Chapter 1

Basic Concepts and Tools

1.1 MODELING AND SIMULATION: WHAT IS IT?

To put it simply, computer simulation is a process of making a computer behave like a cow, an airplane, a battlefield, a social system, a terrorist, a HIV virus, a growing tree, a manufacturing plant, a mechanical system, an electric circuit, a stock market, a galaxy, a molecule, or any other thing. This is done with a specific purpose, mainly in order to carry out some "what if?" experiments over the computer model instead of the real system.

It is known that the best investment one can make is to invest in the field of high technology and basic research, though the benefits may not be immediate or easy to evaluate. One of such fields is computer simulation software. Recall that *simulation* is used to observe the dynamic behavior of a *model* of a real or imaginary system. The subtitle of the International Journal SIMULATION (the most important publication on simulation methods and applications) is *For Understanding* (see http://www.scs.org). Indeed, when we simulate a complex system, we are able to understand its behavior at low cost. Otherwise, we would have to carry out a complicated theoretical research or to build a device (an electric heater, a building, or an airplane), and observe how it crashes to get hints for improvements in the design.

There are many definitions of computer simulation. For example, A. Alan B. Pritsker (1984) defines it as *the representation of the dynamic behavior of the system by moving it from state to state in accordance with well-defined operating rules*. Bernard P. Zeigler in his book (1976) writes: *We can therefore define simulation as the technique of solving problems by the observation of the performance, over the time, of a dynamic model of the system*. By a *system* we mean a set of components that are interrelated and interact with each other. These interrelations and interactions distinguish the system from its environment. The system is supposed to be organized in order to perform one or more functions or to achieve one or more specific goals. The commonly mentioned properties of systems include:

Aggregation, which means that systems can be grouped into categories that can be nested into larger aggregates.

Nonlinearity – A system need not be linear, i.e. it does not necessarily satisfy the principle of superposition. In other words, the behavior of a system cannot be derived from the sum of behaviors of its components. Consult Holland (1995).

Many authors define simulation in a similar way, with emphasis on the changes of the modeled system state in time. In a somewhat more general way, we can define *modeling as the relation between real systems and models*, *and simulation as the relation between models and computers*.

In the article of Pritsker (1979), you can find about 30 different definitions of computer simulation. Most of the authors tend to define simulation in terms of the software they develop or use. However, the common approach is to relate simulation with the changes of the model state in time. Ralph Huntsinger, past president of the Society for Computer Simulation, always says that "*Simulation is Fun"*. This is true, taking into account the interdisciplinary aspect of computer simulation. A specialist in simulation must learn how planes fly, how rice grows, how shoes are produced, how AIDS is spread, how a legal process works and how galaxies are formed among many other things. This is the FUN you find dealing with computer simulation.

The first step in any modeling and simulation (M&S) task is to construct a system model. As Bratley, Bennet, and Schrage (1987) say in their book: *A model is a description of some system intended to predict what happens if certain actions are taken*. This is perhaps one of the most essential definitions of what a model of a real system is, although it needs some additional explanations. Many years ago, A. Rosenblueth and N. Wiener pointed out that modeling is one of the central needs of scientific reasoning. When we deal with models we must take into account many factors, like the level of simplification, experimental frame, model validity, tractability, credibility, and the aim of modeling among others. Bernard P. Zeigler (1976) gives some basic concepts on modeling. First of all, we must define the *model components* as elemental parts of the model, like clients in a bank, ships entering a harbor, cars on a street etc. Each component is described by the descriptive variables that include input, state, and output variables. The set of all descriptive variables in a model forms the *experimental frame*. The same real system may have several different experimental frames. Each experimental frame results in the corresponding simplified model, as shown in Figure 1.1 (following Zeigler, 1976).

Here the *basic model* is that which exactly reflects the behavior of the real system. Such a model normally does not exist.

Note that the aim of the modeling task, as well as the technical limitations (the computer on which we want to run the resulting simulation program) reduce the number of possible simplified models. This helps us select the appropriate simplification. If there is more than one simplification that satisfies these criteria, we must apply other selection rules (e.g. modeling cost).

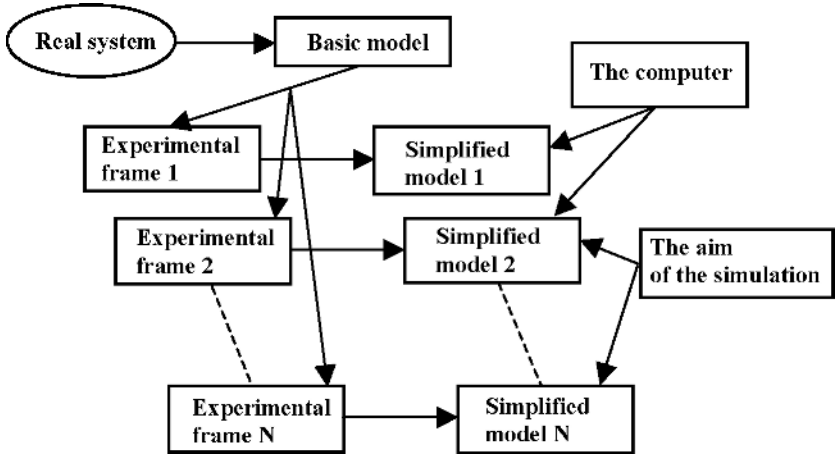

Figure 1.1 Real system, basic model, experimental frames and simplified models.

If no model exists satisfying our aim and technical limitation, then no simulation is possible. Remember that looking for something that does not exist is simply a waste of time. Also note that the same real (or imaginary) system can have several different experimental frames and several simplified models. For example, while modeling an electric circuit, the common experimental frame is the set of the voltages and currents on the corresponding circuit components. But, for the same circuit, someone can define the experimental frame as the set of all voltages and currents, power dissipated on each element, the temperature of each integrated circuit and of the printed-circuit plate, as well as the intensity of the electromagnetic field produced by the circuit. The first frame suggests the use of an appropriate package for circuit simulation, while the second one implies the use of a circuit simulation package, as well as sophisticated heat transfer and electromagnetic wave software.

1.2 VALIDITY, CREDIBILITY, TRACTABILITY, AND VERIFICATION

The concepts of *model validity*, *verification*, *credibility*, and *tractability* are of great importance in any M&S task.

Model *validity* is one of the central problems in modeling and simulation. Before discussing this concept, recall the concept of the system input and output variables. Roughly speaking, by the input we mean all external signals like electric excitations, control signals, or orders that come from the environment. The output variables are those values that we want to observe, measure, store, print, or plot as the result of a simulation run. The concept of input and output comes from the problems of signal processing, automatic control, and similar fields. However, not all systems must be causal, *and the input and output concept may not work*. For example, an electric resistor treated as an isolated, stand-alone system has no input and no output. One can define the input signal as the voltage applied to the resistor

and the output as the resulting current. But the same resistor in another circuit (context) can have forced current (connected to a current source) as the input signal, the resulting variable being the voltage (model output). A new approach to M&S of physical systems rejects the input–output concept, which means that from this point of view, physical systems are not causal (see Cellier, 1993). Other aspects of model descriptive variables and formal model definition will be discussed in the section devoted to the DEVS (Discrete Event Specification) formalism.

But now let us come back to causal systems with input and output signals well defined. Consider a real dynamic system and its model. Let S be the operation of modeling (passing from a real system to its model). By $x(t)$ we will denote the system state in the time instant t, by $y(t)$ the system output, by f the state transition function that maps the state $x(t)$ and input over $[t, t + h]$ into the new state $x(t + h)$. The same letters with suffix s denote the corresponding items for the model. Our model is said to be valid if and only if the diagram of Figure 1.2 commutes.

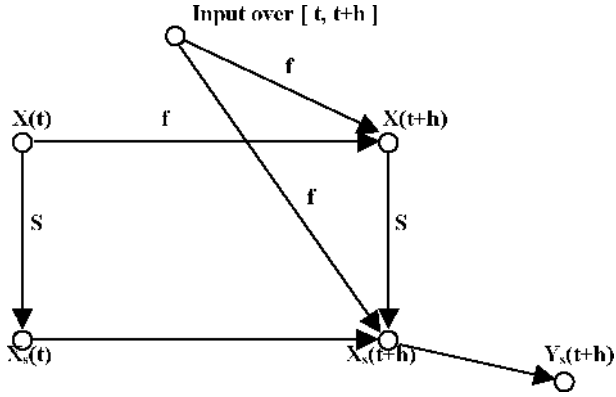

Figure 1.2 Model validity.

In other words, starting with $x(t)$ we must obtain the same model output $y_s(t + h)$ separately from the way we choose. This must be satisfied for *any* possible initial state and input.

The above definition of model validity is somewhat difficult for practical applications. A more practical concept is the *input–output* or *external* validity that can be illustrated by the scheme shown on Figure 1.3.

