Simulation in C-SIM

Principles of Modeling,

Simulation Techniques, and Simulation Tools

W. Kowalk

Fachbereich Informatik der Universität Oldenburg

Contents

Simulation in C-SIM

Principles of Simulation and Tools

Abstract

This paper describes the technique of simulation. Since there are many different simulation methods, we restrict our considerations to those methods that can be used to simulate particular kinds of models, usable for discrete computer evaluation.

The first part considers real systems and shows how they can be modeled. The second part shows how models can be implemented in computer systems. The third introduces the special simulation language C-SIM and its application to models considered in the first part.

Part I

Principles of Modeling Techniques

I.1 Modeling the Real World

Phenomena of the real world are complex in appearance and relationship. Human beings are usually not able to recognize all details in the short times when they occur. To make the world understandable it is necessary to **model** the real world in an adequate way.

Modelling means here to consider only those phenomena of the real world that are assumed to be important for the problem under consideration, and to reduce the interrelationships between different systems as much as possible.

A **system** is a closed set of phenomena, which is reduced in its complexity so that a human being can easily distinguish between different causes and effects.

Thus a model of the real world is always a reduction of the appearances so that it is easy to control, however, it is never the real world itself.

Some interesting models of real world have been invented by ancient mathematicians. An example is Euclidian geometry, used to survey areas for agriculture and to estimate wealth, taxes, and income of people. In this <u>theory</u> the real world is reduced to a flat plane, where no hills and valleys are considered. A boundary post is reduced to a point, a plot (using the light) is reduced to a line, a radius of a pole or a rope to a perfect circle. Using these techniques, mathematicians have proved how to construct special kinds of areas, how to survey their size, how to estimate angles, etc. All they could do, they have done, and what they couldn't (e.g. dividing an angle into three equal parts) they haven't done. So this model of the real world is a reduction of the real world phenomena to some <u>abstract</u> concepts.

Other methods were explored later in the time when the knowledge of humans grow. So they invented the number to distinguish between more and less, they learnt how to compute with them, they found irregularities (e.g. nonrational numbers, that cannot be computed by the fraction of two integers), and other things of theoretical interest.

But they also learnt to apply these numbers. In the midage merchands learnt how to model the flow of worth through their companies by use of numbers, representing the worth of their goods (or the money etc.) by a number. Book-keeping uses accounts which represents the worth of some goods. Buying and selling of goods means to change the values in an account. This is an example of a typical <u>dynamical</u> model, since here the time is recorded as well, when an <u>event</u> occurs.

These examples of models of real world systems used techniques called mathematical. Later on the methods were refined. Another important example is the theory of the movement of planets. Kepler described their behaviour using geometrical terms like circle, ellipses, or arithmetical terms like ratio of squares and cubes. A theory that could explain all observations were invented by Newton, who assumed a central force between heavy bodies and some other mechanical laws like that of inertia, coming to a mathematical derivation of the behaviour described by Kepler. To do this elegantly, he had to invent a new mathematical theory, the calculus.

Although Newton's techniques could explain the phenomena observed by Tycho Brahe (and described by Kepler) we know today that they are only approximations to the real world. Higher accuracy, however, has been found only by Einstein in the beginning of this century. So we see that a model, even if it is only an approximation to the real world phenomena, can already be useful to derive results of important applicability. Also we see that description of time dependent behaviour is always very difficult and needs complex mathematical tools, only understandable by people specially trained for this. And although the calculus can be used to model many time dependent real world systems, it cannot model all of them. Thus we need a technique that can be applied to more systems than the most complex mathematical theory.

Modelling of random phenomena has been started some 400 hundreds years ago with Fermat and Pascal, however, serious progress has been made only in this century. Modeling of time dependent random processes needs a deep lying mathematical theory, which needs much understanding of fundamental problems of mathematics. Thus, neither the method nor the results can be interpreted by average engineers. Also, the method is severly limited to almost trivial problems, or problems of little interest.

However, another invention of this century can help to model and understand real world systems: the computer. Here, the phenomena of the system to be modeled can be stored within variables, and the change of their values during time can be traced so that it is almost trivial to model even very complex systems, and to get an understanding of the behaviour of those systems. Although currently computer models can be applied only to special parameters of a problem, this technique can be extended to gain much better understanding of the real world than with the most advanced mathematical models. Electronical digital computers are the most common tools of engineers of our days, more important than the most other tools.

I.2 Mathematical Models

Modelling of the real world is a long desire of the mankind. Speculative philosophical methodes where the first one, coming out with religions, astrologic and other ancient sciences. The first more accurate models were early mathematical systems, as described in the previous section.

The idea of the mathematical models was to give a formal description of the reality by using symbols and graphics. To use them, a formal relationship between symbols was introduced, where the relationship or the transformation of one symbol into another was done automatically, not considering the meaning of the symbol. This is the starting point of a method called algorithm, which became used when applied to mathematical techniques like multiplication etc, however which became more important after the invention of electronic computers.

When symbols with no interpretation are manipulated, mathematicians call this a formal systems. They mean that a formal relationship is defined by some axiomatic systems, which are correct with no doubt. Thus, all systems satisfying this relationship, can be manipulated in the same way. This method has been applied successfully to simple formal systems, like groups, fields, etc. However, more complex systems, particular very special systems, suffer very much from intricate computation, giving only trivial results. Even some simple systems of interest could not be evaluated until today. For example, the three-body-problem, describing the moving of three bodies under gravitational force, has no closed analytical solution, since some areas suffer from 'chaotic' behaviour. Thus, current mathematics does not offer any solution to this problem. Other examples are queuing networks, where some quantities, like the distribution of the sojourn time, cannot be solved by use of calculus and probability theory. Although this is not yet quite clear, one can guess that there is also some chaotic behaviour inside queueing systems.

Despite its merits in the history of human science, mathematical methods have their limits, and if human people are not ready to confine their knowledge to mathematical bounds, they have to use other models and tools, where the most important one is an electronic computer.

I.3 Other Models

A model of real world systems needs some requirements, which we have mentioned before. They need the possibility to describe the state of a system, to describe relationships between phenomena, and they need to describe the time dependent change of the state. A computer program is obviously the most adequate tool to fulfill these requirements.

The state can be modelled using variables, i.e. memory cells, the values of which can be changed easily. Relationships are expressions or functions, changing the value of some variables, depending on the values of other variables. This can be done in specified sequences, modeling relationships of time. All together, a computer program can be used to model a real world system in a simple and evident way, applicable by people with little formal education and fundamental mathematical knowledge.

Of course, mathematical methods are advantageous when general relationships can be found. However, if this is not possible, computer models are usually much more useful. Thus, modern computer science should try to apply mathematical methods, were feasible, however, where not available, extend them to areas beyond mathematical applicability. Mathematics and computer science supplement each other, where mathematics is more elegant for simple models, however, more cumbersome for complex ones; computer models are easy applicable to really complex systems, where however general relationships are only difficult to detect and to evaluate.

In this report we apply computer models to complex real world systems for which adequate mathematical models have not yet been found.

Other methods for modelling of the real world are known as well. The most obvious ones are **physical models**, e.g. models of cars or planes tested in a wind channel. Here, often models of reduced parameters, e.g. size, are used than in the real world. The reason are economic ones, since building the model and the environment in which this model is tested are usually very expensive. As known from history, the first motor plane was developed in a wind channel by the brothers Wright.

Also, **electrical models** can be used, particular with some units called sometimes analogous computers. They use classical electronic circuits, like capacities, to model some mathematical properties, like differentiation, build a model of the system to be analysed and run this model by use of some input current, observing the output on an oscilloscope. This type of model is coming out of date, because also digital computers can models these subjects in a more elegant way.

Concrete models are often not used to model the total system to be analysed. They investigate usually only some properties, which are embedded into a more abstract model. This type of technique, using different models, is sometypes called: **Hybrid**. Also computer models can supplement mathematical models by estimating some subsystem of a system that cannot be treated mathematically, and use the results to insert them in some adequate way into the mathematical model, which can be analysed completely.

The short overview of modeling techniques should help to understand that there are many ways how mankind can analyse the real world by some modeling techniques. Usually, all these models have their rights on themselves, and all can supplement, provided they are used carefully. Of course, only quantitative models are presented here. The way like philosophs, psychologists, poets, or other humanities consider this world (and particular the human beings) are not the subject of this report.

I.4 Model Concepts

A model of a real world systems can be created in many different ways. However, since much work is to be done before such a model runs credibly and efficient, some concepts have been created by many researchers. In this section we consider some of these concepts. This should help the reader to develop a general understanding of the problems that occur and to introduce a common language how models and their concepts can be talked about.

At first we analyse some properties of the elements of a simulation model. Then we show, which standard functions are useful in simulation systems. We analyse some of the possible relationships between elements, and consider the representation of quantities in models. To classify the models, we define some stages of model concepts and different classes of modeling techniques. Finally, the types of computer models in use are presented and classified.

I.4.1 Elements in a Model Concept

Usually a systems consists of several parallel components. For example in a traffic model the system consists of parallel cars moving on streets. The notion parallel process is used to describe an element that includes an active component. Other elements are changed by theses processes. They seem to wander through the system, altered by the processes, and being generated and disposed. These passive elements are sometimes called messages, packets or jobs.

1.4.1.1 Active Elements

Parallel processes are differentiated in more detail. Processes are **independently parallel** (*nebenläufig*), when they run without influencing each other, either directly or indirectly.

Processes are **cooperating**, when they work together on the same goal. Processes are **concurrent**, when they work on different goals, however, they need the same resources, for which they have to negotiate. A concept introduced in the simulation language Simula 67 is called a **coroutine** and means, that two processes run alternatively, using some time slices of a common processor. This concept will be used to execute our computer models, however, it can also be used to model the behaviour of some real world systems, e.g. computer systems using the time slicing technique.

1.4.1.2 Passive Elements

Passive elements will be generated during the life time of a model. Then they should be able to store arbitrary information (which is in some simulation concepts sometimes too much restricted). If they are no longer needed, they should be delivered to the system which disposes them.

1.4.1.3 Permanent and Transient Elements

In many systems, parallel processes are permanent, i.e. they are to be generated at the beginning of the simulation, and they remain there unchanged until its end. However, in some systems these active elements, as well as the passive ones, can be created in any number, which makes those system more flexible, however, also more complicated.

1.4.1.4 Common Properties of Active and Passive Elements

There are many common properties active and passive elements should have, some of which will be considered here.

Elements can be **generated**, so that a data record is provided. In computer simulation, the corresponding memory is to be supplied dynamically. In other models there is usually a severe confinement on this requirement. Here the generated element is always of a particular type, defined somewhere else, so that its kind is uniquely defined.

An element can be **referenced**, i.e. it is possible to name it or to point at it. In computer programs this is done by variable names or pointer structure, while in other models a name of an object is provided.

It should be possible to group the elements, this collection of elements is usually called a **set**. Examples are sets according to some properties (active/passive, discrete/continuous etc.) or sets according to the meaning of the elements in the real world system, e.g. all servers of jobs, all generators of messages etc. There can be an order in this set, e.g. a sequence (for a FIFO-queue) or a cycle.

Elements can be **disposed** when they are no longer needed. In computer programs the corresponding memory is reorganized, while in other models the corresponding resource is removed from the model.

Elements have the property to **hold information**. In computer programs, data record can be used, while in other models the meaning of an object is usually recorded somewhere else, e.g. in a protocol. Examples of information to be stored are the generation time, the route of an object through a network of processes, waiting and service times and other parameters.

Some elements contain **functions** besides information. There should be a simple structure to ease the programming of these elements.

1.4.1.5 Static Elements

We call an element static if it exists during the execution of a simulation model. These are events, times or phases, which can be organized to processes. Other static elements are variables or complex data structures like histograms, which can be found in computer programs as well as in other types of models.

1.4.1.5 Dynamic Elements

There are some distinguishable phases in a simulation run. During the **initializing phase** the model is set in its initial state. E.g. a physical model is put into its start position; in an electrical model a condensator is loaded; in a computer model the initial values are assigned to the variables, etc.

During the **execution phase**, the model is run by allowing the elements to change themselves or alter some other elements. Here some notions are:

- Generation of elements by a **source**
- Route, history, live, flow, or transfer of elements through a **network**

- Waiting of elements, waiting time, waiting queue
- Service of elements, service time, server
- Disposal of elements in a **sink**

Finally, the execution stops and the observed quantities are evaluted. In this **termination phase** the results are evaluated and displayed; they should be stored or recorded for further use in some permanent files.

I.4.2 Standard Functions

To model systems, the simulation concept should provide some standard functions. There are several reasons to do so. In contrast to ad hoc functions, specified and implemented by the analyst:

Standard functions need less evaluation work
Standard functions are usually less error prone
Standard functions are better documented
Standard functions need less memory and are faster (in computer models)
Standard functions are easier understood and more general

If the model concept supplies standard functions, they should be used. However, in some model concepts, the standard functions are wrong, bad or too primitve, so that there should be the possibility to extend the set of functions of the corresponding system. We consider here three sets of standard functions that are useful for simulation.

I.4.2.1 Probability

The system should supply a good random number generator. Random numbers can be formalized by a mthematical sequence $\{x_i\}$ of real numbers in the intervall $x_i \in [0,1)$, where 0 is included and 1 is excluded. Thus if x_i is the i-th random number, then x_{i+1} ist the (i+1)-th random number, and the random number generator is a function `Random` that maps x_i to x_{i+1} for all i³0:

$$\begin{array}{llll} \texttt{Random :} & [0,1) & \rightarrow & [0,1) \\ & x_i & \rightarrow & x_{i+1} \end{array}$$

Among all those random number generators suggested in the literatur, there is only one good one: The linear congruential random number generator. It multiplies x_i by a number, e.g. 5, and computes the residual value, i.e. the fraction. The analyst should however always test this function before he can use it faithfully.

Other functions should provide distributions of this random number sequence, e.g. true with a probability (IsTrue), equally distributed on an intervall [a,b], either with integer or with real numbers; also some more complex functions like negative exponentially distributed (NegExp), normally distributed (Normal), Erlang distributed (Erlang) and some other important functions.

Finally there should be some functions to evaluate the results, e.g. histograms to store the observed quantities in a differentiated way, mean, variance, moments and SCV of random variables, and also distributions with some confidence intervals. All these functions, carefully applied, might help to improve the results of a simulation run.

I.4.2.3 Special Simulation Functions

There should also be some functions, which help the analyst to use the simulation system. These can reach from simple scheduling functions for coroutines towards functions to evaluate complex subsystems in a hierarchical simulation environment. Some of these will be explained in this text.

I.4.2.3 General Purpose Functions

There should also be some general functions or statements, which are well known from general purpose computer languages. These types of functions should be supplied by the simulation system, as well, although not all of them are rearly useful there. For example, Simula 67 is an extension of Algol 60, GPSS is based on Fortran, and CSIM is based on C.

I.4.3 Relationships between Elements

The model concept should provide simple methods to describe the relationships between the elements of the model. These are of different character and can be used in several model concepts.

I.4.3.1 Mathematical Relationships

Typically, the equation a=b is a relationsship between values. However, mathematics has refined this by using difference or differential equations to express the relationsships after some time. In simulation models mathematical formulae are used to compute the next state from the previous ones.

I.4.3.2 Functional Relationships between Elements

The elements in a system are influenced by internal or external sources. These influences can be provided by a condition of the state of the system, by a message sent by another element, or by some quantity. The latter can be either a

- **constant** quantity
- **minimum** and **maximum** quantity for **sensitivity analysis**
- **mean** quantity
- **randomly distributed** quantity
- another quantity

In this report we will use all of them.

I.4.3.3 Structural Relationships

In some models it is possible to detect some structures of the relationships between the elements. They can be

- no relationship between elements
- **sequential** relationship between elements
- **cyclic** relationship between elements
- **networked** relationship between elements
- **ring** relationship between elements
- **feed back** between elements
- **general** relationship between elements

Also, the elements must be referenced by each other. Usually, a formal model uses names to reference an element. Mathematicians are used to one-letter names, sometimes taken from foreign alphabets (a, b, α, β, $\aleph$, $\Im$, $\Re$ usw.). In computer science self-explaining names are preferred. Here, also names are defined only within a local context, e.g. within one of several processes. Thus different processes can use the same name, meaning however only different elements. Also, the notion of a reference is made more explicite, where a name can mean either an object, or a reference to an object. The latter concept increases the flexibility, however, the difficulties as well.

I.4.4 Presentation of Input, Execution and Results

To increase credibility the quantities used in the beginning of a simulation, during its execution and the evaluation of the results should be made as evident as possible. Thus the used quantities, structures and relationships should be recorded in a protocol, they should be displayed on a graphik screen, if possible, and they should be controlled at several parts of the models.

I.4.4.1 Input representation

The input data should be presented explicitely, e.g. within a list of quantities. In computer models they should be read in interactively, or, if less cumbersome, over a data file. When the model is displayed, the initial structures and values should be plotted, so that failures in these data can be recognized immediately. Also the ranges, e.g. of sampled data, should be displayed. In many cases it is useful to change some quantities during the simulation run.

I.4.4.2 Execution Representation

A model is executed, if after the initial phase the simulation run is performed. Here, only the change of some of the quantities might be of interest and therefore not more should be displayed. Thus a hierarchical representation of the model should be possible.

Usually it is easy to display the alteration of some quantities, like simulation time or queue length. However, if the structure of the model is changed during execution, it is also useful to display this in the external representation.

In some case, intermediate results are computed during the execution; they might be use to determine the length of the simulation run. It ist useful to display such results as well. Of course, the range of some evaluated quantities should be displayed to decide, whether the corresponding parameters were choosen correctly.

I.4.4.3 Output Representation

After the simulation run it is necessary to display the final state of the model and to present the final statistical results. The latter can be computed by the simulation run; if the data are stored in a permanent memory, like a file, they might be analysed by a special computation. This is particular useful when the analysis is lengthy or complicated, or this task needs human interference. The final judgement can only in special cases been made automatically. It is always a matter of common sense and interpretation.

I.4.5 Stages of Model Concepts

There are several products to sustain the modeling of a system. In this section, we will classify them and analyse, how they are related to each other.

I.4.5.1 Simulator

A model that can be used to simulate a real system with no additional work is called a Simulator. We have already seen that these can be physical, electrical, mathematical models, as well as computer models or others. Examples are flight simulators (e.g. used for the training of pilots; also available as computer game); control mechanisms in electronic circuits; a table of solutions of differential equation for a special problem, like the flow of heat on a surface; or some simulation programs ready for use, where only some parameters are to be initialized.

I.4.5.2 Simulation Concept for a Problem Class

Some models can be used to simulate all systems that belong to a class, e.g. wind channels to be used for motor cars, sail boats, or aeroplanes; analog calculator used to solve differential equations; general algorithms to solves difference or differential equation; or programming systems that can be used to build a model for some real system. There are some systems of this type to be purchased.

I.4.5.3 Simulation Language

It is always possible to create a programming language that can be used to model systems and to simulate them. They are usually not applicable to general programming problems. Examples are GPSS, MIMIC, CSIM, and others, some of which will be explained later.

I.4.5.4 Programming Language with Simulation Concepts

Other developments in the area of simulation systems use general purpose language and extent them with some features required for simulation. There is one example, Simula 67, which used Algol 60 a basic language, however, extended this with adequate functions to model parallel processes; also the new concept of an object was introduced, which includes both, data as well as functions, and a defined interface to the environment, inventing the object oriented programming style. We see from this developement that simulation systems require quite different concepts than algorithmic language which were invented for the solution of mathematical problems.

I.4.5.5 General Purpose Programming Language

It is always possible to use a general purpose programming language, like C, Fortran, COBOL or even Assembler. Here, the necessary concepts for simulation programs are to be ‘reinvented‘ and ‘reimplemented‘. Since many people have spent much time and work on this subject, the state of the art should be considered before such a work is done.

I.4.5.6 General Remarks

Although the five stages above can be used to classify a model concept, it is not always quite clear where to classify a given system. A model of stage 1 needs always some parameters, so it can be used to model several systems, i.e. a class of problems, so that it might be considered to be a model of stage 2. If models of stage 2 are defined by a complete programming language, they might be a simulation language of stage 3, etc. Alhough some doubts may be left where to classify a given model concept, this set of stages may be helpful to see which range of possible modelling concepts exists. The following list gives an overview of these stages:

		programming overhead if model		
Model Concept	Example	suitable	not suitable	studying effort
Simulator	X.25	no	not suitable	very low
Simulation Concept for a Problem Class	CSIM+DFN	low	less sensible	low
Simulation Language	CSIM, GPSS, MIMIC	medium	suitable	medium
General Purpose Language with Simulation Concepts	Simula 67	high		high
General Purpose Language with no Simulation Concepts	Pascal, C, Modula, etc.	not suitable	not suitable	too high

Another question of interest is when to apply which type of model. Of course, the lower the stage the better the model. E.g. if there exists already a trustworthy simulator for a particular problem, then this should be applied. But if this is not the case, a simulator for the corresponding problem class is usually much more efficient than a general purpose language. It should be mentioned that there exists some methods to model a system in a stage 4 concept and translate it into a stage 5 concept, e.g. to translate a Simula 67 program into a Fortran program. However, these concepts seem to be too error prone and to cumbersome.

Another problem is to learn to use the corresponding model concept. The analyst has to understand both, the real world system that is to be modeled as well as the model concept. Thus its task is usually very difficult and he is responsible for much spent time and resources.

I.4.6 Properties of Simulation Model Concepts

The following table lists some properties of modeling concepts.

State	Time	Time distance	Randomness
state continuous	time continuous	continuous equal distance	
state hybride	time hybride	continuous equal distance event driven	deterministic stochastic
state discrete	time discrete	equal distance event driven	

The model of a system can be classified according to its dynamic and static elements. The static elements describe the state of the system. The state can be continuous or discrete or hybride, i.e. a mixture of these two. While this difference is in mathematical models very important, it is usually of minor interest in computer models since there all quantities are discrete within a given accuracy, modelled by use of real numbers.

