# 1 Modeling and Simulation

## A. Introduction

Science and engineering is all about studying systems, be it natural or man-made, in order to understand them, to predict their behavior, or to design new systems and/or improve existing designs. There are many ways to study a system, and the traditional approaches are through theoretical analysis or physical experimentation on the systems themselves or physical models of the systems under study. With the development of digital computers around the second half of the previous century, a new and very powerful paradigm to study a system was introduced, that of modeling and subsequent simulation on a computer. With the ever increasing speed of computers, and introduction of new computing paradigms, such as parallel- and distributed computing, [1] the use of computers as an "experimentation environment" has become increasingly important in a wide area of applications. The field of Modeling and Simulation is now very broad, and is applied in many disciplines ranging from the social sciences (to e.g. study group behavior) to the natural sciences (e.g. fluid flow modeling, etc).

The field of *Scientific Computing* provides the very important and necessary glue between on the one hand conceptual or mathematical models of a system under study, and on the other hand a simulation of such a model on a computer. Scientific Computing is the *mapping* between a model and an actually running simulation. As such, Scientific Computing is inherently interdisciplinary, joining application fields with numerical mathematics and computer science and is part of the larger field of *Computational Science*. This course, *Parallel Scientific Computing and Simulation*, gives an introduction to the fields of Scientific Computing and Simulation, with a strong focus toward parallel computing. The latter is because parallel and distributed computers have become the hardware of choice for large scale (i.e. compute intensive) simulations. Moreover, parallel computing includes the field of sequential computing (in the obvious limit of $p$, the number of processors, being equal to one).

The lecture *Simulation and Modeling* [2] has given you an in-depth introduction into this very important field, and here it is assumed that you are familiar with the concepts and ideas as introduced in that lecture. Here we will just give a very condensed repetition of the most important concepts, and we strongly suggest that you go back to your lecture notes to refresh your memory. Moreover, the ideas and concepts as introduced in the lecture *Introduction Parallel Computing* [1] are assumed well known and are not repeated here. The main part of this chapter will cover a more in-depth discussion of a specific type of models, the *Discrete Event* models and discuss at length *Parallel Discrete Event Simulations* (PDES).

## B. Modeling and Simulation, a short review

### B.1. How to Study a System

A system can be anything. It can be blood flow in the aorta, it can be the architecture of the internet, it can be the Golden Gate bridge, it can be the solar system. A system is an isolated part of reality[1] that we wish to study through scientific inquiry. According to

[1] Or may be even more than that, it may also be a construction of the mind, such as e.g. an abstract Mathematical theory.

Bernard Siegler ‘a system is a potential source of data’ [3]. We study the system by experimenting with it, where, according to François Cellier, ‘an experiment is the process of extracting data from a system by exerting it through its inputs’. [4] This sounds very theoretical, but you can easily come up with examples of systems and experiments to study them (do this, write down some examples!). Figure 1 provides a schematic overview of ways to study a system.

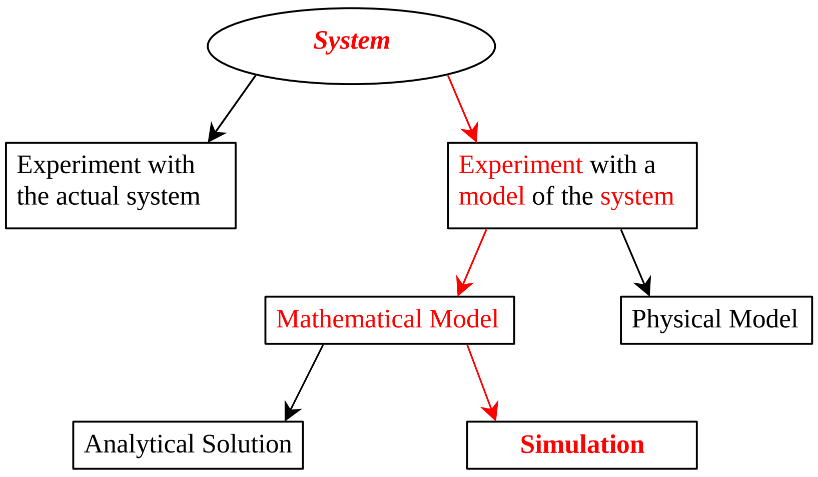
System
Experiment with the actual system
Experiment with a model of the system
Mathematical Model
Physical Model
Analytical Solution
Simulation