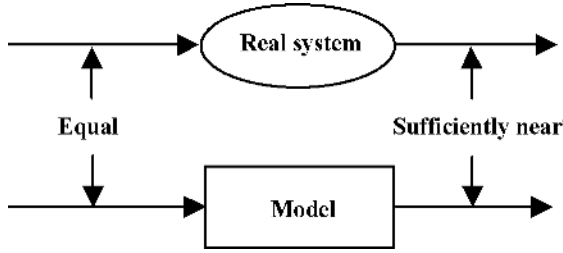

Figure 1.3 Validity "input-output".

The model is supposed to be *I/O valid* if the outputs from the model and from the real system are "sufficiently" near. What "sufficiently" means is the individual judgment of the modeler.

Obviously, the above property must be satisfied for a long time interval, and perhaps for future model inputs. As we do not know what input signals will affect our model, the only way to check the I/O validity is to compare the model behavior and the real system output using some historic data.

Observe that according to the first definition (Figure1.2), *any approximation of a real continuous system by a model with discrete time is invalid*. Indeed, in a real system, a disturbance may come within the time period of discretization, as it is undetected by the discrete-time model. Even if the integration step of our numerical method is small, the input signals may have high frequencies, and the aliasing phenomenon may result in completely wrong results. Another source of error is the time-discretization itself that implies its own errors. The most direct way to make the discrete-time model more exact is to decrease the sampling time for the discrete model. However, there are limitations for such actions, mainly caused by the computer resolution and limitations of a numerical method applied for trajectory integration. In any case, the modeler must look for a reasonable compromise and not forget that what he or she is creating is nothing more than an illusion. Another useful and simple test that may be used to qualify a model as invalid is to prove that the model does not explain certain known properties of phenomena that we observe in the real system.

Invalid models are often results of wrong assumptions. Creating a simplified model, we idealize the real system, which may result in physically wrong models.

Consider a simple example of an electric circuit shown on Figure 1.4.

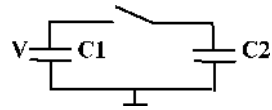

Figure 1.4 An electric circuit.

Let the initial voltage for the capacitor C_1 be equal to V, and initial voltage for C_2 equal to zero. After closing the switch, the two voltages obviously become equal to each other and equal to $V/2$, provided $C_1 = C_2 = C$. Now, consider a simplified model of the circuit (Model 1). We assume that all elements are "ideal", which means that the capacitors have no internal resistance or inductance, the same assumption being taken for the conductors ($R = 0$). Now calculate the initial energy of the circuit. It is equal to $(CV^2)/2$. After closing the circuit, the total energy is $2(C(V/2)^2/2) = (CV^2)/4$. Where did half of the initial energy go? Our model cannot explain this. In other words, *the model is invalid*. The very simple reason is that it is too simplified. Now, consider Model 2, where the connecting wires have resistance R. One could expect that Model 1 is a limit case of a sequence of models of type Model 2, with R approaching zero. Unfortunately, this is not the case. It is easy to show that *half of the energy is always being dissipated in the wire resistance during the transient process*. Moreover, this energy does not depend on the value of the resistance. This means that Model 1 is *not* the limit of a sequence of models (Model 2), with R approaching zero(does not converge to Model2). In more "mathematical" terms, we can say that in the space of all models of type 2, Model 1 (point of models space) with $R = 0$ is a discontinuity point (or singularity). Also note that the model validity depends on the actual experimental frame Modeling. The first model of our circuit is valid if the experimental frame only includes the (static) voltages. However, if we add the system energy to the experimental frame, this model becomes invalid and we must look for another one. Here we only talk about the energy of the electric field and the dissipated energy. In a more complete model we should also take into account the energy of the electromagnetic field. In fact, if we neglect the wire resistance and take into account the inductance of the loop (with ANY value different from zero), than we will see that the circuit will always oscillate, exchanging the energy between electric and electromagnetic fields.

Other remarks on validity can be found in Section 3.1.

Model *credibility* is another important aspect of M&S. Rather, this is the question of the relation between the target user and the model implementation. The model can be valid and well implemented, but this does not mean that the user believes in it. If he/she does not, the whole project is a waste of time, because the user will not use it to solve his problems (design, decision making etc.). A good way to make the model credible for the user is to involve him in the process of model building. If the team that works on the model includes one or more people from the company the model is created for, it is more likely that the model and the resulting simulation experiments will be credible and accepted by the company.

The *model verification* is rather a technical problem, related to the computer implementation. Verification in simulation tasks is nearly the same as testing in any other software development process. Sometimes testing is even more costly and time-consuming than the model development. To perform good testing in any complex software project, it is recommended the test team be separated from the developer team. The goals of the two teams are opposite: a tester wants to find

bugs in the software, while the developer wants to prove that the software is error free. The interactions between the developer and the tester team result in improving the software quality. Remember the law of Murphy! If a complex software package has been well tested, so that the probability of any malfunctioning is nearly zero, and if it contains a small bug, then the first thing the user will do is to introduce data or action that activates this "improbable" erroneous functioning.

The technical limitations are mainly imposed by available hardware. It might appear that the rapid growth of the computing capabilities of our tools implies that what is impossible to implement now, will be quite possible in the near future. However, some problems are so complicated that they are *computationally intractable*, whatever the computer speed will be in the future. It is well known in operations research that some problems that have a nice mathematical description cannot be solved using the hardware we actually have. The same occurs with models prepared to be run on a computer as a simulation task. Some models are computationally intractable, which means that they are too time-consuming or expensive to be realized.

A modeling and simulation task is intractable if its computational complexity increases faster than exponentially or factorially with the number of its descriptive variables. The computational complexity can roughly be defined as the minimal cost of guaranteeing that the computed answer to our question (simulation task) is within a required error threshold.

A mostly cited example of an intractable problem is the salesman problem, namely, the simulation of all possible routes of a salesman that must visit various cities. The problem is to find the route with the shortest distance possible. This problem can be treated with many suboptimal algorithms known in the operation research field. However, we can only obtain a suboptimal solution even with a small number of cities, without knowing if it is really the optimal one. Other examples in continuous system simulation can be found in fluids dynamics applied to problems, like the reentry of a space shuttle to the atmosphere of the earth (modeling the airflow around the craft, and its thermodynamics).

For more discussion on intractability, consult Traub, and Wozniakowski (1994) or Werschulz (1991).

1.3 SYSTEM STATE AND CAUSAL SYSTEMS

Let $U\{s, t\}$ be the input to the system over the interval $[s, t]$, where s and t are two time instants, s $X(t) = F(t, U\{s, t\}, s, X(s))$

In other words, it is necessary that the system state at a moment t can be calculated using some past state in time instant s and the input function over the interval between the two moments of time. For example, the state of a system that has one spring, one mass, and one damper linked together is given by the mass position and velocity. All other descriptive variables of this system are parameters, inputs (e.g. external forces), or some output functions defined by the modeler.

In the case of an electric circuit composed by any number of resistors and one capacitor, the state is a (scalar) value of the capacitor voltage. In this case all the

currents in the resistors can be calculated provided we know the initial capacitor voltage and the external excitations (input signals). The system state may be a scalar, a vector, or an element of a more certain abstract space. For example, the state of the model describing the changes of the temperature distribution inside a piece of metal belongs to the space of all differentiable functions of three variables defined in the region occupied by the modeled body.

Now, let us define what we mean by the phrase "equivalent to", applied to functions. Obviously, two functions equal to each other in all points of a given interval are equivalent. In general, we treat two input signals as equivalent if they produce the same output. For example, if the system is an integrator, two input signals that are equal to each other on $[t_o, t_1]$, except a set of points of total measure zero, are equivalent. Another example is a sampled data system (e.g. a digital controller with an A/D converter at the input) with sampling period T. Two different input signals that coincide only at $t = 0T, 2T, 3T \ldots$ are equivalent to each other because the system cannot observe the values in time instants other than the sampling moments.

Consider a dynamic system whose state is X. Let us suppose that we can define input and output signals for this system as U and Y, respectively. The system is said to be *causal* if and only if for every t_1

$U_1(t)$ equivalent to $U_2(t)$ over an interval $[t_o, t_1]$, implies that $Y_1(t_1) = Y_2(t_1)$

where $Y_1(t_1)$ and $Y_2(t_1)$ are two outputs at the time instant $t = t_1$, obtained with inputs U_1 and U_2, respectively, t_o is a fixed initial time instant, and $X(t_o)$ is fixed. The output signal is supposed to be an algebraic function of the system state. We suppose that the system state in t_o is the same for the two possible trajectories.

An example of a noncausal system is as follows:

$$Y(t) = \int_{-\infty}^{\infty} p(t-s)U(s)ds$$

where $p(t)$ is a function different from zero everywhere. In this case the value of Y at the time instant t depends not only on the past, but also on the future values of p over the time. If we replace p by the Dirac impulse at zero, then the system becomes causal (in this case simply $Y(t) = U(t)$).