The dynamic elements are the time, which is either a continuous time or a discrete time; some hybrid concepts use both types of time. Continuous time models use either continuous time models (like electrical or analytical models), or it uses small time steps of usually the same distance, small enough to imitate the behaviour of the real world system adequately. Discrete time models use either equal distances between two events (where at many time steps no events occur), or they use event driven modelling, where the time depends on the next event that changes the state of the system.

Hybrid models can either use parallel processes of different time type or they use hierarchies of processes of different time type.

All model concepts can use either only deterministic or additionally random elements.

It is also useful to distinguish between hierarchical and non-hierarchical model concepts. While the former are more general, the latter are usual more efficient.

I.5 How to Model a real system

In this section we list some general aspects that should be considered when a system is to be modeled. However, it is not possible to give a unique method how to model all types of systems. This depends on the type of systems as well as on the type of model.

I.5.1 Formulation of the Problem

A system cannot be modeled adequately unless it is carefully understood. Thus the analyst should either be an expert in the corresponding area (which is very often the case), or he should be able to interrogate such an expert. Otherwise, neither the model can be useful, nor the results can be interpreted adequately.

If the system is understood, the characteristics for the analysis are to be determined. This is usually a critical, since too many characteristics might complicate the system too much, so that it is not solvable, while too less characteristics would complicate the system, so that its outcome would be too poor and useless.

If all characteristics are specified, the variable and constant parameters are to be defined. In many systems, the structure of the components is fixed, while some quantities, like arrival rates or service speed, are variable. However, in some models also the structure of the model can varied.

In some models some interactive control is required, which is also to be specified at this phase.

I.5.2 The Formal Model

If the characteristics of a model are known, it is necessary to describe the model formally. Here all relationships between elements, their local and global quantities and their ranges, as well as input and output data are specified. Also, the change of the states should be described in a formal way to ease the implementation.

Formal notation are very often mathematical ones. In many cases, however, the formal notation is derived from the corresponding science, e.g. in in chemistry where a topological model is used, or in mechanics, where a technical draft of an engine is given.

I.5.3 The Executeable Model

It is usually easy to derive from the formal model the working model that can be executed to produce the required results.

If the working model is a computer program in a high level language or a special simulation language it is also possible to use the formal description more or less directly to derive from this the working model. If the formal model is a technical draft, the corresponding mechanical parts are to be realized. If the formal model is an electric circuit diagram, the corresponding electrical wiring is to be made.

I.5.4 Validation of the Model

The required properties of the model should be tested. Since models are usually very complex, it might be possible that errors are introduced during the modeling process. Also, if the model is to rough, it might be too unprecise in many ranges etc.

To validate whether the model is adequate, it must be tested against some hypotheses, which should be stated before the test and independently. Since a model can never be the real system itself, there should be defined acceptable differences between the two. These requirements are difficult to state formally, however, good statistical tools should help to do this task.

I.5.5 Calibration of the Model

If the model does not suit sufficiently, it is sometimes possible to change some parameters to make the model suit. This is of course dangerous, since some typical behaviour of the real system might be hidden by this approach. However, since the model can only be an approximation to the real world, **calibration** is sometimes necessary. If this is done carefully, this method can be justified. In many cases, a more detailed analysis of the system might help to avoid calibration.

I.5.6 Experiments with the Model

Experiments should be made systematically. Usually, some preruns of the simulation model are required to derive the basis properties of the system, however, after this a systematic variation of input parameters should help to understand the system in a wide range of applications. The corresponding steps in the parameters are dependent on the desired results. If the general problem of the system is to be analyzed, or for educational use, the steps might be very large, while for optimization of some paramters the steps will be finer and usually varying.

In many cases it is not possible to analyse all combinations of input variables. Thus some interpolation techniques are necessary. Particularly, to find the optimum of some system properties, systematic methods are required that can handle the variing values of some simulation results. Also, the time of a simulation run (i.e. the number of events) must be limited by some statistical method. Intelligent simulation system would compute during the run time of the simulation intervals of confidence and determine in this way the required simulation time "on the fly".

Numerical results can estimate the **sojourn time** (mean, variance, distribution) of some objects in some components, or derived quantities, like **throughput**, **occupation time** or **utilization** etc. Qualitative properties are **deadlocks**, **liveness**, **reachability** or **coincidence** of events (e.g. do some events happen at the same time). Since the model is completely under control of the analyst, it is usually easy to observe and compute these results.

I.5.7 Evaluation of the Model

The results must be analysed statistically. This means, that confidence intervals are to be computed und the results are to be comparte. If the difference between two results is of no significance, such a difference must not be interpreted into the outcome of a simulation. Also, all assumptions made by the analysist must be documented together with the results. The goal of the simulation should be made clear, and raw data must be interpreted by the analyst. Since simulation is an expensive technique, available results should be used to define new parameters or questions to the models. A good outcome would be a sensitivity analysis, where the dependence of output data on some input parameters is revealed.

I.6 Organization of Discrete Time Simulation Models

In this section we analyse the most important notion of computer simulation concepts, the **event**. The idea behind this is the fact, that usually the change of the state of a system can be described by a simple function. E.g. the arrival of a customer in a queue in front of a bank counter is an event that changes the state of the queue; or the movement of a planet within a very short time can be expressed by a differential equation. Thus the analyst knows usually the incremental behaviour of the considered system, however, he is interested in the long term behaviour. Thus, in a discrete event model the state of the model is altered incrementally, which can be simulated by a computer program in a very simple way. After a long simulation run, the final state (or also some intermediate states) can be analysed, which is the goal of all the modeling effort.

We analyse here some of the properties of events that should be availabe in a simulation language.

When an event is performed it describes the change of a state at that time and the next event after some time. This can be formulated directly in some older simulation languages like SIMSCRIPT, which is a collection of some events that change the state space of the program. Threfore, this method is called **event scheduling approach**.

This method is very flexible, however, it has almost only limited possibilities to define a relationship between the events and their variables or other events. Thus, this method is not useful for complex models.

The next step is to combine some events to build a process. Only events within this process may start each other, and at most one event is active. Thus the naming and referencing of events becomes much simpler.

Now, also local variables can be introduced. They are local to the process (and can be referenced by the events), and they describe the state of the process. Thus the variables together with the events can be used as a model of a component of a real world system, e.g. a computer server, a motor car on a road or planet in the space. These elements confine a space and an action which is to be performed next according to the active event.

This method is sometimes called **process oriented**, at least, if all elements of the systems are processes. However, a passive process, containing only variables, can also be used in these models, so this concept is ideal for modelling jobs and servers of these jobs. Here, all processes can be generated in arbitrary numbers during the simulation time, which differ only in their types. This concept (in a simplified manner) is sometimes called **object oriented programming**.

In the latter approach the next execution time of an event is to be determined by the previous event. In simulation languages like Simula 67 a processes can be **passivated** and later been **activated** by another active process. Thus it is probably possible, to define a general condition when the next event of a process is to be started. However, this is very cumbersome (and thus rarely used).

In the **activity scanning approach** a general central condition can be stated which process is to be executed next. E.g. if a queue is filled up to its upper bound, a process that wants to send a job into this queue should be blocked. This could be implemented by this concept, however, it is much too complicated for really complex systems, since all processes are to be considered. Thus, there has never been any application of this method.

More general, there could be a distributed condition for each process or even event. The process is delayed until the corresponding condition holds. This concept can be found in some simulation languages, e.g. in CSIM.

However, this concept requires much computational overhead, since after each change of the state space the condition is to be reevaluated. Thus, for some applications simplifications of these concepts would be useful. For example in CSIM the condition can be stated that a message from another process arrives. Until this does not occur, the process is passivated.

Another concept can simplify the modeling of some effects. An event has been defined to change the state space and then activate after some time another event. It is sometimes more comfortable to delay the change of the state space until some time is spent, then change the state space, and after some further delay to activate another event. Thus we consider the following general event, which is called a **phase** in CSIM:

```
Phase <name1>
    Delay <time1>
    Await <condition> Until <time2>
    Delay <time3>
    <change state space>
    Next <name2>      After <time4>
```

The semantics of these expression is the following:

When this event becomes active, delay the first time. Then check the condition, and if this does not hold before the second time delays, then proceed (there is a possibility to check, whether the timer ran out or the condition became true). Then delay the process another time, if necessary, and then change the state space. After this, the next event `<name2>` is selected and activated after the `<time4>`.

Concepts of this type, together with some additional syntactical extensions, are supplied in CSIM, which will be introduced in this text.

Part II

Computer Simulation Principles

II.1 Principles of Computer Models

Modelling real world systems means to map the characteristics of a real world system onto a computer. There are many possible different characteristics, the most important of which are numbers, however, others can be colours or shapes, there can be psychological characteristics like intelligence, balance, or beauty, or sociological like democracy, justice, or fairness; other important characteristics are know from medicine, business or art. Some of these can be mapped in a natural way onto numbers, particular integers, others can not. Models that use numbers are usually adequately mapped onto computers, while for others computers are only rarely used. We are interested in models for digital computers (there are also analogous computers, although they are coming out of use).

Mapping a system on a computer means to map this onto a model. Since numbers are the most natural objects of computers, it would be wise to use numbers of objects of the model. However, some simple models, like finite sequences, can be mapped in a very simple way into numbers, so that such sequences are often used instead of numbers, e.g.

(black, brown, red, orange, yellow, green, blue, violett, grey, white) is a typical sequence of colours, and in many applications it is of no importance, on which particular numbers they are mapped. So, a computer language using names of features instead of numbers are helpful in simplifying the model.

Computers use numbers, stored in memory cells. Each number can be retrieved by use of an address. To simplify addressing, the addresses are mapped onto names. Thus, we can assume that a memory cell has as name. A name, together with a memory cell is called a **variable**. Variables can change the contents of their memory cell, which means that the previous value is replaced by a later value. If `VarName` is a name of a variable and `Value1` is its previous contents, then we can write (using the sytnax of the programming language C):

```
VarName = Value2 ;
```

to replace Value1 in memory cell `VarName` by `Value2`. `Value1` is completely forgotten (unless it has been saved somewhere else in another memory cell).

To model a static system on a computer, we map all characteristics of the system into variables. If there are dependences between different characteristics, we can compute the values of some variables by use of others.

Example: If a building is to be build, the weight of the upper stores has to be carried by the lower stores. Thus, the architect starts with some assumptions on the strength of the lower stores, and computes the weight of the upper stores. If they are too heavy, he has to change the strength of the lower stores, changing their weights, which again needs some change in the strength of the fundament, etc. Here, using some known variables, there is an algorithm to compute the values of other variables. The example has, by the way, led to the invention of the computer by K. Zuse.

Thus, although the system is static, the variables can change, however, this is not assumed to be a change over the time but over the relationships between the variables. Computers are useful tools to model such type of systems, e.g. to model buildings, but also for texts, drawings, or to collect data not depending on time.

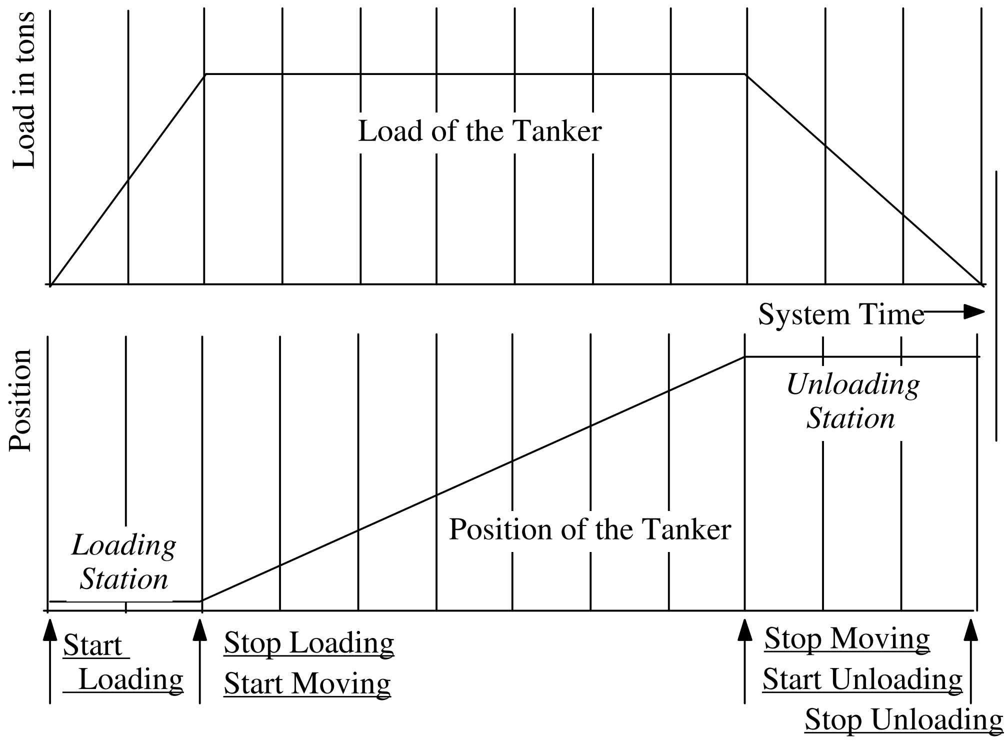
We are here more interested in systems changing over time. Such kind of systems are of course more natural, sind almost everything is changing. Also the pyramides will not stay there forever. But if the change in time is slow enough, static models can be satisfying as well. Dynamic models, however, need always the notion of **time**. We consider time to be a number, usually of type real, increasing slowly but steadily, and at each time there is a particular state of the model. Thus, to model a time dependent system, it is sufficient to add to the variables representing the state of the system, another variable, representing the time when the corresponding state is attained. If time changes, then also the state of the system changes, where the change of state has to be computed by use of the former state and the time increment, which can be of different size.

Computer models are discrete models, which means that there is always a smallest increment of a number, e.g. the time. Thus, computer models are alway **time discrete**. This is not a serious problem, since small changes in time usually allow only small changes in the state of a system.

Example: Movement of planets, attrackting each other by gravitational forces. Using sufficient small increments in time, the resulting force on a body can be computed, and since the time increment is small, one can assume that the force does not change during this time. Then, the acceleration on the body, an so the change in speed and location can be estimated, and doing this for all bodies, one can compute the state of the space after the time increment. Repeating this for all time increments, the resulting state should be very near to the real state.

In many applications, however, the state of the system does not need to be modelled with continuous numbers.

Example: Transport of a liquid (e.g. petrol or milk) is to be modeled. The tanker arrives at a station to be filled with the liquid. This takes some time, where the load of the tanker is changing continuously. However, the model is not very interesting during this time, since nothing else happens but the loading of the tanker. Thus, if loading time is known, one can 'forget' the change of state during this time, leaping to the time when the tank is filled up and the tanker starts its voyage to its destination. Also, during the transport the state of the system changes continuously, since the location of the tanker changes. However, again the analyst ist not interested in knowing the tanker to be at time t at location s. Thus, the next time of interest is the arrival time of the tanker at its destination station. This next <u>event</u> (arrival at station 2) is the time when the model should be inspected again, and the state of the model is changed (i.e. the tanker is now at its destination).



The Model of a Tanker

It is obvious from the last example, that a model is always a simplification of the real system. The analyst considers only values of interest, and characteristics of no importance to the model are ignored. We call a model **finer** than another model, if the first model contains more details of the state or of the change of the state than the second. Thus Einstein's model of the movement of planets is finer than the corresponding Newtonian model. Finer models usually represented the real world better, however, they are also more complicated to be build and they are also much more error prone. Also, they usually increase execution time of computer programs severly, so that also computers can model systems only down to some granulation, which depends on the time the analyst has to wait for results.

If time changes discretely, we call this a **time discrete** model. These are the most common models in computer simulation, where two cases can be distinguished:

1. Time is changed in small **constant step**s to model continuously changing variables (like the movement of planets or other models that can be represented by differential equations).

2. Time is changed in steps depending on the next event of the model. This is also called **event driven** simulation.

If the model used is able to compute the change of the event during some time period, the event driven method is very accurate. E.g. if there is a body moving constantly on a straight line, then its location can be computed exactly by simple calculations. Here, only the state at the beginning and at the end of the movement is of interest, which can be modeled by the event driven method. This simple type of behaviour was assumed in the last example, as well. Since there are many interesting problems that can be adequately modeled by the time discrete, event driven method, this simulation approach has become very common.

II.2 Problems of Computer Simulation

Although computer simulation is very useful, there are also some problems with this method all of which are not yet solved completely. We mention only some of them: Random events, statistics, and concepts of modelling of complex systems on computers.

II.2.1 Random Events

The first problem is that of random events that are observed in real world phenomena. Strictly speaking, random events never occur, since their might be always a cause for any effect. However, in practical life, not all causes can be kept under sufficient control to get a simple, managable model, which is an important aim of modelling technique.

Example: Customers arriving at a cashier desk do not arrive randomly, since anybody is doing this willingly, being quite aware of going now to the counter. However, the clerk behind the counter sees the people approaching according to a plan, he will never be able to understand. Thus it might be a good assumption that the people arrive randomly, where only some parameters must be considered, like the interarrival time between two customers.

Thus the analyst must be able to model the behaviour of some system, using random generating techniques. Also, he should know which type of process is feasible under a given assumption. Little knowledge in this area is usually the source of many poor models, and even of wrong results.

To understand this technique, some knowledge in probability theory is required, which to present here is beyond the scope of this text. Thus we only can mention the corresponding results and assumptions, leaving the unexperienced reader alone with his ignorance.

One possible way of generating random numbers is to use a discrete counter, that is incremented continuously, where incrementation speed or even the increment depends on some parameters that cannot be influenced, e.g. the temperature outside, the speed and direction of winds and so on. This method might produce some numbers that can be assumed to be randomly distributed, however, there are some disadvantages with this method.

The first problem is that these numbers are usually not really independent, since during small intervals of time the parameters do not change very much, so that several probes taken in this period with the same rate have almost the same distant between two values. Another problem is that such sequences of numbers cannot be reproduced, which is in many circumstances required. A further problem is the physical input into the system, which is at least uncomfortable, in some situations even impossible to achieve. Thus this method is usually not taken.

Another method is to compute random numbers. This seems to be a contradiction, since computation yields always predestined results, while random numbers are to be generated. However, this idea works very well in real applications. Let r be a number between 0 and 1: 0²r<1. Then r'=r*a+c is another number, and if a>1, c>0 then we can take the ciphers of r' below the decimal dot, which yields another number between 0 and 1. This method, applied to a memory cell, can be shown to generate numbers, which are very good distributed randomly over the interval [0,1). It is called **congruential random number generator** and is used for the most random number generators within computer systems. For us it is important to see that this method can prodruce **pseudorandom numbers** of high quality.

Many other random number generators have been suggested, but none has been proved to be so good as this technique. Thus we will not consider other techniques, since they do not yield better results.

If we have random numbers we can compute from them other random values. The pseudorandom numbers generated by the congruential random number generator are distributed equally over the intervall [0,1). We call a function in C **Random()** that generates a real random value from this interval. Sometimes, equal distributions over an arbitrary interval [a,b] is required, where the borders can be included (in case of integer results) or not. There are simple mappings from the pseudorandom numbers into such an interval, e.g. for real numbers (called float in C):

```
float a,b;
Equal_float = a + Random()*(b-a)
```

and for integer numbers:

```
int a,b;
Equal_Int  = a + Random()*(b-c+1)
```

(In C, float values are truncated to integer values.) Other important distributions are called memoryless distributions. Here, the quantities of two succeeding values are independent in a stochastic sense. There are only two such distributions, one for discrete values, the other for continuous values. In case of discrete values we assume that there are intervals of equal size, where an event occurs in such an interval or it doesn't. We ask for the number of intervals where no events occur between two events. The corresponding distribution is sometimes called **geometric** distribution. If each interval is considered to be an experiment, then this can succeed (i.e. an event occurs) or fail (i.e. no event occurs). If p is the probability that an event occurs, then

$$p_n = p*(1-p)^{n-1}$$

is the probability that in n-1 intervals no event occurs and in the n-th an event occurs. Summing p_n over all n from 1 to k we get:

$$P_k = \sum_{n=1}^{k} p_n = p* \sum_{n=1}^{k} (1-p)^{n-1} = p* \frac{1 - (1-p)^k}{1-(1-p)} = 1 - (1-p)^k$$

Here, P_k is the probability that a value between 1 and k is attained. Solving for k yields:

$$k = \frac{\log(1-P_k)}{\log(1-p)} + 1$$

If P_k is a pseudorandom number Random() from [0,1), then 0²1-Random()<p is to be mapped on n=1, p²1-Random()<1-$(1-p)^2$ is to be mapped on n=2, etc.; thus all other intervalls are to be mapped on

$$n = \left[\frac{\log(1-Random())}{\log(1-p)} \right] + 1$$

where [x] is the largest integer smaller or equal to x. We compute '1-Random()' instead of 'Random()', since the former function attains values > 0 only so that the logarithm is always defined. The mean number of steps can be computed by the sum:

$$E[Geo] = \sum_{n=1}^{\infty} n*p_n = p* \sum_{n=1}^{\infty} n*(1-p)^{n-1} = p*\frac{1-p}{p^2} = \frac{1-p}{p}$$

Thus we can use the mean number of intervals, instead of the probability:

$$n = \left[\frac{\log(1-Random())}{\log(mean/(1+mean))} \right] + 1$$

The memoryless distribution for continuous time problems can be derived by the **negativ exponential** distribution, the distribution function of which is given by:

$$P[Arrival \leq t] = \begin{cases} 0 & \text{for } t < 0 \\ 1 - e^{-\lambda * t} & \text{for } t \geq 0 \end{cases}$$

which gives the probability that the time between two events is less than or equal to t, where λ is the arrival rate (i.e. $1/\lambda$ is the mean interarrival time). This can be solved for t, yielding:

$$t = - \frac{1}{\lambda} * \ln (1-Random())$$

Here, we have taken 1-Random() instead of Random(). The former yields always a value>0, while the latter can attain the value 0, for which the logarithm is not defined.