*Figure 1: Several ways to study a system through experimentation.*

First we may decide to experiment with the actual system. This may be possible for some systems (e.g. blood flow in the aorta may be measured with Magnetic Resonance Imaging, and we may study e.g. the influence of exercise of a person on the flow), but in other cases this may not be so obvious (how would you experiment with the solar system?). In many cases it is more natural to first construct a model of the system under study and next perform experiments on the model. So, you could study say the influence of a tornado on the mechanical vibrations in the Golden Gate bridge by actually building a small scale physical model of it and putting that in a wind tunnel to see what happens. However, you could also build a *mathematical model* of your system. For instance, for the Solar systems you could write down Newton’s equation of motion for the main bodies in the solar system and then try to solve these equations.

In general, according to Marvin Minski, ‘a Model (*M*) for a system (*S*) and an experiment (*E*) is anything to which *E* can be applied in order to answer a question about *S*’. [5] So, it should be clear by now that a model is not necessarily a computer program. However, in this lecture we will only study models that can be expressed as computer programs, so-called *Mathematical Models*. As the model replaces the original system we must always ask the question if this representation is accurate enough for our purposes. In other words, modeling always requires the act of *validation*, where model validation always relates to an experiment performed on the original system.

Now, mathematical models allow theoretical investigation, and in some (unfortunately not too many) situations it turns out that the mathematical equations can be solved analytically. We will encounter some situations of mathematical models that can be solved analytically later in this reader. However, this is rare, and in many cases one must resort to the final possibility, i.e. *Simulation*, where we may define simulation as ‘an experiment performed on a mathematical model’. [6] Again note the importance of validation. It is easy to perform an experiment on a model in a range of parameters where the model is no longer valid, i.e. where the model is no longer an accurate representation of the system under study. In that case the simulation will produce results, but they have no meaning whatsoever (garbage in, garbage out).

So, within this framework, Modeling and Simulation provides a clear route through Figure 1 and currently is a well-established field in Science and Engineering. Let us now zoom in a little more on the modeling and simulation cycle, see Figure 2. A domain specific problem (a system) is mapped on a conceptual model, which is then mapped on a solver layer. The solvers can be of different types (*Numerical*, *Direct*, or *Natural*) and in the following chapters you will encounter these different types in much detail (remember, Scientific Computing was the glue between the model and the actually running simulation). Next, the solvers are mapped on a virtual machine model. This can be e.g. a message passing parallel computing model (e.g. SPMD in MPI, see [1]), or the data parallel model, or the bulk synchronous parallel model, etc. Finally, the virtual machine model is mapped onto the actual hardware where the simulation is executed.

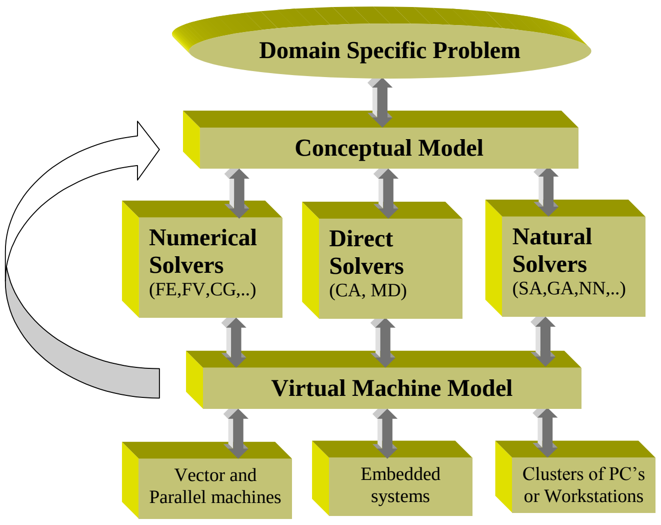
Domain Specific Problem
Conceptual Model
Numerical Solvers (FE,FV,CG,..)
Direct Solvers (CA, MD)
Natural Solvers (SA,GA,NN,..)
Virtual Machine Model
Vector and Parallel machines
Embedded systems
Clusters of PC's or Workstations

*Figure 2: The Modeling and Simulation Cycle.*

In this cycle many feedback loops will exist, but in this lecture the most interesting one is how a specific virtual machine model (read, the assumption of parallel computing in the SPMD paradigm) influences the models and the solvers. Or, is it possible to find models and solvers that are inherently parallel, and therefore allow a straightforward mapping on a parallel computer. In this lecture we will study a large collection of models and solvers, and constantly address this question.