1.4 CLASSIFICATION OF DYNAMICAL SYSTEMS

To correctly select an M&S method and software we must know what kind of system we will be working with. The general classification given below can help us in this task. This is a known classification that frequently appears in texts on automatic control and physical system dynamics. Recall the notion of the equivalence class of input signals. As stated in the previous section, the signals belonging to the same class of equivalence produce the same effect. So, the number of classes of equivalence is the number of possible final system states. Now, we can classify

the dynamic systems as follows (SCE stands for "set of all classes of equivalence" of system input signals).

Finite automata. These are systems with a finite SCE. A simple example is an electric switch, with only two possible states. Another fairly complicated example is a computer, whose SCE is finite, though it may be a huge set.

Infinite automata. For this class the SCE is infinite, but enumerable. This means that for each class (and each system state) we can assign a different natural number. For example, a discrete model of an unlimited growth of a population belongs to this class. Also an unlimited queue, represented by the number of waiting clients, belongs here.

Systems with concentrated parameters. The systems of this class have infinite and not enumerable SCE. The power of the set of all possible system states is equal to the power of the set of all real numbers, so the states cannot be enumerated. This is possibly the widest class of models in continuous simulation. All models of electrical circuits with ideal discrete elements like capacitors, resistors, inductors, or transformers belong to this class. Also, mechanical systems with discrete components are of concentrated parameters. But, an electrical "long line" where the resistance, capacitance, and inductance are distributed along the line, does not fit into this class. The common model used to simulate systems of this class is the *Ordinary Differential Equation* (ODE) in one or *N*-(finite) dimensional space.

Systems with distributed parameters. For this class of systems, the SCE is infinite, and the power of the SCE set is greater than the power of the set of all real numbers. Recall that there are many known sets that are "greater" than the set of all reals. A simple example is the set *C* of all continuous real functions over [0, 1]. No one-to-one mapping from the set of reals to *C* exists. This is the reason why the simulation of systems of this class is so difficult and why we face such serious difficulties in the corresponding numerical solutions. Examples of some models of this class include fluid dynamics, heat transfer, diffusion process, dynamics of electromagnetic wave and spring vibrations among others. The mathematical model of such systems is normally given in the form of a *Partial Differential Equation* (PDE). The corresponding numerical solutions can be found using the numerical methods for PDEs, or the finite-element methods. Corresponding simulation software is rather expensive and the simulation programs are highly time-consuming. To simulate even a simple example in the three-dimensional case, we need several hours on a fast computer. Best results are achieved using systems with multiprocessing or supercomputers. *Do not confuse* the simulation of systems with distributed parameters with *distributed simulation*, which is a completely different thing (see Section 1.6.10).

The above classification can be helpful while deciding which simulation tool (language, package of algorithm) we should use.

1.5 DISCRETE AND CONTINUOUS SIMULATION

When one reads academic texts one may conclude that everything in this world can be divided into two or three parts, sometimes even more. Such divisions allow us to explain things in a comprehensive and ordinate way. However, this sometimes results in creating artificial divisions and limits between concepts, which, in fact, are not very different from each other and should not be separated. So, following this almost obligatory convention, the field of M&S has been divided into *discrete* (or *discrete event*) and *continuous* simulation. Note, however, that the real world does not divide into "discrete" and "continuous" parts. Consequently, the discrete/continuous division may lead to serious methodological errors. Roughly speaking, the discrete models assume that the events (changes of the model state) occur in discrete-time instants and that the duration of each event is equal to zero. In the case of continuous simulation, we assume that the model state changes continuously in time, and that the model time advances continuously. So, a common practice is to put the model in one of these extreme cases. My point is that *this is a fundamental conceptual error* committed in the M&S methodology. As for the discrete case, observe that discrete events do not exist in the real world. The execution of any event must take a finite time T to be accomplished. Passing with this time interval to zero is a risky operation, and can be done only if the case $T = 0$ is the limit case of a series of models with T approaching zero. Unfortunately, to be sure of this, we must prove that the case $T = 0$ is not a singularity in the space of all models of dynamic systems. My point is that in many cases this point IS a singularity and the mapping from the model to the corresponding space of results is not continuous at $T = 0$. Basically, this makes the discrete model invalid and false. For more details, see Section 3.1.

As for the continuous simulation, it is obviously an illusion because in a digital computer nothing is continuous, and any result on continuous simulation must be an approximation of the reality. Note also that, despite the declarations like *x: real* used in many programming languages, real numbers do not exist in a digital computer. The only way to really simulate a continuous system is to use an analog computer. These questions are discussed in the chapter "Continuous simulation". There is also a third category in M&S, called *combined simulation* or *combined models*. The combined simulation puts discrete and continuous parts of the model in one, integrated simulation tool. However, this is not a remedy for the problems mentioned above. Putting two invalid models together cannot result in a valid one.

Despite the above comments, I will keep using the terms "continuous" and "discrete" simulation in the following chapters. After all, this is a traditional classification of models and it turned out to be practical and useful while developing simulation software.

1.6 EVOLUTION OF SIMULATION SOFTWARE

Simulation languages and packages do not form the main topic of this book for two reasons.

First, even if we only select the most relevant M&S tools, it is impossible to mention them all. To see the characteristics of some selected software tools the reader can consult the Directory of Simulation Software edited each year by *The Society for Modeling and Simulation* (formerly *The Society for Computer Simulation International*, SCS). There you can find about 150 software packages, selected as the most relevant and useful. Similar directories are published by the OR/MS (Operation Research and Management Science journal) and by the Chemical Engineering Progress Journal. When you navigate through the Web you will find millions of pages dedicated to computer simulation software, so it is difficult to find what you really need. The prices of simulation packages oscillate between US $25 and tens of thousands of dollars. The price is normally related to the size of the package, but there is little relation between the price and the software quality and its usefulness. Simulation software is not a merchandise in massive demand, so it is rather expensive. The normal price of a good simulation tool with 3D graphics and animation offered by known software companies is about US $1000 or more per licence.

Second, a book on computer simulation should not be a manual of any simulation package. The lifetime of a book is longer than the lifetime of many software tools, so a book dealing with a particular simulation software may quickly become obsolete. However, some tools are mentioned here in more detail. This is done in order to illustrate some general concepts, and not to make any emphasis on the mentioned software.

Simulation software appeared shortly after the first digital machine was constructed. Let us briefly comment on some examples of simulation tools that may be considered as milestones in the M&S software evolution.

The control and simulation language (CSL) language is mentioned mainly for historical reasons. General Purpose Simulation Language (GPSS), Simula67 and DYNAMO are treated separately, for the great importance of these tools. Some other software items are commented on. However, we do not pretend to talk about a complete list of M&S software that includes hundreds of languages and packages here. As stated before, such lists are available from the SCS (The Society for Modeling and Simulation Int.), OR/MS Journal, Chemical Engineering Progress and other sources that publish simulation software directories. As for the Internet, see the ODP (Open Directory Project, www.dmoz.org) category Science-Software-Simulation or the Informs of College on Simulation, a part of the Institute for Operation Research and Management Science (www.informs-cs.org).

1.6.1 CONTROL AND SIMULATION LANGUAGE (CSL)

CSL was one of the first simulation languages, developed in the 1950s. In fact, the name is somewhat confusing; the language has no capabilities for automatic control or similar tasks. It is a discrete simulation language that implements the Activity Scanning (AS) simulation strategy.

CSL is obsolete and has not been used recently. However, for historical reasons, some CSL concepts are described here in more detail. The language was based on

Fortran. Compared to Fortran, it has three new elements: the clock mechanism, entity class definition and sets of entities.

The clock mechanism manages a set of variables named time variables. These variables store various time values. They are subject to the clock algorithm, and may be used in user-defined logical expressions. The time named CLOCK starts with a value equal to zero. Then, all possible model activities are scanned, according to Figure 1.5.

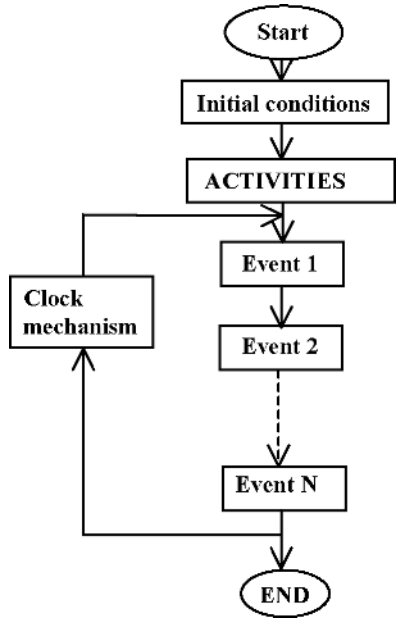

Figure 1.5 The flow diagram of a CSL program.

The events are defined by the user and normally have the structure shown in Figure 1.6.