The negativ exponential distribution has only one parameter, so when the mean is determined, all other parameters are fixed, the most important of which the **variance**. This is a measure for the differences between the values of a probe and their mean. If the variance is 0, then all values of a probe are the same as the mean. The **standard deviation** is the positive square root of the variance. The variance of the negative exponential distribution is always the square of the mean, so the standard deviation of the negative exponential distribution is the absolute value of the mean. If distributions with lower deviations are needed, one can use the **Erlang distribution**. This has a second parameter k, which is of type integer, so that the variance is reduced by the factor 1/k. The function can be computed by:

$$Erl\ (Mean, Steps) = -\frac{Mean}{Steps} * \ln \left(\prod_{i=1}^{Steps} Random() \right)$$

where the product means the product over Steps random values. For the normal distributon we can use an approximation:

$$\text{Norm (Mean},\sigma) = \sqrt{-2*\mathbf{ln}(1-\text{Random}())}*\mathbf{sin}(2*\pi*\text{Random}())*\sigma + \text{Mean}$$

where σ is the standard deviation. Thus, mean and standard deviation are the only parameters of the normal distribution.

Other distribution can be introduced, if required. We do not list them here in all detail. It is of more interest, however, to see which distribution a particular process has. This depends on the type of the corresponding system. The following hints should be helpful to decide, which distributions is the most adequate one in a particular situation:

Equal Distribution

If all events have the same proability, e.g. dying, throwing a coin.

Geometric, NegExp Distribution

If events occur independently, the time between two events should be distributed with the negative exponential distribution or in case of discrete time with the geometric distribution.

Erlang Distribution

If the delay can be considered to be the sum of several NegExp Distributions

Normal Distribution

If the values of the distribution vary about a mean with the same magnitude up and down

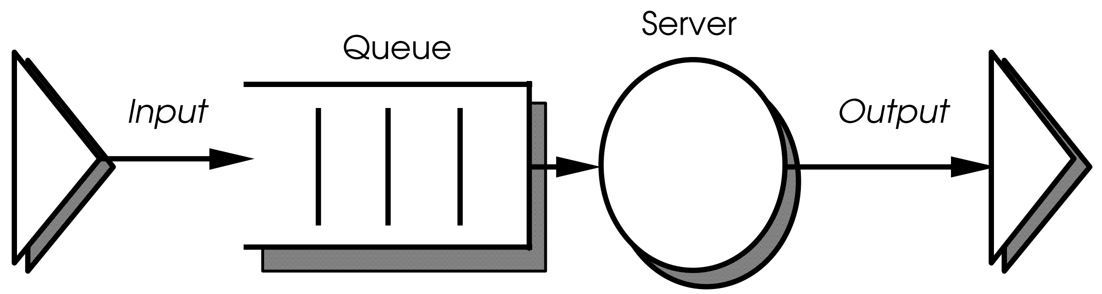
Also, there are situations where no standard distributions can be assumed. In this case the distribution can often be approximated by a piecewise linear distribution, stored in an array in the memory. We will not consider this technique here more closely.

The following considerations are necessary to determine the sequence of random numbers. Usually, we start with an initial value for the congruential random number generator, lets say x_1; this starting value is also called the **seed** of the random number generator. If we proceed, we get the next value, i.e. x_2, x_3, … ,x_n. Since all these values are computed, the first number x_1 determines the sequence of all other values. Thus, if the first number of the congruential random number generator is always put to the same value, then we always get the same sequence of random numbers. This has the advantage that the sequence of random numbers can be repeated as often as required. This is useful for testing of simulation programs as well as in some techniques used for variance reduction.

Another question with sequence numbers is the following: Usually, concurrent processes are modelled, i.e. depending on unforeseeable sequences of events in concurrent systems, random numbers are generated. It is possible to draw the random numbers for all concurrent systems using one seed and variable, or for each process another seed and another variable. Thus the random numbers can be completely independent, or they can (at least in principle) depend on each others. The analyst should be aware, i.e. that in some cases the sequence of all random numbers are changed, if one parameter in only one process is altered, while in other systems the sequence of variables is identical for processes where parameters are not changed.

II.2.2 Output Analysis

Having run a simulation, results occur, usually as sequence of numbers. To be more concrete, let us assume a queueing system is analyzed, where the waiting time and the number of queueing places is to be estimated. A queueing system can be sketched as:



A Model of a Single Server Queueing System

where the following parameters are of interest:

Arrival rate: λ	Gives the average number of jobs that arrived at the queue within a unit of time; measured in jobs per time. $1/\lambda$ is the mean service time.
Service rate: μ	Gives the rate of service per job, measured in jobs per time. $1/\mu$ is the mean service time.

Service discpline	Describes the sequence in which waiting jobs are serverd, e.g. FCFS (first come first serve), SJN (shortest job next) etc.

Having run the system with such parameters the analyst can measure the time each job has to wait, and from this the maximum or mean number of waiting jobs etc. The first measure considers the quality of service for the customers (i.e. there statisfaction), the second the number of required waiting places.

Analysing these quantities can be difficult. The questions we have to consider are:

- How long should a simulation run be?
- Can we estimate the level of confidence of our results?
- Is the mean of a quantity sufficient, or do we need more elaborate quantities, e.g. variances or even distributions?
- Do we get all possible information from our (usually very expensive) simulation runs?

Here, many questions can be answered by assistance of statistical techniques. Thus the analyst should have also some knowledge of these methods. We will give an overview here.

Statistics is a technique to derive knowledge about a set A by observing a subset $B \subset A$. From some combinatorical considerations follows that a subset can be considered to look almost alike the superset, provided the subset is choosen independently. Thus, the distribution of values in the subset can be considered to be similar to the distribution of values in the superset. Thus we have to assume that our observation of values in the subset B is an independent probe of the values to be estimated. If this holds, statistical techniques can be used to determine quantities like mean, variance, or the distribution of random variables, by computing these quantities using the values of the subset and giving some confidence intervals.

Although this can be considered to be a classical statistical technique, there are some special considerations; since the values derived are computed by use of a synthetical 'real world', there are some advantages, since the probes can be very large, and very easily be manipulated. Simulation experiments can be prolonged or repeated, they can be determined (or **controlled**) in a way usually not possible with real systems. We will take use of some of these advantages.

To look somewhat more closely on output analysis, we consider the following conditions:

1) There is a model of a system to be analysed
2) There are some parameters of the model
3) There is an initial state of the model
4) There is an initial random seed

We assume that there is a simulation model of a real world system. Since the following considerations are very general, we do not restrict this model in any way.

The **parameters** of the model, e.g. the arrival rate or the mean service time, have to be given. If they are the same, different simulation runs will be called **repetitions** or **batches**. If they are unlike, different simulation runs will be called **comparisons**.

The **initial state** can be the same or different for different simulation runs. Usually, the initial phase of a system is of minor interest, so that the simulation might start at an arbitrary state, e.g. a non-empty queue with some randomly determined customers waiting. However, to be able to compare different simulation runs, it would be wise to start alway from the same initial state.

The **initial random seed** are the initial values used by the congruential random number generators. If two independent simulation runs where to be made, the initial random seed should be different; here, the last value of the first could be the first value of the second run, etc. which guarantees the best independence of two random seeds. We call this **independent** simulation runs.

In some other cases, it would be useful to supply the same initial random seeds; e.g. if two arrival processes should be identical (to compare only different service disciplines) this can be achieved by use of twice the same initial random seed for the arrival process. In some cases however, the random seed should be **antithetic**, which means that instead of x_n we use $1-x_n$. Here, two values are highly negativly correlated, which is required for some applications. If the same or antithetic random numbers are used in different simulation runs, we call this **controlled** simulation runs.

The quantities measured are called **time series**. They can be the waiting time $\{w_1,w_2,\ldots w_n\}$ at a queue or the number of sold newspapers over a year: $\{s_1,s_2,\ldots s_n\}$. In the second case, two succeeding values of the time series might be independent, while in the first case, they should be dependent: If a job waits for a long time, then the job served after this one will wait for a long time as well. Thus time series can be **dependent** or **independent**.

Time series can be considered to be random variables with some given distribution. They might have some parameters like **expectation** (or **mean**) and **variance**, or **median**, or **most probable value**, etc. Another quantity is the **cumulative distribution function** or the **density** of the corresponding distribution. This can be computed by using **histograms**.

To estimate the mean, one uses normally the arithmetical mean, which is in some theoretical sense the best estimation for the expectation:

$$W = \frac{1}{n} * \sum_{i=1}^{n} w1$$

For the variance, the correct estimator is:

$$s^2 = \frac{1}{n-1} * \sum_{i=1}^{n} (w_i - W)^2$$

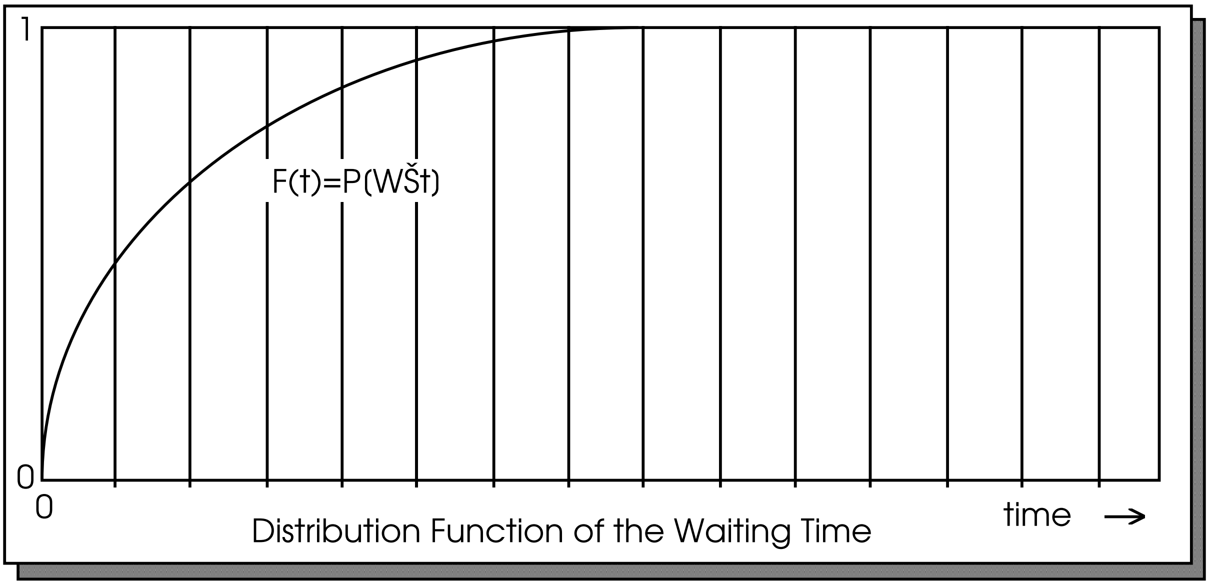
Computation of the median can be done only, if all n values of the time sequence are stored and sorted; then the median is either the value at (n+1)/2, if n is odd, or it is any value between (n/2 … n/2+1).

Let H be a histogram, i.e. an array of length H[0…K+1]. For each value w_i of the time series, H[w_i] is incremented by 1 (if w_i<1, then H[0] is incremented; if w_i>K, then H[K+1] is incremented). After the simulation run, compute H/n (i.e. divide each H[i]:=H[i]/n), which contains an estimation of the density function of the distribution of the time series $\{w_1, w_2, \ldots w_n\}$. If we define

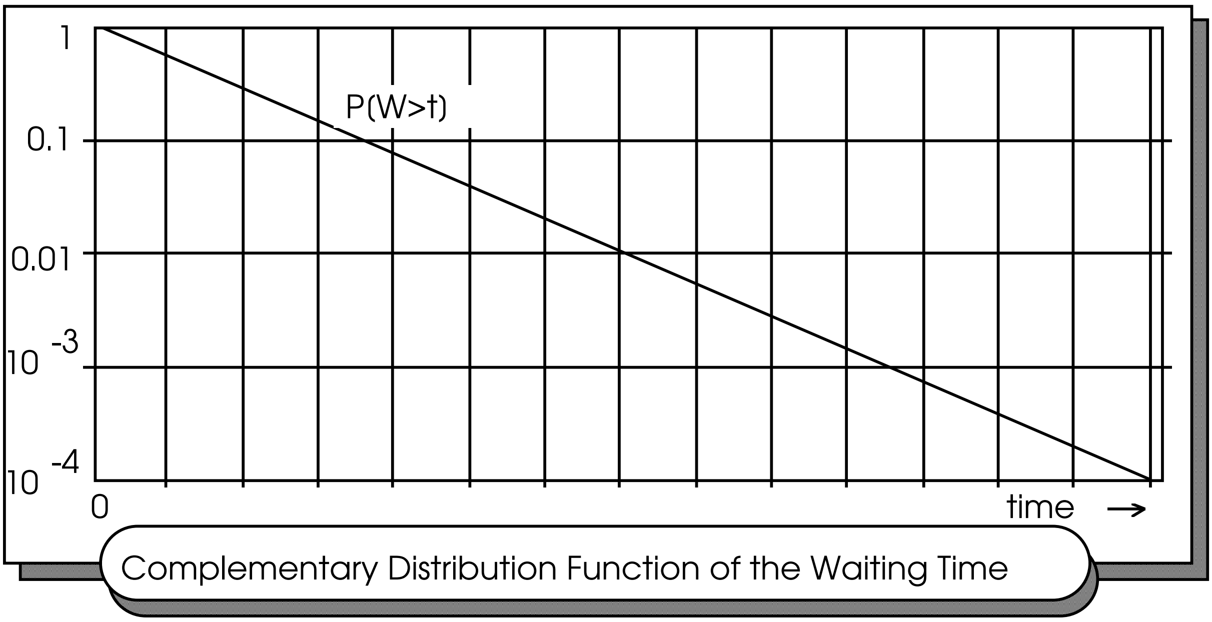
$$W[k] = \sum_{i=1}^{k} H[i]$$

then W[k] is an estimation of the cumulative distribution function of the time series $\{w_1, w_2, \ldots w_n\}$.

Distribution functions give the probability that a quantity is smaller or equal to a given value; e.g. we write $F[W^2t]$ for the probability that the random variable W attains a value that is equal to or less than t. Thus a plot of this function is monotoneously increasing (or constant), but never decreasing:



The disadvantage of this presentation is that many variables are usually interesting only close to the probablity 1. E.g. the last plot might display the probability that the waiting time of a customer is less than t. To compute the number of waiting places required to have a loss probability that is less than a given value (e.g. 0.999 or 'one of thousand'), one has to inspect the distribution close to one. However, the resolution of the plot is very poor in that range. A logarithmic representation would not help directly. However, if we use the presentation of 1-P[W²t]=P[W>t], then a logarithmic representation would be very useful:



This is also called the **complementary distribution function**. In many cases their logarithmic plot is almost linear.

Some random variables model a function of time instead of a sequence, e.g. N_t is the number of waiting customers at time t. The mean of this random variable is usual estimated by:

$$N = \frac{1}{T} * \int_0^T N_t \ dt$$

which can be computed by a computer by use of the sum:

$$N = \frac{1}{T} * \sum_{i=0}^{n-1} N_i*(t_{i+1} - t_i)$$

if N changes its value at time t_i the i-th time and attains at time t_i the value N_i. The variance will then be estimated by:

$$s^2 = \frac{1}{T} * \sum_{i=0}^{n-1} N_i^2*(t_{i+1} - t_i)$$

To compute the density and cumulative distribution function, instead of 1, the time $(t_{i+1} - t_i)$ is to be added to the entry $H[N_i]$. The rest remains unchanged.

To estimate a confidence interval, several (here m) independent repetitions can be done, and each parameter estimated can be assumed to be independent and normally distributed. Then by use of estimators of mean and variance and use of the t-distribution, it is possible to compute a confidence interval for the estimated quantity. If $F(c)=\frac{1}{2}*(1+\gamma)$, where F is the t-distribution with m-1 degrees of freedom, then with probability γ the true mean μ lies within the interval:

$$W - \frac{s*c}{\sqrt{m}} \leq \mu \leq W + \frac{s*c}{\sqrt{m}}$$

Here, W is the estimated mean, and s^2 is the estimated variance of the quantity to be observed. This method can also be applied to compute a confidence interval for histograms.

From some further considerations follow that repetitions with the antithetic random numbers produce quantities with less variance than repetitions with independent random numbers. Although, this does not hold in all cases, it might help to decrease the confidence interval computed above. However, this method needs a much better understanding of the basic statistical ideas, so we will not further work out these ideas.

Other topics of interest in this area will be mentioned here only with no further development.

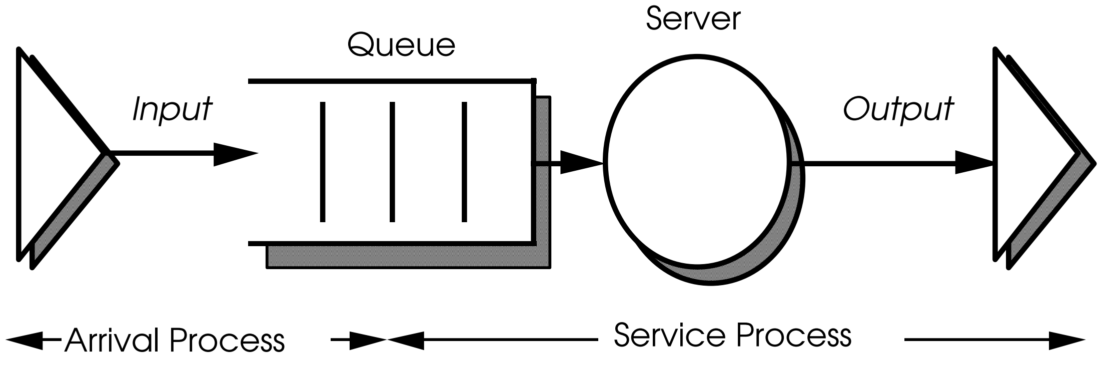
In time series, two succeeding values might be dependent (see the waiting time example from above). This dependence can be analyzed, using techniques like **autoregression** and **spectral analysis**. Here, it can be possible to adapt a parametrical model to the time series.

If parameters are changed, the behaviour of the model will change as well. The question of interest is the magnitude of this change. This is sometimes called **sensitivity analysis**, since the dependence of the output from the input parameters is analysed.

Other problems occur if the parameters of the model are to be changed so that a given quantity becomes an optimum. Here, the corresponding quantity is usually a random variable. Thus, classical optimization techniques are not always feasible.

II.2.3 Programming Simulation Models

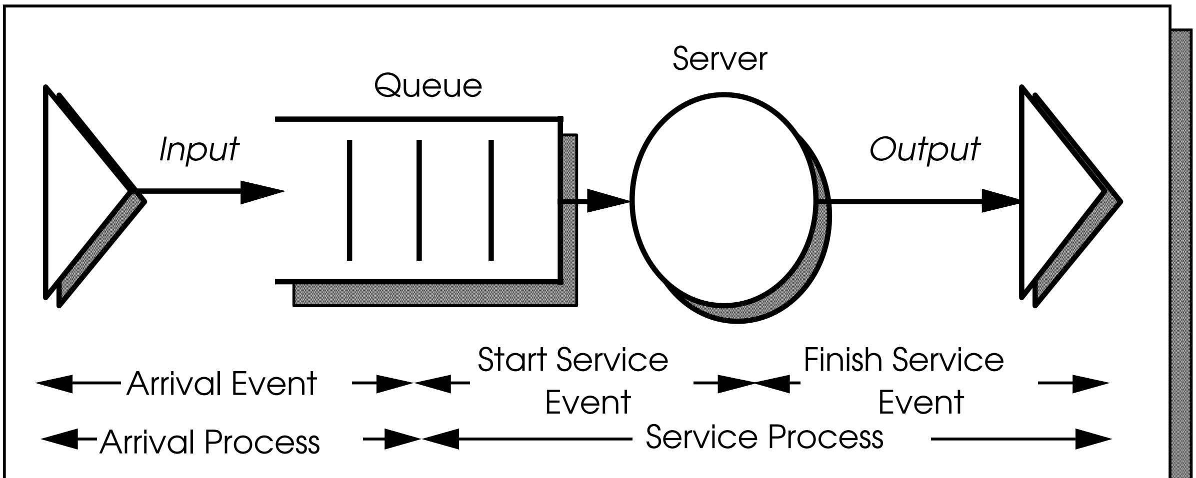
In the example of the last chapter, the model consists of two independent processes. The first one generates jobs and puts them into a queue. The second process gets a waiting job out of the queue and puts it into service. If service is finished, the job is disposed somewhere else:



Model of a Single Server Queueing System with two Processes

Thus we can consider this system to consist of two independent **processes**. This is a simple and very well structured view. However, in many simulation concepts, other models for implementation of simulation programs are choosen. We consider some of them in this chapter.

Instead of processes, all events that change the state of the system can be considered. Here, everything that can change the state of the model is called an **event**. In the example above there are three events: The arrival of a job in the `Queue` changes the state of the queue. The selection of a job from the `Queue` changes both, the state of the `Queue` and the state of the server. Finishing service and disposing the job finally changes the state of the server:



<u>Model of a Single Server Queueing System with three Events</u>

Common programming languages do not have structures like processes or events. Thus, programming of systems of this type in a general purpose programming language like Pascal requires a solution of the following tasks:

1) Model the System Time
2) Model the state of the Queue
3) Model the state of the Server
4) Model the generator of jobs, i.e. the time when the next job arrives
5) Model the service of jobs, i.e. the time when the next job is served
6) Write procedures that change the states and the system times

Usually, the <u>System Time</u> can be modelled by a simple real variable `SystemTime` (we use `Courier` type letters for program text). However, for sake of accuracy, the system time should have highest possible resolution, such a double precision real variable (as possible in some Pascal dialects) or the double float variable (as possible in C) would be a good choice here. An integer variable is not sufficient!

The state of the <u>Queue</u> can be modelled by use of an integer variable `QueueCount`, counting the number of jobs in the queue. Its range is the set of all nonnegative integers: {0, 1, 2, …}

The state of the <u>Server</u> is simply a boolean called: `Busy`. If it is true, the server holds a job. If `Busy=false`, the server is idle, i.e. holds no job.

The job generator can be modelled by a time variable `GenTime`, holding the next time when a job is to be fed into the queue. The type `GenTime` should be the same as that of `SystemTime`.

The job service can be modelled by a time variable `ServiceTime`, holding the time when the job in service is finished. The type of `ServiceTime` should be the same as that of `SystemTime`.

