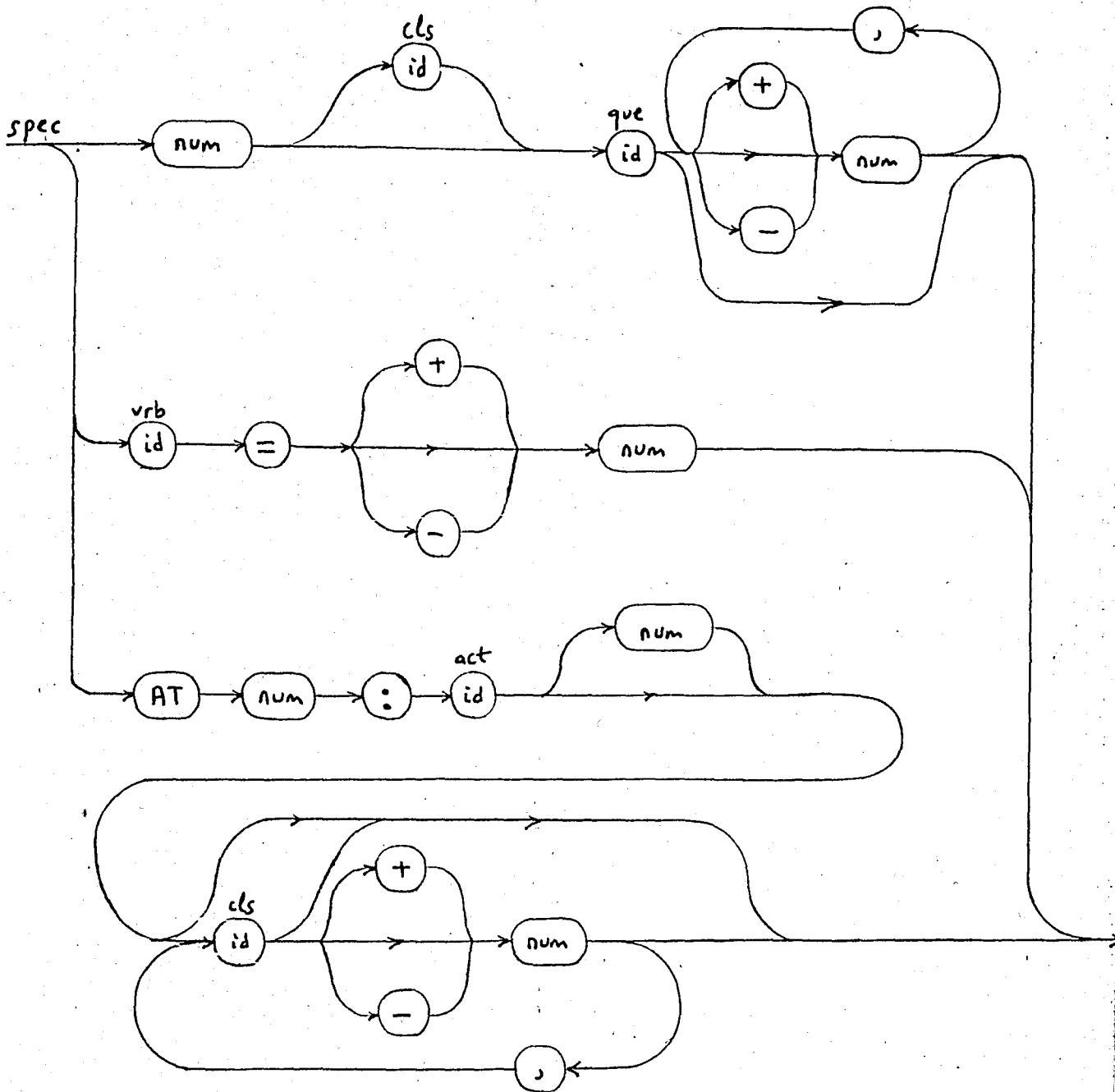












ACSIM 2.0 ERROR MESSAGES

- (1) Number too large (At most 5 digits)
- (2) Undefined symbol in input file
- (3) Program name expected
- (4) 'END' expected
- (5) 'COLLECT', 'DISPLAY' or fullstop expected
- (6) Fullstop expected
- (7) 'CLASS' or 'VAR' expected
- (8) Variable name expected
- (9) Name defined more than once
- (10) Class name expected
- (11) Attribute name expected
- (12) Colon expected
- (13) Queue name expected
- (14) Activity name expected
- (15) Name used more than once
- (16) Question mark expected
- (17) Relation expected
- (18) Class and queue are not compatible
- (19) Left parenthesis expected
- (20) At most 4 variables are allowed in a PRINT statement
- (21) Right parenthesis expected
- (22) Attribute not of this class
- (23) Equal sign expected
- (24) Class or variable name expected
- (25) 'THEN' expected
- (26) 'ELSE' or 'ENDIF' expected
- (27) 'ENDIF' expected
- (28) Class, variable, queue or attribute name expected
- (29) End of file reached in input file
- (30) Data name expected
- (31) Number of entities of denoted class must be at least 1
- (32) Class or queue name expected
- (33) Number expected
- (34) Statement number must be at least 1
- (35) A number, a variable name or 'AT' expected
- (36) Movement of entity not in conditions
- (37) Condition not satisfied by actions
- (38) Invalid factor in expression
- (39) Variable, queue or attribute name expected
- (40) 'OF' expected
- (41) Comma expected
- (42) Perturbation can not be zero
- (43) Nonexisting class name in initial state
- (44) Nonexisting queue name in initial state
- (45) Nonexisting variable name in initial state
- (46) Nonexisting activity name in initial state
- (47) Nonexisting attribute name in initial state
- (48) Too big number for statements in initial state
- (49) Random seed expected
- (50) 'BY' expected
- (51) Clock can not be updated by the user
- (52) Program must have at least 1 activity
- (53) Illegal symbol in activity initialization
- (54) Inappropriate starting statement in initial state
- (55) 'DELAY' or 'LENGTH' expected
- (56) Blank actions not allowed

REFERENCES

1. Crookes, J.G., "Simulation in 1981", European Journal of Operations Research, 9 (1982), pp. 1-7.
2. Eyler, M.A., "ACSIM : A Simulation Program Based on Activity Cycles", Research Report, Boğaziçi University, 1982.
3. Teichrow, D. and Lubin, J.F., "Computer Simulation - Discussions of the Technique and Comparison of Languages", Communications of the ACM, Vol. 9, No. 10, pp. 723-741, October 1966.
4. Ho, Y.C., Eyler, M.A. and Chien, T.T., "A New Approach to Determine Parameter Sensitivities of Transfer Lines", Management Science, Vol. 29, No. 6, pp. 700-714, June 1983.