## B.2. Types of Mathematical Models

Although many ideas exist on how to distinguish between different mathematical models, and although certainly no clear consensus exists as of today, a generally well accepted distinction is in the following three types:

- *Continuous time models*;
- *Discrete time models*;
- *Discrete event models*.

The main distinction lies in the way the state of the model changes as a function of time, see Figure 3.

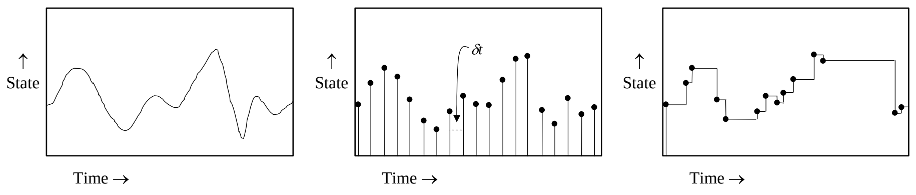

*Figure 3: Trajectory of a continuous time - (left), discrete time - (middle), and discrete event model (right).*

In continuous time models the state of a system changes continuously over time. These types of models are usually represented by sets of differential equations. With discrete-time models, the time axis is discretised. The system state changes are commonly represented by difference equations. These types of models are typical to engineering systems and computer-controlled systems. They can also arise from discrete versions of continuous-time models. The time-step used in the discrete-time model is constant. In discrete-event models, the state is discretised and "jumps" in time. Events can happen any time but only every now and then at (stochastic) time intervals. Typical examples come from "event tracing"' experiments, queuing models, Ising spin simulations, image restoration, combat simulation, etc.

## C. Discrete Event Simulation

### C.1. A Definition

We have seen that in continuous systems the state variables change continuously with respect to time, whereas in discrete systems the state variables change instantaneously at separate points in time. Unfortunately for the computational scientist there are but a few systems that are either completely discrete or completely continuous, although often one type dominates the other in such hybrid systems. The challenge here is to find a computational model that mimics closely the behavior of the system, specifically the simulation time-advance approach is critical.

If we take a closer look into the dynamic nature of simulation models, keeping track of the simulation time as the simulation proceeds, we can distinguish between two *time-advance* approaches: *time-driven* and *event-driven*.

1. *Time-Driven Simulation*

In time-driven simulation the time advances with a fixed increment, in the case of continuous systems. With this approach the simulation clock is advanced in increments of exactly $\Delta t$ time units. Then after each update of the clock, the state variables are updated for the time interval [$t$, $t+ \Delta t$]. This is the most widely known approach in simulation of natural systems. Less widely used is the time-driven paradigm applied to discrete systems. In this case we have to consider whether:

- The time step $\Delta t$ is small enough to capture every event in the discrete system. This might imply that we need to make $\Delta t$ arbitrarily small, which is certainly not acceptable with respect to the computational times involved.
- The precision required can be obtained more efficiently through the event-driven execution mechanism. This primarily means that we have to trade efficiency for precision.

2. *Event-Driven Simulation*

In event-driven simulation on the other hand, we have the next-event time advance approach. Here (in case of discrete systems) we have the following phases:

*Step 1* The simulation clock is initialized to zero and the times of occurrence of future events are determined.

*Step 2* The simulation clock is advanced to the time of the occurrence of the most imminent (i.e. first) of the future events.

*Step 3* The state of the system is updated to account for the fact that an event has occurred.

*Step 4* Knowledge of the times of occurrence of future events is updated and the first step is repeated.

The nice thing of this approach is that periods of inactivity can be skipped over by jumping the clock from *event time* to the *next event time*. This is perfectly save since – per definition – all state changes only occur at event times. Therefore causality is guaranteed. The event-driven approach to discrete systems is usually exploited in queuing and optimization problems. The following sections will discuss in detail Discrete Event Simulation (DES) and Parallel Discrete Event Simulation (PDES).