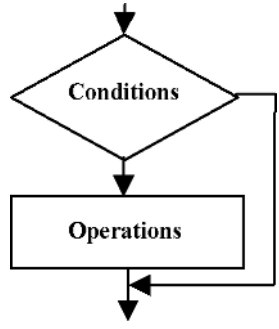

Figure 1.6 CSL event.

In the conditional part of the event, the user should use one or more *time variables*. These variables can be independent or attached to model components. They store certain values of the model time. The main condition of the conditional part should be TX EQ 0 (in Fortran notation this means "TX equal to zero"), where TX is a time variable. So, only the events with corresponding time variables equal to zero will be executed.

After terminating AS, the CLOCK is set equal to the value of the nearest time variable greater than the CLOCK (this means to the time instant of the next possible event). After this, all time variables are being decreased by the same time interval. This way, in the next time loop, the next event(s) will be executed.

The following line is an example of a CSL class definition.

```
CLASS TIME CLIENT SET OUTSIDE, QUEUE, SERVICE
```

This defines a class named client (e.g. clients in a bank). The word TIME indicates that each client has its own time variable. A client may belong to the sets OUTSIDE, QUEUE (waiting to be served) or SERVICE (being served). The reference T.CLIENT.K can be used to refer to the time variable of the client number K.

The program code only includes the initial part (initial conditions and the word ACTIVITIES), and then the event codes. The order the user puts the events in the program is not relevant (except possible simultaneous events). The clock part is hidden and runs automatically. The code of an event should contain a conditional part and some operations. The operations are NOT executed if the conditions fail. For example, the arrivals of a client can be described as the following event code:

```
BEGIN
T.ARRIVAL  EQ  0
T.ARRIVAL = NEGEXP(20)
FIND K OUTSIDE FIRST
CLIENT K FROM OUTSIDE
CLIENT K TAIL QUEUE
```

Note that this event reschedules itself to be executed after a random time interval (with exponential distribution, mean equal to 20 time units). The main condition is T.ARRIVAL = 0, where T.ARRIVAL is a time variable. FIND finds a client being outside the bank. The client is taken from outside and put as the last one waiting in the queue. The instruction FIND also returns a logical value. If it fails (there are no clients outside), the following "operational" part of the event is not executed.

As mentioned above, this description of CSL is only given here for historical reasons. CSL is now obsolete and completely forgotten. However, some important simulation concepts were introduced in this language, perhaps for the first time in the history of computer simulation. In the decade of the 1960s I had been working with CSL at the Computer Center of the Academy of Mining and Metallurgy in

Krakow, Poland. We successfully accomplished many simulation projects using this tool. Perhaps it sounds incredible for the recent generation of computer scientists, but the language was running on computers which had 32 kb (KILOBYTES!) of RAM, with no hard disk, and using magnetic tapes as the only storage facility.

1.6.2 STRATEGIES OF DISCRETE EVENT EXECUTION

The main contribution of CSL was the implementation of the clock mechanism, transparent to the user. This is why a simulation code in an appropriate simulation language is up to 20 times shorter than the equivalent code written in an algorithmic language like Pascal, C or Fortran. A discrete simulation language and its compiler are as good as the algorithm that manages the *event queue*. This event queue should not be confused with a queue we want to simulate, for example, a queue of clients in a mass service system or a buffer in a manufacturing system where the processed part can wait to be processed. The event queue contains a set of *event messages*, each of them telling which model event to execute and specifying the time instant when the execution will occur. The advantage of discrete simulation is that the model time jumps to the next (in time) event to execute, instead of advancing "continuously". This means that the system (the program, which controls the discrete event simulation) must know which the next event to execute is. There are two ways to achieve this. First, we can continue to maintain the queue sorted with increasing execution time and execute the first event from the queue. The second way is to, add new event messages at the end of the queue, and then look for the nearest event to execute. Both options involve the problem of sorting or scanning. This process is simple and fast if there are few events in the queue. However, if the model is not trivial (has more than, say, two queues and servers), the event queue can grow to hundreds of thousands of events, and the event handling strategy becomes crucial to the whole system performance.

Observe that the event queue in the simulation process constantly changes. Any event, while being executed, can generate one or more new event messages or cancel some of the existing ones. Moreover, there are events that cannot be scheduled through the event queue mechanism, being executed due to the changes of the model state and not because of the model time. Such events are called *state events* and must be handled separately.

There are three basic strategies in discrete simulation: AS, Event scheduling (ES), and Process interaction (PI). In this section we treat activity and event as synonyms. More advanced strategies are being developed and can be found in publications on the DEVS formalism. See Chow and Zeigler (1994) and Zeigler (1987).

AS was the first discrete simulation strategy, developed in the 1950s. One of the first implementations was the language CSL. According to this strategy, the model time is set equal to the time instant of the nearest event. Then, all model activities (events) are scanned and those that can be executed are executed, with the others remaining inactive. Next, the time jumps to the next possible event, and the whole process is repeated. This clock loop stops if no possible events remain in

the model. Obviously an event, while being executed, can schedule itself or other events to be executed in the future, so the event sequence can be long and complicated, even if the source program is relatively short.

The ES strategy is somewhat more effective. The event queue is created in the computer memory. Every component or event message of this queue stores the time the event will be executed, and the event identifier. So, the only problem is to maintain the event queue sorted according to the execution time. If we do this, we simply take the first event and execute it, without scanning all possible events. This event queue management is transparent (invisible for the user) and works automatically. The user can schedule events, but cannot redefine the model time or directly manipulate the event queue. The most effective event management algorithms are those using binary tree techniques to sort the event queue.

The PI strategy is more advanced. The model is defined in terms of processes that can run concurrently. The rules of interactions between processes are defined, while the specification of a process includes the necessary event scheduling. PI can be implemented in any object-oriented programming language, and became the main feature of Simula, Modsim, PASION, and other languages.

The Three-phase strategy in discrete simulation is a combination of these three strategies. The three phases are as follows. As a *state-event* we mean an event whose execution depends on the model state rather than the model time.

1. Model time jumps to the next event.
2. The event(s) scheduled to be executed at this time instant are executed.
3. All state events are revised. Those that can be executed, are.

Consult Lin and Lee (1993) and O'Keefe (1986).

1.6.3 GPSS

First introduced in 1961, GPSS is perhaps one of the most popular and widely known simulation tools. The name of the language does not reflect the field of its applications. GPSS is not a "general-purpose" language; rather, it is a discrete event simulation language. It is incredible that such an old and apparently obsolete tool is still used in its nearly original form and proved to be as useful as many other new M&S software. The popularity of GPSS is still very high. The search for "GPSS" on Yahoo! provides more than 40,000 pages related to the language. In fact, GPSS has some elements of object-oriented simulation, though it is not an object-oriented language like C++ or Delphi Pascal. The general idea of GPSS is to manage *facilities* (permanent model resources) and *transactions* that are created, pass through the model and disappear. During its "life", the transaction passes through several GPSS *blocks*, each of them representing an event that is executed by the transaction. As a very rough analogy, using the terminology of the modern object-oriented languages, we can say that the transactions are instances of model objects and the blocks are the methods (see Figure 1.7).

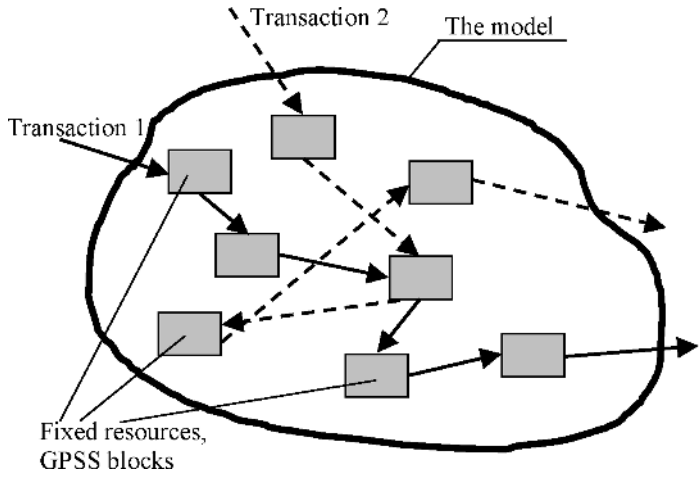

Figure 1.7 A GPSS model, transactions and resources.

A GPSS model takes the form of block diagrams. There are more than 50 block types available, each of them having a name and graphical symbol. Transactions may represent clients in a shop, cars on the street, parts to process etc. The model defines the "life" of a transaction while passing through the GPSS blocks. For example, a simple one-queue-one-server model is represented in Figure 1.8.

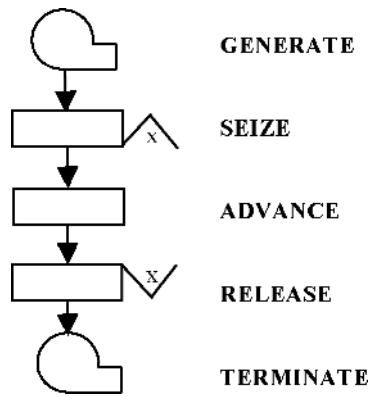

Figure 1.8 Graphical scheme of a GPSS program.