Putting all these variables together, it is possible to model the system above in a straight forward manner:

```
program Simulation(input,output);
type
   EventType = (Arrive, Serve);
   TimeType = real;
var
   GenTime, ServiceTime, SystemTime: TimeType;
   QueueCount: integer;
   Busy: boolean;
   NextEvent: EventType;
begin
   SystemTime:=0;
   QueueCount:=0;
   Busy:=false;
   GenTime:=NegExp(Mean_Arr);
   ServiceTime:=0;
   while SystemTime <= 1000 do begin
      if Busy and (GenTime>ServiceTime) then begin
         NextEvent:=Serve;
         SystemTime:=ServiceTime;
      end else begin
         NextEvent:=Arrive;
         SystemTime:=GenTime;
      end;
      if NextEvent = Serve then begin
         if QueueCount >0 then begin
            Busy:=True;
            QueueCount := QueueCount -1;
         ServiceTime := SystemTime + Equal(Serv_1, Serv_2);
         end else Busy:=false;
      end else if NextEvent = Arrive then begin
            Busy :=True;
            QueueCount := QueueCount +1;
            GenTime:= SystemTime + NegExp(Mean_Arr);
      end;
   end;
end.
```

Although we think that this program is very well structured, there are many dependences between the different sections of the program. For example, when `NextEvent=Serve`, then we can assume that `SystemTime=ServiceTime`. However, if we write

```
ServiceTime := ServiceTime + Equal(Serv_1, Serv_2);
```

this could lead to wrong results, since under some other circumstances, the value of `ServiceTime` is not defined. It takes some time before the reader of this program has completely understood what happens in which situation. It is almost impossible to extent this model, e.g. to a two server model. Usually, a 'redesign' is made, which simply means that the program is constructed completely new and reimplemented.

Since the system starts each loop by scanning all situations and looks for the next 'activity', this method is sometimes called: **Activity scanning approach**. Since it is nothing else but elementary programming, we do not find any advantage in this method. The disadvantages are the complex relationships between different parts of the programs that cannot be kept under control for really complex models. It is important to keep data together that belong together.

We will now consider a model that encapsulates at least all events. The SIMSCRIPT model is sometimes called **event oriented**. Events are implemented by using some structures similar to a subroutine in FORTRAN (the syntax of SIMSCRIPT is unusual and will not be used here). Each event can schedule another event at a given time by putting this into a corresponding schedule table. Here, all active events are sorted according to their scheduled time. In the above example, there are three events:

```
Event Arrival:
   JobType Job;
        Create new Job
        Put this into JobQueue
        If Start_Service is passivated then activate Start_Service
        Schedule this event after NegExp(Mean_Arr)
EndEvent Arrival:

Event Start_Service:
   JobType Job;
        If JobQueue is empty then passivate this event.
        Get a Job out of JobQueue
        Schedule Finish_Service after Equal(Serv_1, Serv_2) (Job)
EndEvent Start_Service:

Event Finish_Service(Job):
   JobType Job;
        Dispose Job
        Schedule Start_Service now
EndEvent Finish_Service(Job):

Main
   JobQueueType JobQueue
        Create a new JobQueue
        Schedule Arrival after NegExp(Mean_Arr)
```

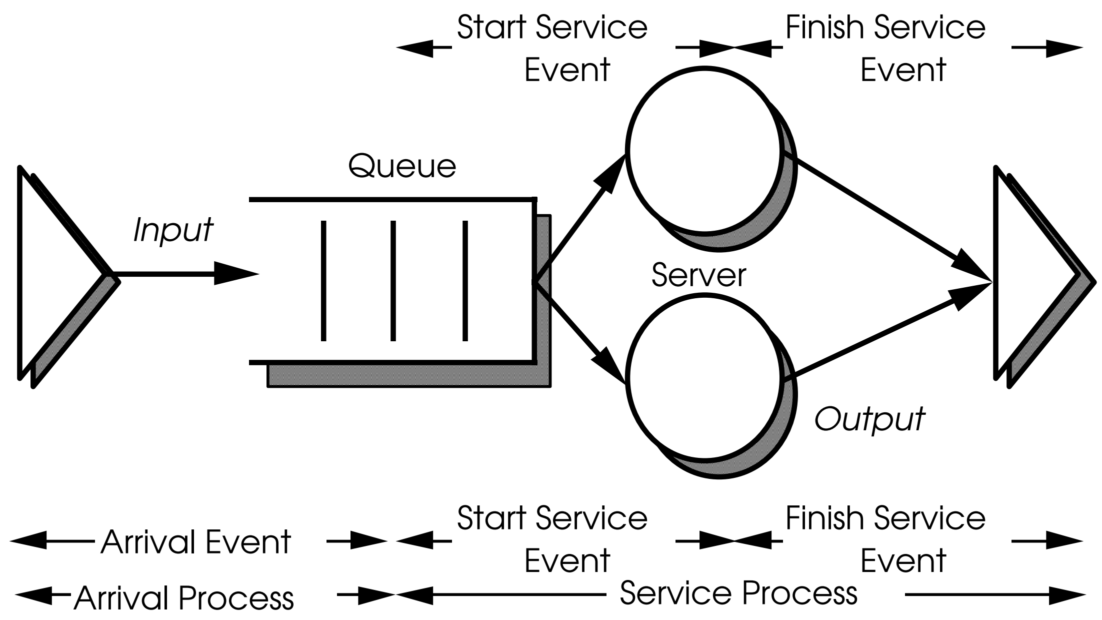
We have omitted evaluations of statistical results. This type of systems looks very clear. However, the main problem is that there is no structure in the model. Each event can schedule an arbitrary other event, independent of any logical structure. Also, each event must explicitly have a name of another event that is to be scheduled. Although this is very simple in this small example, in complex systems, where many similar events are to be scheduled, this can be very confusing. Thus the next step should be to collect the events within more complex structures that display better the natural sequence of events:

```
Process Arrival:
   JobType Job;
   Init
         Next after NegExp(Mean_Arr)
   Event
         Create new Job
         Put this into a JobQueue
         if Server is idle then activate Server
         Repeat after NegExp(Mean_Arr)
EndProcess Arrival

Process Service:
   JobType Job;
   Event
         If JobQueue is empty, then passivate
         Get a Job out of JobQueue
         Next after Equal(Serv_1, Serv_2)
   Event
         Dispose Job
         restart now
EndProcess Service

Main
   JobQueueType JobQueue
         Create a new JobQueue
         New Process Arrival
         Server := New Process Service
```

Although only little has been changed in this program, we see that there is a process `Arrival` which enters jobs into a `JobQueue`, and that there is a process `Service` that gets jobs out of that `JobQueue` and serves them. The two events of the process `Service` are executed one after the other, and if the last event has been finished, the first one is restarted again. In the process `Arrival` only one event occurs continuously. The statement after `Init` is executed only once at the time when this process is created.

Also, the parameter `Job` in the process `Service` is now local to the process, i.e. it can be accessed by both events. Thus it is no longer necessary to transmit this object as a parameter as it was necessary in the former example.

One problem with models of real systems is their variability: How easy is it to change a model. For the example above, it might be required to model a system with two servers instead of one, being fed by the same `JobQueue`. In our model it is very easy to create another server, sharing the task of serving the queue. This can be done by creating a new service processs that must be activated by the `Arrival` process as well. A plot of the model shows the two servers that get jobs out of one queue:



Model of a Single Server Queueing System with three Events

To write this model as a program we use the following technique:

```
Process Arrival:
   JobType Job;
   Init
        Next after NegExp(Mean_Arr)
   Event
        Create new Job
        Put this into a JobQueue
        activate all Service
        Repeat after NegExp(Mean_Arr)
EndProcess Arrival

Process Service:
   JobType Job;
   Event
        If JobQueue is empty, then passivate
        Get a Job out of JobQueue
        Next after Equal(Serv_1, Serv_2)
   Event
        Dispose Job
        restart now
EndProcess Service

Main
   JobQueueType JobQueue
        Create a new JobQueue
        New Process Arrival
        New Process Service
        New Process Service
```

Here, no reference to a process `Service` exists. Instead, all processes of type `Service` are activated by the process `Arrival`. If there is no passivated process, this statement has no effect. Nothing is changed within the declaration of the process `Service`.

Since both, processes and events are modelled, we call this **process-event method**.

A similar method has been introduced by **SIMULA 67**, using a <u>class</u> instead of a process as we have defined here. In that language, each 'process' can be <u>detached</u>; the corresponding process is then sorted into an <u>Eventlist</u> at a given time, and the next event to be executed is always the event with the smallest time in the <u>Eventlist</u>. Thus, if we remove the keywords `Event` within a process and use instead of them only a statement that delays the process (here <u>Hold</u>), we get a SIMULA program.

```
Process Class Arrival;
    JobType Job;
    begin
           Hold( NegExp(Mean_Arr));
           while true do begin
                  Create new Job
                  Put this into a JobQueue
                  if Server is idle then Activate Server;
                  Hold( NegExp(Mean_Arr) )
           end;
  End Process Arrival;

Process Class Service:
    JobType Job;
    begin
           while true do begin
                  if JobQueue is empty then Passivate;
                  Get a Job out of JobQueue
                  Hold( Equal(Serv_1, Serv_2) );
                  Dispose Job
           end;
  End Process Service;

begin
    JobQueueType JobQueue;
    Ref(Service) Server;
           Create a New JobQueue
           New Process Arrival
           Server := New Process Service
end;
```

We see that this is almost the same as the process-event method, described above. However, in this programming style we cannot see, where events occur; also the infinite loops have to be programmed explicitly; finally, the class concept (**`Process Class`** `Service` means that the class `Service` has the standard class **`Process`** as a superclass), which is used in the example above, is probably only hard to understand by people not aquainted with object oriented programming styles.

Thus we think that the process-event method is better structured than the SIMULA concept. However, SIMULA is a programming language on its own, having introduced the concept of classes and so the object oriented programming style already 25 years ago.

The next step to structure the processes would be to forget about the events and consider the processes alone. Using a syntax similar to GPSS, one can write:

```
Simulate
Generate     Job   all NegExp(Arr_Time)     ! Job arrives
Queue        1                              ! … enters Queue
Seize        1                              ! … goes to Server
Depart       1                              ! … leaves Queue
Advance      Equal(Serv_1, Serv_2)          ! … is served
Release      1                              ! … leaves Server
Terminate    1                              ! … is disposed
```

Here, the flow of the program is considered to be the flow of the jobs through the system. **Generate** produces a new job (called **transaction** in GPSS). This transaction enters the queue number 1; then it tries to **Seize** the server 1. If this is busy, the transaction has to wait in a queue inf front of server 1. Otherwise it **Depart**s from the queue 1 and is delayed (**Advance**) for a random time, the service time. After this time the server 1 is **Release**d. To dispose the transcation, **Terminate** is to be executed.

If we want to build a two server system in GPSS, the second server is to be implemented explicitly, i.e. we have to write:

```
    Simulate
    Generate          Job   all NegExp(Arr_Time)    ! Job arrives
    Queue       1                                 ! … enters Queue
    Transfer          0.5, S1 , S2                      ! … select Queue
S1  Seize       1                                 ! … goes to Server
    Depart            1                                 ! … leaves Queue
    Advance           Equal(Serv_1, Serv_2)             ! … is served
    Release           1                                 ! … leaves Server
    Terminate         1                                 ! … is disposed
S2  Seize       2                                 ! … goes to Server
    Depart            2                                 ! … leaves Queue
    Advance           Equal(Serv_1, Serv_2)             ! … is served
    Release           2                                 ! … leaves Server
    Terminate         2                                 ! … is disposed
```

The first disadvantage we see here is that the system cannot be extended to a variable number of servers (or other processes). Each process must be explictly plotted in the program text, and also processes behaving in exactly the same way do need completely new program text. This makes it difficult to see, whether different servers are the same or some slight differences are to be considered.

Also, each server must have its own number (which is the only way to distinguish two objects in GPSS). This makes it difficult to understand the system. There are however other severe disadvantages. GPSS is a 'closed' system, i.e. nothing else can be added to the functionality besides those GPSS-Blocks already predefined (there are about 50 or so, which is a rather big number of key words compared to a common computer language). Thus, the queue disciplines are limited to predefined service sequences, also the evaluation of results takes places automatically, so that only standard parameters can be determined. Thus the system is not flexible.

One possible way to ease these disadvantages is to implement GPSS in a common computer language, where GPSS-Fortran is one example of (other languages were also used, such as PL/1 or Basic). Here, each block is a procedure call. Extensions can be made by implementing them into Fortran. However, to do this, the control of the transaction must be understood by the programmer. This is, however, rather complicate, since transactions are the same as the jobs, so that the service discipline and the time control are to be manipulated together. To make tiny changes, the complete system is to be redesigned. Thus, this method is not applicable by an average analyst or even programmer.

There is however an advantage of the transaction method: The flow of objects through the model is written explicitely in the program. In the process-event method, this flow is modelled by special instructions, sending jobs from one process (or server) to a queue, which is emptied by another process. Thus we extent the process-event method by two constructions. The first is to define standard processes, consisting of a standard JobQueue and the non-standard events:

```
Process Service:
   Enter JobQueue;        /* Declaration of a queue called JobQueue */
   JobType Job;
   Event
         Job := await( JobQueue );
         Next after Equal(Serv_1, Serv_2)
   Event
         Return Job;
         Restart now;
EndProcess Service
```

To insert a job into this queue, the calling process executes:

```
         Enter Job into Server;
```

where `Server` is a reference to a process of type `Service`. Activation and Passivation of the process `Server` is done automatically. The second extension is the introduction of a **sequence process**:

```
Sequence Process Manufacture:
   JobType Job;
   Job := NewJob;
   Enter Job into Server1;
label1:
   Enter Job into Server2;
   Enter Job into Server3;
   if (Job.something_wrong) Next label1;
   Enter Job into Server4;
   …
   Enter Job into Server2;
   Enter Job into Server9;
   Dispose Job;
EndSequence Process Manufacture:
```

Here, the `Servers` are processes holding `JobQueues`. A job that is entered into a `Server` is there manipulated, and then it is `Returned` to the sequence process. Thus, the way of the jobs through a 'job shop' can be modelled easily and in a very obvious was. By the way, each incarnation of a sequence process controls only one job, so this sequence process is to be disposed, if the corresponding job leaves the model.

Conclusions

In this section, we have analysed some methods how to implement simulation programs in a structured way. From the arguments given here follows that the Process-Event method is most advantageous. It is well structured, however, it is also flexible and easy to maintain, alter, extent etc. In the next part, we introduce a simulation environment on basis of the C programming language using the process-event method.

Part III

The Simulation Language C-SIM

This part of this report introduces the programming language C-SIM, which is a comfortable tool to implement a process-event oriented simulation model. In the first section we give a sum up of the requirements needed to implement that language. In the second section we give an informal description of the language C-SIM, and in the third one some examples are displayed and explained.

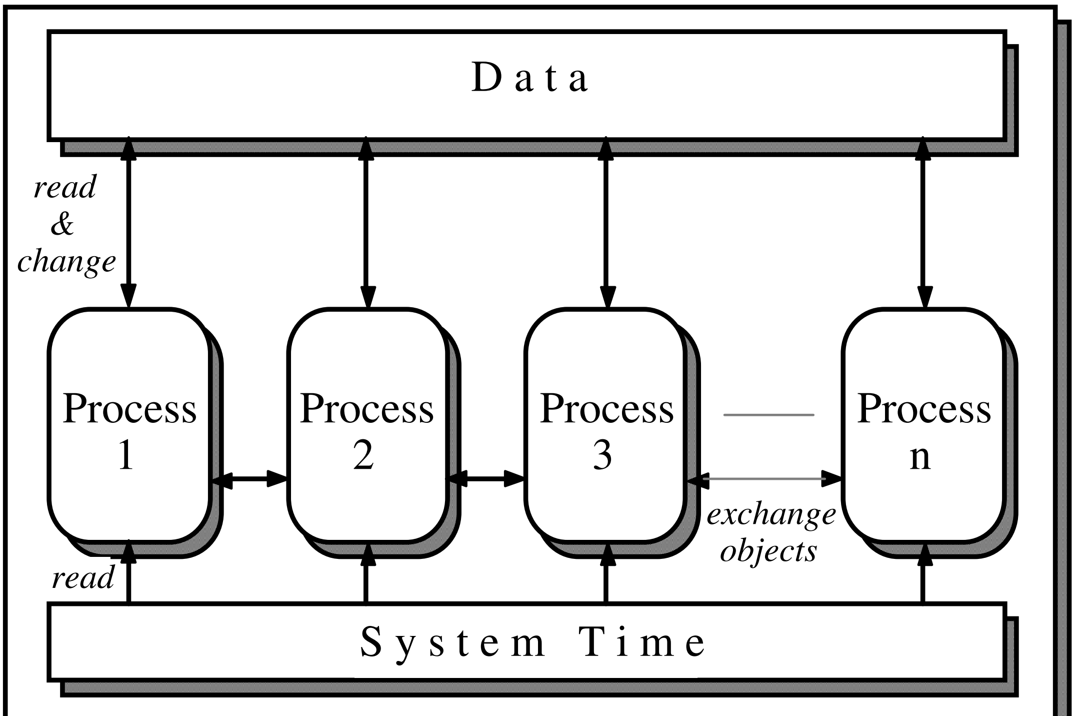
III.1 Requirements of a Process-Event Simulation Language

III.1.1 Data and Processes in C-SIM

A model is assumed to be a representation of a real system. The (interesting) phenomena of the real system are to be represented within the model, using an adequate data structure. Those data structures should be hidden to all processes not dependent on this, however, should be accessed easily by processes that have to use it. This can be done easily by a record (called **structure** in C) which is nothing else but a collection of some data. This feature of C is not changed by C-SIM.

Data structures are usually interpreted in a given way. For example, an integer (in C) can be interpreted as a character or a boolean (and even as an integer). But an integer representing a number can in some circumstances represent the number of queueing places, in others a colour or a planet. Thus, it would be fine if quantities can only be interpreted in a defined way, which means also that they can be altered only in a distinct, well understood way. This is, however, not possible in C; languages that control this are MODULA 2 or Ada, etc. In C-SIM the programmer can protect the access to quantities only within processes or their environments, and so restrict unauthorized change of private quantities.

We understand a model as a collection of data, representing the state of the model, and a collection of **processes**, describing how those data, i.e. the state of the model, is changed. While data are a static part of the model, processes are the dynamic one. The following figure should help to understand this concept:



Dynamic Model of a Real World System

Here we see the data that represent the state of the system, and the processes that observe and change the state of the system. The behaviour of the processes depends on the the system state as well as on the System Time that is changed permanently. The processes exchange information by use of messages, usually hold in records we call jobs. However, in other applications theses objects might be called messages, signals, customers or others. Also, for the processes other names are common, like server or generator.

Thus, there are active instances in such a model, the processes, and passive instances, the jobs. The state of the system (i.e. the global data) is a passive instance, as well, which is accessed by all processes, while the System Time can be considered to be an active instance. Since the structures in C are merely a collection of data, we assume that all processes interprete the data correctly.

The next instance of interest is the process. We consider a process to be a sequence of events, where each event is changing the state of the system. There is a general sequence, in which events are executed. The change of the state by execution of an event does not take any System Time, while however between the execution of two events of the same process some System Time can pass on; this time is called **delay time**. This is an important fact. In the liquid transport example from above, the tanker was filled during some time, however, the model considered only the point of time when the loading started and the point when the loading was finished. Intermediate time was ignored. Usually, the time when the next event occurs must be explicitely declared by the previous event, and also the next event is to be selected by the previous one, if there is a choice. However, there is an exception from this rule that will be mentioned later.

To understand the following better, we define some new notions. A process can be **active**, i.e. being waiting for a time when the next event of the process can occur. This is the general situation within processes as we have introduced them. If a process waits for something else but time, we call it waiting for a condition, then we say the process is **waiting**. For example, if a process wants to resume only if a job is to be served, it can wait for the condition that the corresponding queue is not empty. A process is called **passive**, if it is waiting for another process to (re)start it. Thus neither a time, nor a general condition on the state of the model makes it active, but only the explicite activation by another process can do this. In the last example, a process sending a job to a queue, can activate the corresponding server of the queue. The following table gives an overview of these situations:

Next execution of an event of the process depends on:	Process is called
time	active
restart by another process	passive
general condition	waiting
next execution is impossible	finished or killed

The last situation occurs when a process is not able to be executed anymore. If the data record of the process is disposed, this process is called **killed**; otherwise we use the notion **finished**. These concepts are sustained by C-SIM.

III.1.2 Events in C-SIM

We are coming now to the events, which are the smallest active objects in the process-event simulation method. Each event can change the state of the system, however, for sake of clarity, events should alter only those variables that are either local to the corresponding process, or that can be manipulated only by a small number of processes. Otherwise, the resulting effect may be completely opaque. We consider this point later in this section more closely.

An event is called **e-active** (which means event-active), if it will be executed the next time the process is executed. I.e. an event can be active, if the corresponding process is active; or if the process is passive, however, after its activation the corresponding event will be executed (before any other event of this process); or if a process is waiting, and if the waiting condition becomes true, then at first this event is executed. Thus, the notion e-active can mean something else but the notion active for processes. Careful distinction is necessary here.

If an event is executed and the process remains <u>active</u>, then the event schedules another event; this includes two activities: Select the next event and determine the time when this event is to be executed. Thus, the selected event becomes e-active, and its **activation time** becomes the time when this event is to be executed. The notion activation time can also be used for processes and means then the next time, when an event of this processes will be executed.

If the process becomes <u>passive</u> after execution of an event, then also the next event is to be choosen. This event will be the event the process executes when it is activated again; thus we call it the e-active event. However, no time can be selected, since activation depends on other processes, only.