## C.2. A Prototypical DES example

The prototypical DES example is a queuing system. Consider a store, with clients entering. The store has just one clerk that handles the clients. The clients enter at certain time intervals, and each client needs a certain (not constant) time with the clerk to finalize his shopping. In short, clients enter the store, they queue up, are serviced, and leave again. Depending on the amount of clients that enter the store per time unit and the handling times per client, a queue may or may not form. Interesting parameters are mean waiting times for each client, mean queue length, probability to have a certain number of clients in a queue, etc.

We may formalize this example in terms of a single server queuing system, see Figure 4. Here we assume an infinite population of units, and according to some probabilistic model[2] for arrival times, single units arrive in a waiting line (the queue). Units are serviced by a server, again assuming some probabilistic model for the service times, and after servicing the unit departs the system.

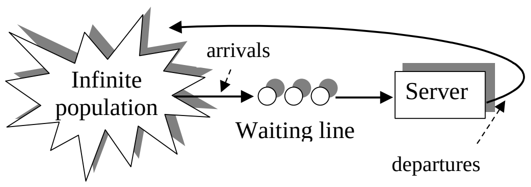

*Figure 4: A single server queuing system.*

We may distinguish a number of actions in the model:

- arrival of a unit,
- entry of a unit in the queue,
- servicing of a unit,
- departure of a unit.

An event, which in the model takes zero time, is the moment at which an action starts or terminates. The minimal number of events needed to model the single server queuing system is two: an arrival event and a departure event, see Figure 5.

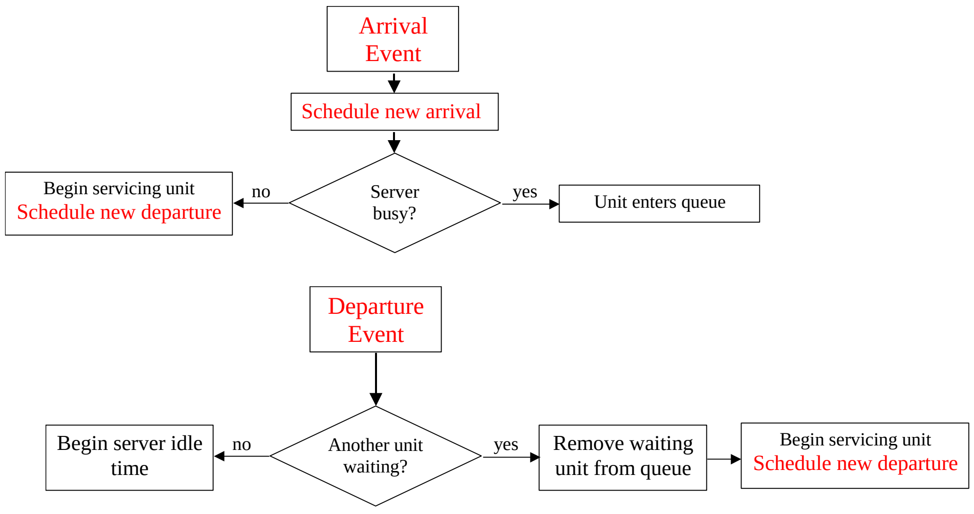

*Figure 5: The arrival – and departure event in the single server queuing model.*

After arrival of a unit the event handler first checks if the server is busy. If so, the unit enters a queue, if not it is immediately serviced. Note that the arrival event again schedules new events. This is a important feature of discrete event models. So, upon arrival of a unit, the first thing that happens is scheduling of a new arrival event,

[2] The appendix to this chapter gives a very short overview of probability distributions.

according to some probability distribution. Here arriving units can be modeled as a Poisson process[3] (why?), so the inter arrival times are exponentially distributed. Furthermore, if servicing of a unit begins, a new departure event is scheduled, according to a probability distribution for service times (e.g. exponential or normal distributions). In the simulation pseudo random number generators are used to draw times from the distributions (see e.g. [7], appendix A). The departure event has a comparable structure. It first checks if a unit is waiting. If not it idles, if yes, a unit is removed from the queue and another departure event is scheduled.