GENERATE block creates clients (transactions). The client occupies server *x* (SEIZE) and remains in the service for a time interval (ADVANCE), after which it releases the server and disappears, entering the TERMINATE block. If the server is occupied, a client that attempts to enter it waits in a queue.

After running the simulation, GPSS provides basic statistics like server occupation, idle time, maximum, minimum, and average queue length etc.

Figure 1.9 shows another example of a GPSS model. Now, there are five servers with corresponding queues. Arriving clients SELECT the queue with minimal length and the corresponding server. The result of the selection is stored in a halfword transaction attribute PH1.

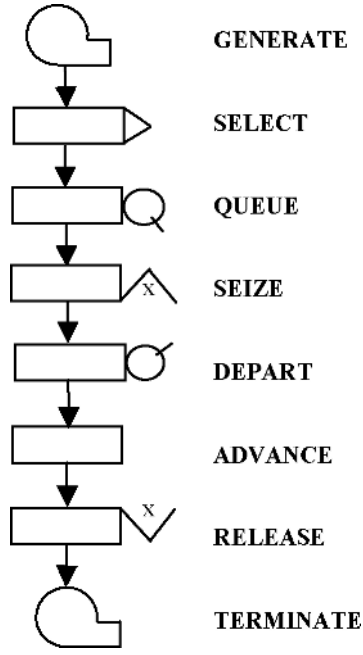

Figure 1.9 A GPSS program, multiple servers and queues.

The corresponding GPSS code is shown below:

```
SIMULATE
GENERATE   RVEXPO(1,0.2),,,,,1PH
SELECT MIN 1,1,5,,Q
QUEUE      PH1
SEIZE      PH1
DEPART     PH1
ADVANCE    1.05,0.3
RELEASE    PH1
TERMINATE  1
START 500
END
```

SIMULATE means the beginning of the simulation program. GENERATE generates clients with exponentially distributed time intervals between arrivals. The first parameter of GENERATE is the source of the random number stream (normally set to 1), and the second is the expected value for interarrival intervals. 1PH means that each client has one half-word attribute, referred to as PH1 in the following instructions. SELECT selects a queue and stores the result in the first attribute (PH1). The auxiliary word MIN means that it selects the queue (parameter Q) with minimal length. The second and third parameters define the range of queues (1–5). The QUEUE block is used to create statistics for the queue whose number (index) is stored in PH1. SEIZE is the beginning of the service. DEPART tells GPSS that the client departs from the queue. ADVANCE is the service time. It has the uniform distribution with mean 1.05 and width 0.6 (0.3 in the code means ±0.3). RELEASE means the end of the service (the server PH1 becomes free). TERMINATE terminates the client activities. START is the beginning of the simulation. The transaction counter is set to 500. Each execution of TERMINATE decreases this counter by one, so 500 clients are to be simulated.

Consult Gordon, 1975.

1.6.4 SIMULA67

This is an object-oriented high-level algorithmic language, developed by Ole-Johan Dahl, Bjorn Myhrhang, and Kristen Nygaard in the Norwegian Computer Center in 1967 (Dahl and Nygaard, 1967).

Simula67 is an extension of Algol60. The basic concept of Simula is an *object.* It is a structure composed of attributes and instructions. Schematically, the object is represented as follows:

```
   a1,a2,a3,....., an
< ------------------------, I >
   v1,v2,v3,.....,vn
```

where *a*1, *a*2,..., *an* are object attributes, *v*1, *v*2, *v*3,..., *vn* are the corresponding attribute values and *I* is the instruction list. An attribute may represent a single variable, a data structure or a procedure. The value of a data attribute is a value of corresponding type, while the value of a procedure attribute is the procedure body. The instruction list is a list of Algol instructions that is being executed when the object is created.

When created, the object executes the instruction from the list and then detaches itself, remaining inactive. For example, the following object may represent a point on a plane that can calculate its norm (a distance from the origin of the coordinate system):

```
   x, y, real procedure norm
< ----------------------------------, 0 >
      begin
```

```
        norm:=abs(x)+abs(y)
        end;
```

The object position is not defined by the object itself. The procedure norm is fixed as a "taxi" distance from the origin, and the instruction list is empty. While created, the object does nothing. It may be given the values of *x* and *y*. Once the object position is defined, the procedure *norm* can be called from outside the object to calculate its norm. The basic instructions that manage objects are *detach* and *resume*. The detach instruction interrupts the object activities (the object waits) and resume activates the object (it continues to execute its instructions). Consider the following example:

```
Producer
  ....x....
<----------,  begin >
            while true do begin
            ........
            x:=.......;
            stock:=x;
            resume(Consumer);
            detach;
            end; end Producer;

Consumer
  ....y....
< ----------, begin >
            detach;
            while true do begin
            ......
            y:=stock;
            stock:=0;
            ......
            resume(Producer);
            detach;
            end; end Consumer;
```

In this example *stock* is a global variable. After creating these two objects, the Consumer detaches itself and the Producer starts to execute his instructions. The dotted part of this example is other instructions. In the case of the Producer, it should "produce" something (variable *x*) and put it into the *stock*. Then, it activates the consumer and detaches itself. The Consumer takes the produced goods (variable *y*) from the stock and "consumes" them. Then, it activates the Producer and remains waiting.

This is a very primitive example, only to show how detach and resume can be used. In the environment of Simula67 there is a class Process where the clock

mechanism and event queue management are defined. It allows us to locate (schedule) the events in model time. In the Process class we should use the instructions *passivate* and *activate* instead of *detach* and *resume*, respectively. The *hold(v)* instruction makes the object wait during *v* model time units, so the object activities can be executed in proper model time instants.

The inheritance in Simula67 is realized by simple prefixed declarations. For example, *Process class Producer* can be the heading of the producer class. If so, the Producer inherits all the properties of a Simula process. This means that it can use the time-related instructions like hold and other mechanisms needed for dynamic modeling. The objects of Simula are instances of the corresponding class definitions.

Simula67 was perhaps the first (and best) object-oriented language with simulation capabilities. More than 30 years have passed since its creation, yet, it remains one of the most elegant and sophisticated object-oriented languages. The only disadvantage of Simula is its relation to Algol, which is not widely used, nor is it a practical programming language. Simula was another (along with CSL) simulation language used by our team at the Computer Center of the Academy of Mining and Metallurgy in Krakow in the late 1960s and 1970s. As I have mentioned before, this may sound incredible, but this complicated and highly algorithmic and object-oriented language had a good and relatively fast compiler working on the machines with 32 Kb (not Mb!) of RAM.

1.6.5 DYNAMO AND SYSTEM DYNAMICS SOFTWARE

The DYNAMO language and the methodology developed by Jay Forrester (1961) became milestones in the evolution of M&S. The resulting field of System Dynamics and the series of simulation packages developed after DYNAMO had a great impact on the "system approach" or "system thinking" (Checkland, 1981). Although the methodology is rather simple and may result in invalid models (see Chapter 8, "System Dynamics"), its simplicity made it possible to simulate systems that had never been simulated before. In many cases the lack of understanding of the dynamics of a company may lead to its failure, even if no evident errors in the company management have been committed. DYNAMO and the Forrester method permit a better understanding of the company dynamics and avoid errors in strategic planning.

DYNAMO was the first language supporting the Forrester approach (Forrester, 1961) to dynamic system modeling. It provided a powerful tool for a wide class of simulationists that needed an easy-to-use-modeling methodology and software. The name DYNAMO is recently used for several other, different software tools, like "dynamic logic programming" or the Russian Dynamo language (1993) for multiprocessing and Java-like classes. These tools have nothing to do with the original DYNAMO software.

A DYNAMO model is expressed in the form of ordinary differential or difference equations. The model is described in terms of flows and levels. The levels are integrals of the sum of flows affecting the level. The flows can be controlled by

algebraic functions of the model state (the levels), so any complex feedback system can be simulated.

In models of organizations, there are defined flows of information, material, orders, working power, capital, funds etc. The corresponding levels represent the actual state of the organization.

After DYNAMO, a series of similar tools have been created, most of them equipped with graphical user interfaces, which made them more user friendly. The most important one was the Stella package, followed by Powersim, VenSim, Ithink, and others.

DYNAMO and other System Dynamics tools belong to continuous simulation. However, all that family of languages should be treated separately, for their specific field of applications. Other tools have been developed independent of the System Dynamics largely for general use in ODE modeling. In the early 1960s, analog computers were still in use, primarily in the field of automatic control system design, signal processing, and general dynamic problems of engineering design. Consequently, the first continuous simulation languages were developed bearing in mind the analog schemes. Then, the developers put more effort on the implementation of numerical methods and on the ODE models. The most classic tools were Continuous System Simulation Language (CSSL) and Advanced Continuous Simulation Language (ACSL), mentioned in the following sections.