If the process becomes <u>waiting</u> after execution of an event, also the next e-active event is to be selected. In C-SIM, we use the following concept: The corresponding event computes the condition whether the process has to wait. If it has to wait, the process is put in a 'wait-list' that is always executed before any other nonwaiting process is executed. Thus, if a non-waiting process changes the state of the model, there is a chance that the waiting process can resume. Thus it performs the <u>same</u> event again, which means it computes the condition another time, and when it is now changed, i.e. the wait condition is false, then the <u>same</u> event is resumed. Thus, after a 'wait-statement' the next e-active event should always be the same event that has executed the wait-statement. This means, however, that neither the next event nor a scheduled time can be determined for a waiting process.

if process becomes …	**then** select …
… active	… next event & activation time
… passive	… next event
… waiting	… nothing

There is an exception from the concept of an e-active event. In C-SIM we have defined a preempt-event. This event is executed always when a job is sent to the process by another process, even if another event of this process is e-active. Thus, it is possible to model the situation that because of a particular condition a process must interrupt the current activity and performs some other actions. Classical modeling languages like SIMULA do not sustain this concept. When the process performs a '**Resume**-statement' the interrupted event is continued, which means that the process will be delayed the residual delay time of the event that was e-active before the interruption, and then this event will be executed. This concept can be used recursively, i.e. also interruptions can be interrupted again.

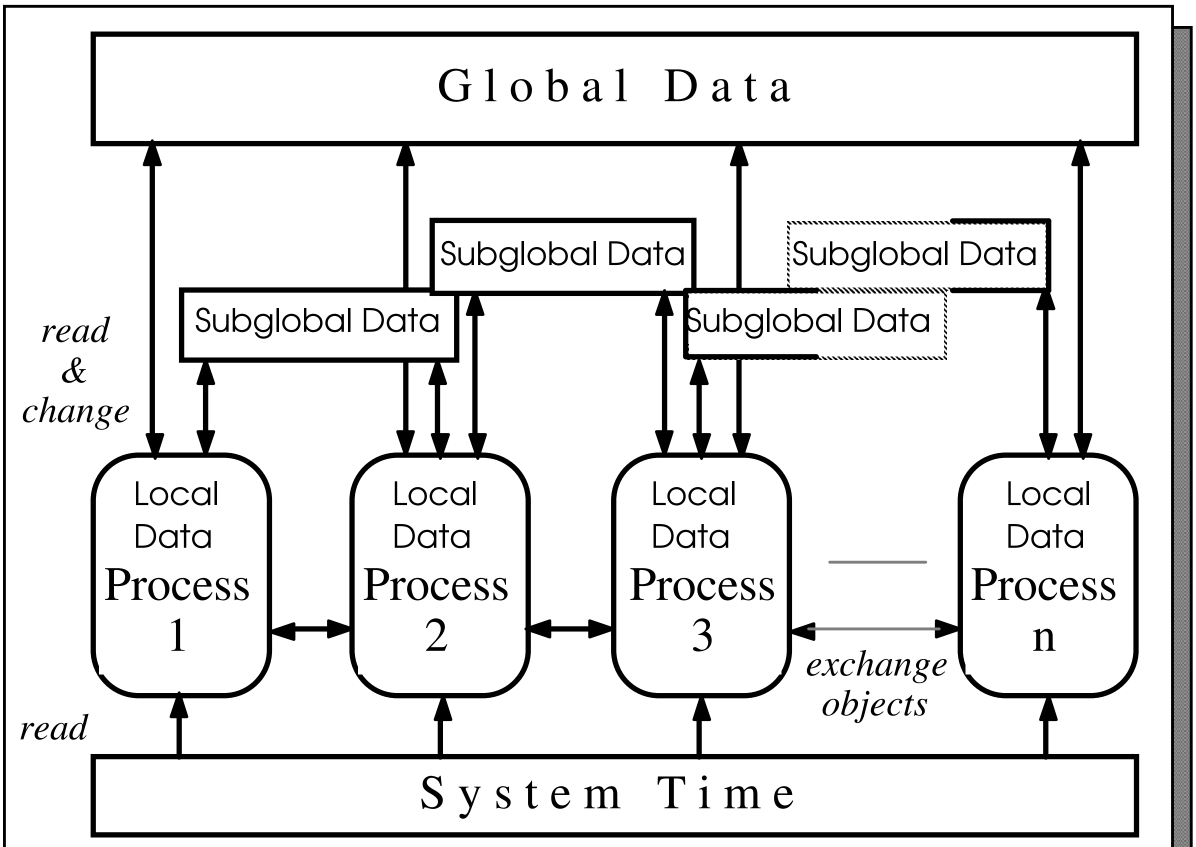
III.1.3 Subglobal Data in C-SIM

We will now continue the discussion about locality of data that are changed by events and start with an example:

Example: A variable counting the number of jobs that have passed through a server should be local to the corresponding server. Thus all servers could have such a counter and it is (almost) impossible of another process to overwrite the corresponding counter variable of a server. This improves the reliable of such systems.

In other cases, global variables must be shared by different processes. For example, a queue can be fed by several processes. Here it is important that the processes know the queues in which they have to insert their jobs. Although this is simple for tiny models, it can be a complex task for medium size or big models to keep control of some tens or hundreds of queues that get jobs from some processes and send them to other processes. To ease this problem, the following concept has been introduced in C-SIM:

A variable can be **local** to a process, it can be **subglobal** to a well defined set of processes, and it can be **global**, which means that all processes can access this variable. The next diagram should visualize this concept:



Subglobal Data in a Simul ation Model

The reader can see from this diagram that the local data of the processes can be accessed only by the corresponding processes. Other processes should never use them, they should not even be interested in them, so that no legal reference to them should exist. Subglobal data can be accessed by the corresponding processes these data are subglobal to. Global data can be accessed by all processes. This intermediate stage 'subglobal' has been introduced to achieve higher reliability. While in the programming language C variables are either local or global, we get with this concept variables that are global only to some particularly selected processes, and to all other they are local (i.e. outside of their scope). Since one process can have several disjunct sets of subglobal variables, communication between processes can be very well structured.

There are two matters to be considered with this concept, the first is the syntactical description, and the second one is the implementation. Since our simulation method is to be translated into C, we have to find a method to implement these concepts in that programming language. For the syntactical problem we suggest the following notation:

```
Process Service;
subglobal QueueType InQueue, QutQueue;
subglobal int Counter;
…

Process Service Server1, Server2;
…
(Server1->OutQueue,
 Server2->InQueue)   = New Queue(Length=20);
(Service1->Counter,
 Service2->Counter)   = int;
```

Before we can explain this notation, we have to mention that in C-SIM a process is not the instance of an object but only a description of such an object. This corresponds to the class concept of SIMULA and the pointer concepts of Pascal, i.e. the declaration `Process Service Server1, Server2;` yields pointers (or references in SIMULA) to objects of type `Process Service`. To get the corresponding record of a variable, one has to initiate that process by calling the function: `New Process()`. The return value of this function is a pointer to a record in the memory, with enough space to store the corresponding local data of this process.

Also, if we call the function `New Queue` in C-SIM, we get a reference to a record that holds a queue of jobs. This reference is now stored inside the processes Server1 and Server2. Both servers hold the same object, however, they know this by locally different names. Thus, if `Server1` sends a job into its `OutQueue`, `Server2` observes that its `InQueue` is altered. In the second example, an integer is used by two processes in common. The meaning of this statement is that both processes share the same variable of type integer, using this time also the same name for it.

The subglobal concept is a dynamic feature, i.e. both processes do not need to have a global name of the variables used in common, nor do they need to know anything about the number of subglobal variables in the program. Thus the following loop generates an arbitrary number of processes, using subglobal queues to communicate:

```
{ Process Service Server1, Server2;

  Server1  = New Process(Service);

  for (i=2; i<=N; i++) {
   Server2 = New Process(Service);
          (Server1 ->OutQueue,
           Server2 ->InQueue)   = New Queue(Length=20);
          Server1 = Server2;
     }
 }
```

This program fragment generates N processes. All these N processes are linked together by N-1 queues. While the processes can all access their input and output queues using only <u>one name</u> as if they were <u>global</u>, the names of these queues are not known anywhere outside these processes. This concept helps to simplify programming, since the programmer has not to 'invent' N-1 different names for the output queues, and inside of the processes no different names for the queues have to be used. An alternative method were to define a global array that can hold all queues, which is, however, not only cumbersome, it is also unsafe, since any other process might try to access one of these queues (and also, the array concept of C is not very secure).

Because of this dynamic structure, it is not possible to implement subglobal variables in C using global variables, which might be used only by some processes (and this would be checked by a preprocessor). Thus we have to introduce a record for all such subglobal variables, and to transfer a pointer to this record when defining a subglobal variable. Inside the process the compiler has to solve this by recognizing, that all subglobal variables are called by reference. The example above programmed in C would look like:

```
{ struct ProcessService Server1, Server2;

  Server1  = New Process(Service);

  for (i=2; i<=N; i++) {
   Server2  = New Process(Service);
          AuxQueue = New Queue(Length=20);
          Server1  -> OutQueue = AuxQueue;
          Server2  -> InQueue  = AuxQueue;
          Server1  = Server2;
     }
 }
```

and for an integer variable we have to write instead of:

```
(Server1 -> Counter,
 Server2 -> Counter)   = int;
```

the following:

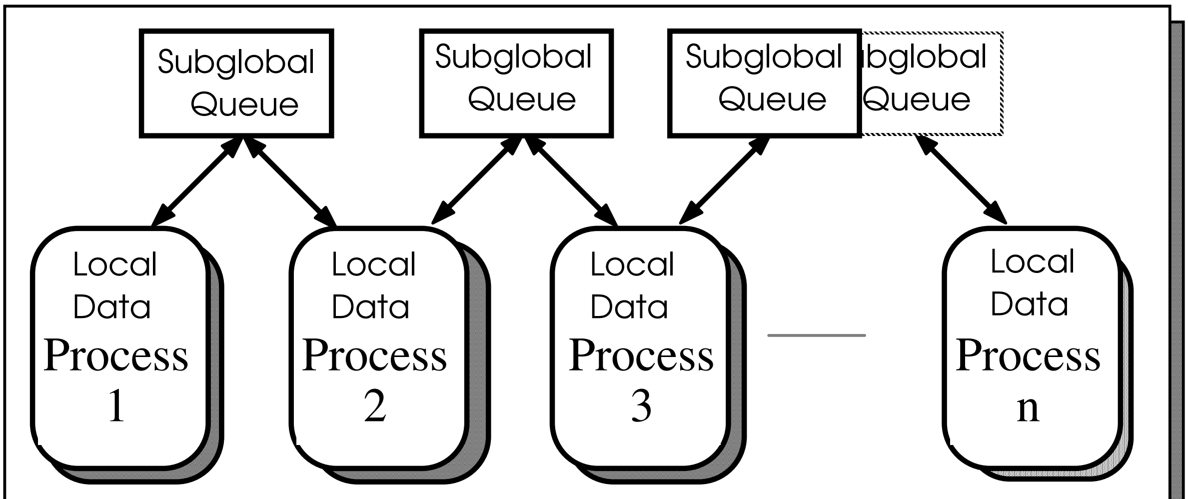
```
struct IntRef (int Value);
AuxInt = (struct IntRef *) malloc((size(struct IntRef));
Server1 -> Counter = AuxInt;
Server2 -> Counter = AuxInt;
```

Inside a process nothing is changed for a reference type such as the reference to a queue. However, for a value type such as an integer, we have to substitute a simple increment: `Counter += 1;` by:

```
Counter -> value += 1;
```

Thus, such variables are to be treated as reference variables in programming languages like Pascal or Ada. We see that this concept can be very simply implemented also in C. However, usually the programmers do not learn to use such techniqes, and so we think that it is useful to introduce global variables in a simulation system, to present a new technique for structuring of programs (particularly parallel programs) and also since otherwise such complex models cannot be treated safely and easily.

For the example above, we can plot the following figure for the structure of the processes and their subglobal variables:

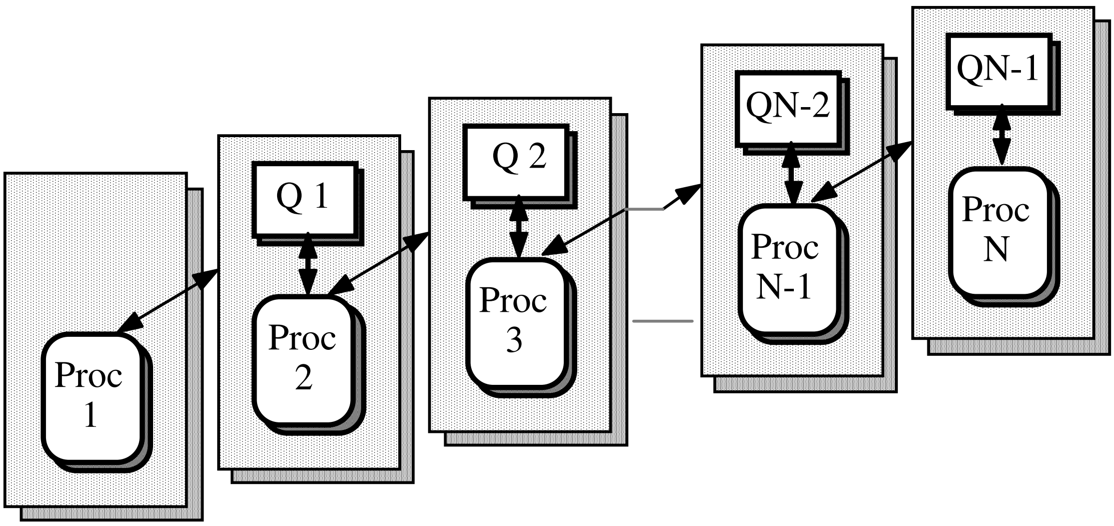


Subglobal Queues in a Sim ulation Model

This figure shows that all processes are on the same block level (if we would speak in a block level language, which C is not). However, a process has to be activated by another process, if its input queue gets another job. This can be done by use of the subglobal concept in the following way:

```
Process Service;
subglobal QueueType InQueue, QutQueue;
subglobal Service NextProcess;
…
{ Service Server1, Server2;
  Server1  = New Process(Service);
  for (i=2; i<=N; i++) {
   Server2  = New Process(Service);
         (Server1->OutQueue,
          Server2->InQueue)     = New Queue(Length=20);
         (Server1->NextProcess) = Server2;
         Server1 = Server2;
     }
 }
```

To understand this example better, we also plot the figure of the subglobal relationships, where an object connected to a process by a double arrow is defined to be a subglobal object of that process; objects confined by a dotted area are common subglobal variables to another process:



Subglobal Queues and Proc   esses in a C-SIM-Model

Thus we see that Q1 as well as Proc 2 are subglobal to Proc 1, while Q1, Q2 and Proc 3 are subglobal to Proc 2, etc. Thus, the processes are not all at the same block level, since different processes have different subglobal variables. There can exist an arbitrary structure of relations between variables; thus it is better not to consider this as a hierarchical struture, but as a network structure. It is also possible that a process 1 is subglobal to process 2, while process 2 is subglobal to process 1:

```
Process Service;
subglobal Service NextProcess;
…
{ Service Server1, Server2;
  Server1 = New Process(Service);
  Server2 = New Process(Service);
     (Server1->NextProcess) = Server2;
     (Server2->NextProcess) = Server1;
      }
 }
```

This can be extended to arbitrary cyclical relationships. This is, of course, more complicate to manage, however, closer to real world systems, which usually are not hierarchical.

Some further comments on the concepts of subglobal variables could be made. However, since this report is not about programming concepts, we will write about this somewhere else.

So far we have analysed requirements of an event-process oriented method concerned with the data structure and the dynamic facilities. The next important features are functionality. We have to describe, which functions for generation of random numbers, for statistical analysation and others are to be supplied by a simulation system.

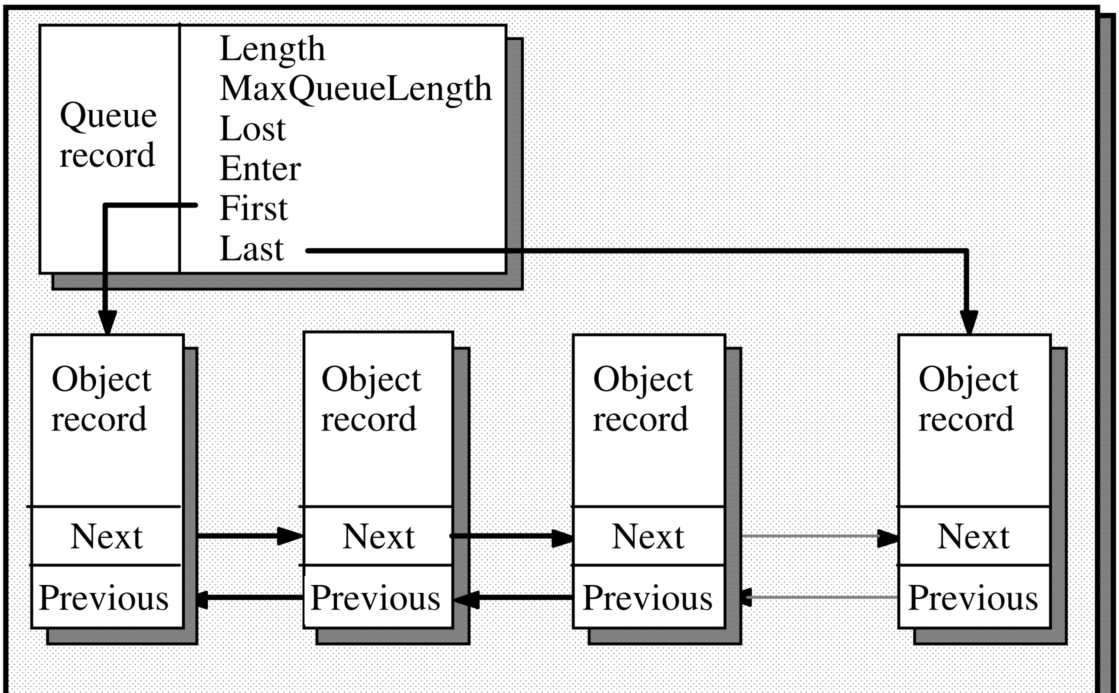
Random number generators and random distributions will be supplied as described in section II.2.1. Statistical analysation will be made by evalutation of moments and histograms, using batch techniques for determining intervals of confidence (see II.2.2). Other functions are those for management of queues and for graphic.

Queues are linear lists used to store jobs or processes. In C-SIM, different types of jobs can be defined, which can all be stored in one type of queue, called `JobQueue`. To store processes, queues of type `ProcessQueue` are supplied.

Queues have the property to store and to retrieve an object. Thus, a queue needs a method to insert an object into a given queue. There is one reason why a queue should not accept a new object: When it is filled up. We allow that a queue can have a finite length by generate them with the statement:

```
JobQueue = New Queue(Length = <number>);
ProcessQueue = New ProcessQueue(Length = <number>);
```

Here, `number` must be a constant expression, resulting in a nonnegative integer value. If this is omitted, the length of the queue is assumed to be infinite. The record of queue is the following:



Structure of a Queue in C-SIM

/If an object is to be inserted into a `Queue`, we use the function:

```
bool Insert(<queue ref>, <object ref> [, <strategy>  ] );
```

In this declaration, `<queue ref>` is a reference to a queue, and `<object ref>` is a reference to an object. `Strategy` is a parameter that can be omitted. If no parameter is given, then the object is inserted as the last one in the queue. This strategy is also called FIFO (for First In First Out) or FCFS (for Fist Come First Serve). Other possible strategies are: LIFO (Last In First Out), PRIO (Priority) and RANDOM: The following list describes the functionality of the corresponding strategies:

Strategy	Meaning	Insert at location:	Remove from location:
FIFO, FCFS	First in first out	last	first
LIFO, LCFS	Last in first out	first	last
PRIO	Priority	Behind first with higher priority	First with highest priority
RAND	Random	After Random()*Length	Random()*Length

Here, `Length` is the number of objects in the queue before the function is called. The function call returns false, if the object could not be inserted into the Queue, otherwise true.

To insert the object with a priority, it must have a local field called `Priority` which is of type integer; the higher the number in this field, the higher the priority of the corresponding object. We recommend to insert an object with a priority and remove it with FIFO, since these are the more efficient operations.

If an object is to be removed from a queue, the function:

```
<object type> Remove(<queue ref> [, <strategy> ] );
```

is to be used.This function delivers a reference to the corresponding object, provided there is one. Otherwise, the return value is `Null` (the pointer to nothing; its C-value is `Null=0`).

The functions `Insert` and `Remove` do not change the record of the corresponding object. They just change some pointers within the objects and within the Queue. Thus, an object can never be contained within two different lists, since it has not the corresponding pointers available. If a job is to be disposed, there is a special statement for this in C (`free(Job)`).

If another strategy is to be implemented by the subscribers this is sustained by C-SIM in the following way. Within a particular part of the program the programmer can write a new strategy by using the pointers within the objects, i.e. `Next` and `Previous`, which are references to other objects. The queue itself holds the pointers `First` and `Last`, which references to the first and to the last object of the queue, respectively. Also, there is a variable called `Length`, which counts the number of objects in the queue (it should never be altered by the programmer, but only read). Both, the `Previous` pointer of the first element as well as the `Next` pointer of the last element points to `Null`. A strategy inserting an object before the last one in the queue could look like this:

```
Before_last (Insert):
{     if (Length <= 1) Append(Null);
      else Append(Last -> Previous);
}
```

Here, a function `Append(Object)` defined locally is used to append an object behind the parameter of the function; if this parameter is `Null`; i.e. `Append(Null)` or `Append( )`, then the object is inserted as first object into the queue. The object that is transferred as parameter is referenced by the name: `This`. The following example removes the object before the last one from the queue (if there is any):

```
Before_last (Remove):
{     if (Length <= 1) return(Null);
      else return(Remove_This(Last -> Previous));
}
```

The function `Remove_This` returns the referenced object, unlinking it properly from the other objects in the queue.

There are also some functions to get information of the state of the queue. `Length(Queue)` delivers the number of objects in the queue, `Empty(Queue)` delivers true, if there is no object in the queue; `Full(Queue)` delivers true, if there is no space for another object in the queue. The maximum number of objects in the queue can be asked for by use of the variable: `Queue->MaxQueueLength`. There are two other variables in the queue record; `Enter` counts the number of objects that have entered the queue (including those which are lost); `Lost` counts the number of objects that were lost because the queue was full. Both variables should not be altered by the programmer.