All scheduled events, with their associated times, are put on a list and sorted in increasing time. This *future event list* is the core of a discrete event simulation. The first event on the list, the imminent event, is taken from the list and handled, next the then imminent event is taken and handled, as so on. The time advance algorithm is shown in Algorithm 1.

```
While not stop {
1. Advance the simulation time to time of imminent event.
2. Remove imminent event from event list
3. Execute imminent event
4. Generate (or cancel) future events (if necessary) and update
   event list.
5. Update statistics and counters
}
```

*Algorithm 1: The Time Advance Algorithm in DES.*

## C.3. World Views in Discrete Event Simulation

All simulations contain an executive routine for the management of the calendar and clock, i.e., the sequencing of events and driving of the simulation. This executive routine fetches the next scheduled event, advances the simulation clock and transfers control to the appropriate routine. The operation routines depend on the worldview, and may be events, activities, or processes.

A worldview is the point of view from which the modeler sees the world or the system to be modeled. Most of the discrete event simulations use one of the three following perspectives [8]: *event scheduling*, *activity scanning*, or *process interaction*.

In *event scheduling* each type of event has a corresponding event routine. The executive routine processes a time ordered calendar of event notices to select an event for execution. Event notices consist of a time stamp and a reference to an event routine. Event execution can schedule new events by creating an event notice and place it at the appropriate position in the calendar. The clock is always updated to the time of the next event, the one at the top of the calendar. In fact, the example of the single server queuing system was presented in this view.

In the *activity scanning* approach a simulation contains a list of activities, each of which is defined by two events: the start event and the completion event. Each activity contains test conditions and actions. The executive routine scans the activities for satisfied time and test conditions and executes the actions of the first selectable activity. When execution of an activity completes, the scan begins again. The activities in the single server queuing model were mentioned. You could try yourself to write down the main loop of the DES in the activity scanning approach.

The *process interaction* worldview focuses on the flow of entities through a model. This strategy views systems as sets of concurrent, interacting processes (objects). A process class describes the behavior of each class of entities during its lifetime. Process classes can have multiple entries and exits at which a process interacts with its environment. The executive routine uses a calendar to keep track of forthcoming tasks. However, apart from

[3] See the appendix to this chapter.

recording activation time and process identity, the executive routine must also remember the state in which the process was last suspended.

## C.4. A Prototypical DES example, continued

Let us continue the example of the single server queuing system by now taking the process interaction point of view. We can identify two processes in the model

1. A *unit generator process*, which generates new units that enter the system.
2. A *unit process*, which requests a service (i.e. queue until server is ready for this unit), will be serviced, and leaves the system.

A simulation is now prepared by defining both processes and then starting a first instance of the unit generator process. Below we show a program for such a simulation, in the SIMSCRIPT II.5 language. SIMSCRIPT is a high-level simulation language that allows construction of models either in the event scheduling or process interaction worldview. Here we show a program in the process interaction worldview. It should be noted however that SIMSCRIPT, and all other high-level simulation languages for that matter, will translate the process interaction view into a execution based on event scheduling. As the process interaction view has a very intuitive feel and is amenable to object oriented programming, this worldview has become very popular. But, the message is that in the final execution the event scheduling view prevails. Anyway, as announced above, in Algorithm 2 the process interaction implementation of the single server queuing model in SIMSCRIPT is shown.

```
Preamble
 Processes include UNIT,
 UNIT_GEN
 Resources include SERVER
 ..
End

Main
 ..
 Create SERVER
 Create a UNIT_GEN
 Activate this UNIT_GEN now
 Start Simulation
 ..
End
```

```
Process UNIT_GEN
 ..
 While not STOP
  Do
   Wait exponential(..) minutes
   Create a UNIT
   Activate this UNIT now
 Loop
End

Process UNIT
 ..
 Request SERVER
 Work exponential(..) minutes
 Relinquish SERVER
 ..
End
```

*Algorithm 2: A SIMSCRIPT implementation of the single server queuing model in the process interaction worldview.*