1.6.6 SPICE

The Simulation Program with Integrated Circuit Emphasis (SPICE) is a general-purpose, continuous electronic circuit simulation tool. SPICE was developed at the University of California at Berkeley between 1972 and 1975. The core of SPICE, coded in Fortran, is an advanced simulation program that includes the analysis of big electrical circuits including all known linear and nonlinear devices. In particular, the model of a transistor (BJT, FET, MOSFET etc.) is based on highly sophisticated mathematical models that simulate all the nonlinearities and physical properties of transistors.

After the first versions had been released, several enhancements were made to eliminate the long standing claims that SPICE is unfriendly and difficult to use. Those are:

- Schematic capture – The process of entering a circuit description into a computer by drawing a picture of the circuit.
- Monte Carlo analysis – The evaluation of circuit performance based on the statistical deviations of parameter tolerances.
- Circuit optimization – The ability to automatically select parameter values based on analysis findings and a particular circuit performance specification.
- Worst case – The finding and reporting of circuit parameter value(s) which will cause the poorest circuit performance based on some operational criteria.
- Parameter sweeping – The process of changing a given parameter's value over a user specified range and evaluating circuit performance as the parameter varied.

- Parameter extraction hardware and software – Tools used to develop SPICE model parameters for use in simulation.
- Graphics output and postprocessing – Process of reading SPICE output files to perform analysis operations on the output data. Operations such as integration, differentiation, fast Fourier transform, algebraic waveform manipulation, filtering, and other measurements of the voltage and current are included.

There are several types of analysis supported by Spice. The basic ones are direct current analysis, transient response, and frequency response.

- The direct current analysis provides a steady state (static) solution of the simulated network. This analysis is always performed before starting other types of simulations.
- The transient response mode; SPICE simulates the circuit and displays the transient processes. The circuit components may be linear, or nonlinear and the simulation is performed with the highest possible accuracy. SPICE automatically generates and solves the set of ordinary nonlinear differential equations of the simulated circuit.
- In the frequency response analysis mode, the circuit is first resolved in the DC mode. Then, all nonlinear elements are linearized in the neighborhood of the DC operation point. Once the model becomes linear, the state transition matrix is calculated and the frequency analysis performed. The results are presented in the form of the Body plots.

SPICE has huge libraries with thousands of models of electronic devices (diodes, transistors, ICs, SCRs, TRIAC etc.) stored along with their parameters. The format of SPICE libraries became a standard in electronic devices specification and they are accepted by many other circuit simulation packages.

1.6.7 DEVS: DISCRETE EVENT SYSTEM SPECIFICATION

From the mid-1970s, an advanced theoretical research on discrete event simulation has been conducted by Bernard Zeigler and his collaborators. The most known publication on the first results of this research is Zeigler's book (1976). The newest edition of the book appeared in 2000, with more results. It can be observed that, during the past two decades, no relevant progress has been made in commercial simulation software. The DEVS development still seems to be too advanced for software development companies to follow. However, research works like DEVS or DYMOLA (see the next section) are of great importance for future advances of simulation methodology. As stated in the Preface, our aim instead is to show some possible new directions, mathematical tools, and methods instead of focusing on the conventional ones. In the next chapters some of those proposals will be given. In this section you will find short characteristics of the DEVS approach.

DEVS is based on the formal M&S framework. It is derived from the mathematical theory of dynamical systems. DEVS supports hierarchical and modular compositions of models, and their object-oriented implementation. It supports both

discrete and continuous modeling paradigms, and exploits parallel and distributed simulation techniques.

The basic DEVS concept is the *atomic model*. It is the lowest-level model. It contains structural dynamics and model level modularity and is described by state transitions. On the next level is the *coupled model*, composed of one or more atomic and/or coupled models. This allows the hierarchical construction of more complex models. The theoretical properties of DEVS models are as follows:

- Closure Under Coupling
- Universality for Discrete Event Systems
- Representation of Continuous Systems
  - quantization integrator approximation
  - pulse representation of wave equations
- Simulation program correctness and efficiency

The experimental frame is one of the basic concepts in DEVS. This term was mentioned in Section 1.1. Recall that experimental frame is the specification of the experimentation to be done on a model. Frames represent the objectives of the experimenter, tester, or analyst. The hierarchical model construction allows maintaining repositories of reusable models and frames.

In the DEVS formalism, a model M is defined as follows:

$$M = < X, S, Y, \sigma_{\text{int}}, \sigma_{ext}, \lambda, \tau > \tag{1.1}$$

where X is the input space, S is the system state space, Y is the output space,

$$\sigma_{\text{int}} : S \to S$$

is the internal state transition function,

$$\sigma_{ext} : Q \times S \to S$$

is the external transition function, and Q is the "total state" defined as follows:

$$Q = \left\{ (s, e) \middle| \, s \in \mathrm{S}, 0 \le e \le ta(s) \right\}$$

Here e is the elapsed time since the last state transition. The term $ta(s)$ means the time of an internal state transition when no external event occurs.

$$\lambda : Q \to Y$$

is the output function, and τ is the time-advance function. The DEVS model (1.1) is also called the *basic model*. To treat complex models with variable structure, the Dynamic Structure Discrete Event System Specification (DSDEVS) is used. This formalism includes DSDEVN dynamic structure network, that is, the pair, where h

is the *network executive name*, and M_h is the model of h. We will not discuss the DSDEVS formalism here. In fact, it is not necessary because the DSDEVS network is equivalent to the DEVS model (1.1) and DSDEVS formalism is closed under coupling.

For more information on DEVS, consult Chow and Zeigler (1994), Zeigler (1976, 1986), and Srivastava and Rammohan (1994).

1.6.8 DYMOLA

In Chapter 1 you have found some comments on the causality concept. The comments were about the causality in physics, which is questionable. This lack of causality is the main assumption taken by F. Cellier and his collaborators while developing the new approach to M&S of physical systems, which resulted in the Dymola software. Let us give a citation from one of the courses on the simulation of physical systems, delivered by F. Cellier: *Maybe*, *thinking about cause and effect relationships helps the engineer conceptualize his or her task better or more easily*, *and therefore*, *it makes sense to pretend that the universe is a causal one*, *even though physics seems to indicate that the causality is a mere illusion?*

The heart of the Dymola system had originally been designed at the Lund Institute of Technology by Hilding Elmquvist in 1978, and further developed at the University of Arizona. The simulator, named Dymosim, had originally been developed at the German Aerospace Research Establishment in Oberpfaffenhofen in 1992.

The main point of the Dymola developers is that the classical and commonly used state-space representation of dynamical systems may lead to serious errors and result in invalid models. The right way to create models of physical systems is to follow the principles of conservation of energy and momentum, and not to blindly apply the space-state models.

In Dymola, the model elements are specified as objects. Each object has its descriptive variables, parameters, and governing equations. No cause effect is given. For example, the electric resistor is simply described by the Ohm equation for its current and voltage. The object-oriented mechanism includes class hierarchy and inheritance. Analyzing the model, Dymola instantiates all submodels (objects) from the user-defined model classes. Then, it extracts the equations from these objects, and expands them with the coupling equations that are being generated from the description of the interconnections between objects. The equations are then manipulated in order to obtain a computable simulation program. However, Dymola itself is not a simulation program and should instead be viewed as a program generator. Note that a very similar mechanism is used to simulate models given by bond graphs (described later in this book). Both the bond graph and signal flow module of the PASION system do exactly the same. However, the Dymola is much more advanced, mainly in its object-oriented character.

Consult: Cellier (1995), Elmqvist (1978).

1.6.9 CHRONOLOGY OF M&S SOFTWARE DEVELOPMENT

1952. The *Society for Computer Simulation (SCS)* was founded by John McLeod. Society activities, like conferences, publications, committees, and working groups, have been of great importance for the development of M&S methodology and software during the last five decades. SCS is a nonprofit international association of men and women who are leaders in their professions: people who are responsible for creating new developments and applications in the rapidly expanding field of computer simulation. The SCS has American and European chapters and covers with its activities in nearly all the countries of the world. The McLeod Institute for Simulation Sciences is a part of the SCS, dedicated to research and academic issues. Each year the SCS organizes several simulation conferences in different countries. The most known publications of SCS are the journal Simulation and the Transactions of the Society for Computer Simulation (consult http://www.scs.org)

The McLeod Institute of Simulation Sciences (*MISS*) is a part of The Society for Computer Simulation International (SCS).

The McLeod Institute was founded for the following reasons:

- To encourage and coordinate research and application in computer simulation;
- To seek and secure external funding for Institute activities;
- To enhance opportunities for undergraduate and graduate teaching, research and training;
- To stimulate the development, use, and evaluation of new instructional techniques and materials that make use of simulation.