III.1.5 Graphics in C-SIM

The graphic concepts of C-SIM have been developed to present the behaviour of the system in an easy manageable way, i.e. on a graphic screen. This can help the programmer to detect errors and to convince other people that the program is free of errors. Graphic is also useful to demonstrate a system that has been analysed. So this is a very useful tool. However, there are also some complications with graphics, the most severe one is its complexity.

Because of these complications, graphics should be used in a very disciplined way; only then it can be helpful to understand a model and to analyse it with scientific methods. We list some hints, the programmer should follow, if he applies the graphics of C-SIM:

1) Use only functions supplied by C-SIM. Avoid to apply any procedure of the basic graphic system, otherwise the software cannot be ported to other hardware

2) Use standard functions of C-SIM, i.e. do not plot your own symbols or icons; otherwise C-SIM cannot assist you to switch graphic on and off, as required.

3) Use graphic confinements, where you use your own graphics.

4) Avoid to plot more than necessary to understand the model. Too many graphics on the screen might confuse the user, instead to inform him.

5) Use animation, where necessary, but avoid too many flickering symbols on the screen.

6) Switch graphics off, if you run the simulation model for performance measures. Graphics always decrease execution speed to a minimum.

7) If no graphics is needed anymore, remove it from the program. Graphic programs are usually twice as long as other programs.

8) If any problems arise with a simulation, try at first to solve it by animating the program graphically. Usually, testing of parallel programs with non-graphical output is impossible.

The concept of graphics in C-SIM is to supply the user by a set of graphical functions, which allow him to plot the model he is running on a screen, animating the moving of objects between the processes, visualize the change of the state of the system, and display immediately statistics. Thus a process is plotted as a firm unit, while the objects are transferred between these units. Some special symbols are choosen for sources of objects, sinks and queues.

Graphics is to be controlled by the simulation program. Thus, the main problem is to program graphics in the simulation program in a comfortable way. We have added the following features for this:

A **Table** can be used to display where the objects are to be put on the screen and which connections exist between them. Thus, it is very simple to create the layout of a plot. The following example shows this more closely:

```
Table:
      |      Headerline of this Simulation Run
      |
      |            @1          @2          @3
100   |
      |      @10   @11         @12         @13         @14
      |
      |            @21         @22         @23
      |
      |      Simulation Time @101   Average Load @102
20    |
---------------------------------------------------
10             300
```

After the keyword `Table` a description of the size of the graphics and the objects in the table, referenced by numbers, is given. The height of the table is plotted by the bars | on the left hand side and the number of pixels (here 100), its width by the dashes — and the number of pixels in that direction (here 300). The other two figures give the locaction of the graphics within the output window: 20 from the left border, 10 from the lower border. Within the program the objects instantiated have to reference the number `n` when to be placed at location `@n.` If a statical model is to be displayed, one can use an instantiation board, describing the class of the objects and their interconnection:

<to be continued>

III.2 The Syntax of C-SIM

This section starts the description of the complete syntax of C-SIM. C-SIM uses C as a basic language, however, with many extensions, functional as well as conceptional. Thus, the C-SIM language should also influence the programming style and extend the programming concepts of modern programming languages. While the concept of subglobal variables has already been introduced, the concepts of parallel programming will also add something to the knowledge of the current programming techniqe.

III.2.1 The Syntax of a C-SIM Program

C-SIM is based on C; so the syntax of C-SIM contains the programming language C as a subset. Unless constructions of C-SIM are applied, particularly outside processes, C can be used as usual; also functions can be defined which however may not contain any parts of the event switching features of C-SIM.

A simulation program is defined within the `main()` program. It starts by listing the parameters, needed within the respective C-SIM simulation program, followed by some declarations of variables and C-SIM objects, including declaration of processes. The statements of the program are then needed to instantiate the objects of the simulation systems, which are usually the processes, and if required some queues, initial jobs etc.; then the simulation is started by use of the statement: `Simulate for 10000;`. Having finished simulation, the residual statements of the program are executed and the program is finished. Thus, we have the following scheme of a simulation program in C-SIM:

```
main()
{ Simulation <name> Include <parameter-list>
  <global parameter-list>
  <process list>
  <declaration-list>
  <initial statement-list>
  Simulate [ for <number> ];
  <final statement-list>
}
```

Here, keywords of the C language are written in lower case letters (`main`), keywords of C-SIM are underlined and start with an upper case letter: `Simulate`, `FIFO`, or `EndProcess`. User defined variables are starting with upper case letters, as well, however are not underlined: `Server_1`, `Motor_Car`. Because of these conventions, it should be easy to understand examples of simulation programs.

In the first line: <u>Simulation</u> <name> <u>Include</u> <parameter-list> the entity <name> is an arbitrary but unique name in the C language. It is used to plot the name of the program within any output, to ease interpretation of the results. The second entity <parameter-list> is a list of C-SIM keywords, which control inclusion of functions to the simulation program. If the corresponding keywords are omitted, the functions are not included into the simulation system; also, guarded blocks that use the corresponding functions can be excluded in this way. The syntactical definition looks like:

```
{<paramter-name>:: <statement-list> }
```

An example is the following one:

```
{Graphics:: Move_Job(Station1,Station2); }
```

Here, an icon called `Job` is moved from `Station1` to `Station2` on the screen only, if the <u>Graphics</u>-parameter is set. Otherwise, this statement is ignored. Guarded blocks can appear anywhere in the simulation program.

The second line: <global parameter-list> is a list of parameters used in the simulation program. The most important parameter is an explicite declaration of the variables within a job. They can be declared by the syntactical expression:

```
{ JobType ::
<declaration-list>
}
```

where <declaration-list> is a list of types and variable names that can be accessed within a structure of type `JobType`. An example is:

```
{ JobType ::
  int JobNumber;
  float Load;
  TimeType CreationTime;
}
```

Since <u>JobType</u> is underlined, it is a keyword of the language C-SIM and cannot be used in another context within a C-SIM program. This structured type is also used to create <u>Queues</u>, which can store structures of type <u>JobType</u>. Further examples of parameters are special queueing disciplines or a graphics table, as defined in the previous section. `TimeType` is a C-SIM defined type and can be changed by the control parameter `TimeType` = <newtype>.

In the next line: <process list> is a formal description of the processes used within the simulation program. They can be considered to be types of processes, which still need instantiation. Processes are the central feature of the C-SIM simulation language, and thus they will be explained in the next section of this report.

`<declaration-list>` is the list of declarations of variables and so on, as they are used within common C programs. The type of the variables can also be processes, Jobs, JobQueues, etc, which are defined in the previous parts of this program. It is also feasible to use declarations that are globally defined, i.e. before the `main()` programm.

Followed by the declaration, the `<initial statement-list>` is used to initiate the variables and instantiate processes and queues; then the simulation run is started by calling the statement: `Simulate [ for <number> ];`. This starts simulation until the maximum simulation time, which is either set by the <number> in this statement, or wich can be explitely set by a statement like: `SimulationTime=10000`. This statement must have occured before the start of the simulation, and then `for <number>` can be omitted (the brackets [] are not a part of this statement). Otherwise `SimulationTime` is set to 0 (from the C compiler).

After simulation, the `<final statement-list>` is executed, which is again a list of statements from C. The program is finished by the closing '}'. C-SIM programs are translated by use of a preprocessor that maps the C-SIM declarations into a standard C-SIM program. This method allows flexible portation of programs between different computer systems, and still using highly optimized compilers. The disadvantages of this method are usually the problems by locating error reports. Since in C the standard preprocessor can handle this problem in a rather intelligent way, we are not intending to change this method.

III.2.2 The Syntax of C-SIM Processes

C-SIM processes are the central part of a C-SIM programming language. The system to be modeled is assumed to be mapped onto processes, which run almost independently, depending on time and some particular interferences between some processes at some time. Thus a process should confine everything that is local to it, where no other process is allowed to access these variables, or even constants or types. This might be compared to some other programming paradigma like functional programming, where the change of some variables is always local to a function (and can only be transferred to the 'outer world' by return values).

However, there are some important differences between functional programming and process oriented programming, since the latter entity have a 'memory', i.e. between two different interferences with the same process, this might have changed its state so that it behaves completely differently. Also, functional programming is used to compute results, never to change the state of a system. So a call to a function always implies that the calling entity has to wait for the result. Interference with a process, e.g. sending a job to a process, always means that the sending process can proceed with its execution, and accepting a result, e.g. the changed values of the job, some time later.

Of course, time is another important factor in simulation programs that does not occur in other programming paradigma; in C-SIM a virtual `SimulationTime` is introduced that is used to define the relative speed of different processes.

In C-SIM a process is a collection of events, where each event is a C program that can change the state of the system. Here, the state is a set of variables, which are either local to the process, or which are global to all process. In C-SIM, also the subglobal concept is used.

An example of a process has been given in the previous part of this report by the process declaration:

```
Process Service:
  Enter JobQueue;
     JobType Job;
    Event;
           If JobQueue is empty then passivate
           Job := await( JobQueue );
           Next after Equal(Serv_1, Serv_2)
    Event;
           Return Job;
           Restart now;
EndProcess Service
```

Thus, a process can be defined in C-SIM by the syntactical rules:

```
Process <process name>;
  <local parameter list>;
  <local declarations>
  [ <event indication> :
           <statement list>
           <selection statement>  ]+
EndProcess <process name>;
```

A process is enclosed by the keywords `Process` and `EndProcess`, followed by the name of the process, which is an arbitrary, but unique C name. This name can be considered to be a structure name, which however, is not preceded by the keyword `struct`. The general form of the declaration of a process is:

```
<process name> <name list>;
```

In C, any type can be defined by use of typedef. Thus, one might consider that a process in C-SIM has been defined by a definition of this kind:

```
typedef struct {<local declarations>} <process name>;
```

where the structure needs some more variables, which are however hidden to the programmer.

A process is instantiated by the function call:

```
New Process(<process name> [, <initial values>]* );
```

This function call returns a reference to the corresponding process of type <process name>, i.e. a record containing the variables defined in the <local parameter list>.

The <local parameter list> is a sequence of C-SIM keywords that can be used to extend the functionality of a process. An example is the **queueing system**: This can be created by use of the keyword Enter <queue name>;. If a job is send to this process by use of the Enter-function:

```
Enter (<job reference>,<server reference>);
```

then this job can be picked up from a queue called <queue name>. Although it is possible to use common queue functions like Remove(), it is better to apply the new function Await(<queue name>), the result of which is the next job in the queue. If the queue is empty, this process is passivated, being activated automatically if a new job is entered into its Enter queue.

Another example of a **local parameter** is preempt, which can be used to preempt the execution of an event. This will explained more closer when the preempt event is explained.

The list of <local declarations> defines variables used within this process. Their type is common to all processes of this type, however, their memory is different, so that they are local variables in the sense of a recursive function. Each instantiation of a process creates new memory for these variables. There are special types of variables possible here, the most important one is the subglobal type. As explained in the last chapter, a subglobal variable references an object outside this process, which can however been accessed by one or more other processes itself. Thus, there are some restrictions on this variable, the most important one is that its reference cannot be changed; the object must be assigned to these variables by use of the syntax:

```
([<process reference> -> <var name>,]*
  <process reference> -> <var name>) = <object reference>; |
                                        <simple type>;
```

Here, [<obj>]* means an arbitrary repetition of <obj>, and | means logical or. The syntax is to assign to a list of subglobal variables, enclosed in parantheses, either the object reference of the variable, or a type name of a simple type (like `int`, `float` or `char`). If the object is globally anonymous, i.e. there is no other reference to this object outside of the processes, it is impossible to access this object from anywhere at any time in the future. Thus, although this object can be used as a global communication mechanism between processes, it is unknown to all processes which are not included in the list above. This mechanism increases security of programming, and extends the notion of object oriented programming as well.

If a variable or a list of variables is initiated by a constant, these variables are initiated during instantiation of the process:

```
Process Server;
  int Counter = 0;
  int ProcessNumber = Init;
```

If a variable is initiated by the keyword `Init`, this variable gets its initial value by the corresponding quantity in the instantiation function `New Process`:

```
New Process(Server, ProcessNumber = 12 );
```

This instantiation is required, if `Init` is used here; if it is omitted, an error is recorded by the compiler. Also, all local variables can be reinitiated by use of a corresponding actual parameter, i.e. in the latter example, also

```
New Process(Server, Counter = -1 , ProcessNumber = 12);
```

is a legal instantiation of a process of type `Server`.

The next part of a process is a list of events, starting usually with at least one init-event:

```
Init :
         <statement list>
         <selection statement>
```

This event is executed exactly once at the beginning of the simulation, and after this it is never executed again (with perhaps one exeption). There can be more than one init event, and there can even be no init events, if this is not required.

After the list of init events a preempt event can follow. This event is executed when a preemption is made on this process by use of the preempt function:

```
preempt(<process reference>);
```

Then the event of the preempted process currently in execution is interrupted. This means, that the state of the process variables, as well as the next event and the residual daley time of this process until the next event is to occur, are stored in an extra preempt queue, not accessible by the programmer. The preempt event is then executed, including all other events activated by this event. If the function `resume(<discipline>)` is executed, one of all interrupted events (according to the `<discipline>`) is restored and its execution is resumed. Nothing is changed, unless the events executed during preemption have changed the state of the process or some global variables. A preempt event looks like:

```
Preempt :
  <variable list>;
         <statement list>
         <selection statement>
```

where the `<variable list>` is a list of local variables that are to be restored when the event currently in action is resumed. This list usually contains the job the process is executing at the time of interrupt; however, any local variable can become a member of this list. There can be only one preempt event (or none).

After these, the events are listed in the sequence of their common execution:

```
 [ Event [<event name>]:
         <statement list>
         <selection statement>  ]+
```

`<event name>` is the name of this event; it can and should be omitted, if it is not needed. It can be used as a reference, if an event that cannot be accessed otherwise, is to be activated. This feature is not often required.

The statements executed within the events are used to change the state of the system. They can access all variables that are local, subglobal or global to this process.

`<selection statement>` is a statement used to select the next event and the time when this is to occur. There must be at least one selection statement at the end of an event, however, there can be more than one selection statement, e.g. within an if-statement:

```
Event:
        if (value>=0.0) Next After 10;
        else Next After 20;
```

This event schedules the next event in the sequence after a time, depending on `value`. There are several possibilities to select an event. They are listed in the following table:

Keyword	Next active event
Next	if not last Event: Event following the e-active one if last Event: First Event (but not Init or Preempt)
First	The First Event (but not Init or Preempt)
Last	The last Event
Previous	if not first Event: Event immediately before e-active one if first Event: Last Event
Next <name>	Event with <name>
ReStart	The First Event (but not Init or Preempt) (same as First)
ReInit	The First Init Event
Repeat	The currently active Event (i.e. no change)

Here, we assume that there is a sequence of events, starting with `Init:`, followed by the first event, and finished by the last event (here we do not consider preempt events); there can also be named events.

```
Init:
        <statement list>
        <selection statement>
Event:              /* First Event */
        <statement list>
        <selection statement>
…
Event <name>:  /* Event with <name> */
        <statement list>
        <selection statement>
…
Event:              /* Last Event */
        <statement list>
        <selection statement>
```

The time when this event is to be activated again, can be determined by an absolute or by a relative quantity. We define:

Keyword	Time when event is activated
	<immediately>
After <time>	SystemTime + <time>
At <time>	<time>
Behind <proc ref>	Activation Time of process <proc ref>
Passivate()	Passivates this process

If the resulting time is lower than the current `SystemTime`, the current `SystemTime` is the next activation time of the event; otherwise the next activation time is the evaluated time. If the keywords are omitted, then this event is executed immediately, with no activation of another event in the mean time. If `After 0` or `At SystemTime` is written, the event is executed after all other active events with the same activation time. This difference is carefully to be considered. If Behind <proc ref> is written, the event is activated after process <proc ref> with the same activation time as that process. A run time error occurs, if either <proc ref> is not `Active`, or if <proc ref> is a reference to Null. If `Passivate` is used, the process is passivated, however, the active event (i.e. next event to be executed when this process is activated) is the event seleceted by the expression before this statement.

In case of sequence processes, a `<selection statement>` can include an `Enter` statement that sends a job to another process. The sequence in which these three parts of a `<selection statement>` are listed is arbitrary. We write as the syntax of a `<selection statement>`:

```
<selection statement> ::
 [<enter statement>] <event selection> [<execution time>]

<enter statement> ::   '' | Enter(<process ref>)

<event selection> :: Next [<label>] | Previous | First |
                          Last | Restart | Reinit | Repeat

<execution time>  ::   '' | After <time> | At <time> |
                          Behind <process ref> | Passivate ()
```

The first entry in the `<execution time>` and the `<enter statement>` the empty, i.e. this can be omitted; only `<event selection>` must always be given (in case of a sequence, the last event has even no `<event selection>`).

This finishes our considerations of the syntax of a process in C_SIM. Some further details can be found in the list of control parameters in the next section or in the examples following them.

III.2.3 Control Parameters of a C-SIM Program

C-SIM achieves its high flexibility by a concept called **control parameter**s or shorter **parameter**s. These are used to control the functionality of the system, and to specify special functions. Control parameters are listed before the simulation program specification after the C-SIM keyword `Include`. Global parameters, like queuing disciplines, job variables, or output specifications, are defined in the `<global parameter-list>`; local parameters are defined within processes in the `<local parameter-list>`:

```
main()
{ Simulation <name> Include <control parameter-list>;
  <global parameter-list>
…
Process <process name>;
  <local parameter-list>;
…
}
```

The functionality of these parameters is explained in this section. They are listed in alphabetical order. The general concept is explained in the first subsection:

III.2.3.1 General Parameter Concept of C-SIM

A control parameter is represented by a particular keyword, written in the standard C-SIM format (i.e. starting with an upper case letter, like: Queues, Histogram, Grapics). Control parameters can have actual parameters, like queue lengths or histogram resolution:

```
Queues(QueueLength=10),
Histogram(HistogramLength=200);
```

Only if a parameter is listed in the control parameter-list, its functions are included in the C-SIM program. If a parameter is not listed there, some of its functions are excluded from the program. If functions are confined by a guarded block, they are excluded if the corresponding parameter is omitted in the control parameter-list:

```
guarded_block :: {<parameter-name>:: <statement-list> }
```

An example is the following one:

```
{Graphics:: Move_Job(Station1,Station2); }
```

The statement in this block is excluded from the program, if the control parameter `Graphics` is not listed. Instead of removing the keyword from the control parameter list, it is also possible to prefix this by `No`:

```
Simulation Example Include Queue, No Graphics,  New MyParameter;
```

The programmer can define its own control parameter to exclude guarded blocks; this is done by preceding the corresponding user defined parameter with the word: `New`.

The parameter concept is also used to extend the functionality of the C-SIM defined parameters. Two examples (already mentioned earlier) are the Job structure and the queueing discipline. In C-SIM, a parameter is extended by writing the parameter name followed by the function that is to be extended. E.g. a Job is a control parameter (which has to be listed in the control parameter list), the type of which can be extended by

```
{Job :: struct JobType {int Count; TimeType Delay;}
}
```

If `Job` is omitted from the control parameterlist, this guarded block has no effect on the program. Otherwise, `JobType` is redefined, i.e. the standard definition of `JobType` is replaced by the definition as given here. The local parameters of a job are automatically extended by other parameters, needed to link jobs into a queue. They are: `Next` and

Previous, as defined in III.1.4, and int Priority, if the parameter Priority is listed in the control parameter list. (Since Priority is local to a job, this variable name can be the same as the name of the control parameter.)

To apply other queuing disciplines, the feasibility of a queueing discpline other than FIFO is to be stated by use of the control parameter: QueueDiscipline. Having done this, other disciplines but FIFO are available, such as LIFO, Random, and Priority. They are accessible by use of the keywords: FIFO, LIFO, RAND, PRIO. They can be accessed by call of the functions Insert and Remove. If another strategy is to be implemented, we use the syntax:

```
( QueueDiscipline ::
    Insert(Before_last)
        {  if (Length <= 1) Insert(Null);
           else Insert(Last -> Previous);
        }
    Remove(Before_last)
        {  if (Length <= 1) return(Null);
           else return(Remove(Last -> Previous));
   }    }
```

This is a guarded block to remove these definitions, if no queuing discipline is required. The functions names are used to distinguish between insert and remove functions. Although it is not necessary, it should be useful to define for both functions the corresponding disciplines. Since queuing disciplines can be applied to all queues (e.g. besides job queues also to process queues), the queueing disciplines are defined for arbitrary queues.

Some control parameters are defined implicitely. They are those for processes (the control parameter of which is Process), Activate, and Passivate. If these functions are not needed, these control parameters can be switched off by use of the No prefix.

C-SIM supplies the programmer with some standard lists, where two of them are important for the simulation. The first one is called ProcessList and is needed to schedule the events. The second one is called PassivList and holds the processes passivated. The other two standard lists are KillList and DeleteList, which are used to hold killed and deleted processes.

If the Passivate parameter is set, C-SIM knows the type ProcessQueue, which can be used to create queues that hold passivated processes. Theses processes can be scheduled explicitely by the programmer.

III.2.3.2 Activate in C-SIM

C-SIM supplies the programmer with functions to queue processes in a standard list called `ProcessList`, which is used to schedule the active events. The control parameter is set implicitely, and removed explicitely by:

activate parameter :: `No Activate;`

By call of the function:

```
Activate(<Process name>) [at <time> | after <time>] ;
```

the process `<Process name>` is removed from the `PassivList` and inserted into the `ProcessList` according to the `<QDiscipline>`; the activation time (i.e. the next time when the active event of this process is executed) is:

$$\text{Activation Time} = \begin{cases} \texttt{max(<time>,}\underline{\texttt{SystemTime}}\texttt{)} & \text{if } \underline{\texttt{at}}\ \texttt{<time>} \\ \underline{\texttt{SystemTime}}\texttt{+<time>} & \text{if } \underline{\texttt{after}}\ \texttt{<time>}\ (^3\ 0) \\ \underline{\texttt{SystemTime}} & \text{otherwise} \end{cases}$$

Instead of a process name, also a process type can be used. Then all processes of this type, which are stored in the `PassivList`, are activated. I no argument is given, all processes are activated. The parameters are always optional.