In the preamble the two processes are requested, as well as a standard SIMSCRIPT resource (a `SERVER`). The `SERVER` is just a high level construct that implements the queue. Processes can request the `SERVER`, but must wait for servicing until it is their turn. When a unit is serviced, it blocks the `SERVER` for use by other units. The exact definitions of the `UNIT_GEN` and `UNIT` are given as well. The `UNIT` process requests service from the `SERVER` resource. When the `SERVER` is still busy, the `UNIT` process will wait. When the `SERVER` becomes available, a service time is established (the call to the `exponential(..) minutes` function, i.e. drawn from an exponential distribution) and during this period the `SERVER` is busy, after which the `SERVER` is relinquished (i.e. freed to service other units). The `UNIT_GEN` only injects `UNIT` processes into the system, with inter arrival times drawn from an exponential distribution. Finally, the `..` are places for other useful code, like measuring all kinds of interesting statistics of the simulation. Finally, the main loop of the simulation first creates the `SERVER` and the `UNIT_GEN` and activates the later. Note that in the simulation just one `UNIT_GEN` is active, whereas many `UNIT` processes can be active.

One final note about this example. In the Kendall notation for queuing models (see e.g. [9]) our example of the single server queuing model is a M/M/1/∞/∞ queue. This queue has analytical solutions for e.g. the expected waiting time, the expected queue length, etc.. Therefore, simulation of such a system is not really necessary (but a nice test case or practice). However, by making the model more complex (e.g. including rush hours in arrivals, having more servers with different statistics and possibility of breakdown, allow for priority clients, etc.) analytical solutions no longer exist and one must turn to simulations.

### C.5. Simulation Languages

The paradigm of Discrete Event Simulation is very powerful and has a wide range of applications. Many special high level languages for DES have been developed. Typically, these languages contain constructs to express the entities of interest, such as events, objects, resources, etc, as well as random number generators to draw random numbers from a variety of distributions, statistical analysis routines, time advancement mechanisms providing an explicit representation of simulation time, and report generation tools. Over the years many languages were developed in all three worldviews, see Table 1 for a small overview.

| *Activity Scan* | *Process Interaction* | *Event Scheduling* |
|---|---|---|
| GSP | GPSS | Simscript |
| Simpac | Simula(76) | Quickscript |
| CSL | Simscript | SLAM |
| ECSL | SOL | Simfactory |
| Edsim | APL | Sim++ |
| | COSMOS | |
| | ModSim III | |

*Table 1: Some high level languages for DES.*

As was already mentioned before, especially the process interaction gives many opportunities for object oriented programming. For instance, the ModSim III system is completely based on this programming paradigm.

## D. Parallel Discrete Event Simulation

### D.1. From Sequential to Parallel Discrete Event Simulation

Discrete event models can become very large, requiring large-scale simulations on high end computing systems. It is therefore very important to study the possibility of Parallel Discrete Event Simulations (PDES). Before we dive into the fundamental problem in PDES, that of *causility*, we first examine another simple example of a discrete event model and investigate at what level we may expect parallelism that could be exploited. As the main entities in a discrete event model are the events, we may hope that large scale discrete event models contain many *independent* events (independent with respect to data dependencies, see [1]) that may be executed in parallel.

Now consider a slightly more complicated queuing system, that of a traffic network as drawn in Figure 6. We have a roundabout with three stop signs, three entry roads and three exit roads. The roundabout and the entrance and exit roads are controlled by three traffic lights. We may now be interested in issues of capacity of the roundabout, the appearance of traffic jams, etc. This traffic network can now be modeled as three connected servers, each with two input queues and two output queues. One of the output queues connects to the input queue of another server. Each unit (car) departing from a server on such output queue will induce an arrival event at the other, connected server. As an exercise you could try to write down a simulation pseudo code in either the event scheduling worldview or the process interaction worldview. We will not do that here. We will continue to analyze this model and to try to find parallelism on the level of events.