The establishment of a center of the McLeod Institute of Simulation Sciences within a university or research center setting provides a mechanism through which faculty, students, and associates from various disciplines can bring their talents to bear in the general area of computer simulation or can apply simulation to new areas.

The Institute is named after Mr. John McLeod, P.E., the founder of the Society for Computer Simulation International and the founder and first editor of SIMULATION.

1961. Jay Forrester publishes his book “Industrial Dynamics”. Then, the DYNAMO language appears (see previous sections).

1961. GPSS is developed (see previous sections).

1964. The CACI Company, founded in 1962, releases SIMSCRIPT.

SIMSCRIPT II.5 is a powerful, free-form, English-like simulation language designed to greatly simplify writing programs for simulation modeling. SIMSCRIPT II.5 has been fully supported for over 40 years. The most important SIMSCRIPT features are:

DESIGN: A powerful world view, consisting of Entities and Processes, provides a natural conceptual framework with which to relate real objects to the model.

PROGRAMMING: SIMSCRIPT II.5 is a modern, free-form language with structured programming constructs and all the built-in facilities needed for model development. Model components can be programmed so they clearly reflect the organization and the logic of the modeled system. The amount of program needed to model a system is typically 75% less than in its FORTRAN or C counterpart.

DEBUGGER: A well designed package of program debug facilities is provided. The required tools are available to detect errors in a complex computer program without resorting an error. Simulation status information is provided, and control is optionally transferred to a user program for additional analysis and output.

EVOLUTION: This structure allows the model to evolve easily and naturally from simple to detailed formulation as data becomes available. Many modifications, such as the choice of set disciplines and statistics, are simply specified in the Preamble.

Contact: CACI Products Company, 3333 N Torrey Pines Ct, La Jolla, CA 92037, USA.

1967. Simula67 language is developed in the Norwegian Computer Center (see previous sections).

1967. Continuous System Simulation Language Committee of The Society for Computer Simulation (SCS) elaborates the standard in continuous simulation, named CSSL. This resulted in a whole family of languages, providing simulation software designed to assist engineers and scientists to mathematically model dynamic systems and to interactively conduct experiments on these models in order to gain insight into the model behavior. CSSL-IV is a version of CSSL that includes a complete simulation environment, which facilitates rapid model definition, accurate experimentation, and graphic display of data.

CSSL-IV provides advanced tools for nonlinear optimization, stiff equations integration, frequency response, spectral analysis, and linear algebra. It also supports bond-graph front end.

An important product fully compatible with CSSL is the ACSL, released by Mitchell and Gauthier Associates, Inc., 200 Baker Ave., Concord, MA 01742-2100, USA.

1975. SPICE package for electrical circuit analysis is developed at the University of California in Berkeley (see previous sections).

1979. A. Alan B. Pritsker develops SLAM, a general-purpose computer simulation language. SLAM relies heavily on GASP and Q-GERT, combining them to provide a single integrated framework. Developed by Pritsker and Associates, SLAM II is the most widely known version of SLAM.

SLAM is an advanced FORTRAN-based language that allows simulation models to be built, using three different worldviews. It provides network symbols for building graphical models that are easily translated into input statements for direct computer processing. It contains subprograms that support both discrete event and continuous model development, and specifies the organizational structure for building such models. By combining networks, discrete event, and continuous modeling capabilities, SLAM allows the systems analyst to develop models from a PI, next-event, or activity-scanning perspective. The interfaces between the modeling approaches are explicitly defined to allow new conceptual views of systems to be explored.

Consult: Pritsker (1984).

1983. First release of the ANSYS software, by ANSYS Inc. ANSYS is a finite-element analysis (FEA) package for distributed parameter systems, used widely in industry to simulate the response of a physical system to structural loading, and thermal and electromagnetic effects. ANSYS uses the finite-element method to solve the underlying governing equations and the associated problem-specific boundary conditions. The following list shows the date, product name, release, and physics capability for the components of the ANSYS software.

1983, ANSYS/EMAG3D, 4.4, Low Frequency, statics

1997, ANSYS/EMAG3D, 5.4, Low Frequency, statics and High Frequency

2000 Q4, ANSYS/EMAG3D, 5.7, Low Frequency, statics and High Frequency, ANSYS/Multiphysics, High and Low Frequency

2001 Q4, ANSYS/EMAG3D, 6.0, Low Frequency, statics, ANSYS/Multiphysics, High and Low Frequency

2002 Q1, AI*EMAX, 6.0, High Frequency

2002 Q4, ANSYS EMAG, 7.0, Low Frequency, statics, ANSYS EMAX, High Frequency, ANSYS Multiphysics, High and Low Frequency

2003 Q2, ANSYS Emag, 7.1, Low Frequency, statics, ANSYS Emax, High Frequency and coupled thermal, ANSYS Multiphysics, High and Low Frequency

2003 Q4, ANSYS Emag, 8.0, Low Frequency, statics, ANSYS Emax, High Frequency and coupled thermal, ANSYS Multiphysics, High and Low Frequency

1984. The Math Works, Inc., releases MATLAB. It is a high-performance, interactive numeric computation and visualization environment that combines hundreds of advanced math and graphics functions with a high-level language. The open system architecture enables users to view the prepackaged functions, customize these as needed, or add new functions. MATLAB toolboxes extend the power of the package by providing leading-edge algorithms and functions developed for signal processing, control system design, neural networks, optimization, and other applications. One of the most important parts of MATLAB is Simulink. This is a software package for the continuous simulation of dynamic systems represented by block diagrams. Simulink is a part of the Matlab software family.

The Simulink model is created by dragging block icons into the screen. No written code is needed. More than 200 built-in block types are provided. The user can also design his/her own blocks with custom icons. Principal applications: control systems, signal processing.

Contact: The Math Works, Inc., 24 Prime Park Way, Natick, MA 01760, USA.

Mid-1980s. Mechanical Dynamics, Inc. develops ADAMS (Automatic Dynamic Analysis of Mechanical Systems).

ADAMS is a simulation package that enables engineers to parametrically model 3D mechanical systems and study alternative designs as "virtual prototypes". ADAMS simulations are used to evaluate system performance, range of motion, collision detection, packaging, peak loads, and FEA input loads. Two-way interfaces with CAD, FEA, and control design packages are supported.

ADAMS version 10 is the world's most widely used software for mechanical system simulation. The ADAMS line of software products can help improve the design of anything that moves – from simple linkages and small electromechanical devices to complex vehicles, aircraft, and machinery. ADAMS operates within a simple four-step process:

Modeling: Within the easy-to-use interface, you can build a mechanical system model by importing data from CAD software or by starting from scratch, using the ADAMS libraries of part shapes, joints, motion generators, and forces.

Solution: Will your system function properly, avoiding lock-up positions and interferences with other parts or systems? Will the system movement and performance match specifications? Will the internal forces be too high? ADAMS can tell you. It automatically formulates and solves the equations of motion, and computes all displacements, velocities, accelerations, and forces.

Visualization: Want to see your system in action? With a click of a button you can examine x-y plots, or view fast, realistic animations superimposed, or in still-frame.

Design Optimization: Your system needs refinements? No problem. Change one or more variables, press a button and ADAMS will perform a simulation of the revised model and make comparisons. ADAMS' parametric design simulation capabilities allow you to evaluate an unlimited number of designs, and pinpoint the optimal design – before a single piece of metal is cut or cast.

To facilitate the virtual prototyping process, a number of ADAMS software modules are available:

ADAMS Full Simulation Package: including ADAMS/Solver and ADAMS/View.

ADAMS/Animation: to provide very-high-speed photo realistic animations.

ADAMS/Car: modeling and analysis of car suspension systems, and full-vehicle modeling of dynamic ride and handling studies.

ADAMS/Controls: sophisticated control systems can be developed without writing the mechanical system equations directly.

ADAMS/Driver: simulation of control actions of a vehicle driver such as steering, braking, accelerating, gear shifting, and clutching.

ADAMS/Engine: offers specialized utilities for valve train design and testing of timing mechanisms, belts, cranks, and pistons.

ADAMS/Flex: allows flexible parts to be created within ADAMS models from modal frequency data.

ADAMS/Hydraulics: helps you smoothly integrate system-level motion simulation and hydraulic system design.

ADAMS/Insight: allows you to plan and run a series of experiments for measuring the performance of your mechanical system design in operation.

ADAMS/Linear: linear and eigenvalue analysis, particularly when integrating mechanical simulation with control design software.

ADAMS/Pre: provides automakers and their suppliers with a suite of specialized preprocessing capabilities to quickly create a complete computer model of a full-vehicle or vehicle-subsystem design.

ADAMS/Rail: modeling and analysis of rail suspension systems and full-vehicle modeling of dynamic ride and handling studies.

ADAMS/Tyre: accurately predicts the influence of tyre characteristics on vehicle operation.

ADAMS/Exchange: automates the exchange of geometry data between ADAMS and CAD systems supporting any of a variety of standards including IGES, DXF, STEP, VDAFS, and STL.