To transfer a process from a `ProcessList` into a `PassivList`, the function `Passivate()` is to be used.

III.2.3.3 Enter in C-SIM

`Enter` is a local parameter, used to create a **queueing system** with one queue and one server. This parameter is to be defined within a process:

queueing system ::

```
          Process <process name>;
             { Enter :: <queue name> [, <QDiscipline>];}
```

A queueing system is created by use of the local parameter keyword `Enter` followed by the name the queue is referenced to inside the process: `<queue name>;`. If the optional parameter `<QDiscipline>` is set, the jobs inserted into this queue are inserted according to this queueing discipline. If a job is send to this process by use of the `Enter`-function:

```
Enter (<job reference>,<process reference>);
```

this job can be picked up from a job queue named `<queue name>`. The function `Await(<queue name>)` removes the first job from the queue. It can also have a second parameter, specifying a particular queueing discipline defined as global parameter. If `AwaitJob(<queue name>)` is applied to an empty queue, the process is passivated, and the next active event is the same event that has executed this function. The `Enter()` function reactivates this process. The function `Remove()` can also access the queue `<queue name>`.

III.2.3.4 Graphics in C-SIM

C-SIM supplies the programmer with some graphic functions.

graphics parameter :: `Graphics ;`

III.2.3.5 Histograms in C-SIM

C-SIM supplies the programmer with some statistical functions. They can be used to evaluate observed quantities during the simulation run. A **histogram** can be used to estimate distribution functions. For this, the control parameter `Histogram` is to be set. An actual parameter of `Histogram` can be used to determine the length of the array used to store the observed data:

histogram parameter :: `Histogram [(Length = <number>)];`

Data are accumulated in an array of length:

```
double Histogram [Length + 2];
```

where the following mapping of a NewValue to a FieldIndex is used:

$$\text{FieldIndex} = \begin{cases} 0 & \text{if } \dfrac{\text{NewValue-Start}}{\text{Width}} < 0 \\ \text{Length+1} & \text{if } \dfrac{\text{NewValue-Start}}{\text{Width}} > \text{Length} \\ \dfrac{\text{NewValue-Start}}{\text{Width}} & \text{otherwise} \end{cases}$$

There are three functions to generate a histogram record, insert values into the histogram, and evaluate the results. They are:

```
struct HistoRec *NewHistogram(Count|Time)
Histogram(<histo ref>, val) TimeType val;
HistogramOutput(<histo ref>)
```

This function `NewHistogram()` generates and initializes a record of type `HistoRec` that contains all relevant data of a histogram. This record is to be transferred as first actual parameter at each function call that accesses this record. There are a couple of data in this record; known to the user are:

```
HistoNumber: an integer number to distinguish histograms
Max_Val_in_Batch: number of values in each batch
Start: lower border of intervall to be tested
Width: Width of each index
Moments: Reference to a record that computes moments
```

There are several other variables in this record, which should however never be accessed by the programmer. They are only accessed via the other two functions. The constants `Count` or `Time` are used to consider two different cases. If `Count` is set, a sequence of values is cumulated, e.g. the waiting time of a sequence of jobs arriving at a given queue. If `Time` is set, the quantity is considered to be a function on time, e.g. the number of jobs in a queue. The computation for these two types of quantities is different, so the programmer should be aware for which type of variable the histogram is to be estimated.

The function `Histogram()` is used to insert a value into the histogram. The first parameter is a reference to the corresponding histogram, the second is the value to be accumulated.

The function `OutHistogram()` is used to generate the corresponding output file where the histogram and some other values are listed in readable form. By use of some targets it is possible to read from these files the data to estimate other quantities, or to generate diagrams to plot the histogram. To produce readable text, it is useful to write a corresponding environment. This can be done by use of the global parameter `Histogram` and some additional output statements:

```
Histogram
[ <histo name>: { [ fprint(<text>,parameter); ]* }
 ]+
End Histogram ;
```

Here, the global parameter is followed by a label, used as name of this histogram output. Then some `fprint` statements follow, enclosed in {}, as if the output were written on the screen. Inside the output procedure the variable <u>`Histogram`</u> is a reference to the histogram record. An example should clarify this:

```
Histogram
 OutSojourntime; {
    fprint("Simulation Model ARGUS\n");
    fprint("Simulation Time = %f\n", SystemTime);
    fprint("Mean Sojourn Time = %f\n",
Moment1(Histogram->Moments));
 }
 OutQOccupancy: {
    fprint("Simulation Model ARGUS\n");
    fprint("Simulation Time = %f\n", SystemTime);
    fprint("Mean Queue Occupancy = %f\n",
Moment1(Histogram->Moments));
 }
End Histogram ;
```

Inside the histogram record the variable <u>`Output`</u> can be used to control the output by setting its value to the corresponding name:

```
Histogram->Output = OutQOccupancy;
```

III.2.3.6 Interrupts in C-SIM

C-SIM supplies the programmer with some control functions that can be used to input some data or instructions into the running simulation program. This can be used to control graphics output, to change some parameters, or to activate some processes etc. The control parameter is set by:

Interrupt parameter :: <u>`Interrupt`</u> `;`

If this is set, a global parameter <u>`Interrupt`</u> should be specified. This would look like:

```
Interrupt
[ '<character>': { [ <statement>; ]*; break; }
 ]*
End Interrupt ;
```

An example is the following:

```
Interrupt
 case 'T':  Tracing = True; break;
 case 't':  Tracing = False; break;
 case '0':
 ..
 case '9':  Speed = key - '0'; break;
 default :  break;
End Interrupt ;
```

The first two cases set the `Tracing` parameter, the other the `Speed` parameter. `key` is a variable that holds the character the key of witch was pressed on the keyboard.

III.2.3.7 Jobs in C-SIM

C-SIM supplies the programmer with a record that can contain data. This record is usually sent from one process to another, i.e. while processes are the active objects, jobs are the passive objects of C-SIM. The control parameter is set by:

job parameter :: `Job ;`

If this is set, a global parameter `Job` should be specified to distinguish between different types of jobs.

```
Job
[ <job name>: { [declaration; ]* }
 ]*
End Job ;
```

An example is the following:

```
Job
 WorkLoad; {
         float Load; TimeType ArrivalTime;
 }
 Signal; {
         int Counter; TimeType LastTime;
 }
End Job ;
```

If this is omitted, then only standard jobs of type `JobType` can be used; they do not have any additional variables to hold other information than that provided by C-SIM. A job record can be used to transport information between different processes. It is generated by the function:

```
JobType New Job([<job name>]);
<job ref> = New Job([<job name>]);
```

If the parameter is omitted, a job of standard type JobType is generated. As soon as a job is generated, its type cannot be changed anymore. However, all types of jobs can be inserted into standard queues (if the Queue parameter is set), and they are all generated by the same function. Their type can be tested by the function call:

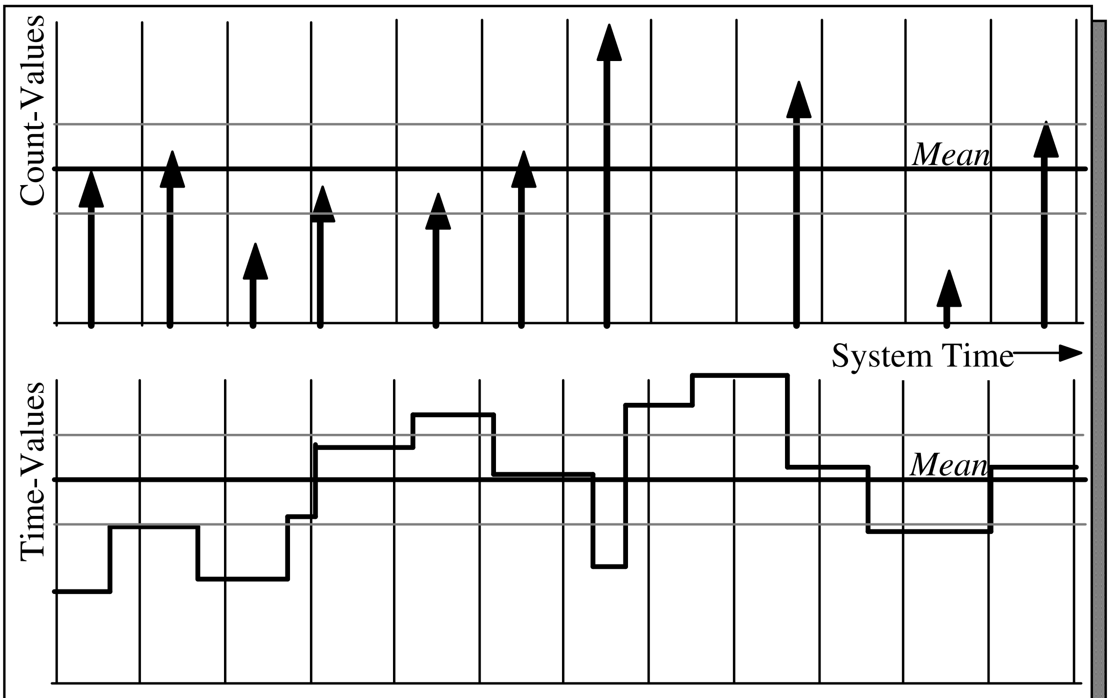
```
bool IsJobType (<job ref>, <job name>);
```

This returns `true` if the job `<job ref>` is of type `<job name>`. The second parameter `<job name>` can also be JobType or Sequence; for the meaning of the latter see the parameter Sequence.

Jobs are killed by the standard C function `free()`, however, jobs that are to be killed may not be linked within a JobQueue.

III.2.3.8 Moments in C-SIM

C-SIM supplies the programmer with some statistical functions. Moments are defined to estimate some average values of some statistical quantities. We have to distingish between Count- and Time-Values:



Count- and Time-Values for Statistical Analysis

If $\{w_1,w_2,\ldots,w_N\}$ is a finite sequence of values, the first two moments of a `Count` quantity are defined to be:

$$\overline{w} = \frac{\sum_{i=1}^{N} w_i}{N} \qquad \text{(1. Moment)}$$

$$\overline{w^2} = \frac{\sum_{i=1}^{N} w_i^2}{N} \qquad \text{(2. Moment)}$$

and if $\{(w_1,t_1),(w_2,t_2)\ldots,(w_N,t_N)\}$ is a function on time, where at time t_i the quantity attains the value w_i, then we define the moments to be:

$$\overline{w} = \frac{1}{t_N - t_1} * \int_{t_1}^{t_N} w(t)\,dt = \frac{\sum_{i=1}^{N-1} (t_{i+1}-t_i)*w_i}{t_N - t_1}$$

$$\overline{w^2} = \frac{1}{t_N - t_1} * \int_{t_1}^{t_N} w(t)^2\,dt = \frac{\sum_{i=1}^{N-1} (t_{i+1}-t_i)*w_i^2}{t_N - t_1}$$

The first moment is also called **mean**, since it is the average value of all quantities, or it is the average height of the function. The second moment can be used to estimate the variance. This is defined by the quantity:

$$\sigma_w^2 = \frac{\sum_{i=1}^{N} (\overline{w}-w_i)^2}{N-1} \approx \overline{w^2} - \overline{w}^{\,2} \text{ (Variance)}$$

The control parameter is set by:

moment parameter :: <u>`Moment`</u> ;

The functions to generate, accumulate and evaluate these quantities are definied to be:

```
struct MomentRec *NewMoment(Count|Time)
Moment(<moment ref>, val) TimeType val;
TimeType Moment1(<moment ref>)
TimeType Moment2(<moment ref>)
TimeType Variance(<moment ref>)
TimeType ConfInt(<moment ref>,gamma) TypeType gamma;
```

The function <u>`NewMoment`</u>`()` is used to generate a record of type <u>`MomentRec`</u>, where during creation the programmer has to determine of which type this variable is. The procedure <u>`Moment`</u>`()` is used to insert another value into this record. The functions <u>`Moment1`</u>`()`, <u>`Moment2`</u>`()`, and <u>`Variance`</u>`()` generate the corresponding result, where the actual parameter is always a reference to the corresponding moment record.

Instead of computing only the mean, it is sometime useful to compute a confidence interval, the mean lies in with some probability. Since the quantities computed are usually highly dependent, we use a batch technique: All sequences of values are computed in batches of equal length, which are for some theoretical reasons normally distributed and independent. Then the t-distribution can be used to estimate an interval, the real values μ lies in with some probability:

$$\overline{w} - \frac{s*c}{\sqrt{b}} \leq \mu \leq \overline{w} - \frac{s*c}{\sqrt{b}}$$

Here, s is the estimated variance of the batches, c is the solution of the equation F(c)=(1+γ)/2, where F is the t-distribution with n degrees of freedom, and γ is the probability that μ lies in this interval. This can be estimated by the function call `ConfInt`, which returns the quantity c. The value $\overline{w}$ can be achieved by a variable in the moment record, called: `Mean`.

III.2.3.9 Passivate in C-SIM

C-SIM supplies the programmer with functions to queue processes in a standard list called `PassivList`, or in a user defined list of type `ProcessQueue`. The control parameter is set implicitely, and removed explicitely by:

passivate parameter :: `No Passivate;`

A queue is created by use of the function:

```
ProcessQueue New Queue(ProcessQueue [,Length=<number>]);
```

which returns a reference to a `ProcessQueue`. The optional parameter `Length` is used to control the maximum number of processes in this queue; if it is omitted, the length of the queue is not limited. By call of the function:

```
Process Passivate(<ProcessQueue name>[,<QDiscipline>]);
```

the process performing this function is removed from the `ProcessList` and inserted into the `<ProcessQueue name>` according to the `<QDiscipline>`. If the latter is omitted, FIFO is assumed. If both actual parameters are omitted, the process is transferred into the standard passiv list.

Access of a process list is gained by use of the standard functions `Remove()` and `Enter()`, which can also have queueing disciplines as parameters. All queueing disciplines, standard or user defined, are available.

To transfer a process from a PassivList into the ActivList, the function Activate() is to be used.

III.2.3.10 Preempt in C-SIM

C-SIM supplies the programmer with a concept to interupt events in execution. This feature can be used to implement disciplines, which are found for example in processor interrupts. This concept is called preempt, since the a selected event is executed before all others. The control parameter is set by:

preempt parameter :: Preempt;

The local parameter is defined by:

preempt parameter ::

```
{Preempt :: <save variable list> [,<QDiscipline>];}
```

The selected event is specified by:

```
Preempt :
        <statement list>
        <selection statement>
```

Functions used to apply this feature are:

```
Preempt(<process ref>)
Resume(<Resume_QDiscipline>)
```

If another process executes the function Preempt(<process ref>), the event in action is interrupted, its residual delay time, including the values of the variables in the <save variable list>, is saved in a special list, not accessable by the programmer, according to the <QDiscipline>; the statement list is executed, including all other events activated by this event. If the function Resume(<Qdiscipline>) is executed, one of all interrupted events (according to the <Resume_Qdiscipline>) is restored (i.e. residual delay time, active event and all variables fromt the <save variable list> are set on their previous values) and its execution is resumed. Nothing is changed, unless the events executed during interrupt have changed the state of the process or some global variables.

If a server system is modeled by use of this concept, the job in service should always be saved in the <save variable list>. Other local objects are usually changeable by the preempt actions, so they should not be saved. However, here not general rules can be given, since this concept is very general and can be used to model very complicated systems.

III.2.3.11 Queues in C-SIM

C-SIM supplies the programmer with functions to queue jobs in a sequence. This data structure is called a `JobQueue`, and it holds a reference to a sequence of jobs of type `JobType`. The control parameter is set by:

queue parameter :: `Queue ;`

A queue is created by use of the function

```
<ref type> New Queue([ProcessType][,Length=<number>]);
bool Insert(<queue ref>,<obj ref>[,QDiscipline>])
<obj ref> Remove(<queue ref>[,QDiscipline>])
```

which returns a reference to a `Queue`. The optional parameter `Length` is set to the maximum length of the queue, i.e. the maximum number of objects this queue can hold. Objects can be inserted into the queue using the function `Insert(<queue ref>,<obj ref>)`, and removed by use of the function `Remove(<queue ref>);`. The jobs are stored and retrieved FIFO. Details can be found in section III.1.4, extensions at the parameter `QueueingDisciplines`.

Graphics is inserted <to be continued>

III.2.3.12 Queueing Disciplines in C-SIM

C-SIM supplies the programmer with some additional queueing disciplines, and also the possibility to program special queueing disciplines. The control parameter is set by:

queueingdiscipline parameter :: `QueueingDiscipline ;`

Standard functions to control the queuing discpline are: FIFO, LIFO, RAND and PRIO (for details see section III.1.4). Additional disciplines can be programmed by the global parameter `QueueingDiscipline`.

```
{QueueingDiscipline ::
 [Insert|Remove (<queueingdiscipline name>):{<statement>;*}
 ]+
} ;
```

The actual parameters Insert and Remove determine, whether the statements insert or remove a job. They can be accessed by the functions `InsertJob()` and `RemoveJob()`, where the last parameter is used to determine the queueing discipline. An example:

```
{QueueingDiscipline ::
   Insert(Before_last)
   {     if (Length <= 1) Append(Null);
         else Append(Last -> Previous);
   }
   Remove(Before_last)
   {     if (Length <= 1) return(Null);
         else return(Remove_This(Last -> Previous));
   }
} ;
```

Details on the functions and variables `Last`, `Previous`, `Length`, `Insert`, and `Remove` are found in section III.1.4. The reference `This` hold a reference to the object to be inserted into this queue.

III.2.3.13 Random Distributions in C-SIM

C-SIM supplies the programmer with some random functions to generate randomly distributed quantities. The functions use the standard UNIX function drand48() and is called `Random()` in C-SIM. It returns a double precision float value in the range of 0 to 1 (excluding 1): `Random()` $\in$ [0,1). This is generated by use of a congruential random number generator starting with the inital value:13332345. This initial value can be changed by the function call: srand(<number>). If the same initial value is used, the sequence of random numbers will always be the same. The control parameter is set by:

random distribution parameter :: `RandomDistribution ;`

The following list gives the names, declarations and a short description of the random functions supplied by C-SIM:

```
double Random(): Returns a double random value from [0,1)

bool Is_True(double p): Returns true with probability p

int Equal(int low,up): Returns an integer value from [low,up]

TimeType Random(TimeType low,up): returns a time from [low,up)

TimeType NegExp(double Mean): returns a time from
               negative exponential distribution with mean Mean

TimeType Normal(TimeType Mean,Stand): returns a time from
normal distribution with mean Mean and
Standarddeviation Stand

TimeType Erlang(TimeType Mean,int Steps): returns a TimeType
value with mean Mean and variance 1/Steps.

TimeType Geometric(TimeType Mean): returns a time from
geometric distribution with mean Mean
```

III.2.3.14 Sequence in C-SIM

C-SIM supplies the programmer with a mechanism to control the routing of jobs through a network of queuing stations. This is done by a process with local parameter sequence. The control parameter is set by:

sequence parameter :: <u>Sequence</u> ;

and the local parameter is set by:

```
Process <process name>;
{ Sequence :: <job type> <job ref name>; }
```

Arbitrary events can be executed, where the control of a process is transferred to another process by the function call:

```
Enter(<process ref>);
```

which is obviously the same function as the `Enter` parameter function. If such a process is generated, it generates a process of type `<job type>` and executes the events in the given sequence. Additional to the selection statement, an `Enter` function is executed; the job is then transferred to the corresponding queueing system. This has to perform the function:

```
Return( <job ref name> )
```

to return the job to the sequence process. This resumes execution with the next event. An example follows:

```
Process Service:
{ Enter :: JobQ; }
   JobType Job;
   Event :
         Job := Await( JobQ );
         Next After Equal(1,10)
   Event
         Return Job;
         Restart ;
EndProcess Service ;

Process Manufacture;
{ Sequence :: JobType Job; }
Event :
         Enter( Server1) Next After 50 ;
Event Label1:
         Enter( Server2) Next ;
Event :
         if (Job.something_wrong) Next label1;
         Enter( Server3) Next ;
Event :
         Next After 10 ;
Event :
         if (Is_True(0.2)) then Previous After 0;
         Enter( Server4) Next ;
...
Event :
         Enter( Server9);
EndProcess Manufacture;
```

Here the job is sent at first to server1; if it returns, there is a delay of 50 units, and then it is sent to server2; if it returns and something is wrong, it is sent back to server2; otherwise the job is sent to server3; after a delay of some time (10 units) the job is sent with probability 80% to server4, otherwise back to the delay of 10 units, etc. Being served in the last server9, the job as well as the sequence process is killed.

The processes the job is sent to must be a queueing system, i.e. have the local parameter `Enter`. A job can be returned only, if it was sent by a sequence process. Otherwise a run time error occurs (since enter queues can hold all types of jobs, this cannot be checked statically). To check the type of a job see the function `IsJobType()` at the parameter: `Job`.

III.2.3.15 Table in C-SIM

C-SIM supplies the programmer with some graphic functions; so the change of the state of the simulation model during execution can be plotted on a screen. In many cases, the model consists of servers or similar stations, exchanging jobs or other objects. To simplify computation of the coordinates of these objects (if they are drawn on the screen), C-SIM supplies the table concept. The control parameter is set by:

table parameter :: Table ;

and the global parameter is set by:

```
{ Table :: [<screen name>]
            |     <text>
<height>    |     <text>
<L-Dist>    |     <text>
---------------------------------------------------
<B-Dist>    <width>
}
```

The <text> inside the bars and dashes is plotted on the screen within a field of the size of the pixels: <height>x<width>. If the text '@'<number> is plotted, the number is a reference in the function coord(<number>) to get the coordinates of the symbol: '@'. This can be used in the plot functions to place the station on the screen. The numbers <L-Dist> and <B-Dist> are used to place the lower left corner of the diagram to the point (<L-Dist>x<B-Dist>). An example follows:

```
{ Table :: BarberModel;
     |
     |      Barber Shop Model, Monday, 12.1. 1999
     |
     |            @1          @2          @3
100  |
     |      @10   @11         @12         @13         @14
     |
     |            @21         @22         @23
     |
     |      Simulation Time @101   Average Load @102
20   |
---------------------------------------------------
10          300
}
```

III.2.3.16 Traces in C-SIM

For graphics output during a simulation run C-SIM uses the parameter Tracing, which supplies the programmer with a boolean variable Tracing. The control parameter is set by:

tracing parameter :: Tracing ;

The global variable Tracing can be set anywhere in the program; for an example see the parameter Interrupt. If Tracing==True, then all graphics output is plotted on the screen; otherwise, no graphics is plotted. If the user implements a guarded block using Tracing as parameter, all statements inside this block are executed only, if Tracing was true when this block was entered. This is equivalent to an if (Tracing) statement:

```
{ Tracing ::   <statement>;* }
```

or

```
if (Tracing) { <statement>;* }
```

Tracing is true when simulation starts.