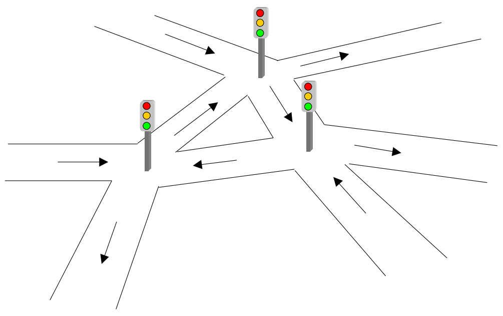

*Figure 6: A small traffic network.*

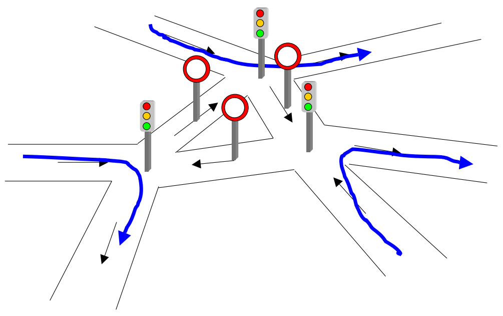

*Figure 7: The traffic network of Figure 6 with closed interconnecting roads.*

Let us first suppose that the interconnecting roads between the traffic lights are blocked, as in Figure 7. All traffic entering on a entry road must take the exit road of the traffic light on which it enters the roundabout. This means that all three traffic lights are completely independent of each other and that every event scheduled for one traffic light can be handled completely in parallel from events scheduled at other traffic lights. Clearly this is not a very interesting situation, but now let us assume that the connection are again open, but that most traffic still takes the routes as drawn in Figure 7, and that just a small portion of the traffic enters the connecting roads. This is drawn schematically in Figure 8.

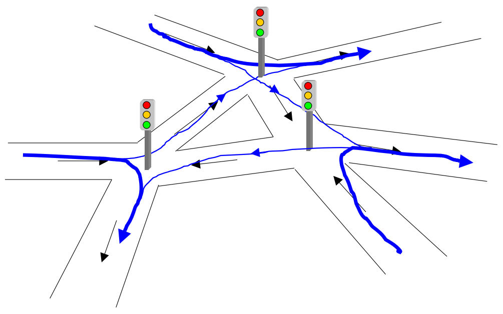

*Figure 8: Traffic network of Figure 6 with a small portion of the traffic on the roundabout.*

In this situation we still have many independent events, that can be handled in parallel and a relative small amount of dependencies that must be resolved.

The main recipe now to introduce parallelism into DES is through the following steps:

1. Indicate physical components in the model,
2 Map physical components to a set of logical processes $LP_i$,
3. Run each $LP_i$ in parallel as a separate DES,
4. Resolve dependencies between LP’s.

The final point, resolving dependencies between Logical Processes is far from trivial and will be discussed in detail in the next sections. The first three steps will now be demonstrated in the framework the of example of the traffic network.

In the example each traffic light can be identified as a physical component, connected through stretches of road. A “domain decomposition” splits the model into three components (see Figure 9) with their connections. These three components are now mapped to Logical Processes $LP_A$, $LP_B$, and $LP_C$. Figure 10 shows the LP’s and their connections. Each LP is executed as a DES, and may schedule events in the other LP’s through the connections.

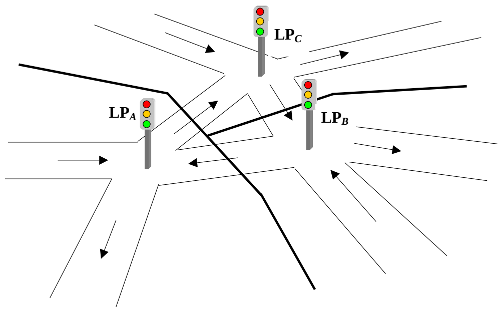

*Figure 9: The traffic network decomposed in three LP’s.*

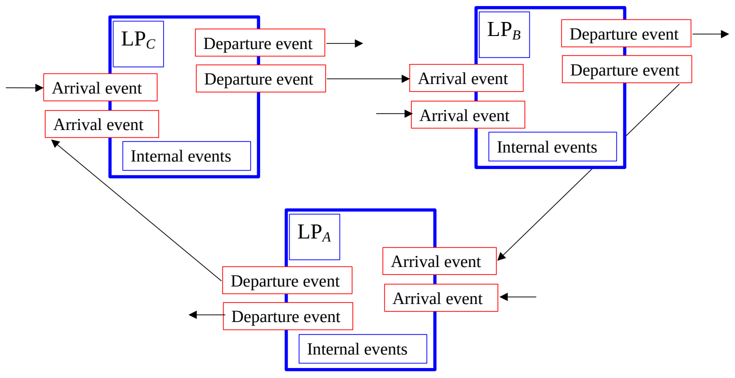

*Figure 10: The traffic network in terms of its Logical Processes and their connections.*

Each Logical Process $LP_i$ has its own internal state $S_i$. It also has its own clock, so each LP has a local simulation time. We call this the *Local Virtual Time* (LVT). Furthermore, for each LP we can distinguish *internal events* that only affect the internal state $S_i$ and *external events* that may affect other states $S_j$. The interaction between LP’s is through the external events.