MECHANISM/Pro: is seamlessly integrated within the Pro/ENGINEER environment, allowing users to easily convert assemblies created with PTC's Pro/ASSEMBLY software into realistic, fully 3-D mechanical system models by adding joints, forces, and motion generators from MECHANISM/Pro.

CAT/ADAMS: allows quick and easy transfer of CATIA-generated component and system geometry, as well as mechanism data from CATIA Kinematics, to the ADAMS solution engine for dynamic simulation of your complete mechanism in motion.

Contact: Mechanical Dynamics, Inc., 2301 Commonwealth Blvd., Ann Arbor, MI 48105, USA. In 2002, Mechanical Dynamics was acquired by MNS Software Company.

1986. CACI Company releases Modsim II. This software provides libraries of objects that can be reused while creating new models. There is a graphical model editor that allows the user to graphically place the object icons and add parameters through dialog boxes.

1988. The first release of the package *Mathematica*, by the Wolfram Research led by Stephen Wolfram. This is a fully integrated environment for technical computing. The concept of *Mathematica* was to create a single system that could handle all the various aspects of technical computing in a coherent and unified way once and for all. For the first time, there was a new kind of symbolic computer

language that could manipulate the very wide range of objects involved in technical computing by using only a fairly small number of basic primitives. Though this is not exactly a simulation package, the importance of the program for simulation applications cannot be overlooked. *Mathematica*'s impact was felt mainly in the physical sciences, engineering, and mathematics.

Mid-1990s. SWARM package.

The basic objects in the Swarm system are *agents*, the simulated components. A schedule of discrete events on these objects defines a process occurring over time. In Swarm, individual actions take place at some specific time; time advances only by events scheduled at successive times. A schedule is a data structure that combines actions in the specific order in which they should execute. For example, the coyote/rabbit simulation could have three actions: "rabbits eat carrots", "rabbits hide from coyotes", and "coyotes eat rabbits". Each action is one discrete event: the schedule combines the three in a specific order, e.g. "each day, have the rabbits eat carrots, then they hide from the coyotes, then the coyotes try to eat the rabbits". The passage of time is modeled by the execution of the events in some sequence.

Consult: Minar, Burkhart, Langton, and Askenazy (1996), and the page is http://www.santafe.edu/projects/swarm/overview/overview.html.

1996. MODSIM III is released by the CACI Company. It is an environment for building advanced models. It generates readable C++ code, so the user can access libraries written in C++. The system under study is described as a collection of interacting objects. Each object module describes the object behaviors, called *methods*. Inheritance allows new objects to be defined as a special case of one or more general classes. The MODSIM concepts are similar to those of SIMULA, PASION, and other object-oriented programming languages. The advantage of MODSIM is that it generates C code (Simula is based on Algol and PASION generates Pascal code).

1996. Integrated Systems company releases MATRIXx, a package for design automation and simulation. It is oriented to automotive, aerospace, process control, servo design, and others. It includes Xmath, an object-oriented analysis and visualization tool with a user-programmable GUI.

Contact: Integrated Systems, 3260 Jay Street, Santa Clara, CA 95054, USA.

1.6.10 DISTRIBUTED SIMULATION

While there are many ways that simulation speed can be improved, clearly the most fundamental is to move from sequential to parallel execution. This means that the simulation task is distributed between multiple processors at the same mainframe or in separate computers running in a network. The fundamental issue is the extent to which simulations should be synchronized in a conservative manner (i.e., without rollback) as opposed to an optimistic manner (i.e., with rollback).

In distributed simulation, each processor runs with its own model time clock. Several distributed simulation techniques have been developed, like the Chandy-Misra algorithm (Chandy, Holmes, and Misra (1979)), and the Time Warp algorithm (Jefferson and Sowizral (1985)). The former is pessimistic or conservative, advancing the processor simulation clocks only when conditions permit. In contrast, Time Warp assumes the simulation clocks can be advanced until conflicting information appears; the clocks are then rolled back to a consistent state.

Such a conflictive situation arises when one of the local clocks (say, that of processor A) advances more quickly than some other (processor B). If the object simulated on processor B sends a message to that of processor A, the message arrives in the past of the receiving object. But every received message can change the object behavior, so the object on A must return to the (past) moment of the message and repeat its, perhaps different, trajectory. The situation becomes complicated if the object A has issued messages to other objects after the conflictive model time instant. All these messages must be cancelled and the receiving objects must also roll back their clocks. This may result in a chain of rollbacks, and make the simulation even slower than in the pessimistic mode.

Distributed simulation has no application on single machines and PCs. It is being implemented on supercomputers and computer networks, to a certain extent. Applications mostly belong to aircraft simulation, communication networks, defense strategy, VLSI(Very Large Scale Integration) chips, and similar great-scale models.

1.6.11 HIGH LEVEL ARCHITECTURE (HLA)

HLA is a standard for constructing distributed simulations. It is intended to facilitate interoperation between a wide range of simulation types and to promote the reusability of simulation software. HLA will encompass virtual, constructive, and live simulations from the training, engineering, and analytic domains, including simulations from the Distributed Interactive Simulation (DIS) community, such as logical time and aggregate level simulations. A group of simulations interoperating under HLA is called a *federation.* The HLA defines the rules of interoperability, a semiformal methodology for specifying simulation and federation object classes, and the interface specification that is a precise specification of the functional actions that a simulation may invoke. Parts of HLA are the Run Time Interface (RTI) and Application Programming Interface (API) through which the simulation of a federation (*federates*) communicates.

The High Level Architecture (HLA) provides an architecture for modeling and simulation in order to encourage the interoperation of simulations and the reuse of simulation components, and to encompass a broad range of simulations, including categories of simulations not previously addressed within the distributed simulation community. The challenges involved in doing this include enabling continuous, real-time simulations to interact with time-stepped or event scheduling simulations. The HLA is being developed by the US Department of Defense (DoD).

The basic HLA concepts are the federation, that is a named set of interacting federates, and the federate which represents a member of a HLA federation. All applications participating in a federation are federates. In reality, this may include Federate Managers, data collectors, live entity surrogates simulations, or passive viewers.

An HLA Simulation Object Model (SOM) is created as an individual federation member. The Federation Object Model (FOM) describes a named set of multiple interacting federates. In either case, the primary objective of the HLA Object Model Template (OMT) is to facilitate interoperability between simulations and reuse of simulation components.

The primary purpose of an HLA is to provide a specification of the exchange of all public data among federates in a common, standardized format. Thus, the components of an HLA establish the "information model contract" that is necessary to ensure interoperability among the federates. The content of this FOM describes an enumeration of all object classes chosen to represent the real world for a planned federation; a description of all interaction classes chosen to represent the interplay among real-world objects; a specification of the attributes and parameters of these classes; and the level of detail at which these classes represent the real world, including all characteristics. Every object found in a federation execution is an instance of an object class that has been defined in the FOM. The FOM also allows for interaction classes (e.g. an explicit action taken by an object, that can optionally be directed toward another object) for each object model. The types of interactions possible between different classes of objects, their affected attributes and the interaction parameters are specified.

The ownership management group of services allows federates to transfer ownership of object attributes. An attribute is defined as a named portion of the state of an object. Owning an attribute gives a federate the privilege to provide new values to the federation execution for that attribute.

The main difference between distributed simulation and HLA is the fact that the simulations do not have direct connectivity with each other. They do not "speak" to each other at all. Rather, they communicate with the Run Time Infrastructure (RTI) which then communicates with the other simulations.

The following glossary explains some of the HLA terms. **attribute** – A named portion of an object state. **attribute ownership** – The property of a federate that gives it the responsibility to publish values for a particular object attribute. **federate** – A member of an HLA federation. All applications participating in a federation are called *federates*. In reality, this may include Federate Managers, data collectors, live entity surrogates simulations, or passive viewers. **federate time** – Scaled wallclock time or logical time of a federate, whichever is smaller. Federate time is synonymous with the "current time" of the federate. At any instance of an execution, different federates will, in general, have different federate times. **federation** – A named set of interacting federates, a common FOM, and supporting RTI, that are used as a whole to achieve some specific objective. **federation time axis** – A totally ordered sequence of values where each value represents an instant

of time in the physical system being modeled, and for any two points T1 and T2 on the federation time axis, if T1 < T2, then T1 represents an instant of a physical time that occurs before the instant represented by T2. Logical time, scaled wall-clock time, and federate time specify points on the federation time axis. The progression of a federate along the federation time axis during an execution may or may not have a direct relationship to the progression of wallclock time. **FRED** – The Federation Required Execution Details (FRED) is a global specification of several classes of information needed by the RTI to instantiate an execution of the federation. Additional execution-specific information needed to fully establish the "contract" between federation members (e.g., publishing responsibilities, subscription requirements, etc.) is also documented in the FRED. The set of management requirements provides one source of input to the FRED specification, which will be recorded in a standardized format.