III.2.3.17 Wait in C-SIM

If execution of an event depends on a general condition, C-SIM supplies the programmer with a concept to implement this elegantly. However, this method takes much execution time since dynamic wait is used. The control parameter is set by:

```
wait parameter  ::  Wait ;
```

Inside a process, the statement Wait can be executed. Doing this, the event is finished, the process is inserted into another queue, called Wait-list, and the process is executed next only after another process from the (regular) process list is executed (to be accurate: Being put into the Wait-list the first time, the process is executed after the first element of the process list is executed, etc.) The active event of this process is the current event. No time is scheduled with this event. Thus, an event using the Wait construct should look like:

```
Event :
  <evaluate a condition>;
  if (<condition>) Wait;
  <statement>;*
…
```

Before executing Wait in this event, only statements for evaluation of the condition should be executed. Also, the state of the system should not be changed by these statements. Then Wait is executed, or if <condition> is false, the residual statements of this event are executed.

Thus, this event behaves like a 'busy wait' statement: It permanently checks the condition, and since this can be altered only by executing another process, this event is performed only after the next process in the process list. A typical example is to check a queue:

```
Event :
  if (Queue->Empty) Wait;
  Job = Remove(Queue);
  …
```

Here, the event checks, whether Queue is emtpy. If it is, the process waits until a job is inserted into this Queue. This job is removed immediately, where no other event occurs in the intermediate time. Thus, when executing Remove, there is definitely a job in the queue. If there is another process waiting for this condition, it will be executed after this event, i.e. finds the queue empty.

III.3 Simulation Model Examples in C-SIM

In this section we describe some example models which can be modelled by use of C-SIM. In the first subsection we analyse queueing systems, particularly to model computer systems. The second subsection models systems that use dynamic processes, i.e. processes that are to be generated and destroyed during the simulation run, and the third subsection analysis some miscallaneous systems.

Since C-SIM is not yet available, we implement the examples below also in plain C, using a systematic technique describe in the last subsection.

III.3.1 Queueing Model Examples in C-SIM

The first example is a simple model where two processes communicate by exchanging jobs. The first process gets on intialisation a (new) job, the second process has no job (i.e. references Null). Both processes reference each other by `OtherProcess`. Thus, if the first process stores its job to `OtherProcess`, it stores this into the second process, and vice versa. While a process does not hold a job, it is passivated, however, will be activated immediatly after receiving the job. The activated process is delayed for 10 units of time, and then it proceeds by sending the job to the `OtherProcess`, which will also be activated:

```
main()
{ Simulation Example_1 Using Job;

Process SendJob;
JobType Job;
SendJob OtherProcess;
Event
         if (Job == Null) Passivate();
         Next After 10;
Event
         OtherProcess -> Job = Job;
         Job = Null;
         Activate(OtherProcess);
         Next After 0;
EndProcess;

SendProcess Process1;

Process1 = New Process(SendJob, Job=NewJob());
Process1->OtherProcess = New Process(SendJob,
Job=Null, OtherProcess=Process1);

Simulate for 1000;
}
```

The job is sent between those two processes, where no further information is transferred. To transmit information, the global parameter `Job` is to be set. In the many applications, a job is generated by one process and served and killed by another process. Thus we can implement another example of a queueing system, a server model:

```
main()
{ Simulation Example_2 Using Job, Queue;
Process Arrival;
Init
         Next After 5;
Event
         InsertJob(Server1->Queue, NewJob());
         if (Server1->Status == Passiv) Activate(Server1);
         Next After 12;
EndProcess;

Process Server;
JobType Job;
JobQueue Queue;
Init
         Queue = New Queue();
         Next After 0;
Event
         if (Queue->Length == 0) Passivate();
         Job = RemoveJob(Queue);
         Next After 10;
Event
         free(Job);
         Next After 0;
EndProcess;

Server Server1;

New Process(Arrival);
Server1 = New Process(Server);

Simulate for 1000;
}
```

This program models a simple single server queue; if the server's queue is empty, the server passivates. If the arrival process sees that the server is passivated, it has to activate the server. The job is served by a delay of the server, and then the record holding the job is freed.

This program can be extended model a system with two (or more) servers. Besides extending to more than one process we also introduce some random distributions:

```
main()
{ Simulation Expl_3 Using Job, Queue, RandomDistribution ;

Process Arrival;
Init
         Next After NegExp(12);
Event
         InsertJob(Queue, NewJob());
         Activate(Server);
         Next After NegExp(12);
EndProcess;

Process Server;
JobType Job;
Event
         if (Queue->Length == 0) Passivate();
         Job = RemoveJob(Queue);
         Next After NegExp(16);
Event
         free(Job);
         Next After NegExp(1);
EndProcess;

Server Server1;
JobQueue Queue;

Queue = New Queue(Length = 5);
New Process(Arrival);
New Process(Server);
New Process(Server);

Simulate for 1000;
}
```

Here, some parts look simpler than in the previous example. Instead of checking the status of a process, we simply activate all processes of the corresponding type, provided they are passive. They will passivate themselves, if no job can be found in the queue; thus also no global reference to a server is required. Instead of only two servers, an arbitrary number of servers could be instantiated.

This could be the model of a barber shop. Customers arrive according to a negative exponential distribution. They have to wait in a queue (i.e. on one of five chairs); if no chair is available, the arriving customer will not wait and leave the shop. Having served a customer for some time (16 units of time), the customer leaves the shop. The barber will clean up the place (1 unit) and then get the next customer from the queue. This model is already more refined than the previous one.

Further details can be implemented easily. E.g. the service time of a customer will depend on the service required (washing, cutting, shaving, drying, laying etc.). Also, charging the customer will cost some time. There should be a coffee break for the barbers (not all together), and the customers could reject service, depending on the expected waiting time.

Questions that could be answered by this model are: How many chairs should be available? How many barbers would be economical? How long is the average waiting time of a customer? What would be changed, if some coiffeurs would offer only some services (no shaving by a learner)? Can there be a gain in utilization, if some works would be done by some auxialiary persons (like charging, cleaning up the places, etc.)?

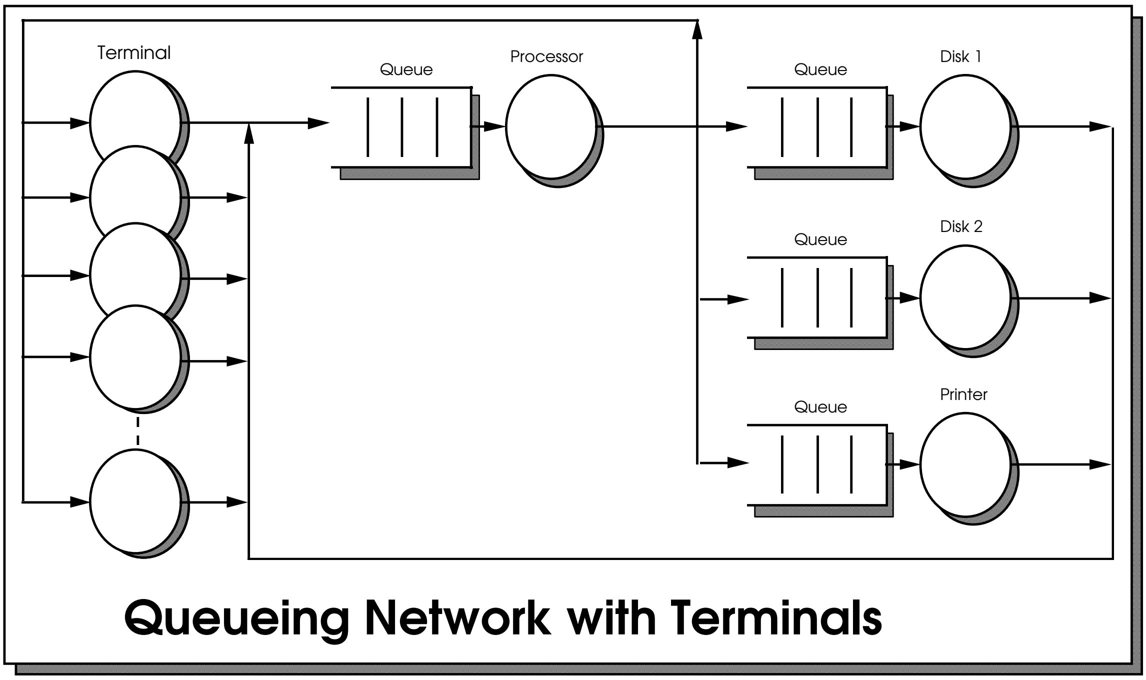
Other types of queueing systems use several independent arrival processes, where each process generates its own class of jobs. Problems can occur if removing of jobs from a queue takes a finite time and several servers access that queue. In this case, an auxialiary process, serving the queue, should be used to model this in a quite correct way.

In C-SIM a process can be passivated in a particular queue of type `ProcessQueue`. A process that performs the passivate function can be inserted into the corresponding queue, and be activated by another process.

```
main()
{ Simulation Expl_4 Using
                Job, Passivate, Queue, RandomDistribution ;

Process Arrival;
Init
         Next After NegExp(12);
Event
         InsertJob(JobQ, NewJob());
         Activate(QService);
         Next After NegExp(12);
End Process Arrival;

Process QueueServer;
ProcessQueue ProcessQ;
Init
         ProcessQ = New ProcessQueue();
         Next After 0.5;
Event
         if (Empty(ProcessQ) or Empty(JobQ)) Passivate();
         Next After 0.5;
Event
         Activate(Remove(ProcessQ));
         Next After 0;
End Process QueueServer;

Process Server;
JobType Job;
int ProcessNumber;
Event
         Activate(QService);
         Next Passivate(QService->ProcessQ);
Event
         Job = RemoveJob(JobQ);
         Next After NegExp(16);
Event
         free(Job);
         Next After NegExp(1);
End Process Server;

QueueServer QService;
JobQueue JobQ;

JobQ = New Queue(Length = 5);
New Process(Arrival);
QService = New Process(QueueServer);
for (i=1; i<=20; i++)
                     New Process(Server, ProcessNumber=i);

Simulate for 1000;
}
```

This simulation program generates jobs in the `Arrival` process; the jobs are inserted into the global `JobQ`. If a server wants to serve a job, it inserts itself into the local `ProcessQ` of the `QService` process. This process delays for some time (0.5), and after this time it activates the first process in its process `ProcessQ`; before the next process can be activated, there is another delay of 0.5, etc. Since the `QService` process is passivated if either the `JobQ` or the `ProcessQ` is empty, execution continues only if a job is ready to be served, and a process is ready to serve this job. Thus service time of `JobQ` is modeled by a constant delay time of 0.5.

The next example models a queueing network, which is offently used to model computer systems. The model consists of a set of queueing systems, each represents an unit like processor, disk, drum or terminal. Jobs are transmitted between these units, occupying the processor for some time, accessing disks and drums, etc. until they return to the terminal. Thus the terminal generates jobs, while the other units serve them. Performance measures are utilization of some units, turn around time (time from sending a job from the server and receiving the answer), etc. A queueing model can be drwan as follows:



The simulation program for this model can be implemented in several ways. To demonstrate this, we implement this at first using a process concept where all jobs are routed by statements within the corresponding processes, and another implementation, using the sequence technique:

```
Process Terminal;
 BatchJob Job;
   Event : Next After NegExp(10)
   Event :
         Job = New Job (BatchJob);
         Job->TerminalProcess= ThisProcess;
         Enter( Job, Processor->Queue );
         Next Passivate();
End Process Terminal ;

Process ProcessorServer;
 {Enter:: JobQueue Queue;}
 BatchJob Job;
 double Prob;
   Event :
         Job = Await( Queue );
         Next After NegExp(12);
   Event :
         Prob = Random();
         if (Prob < 0.3))    Insert( Job, Disk1->Queue };
         else if (Prob < 0.6)) Insert( Job, Disk2->Queue };
         else if (Prob < 0.9)) Insert( Job, Printer->Queue };
         else { Acivate(Job->TerminalProcess); free(Job); }
         First After NegExp(1);
End Process ProcessorServer;

Process Server;
 {Enter:: JobQueue Queue;}
 Init double MeanServiceTime;
 BatchJob Job;
 double Prob;
   Event
         Job = Await( Queue );
         Next After NegExp(MeanServiceTime);
   Event
         Insert( Job, Processor->Queue );
         Next After NegExp(1);
End Process Server;

ProcessorServer Processor;
Server Disk1, Disk2, Printer;

for (i=1; i<=20; i++) New Process(Terminal);
Processor = New Process(ProcessorServer);
Disk1 = New Process(Server; MeanServiceTime=5.5);
Disk2 = New Process(Server; MeanServiceTime=11);
Printer = New Process(Server; MeanServiceTime=35);

Simulate For 10000;
```

This program fragment consists of three process types, where the process `Terminal` has 20 incarnations. The corresponding reference of the `Terminal` process is hold in a variable local to `Job`. Thus, there must be a declaration of the global variable `Job`:

```
    job parameter :: Job ;
    …
Job
 BatchJob; {
         Terminal StartProcess;
 }
End Job ;
```

If the `Terminal` process has sent its job to the `Processor` it passivates being activated again when the job will have been finished its tour through the other servers being sent back to the Terminal. Since we use a local reference to the `Terminal` process, it is easy to verify that a job is sent back to the `Terminal` process it started from. Other methods like global arrays of references and indices identifying the jobs and their terminals would be less secure.

Ther server serves the job for 12 units of time on the average. Then the jobs is sent to one of the three other servers, with probability 0.3 each. With probability 0.1 it is sent back to its terminal. The three servers serve each job according to a mean service time set at their instantiation. If they finish service they send the job back to the processor.

The next program uses the sequence technique. The sequence process is here called `Router`. The reference to the creating terminal process is here hold inside a variable of the `Router` process, which is set during instantiation of the router process; a `Router` process is instantiated by a `Terminal` process. This sends the job at first to a server, acting as processor. Returning from this process, the job is sent to one of three other servers, and returning they start by being served from the processer again. With probability 0.1 the sequence process is finished by executing the last event of this process. Since the last statement activates the `Terminal` process the job was sent from, the next job can be sent from this process after a delay time of 10 on the avarage.

The difference between these two simulation programs is obvious. We see that the sequence example presents in a very simple way the route the jobs are taking through the network. Thus, it should be easy to change this route and still to keep overview. In the previous program, a change of a route means to change each server, which might be difficult to manage for many more stations than in our example.

```
    Process Terminal;
    Event : Next After NegExp(10)
    Event :
          New Process (Router, TerminalProcess = ThisProcess;);
          Next Passivate();
EndProcess Terminal ;

Process Server;
 {Enter:: JobQueue Queue;}
 Init double MeanServiceTime;
 BatchJob Job;
    Event :
          Job = Await( Queue );
          Next After NegExp(MeanServiceTime);
    Event :
          Return(Job);
          Next After NegExp(1);
End Process Terminal ;

Process Router;
{ Sequence :: BatchJob Job; }
 Init Terminal TerminalProcess;
Event :
          Job->MeanServieTime = NegExp(12);
          Next Enter( Job, Processor) ;
Event :
          Prob = Random();
          if (Prob < 0.3))
                Enter( Disk1 ) First After NegExp(1) ;
          else if (Prob < 0.6))
                Enter( Disk2 ) First After NegExp(1) ;
          else if (Prob < 0.9))
                Enter( Printer ) First After NegExp(1) ;
          Acivate(TerminalProcess);
End Process Router;

for (i=1; i<=20; i++) New Process(Terminal);
Processor = New Process(Server; MeanServiceTime=12);
Disk1 = New Process(Server; MeanServiceTime=5.5);
Disk2 = New Process(Server; MeanServiceTime=11);
Printer = New Process(Server; MeanServiceTime=35);

Simulate For 10000;
```

III.3.2 Simulation Models With Varying Numbers of Processes

Its possible to create and kill processes during simulation. This method can be used to model systems, where a varying number of population exists. The first example is a simple model of some animals, which have three phases in their lives: Childhood, adults, and senile. Only in the second phase they can produce knew children (i.e. generate new processes). The total population is counted on the global variable `population`:

```
Process Animal;
 Init int Sex; /* female =1, male = 2 */
 Init TimeType BirthYear;
 Event : /* Childhood */
         Population =+ 1;
         if (Random() > DyingProb) kill();
         Next After 12 + NegExp(2);
   Event : /* fertile adult */
         while (Random() <= 0.7 && SystemTime-BirthYear <= 50)
         {     if (sex == 1) New Process(Animal,
                     Sex = Equal(1,2); Birthyear = SystemTime);
               Repeat After 1 + NegExp(1);
         }
         Next After 0;
   Event : /* senile adult */
         Next After NegExp(120-(SytemTime - Birthyear));
   Event : /* dead */
         Population =- 1;
         kill()
End Process Animal ;

int population = 10;

for (i=1; i<=population; i++)
         New Process(Animal,
                     Sex = Equal(1,2); Birthyear = SystemTime);

Simulate(1000);
```

This program consist of one process only. Its three events are executed once each. Only the third event has a loop, so there are several chances to generate a child, provided the animal is female. Several other environmental conditions, like change of food or deseases could be implemented in a similar simple way. Questions could been ask, like the chance that the population dies out, that there are too many animals in that area, or how many children of a couple would become adult. Here, `DyingProb` might depend on some conditions like food supply, temperature or number of other species, which could be modelled in a similar way in the same program by adding simply another process. The program ends if either the simulation time (1000) ends, of if all animals die out, i.e. the process list becomes empty.

This simple example shows that it is very easy to program models with varying number of processes, where for the example above, this is also a very natural way to model this real life system.

The next example models a system of moving objects, like cars or trains, that communicate with one of several fixed communication antennas. Since the objects moves, they can leave an area where an antenna can receive its signal, and then it has to change its connection to another antenna. This is done by some protocols between those antennas and the moving object.

Generation and moving of objects occurs either randomly or according to some conditions, like the trail of a railroad or a highway. For our model, we assume that there is a function `NextAntenna`, which determines the next antenna an object is to be connected to. Conflicts occur if an antenna is overutilized since all channels are occupied. In this case a connection is lost. The process modelling a moving object, called `Mobile`, can be implemented as follows:

```
Process Mobile;
 Init int Antenna; /* number of antenna the mobile
                          communicates with */
 TimeType StartConnection;
 Init  :
         StartConnection = SystemTime;
         if (FreeAntenna(Antenna))
               Connect(Antenna);
         else {
               CountLoss(Antenna);
               kill();
         }
         Next After NegExp(10);
   Event :
         Disconnect(Antenna);
         if (Random() <= 0.3) kill();
         Antenna = NextAntenna(Antenna);
         if (FreeAntenna(Antenna))
               Connect(Antenna);
         else
         {     Countloss(Antenna);
               kill();
         }
         Next After NegExp(10);
End Process Mobile ;
```

The functions `Connect(Antenna)` and `Disconnect(Antenna)` are used to count the number of connections, the number of losses etc. They will be used for statistics. Other statistics can compute the mean connection time of a mobile by use of the variable `StartConnection`. Further processes are required to start these mobiles etc. They depend very much on the concrete model, so we omit them here.

III.3.3 Miscellaneous Simulation Models in C-SIM

The first example

<to be continued>

III.3.4 Implementing Simulation Programs in C

To implement a simulation program in C, we use a switch, the cases of which represent the events of processes. Each Process holds two local variable, one contains the next activation time of this process, and the other the number of the event that is to be executed. Before leaving an event, the next event and the next activation time is set inside the process record. The active process is always the first process in the ProcessList. To avoid multiple references

```
while (SystemTime <= <max sim time>) {
      SystemTime = ProcessList->Time;
      switch(ProcessList->Event) {
            case <n>:
                  ...
                  ProcessList->Event = ...;
                  ProcessList->Time  = ...;
                  break;
            ...
      }
      SortProcess();
}
```

The procedure `SortProcess()` sorts the first element of the `ProcessList` according to its `Time` behind the first process with an activation time equal to or smaller than `Time`. Since this is repeated until the `SystemTime` is larger than the maximum simulation time, this is already the complete simulation system.

Complications come from the fact that each process has its own record, that processes can be passivated or waiting, and that there are distinct processes to be executed within this loop.

Different process types are implemented by using different labels. E.g. the first process uses the label 0, 1, 2 etc, the seconde one 10, 11, 12 etc. The `After` statement is implemented by adding the corresponding quantity, and the `At` statement by assigning the corresponding quantity. For the second process, its `Next`, `Repeat`, and `Restart` statement is be implemented as:

```
case 10: /* Init */
      ...
      ProcessList->Event = 11; /* Next */
      ProcessList->Time  =+ 1; /* After 1 */
      break;
case 11: /* Event */
      ...
      ProcessList->Event = 12; /* Next */
      ProcessList->Time  =* NegExp(19);
                                /*After NegExp(19) */
      break;
case 12: /* Event */
      ...
      If (..)
            ProcessList->Event = 12; /* Restart */
      else
            ProcessList->Event = 13; /* Next */
      ProcessList->Time  = 1.1*SystemTime;
                                /*At 1.1*SystemTime */
      break;
case 13: /* Event */
      ...
      ProcessList->Event = 11; /* Repeat */
      ProcessList->Time  =* NegExp(2);
                                /*After NegExp(2) */
      break;
```

If the statements `Passivate` or `Wait` are executed, the corresponding process record is unlinked from the process list. Thus the reference `ProcessList->` yields quite another process and must not be used after execution of those procedures. `Passivate` requires that the next event is set.