## D.2. The Fundamental Problem in Parallel Discrete Event Simulation

We are especially interested in parallelization of asynchronous system simulation, where events are not synchronized by a global clock, but rather occur at irregular time intervals. In these simulations few events occur at any single point in simulated time and therefore parallelization techniques based on synchronous execution using a global simulation clock performs poorly. Concurrent execution of events at different points in simulated time is required, but this introduces interesting synchronization problems.

These problems become clear if one examines the operation of a sequential discrete event simulator. The sequential simulator typically uses three data structures: the state variables, an event list (the calendar), and a global simulation clock. For the execution routine it is crucial that the smallest time stamped event ($E_{min}$) from the event list is selected as the one to be processed next. If it would depart from this rule and select another event with a larger time stamp ($E_x$), it would be possible for $E_x$ to change the state variables used by $E_{min}$. This implies that one is simulating a system where the *future could affect the past*. We call errors of these kind *causality errors*.

Let us next consider the parallelization of a simulation based on the above paradigm. Most parallel discrete event simulation (PDES) strategies adhere to a process interaction worldview that strictly forbids processes to have direct access to shared state variables. To this methodology some extensions have been made to support the parallel execution of the simulation [10]. The system being modeled is viewed as being composed of some number of *physical processes* that interact at various points in simulated time. The simulation is constructed as a set of *logical processes* $LP_0$, $LP_1$, …, one per physical process, as explained in the previous section. All interactions between physical processes are modeled by *time stamped event messages* sent between the corresponding logical processes. Each logical process contains a portion of the state corresponding to the physical process it models, as well as a local clock that denotes the progress of the process.

One can assure that no causality error occurs if one adheres to the *local causality constraint*:

> **Local Causality Constraint**: A discrete event simulation, consisting of logical processes that interact exclusively by exchanging time stamped messages, obeys the local causality constraint *if and only if* each logical process executes events in non decreasing time stamp order.

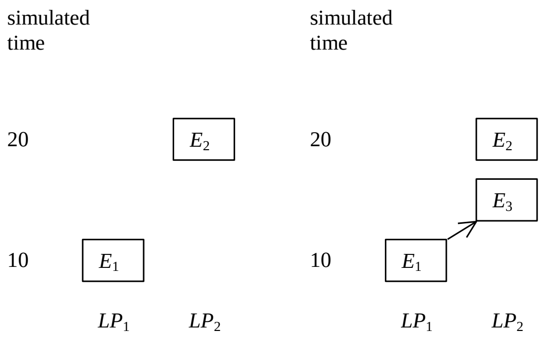

*Figure 11: Causality error.*

Consider two events; $E_1$ at logical process $LP_1$ with time stamp 10, and $E_2$ at $LP_2$ with time stamp 20 (see Figure 11). If $E_1$ schedules a new event $E_3$ for $LP_2$ containing a time stamp less than 20, then $E_3$ could affect $E_2$, necessitating sequential execution of all three events. If one had no information what events could be scheduled by other events, one would be enforced to process the only save event, the one containing the smallest time stamp, resulting in a sequential execution.

During the simulation we must therefore decide whether $E_1$ can be executed concurrently with $E_2$. But how do we know whether or not $E_1$ affects $E_2$ without actually performing the simulation for $E_1$? It is this question the parallel discrete event simulation strategies must address.

We classify parallel discrete event simulation strategies by two categories: *conservative* and *optimistic*. Conservative approaches strictly avoid the possibility of any causality error ever occurring. These approaches rely on some strategy to determine when it is safe to process an event. The optimistic approaches use a detection and recovery approach: whenever causality errors are detected a rollback mechanism is invoked to recover. We will describe some of the concepts behind conservative and optimistic simulation mechanisms.

## D.3. Conservative Methods

Conservative approaches to PDES strictly avoid the possibility of any causality error ever occurring. The conservative approaches were the first distributed simulation mechanisms to be developed for PDES. The basic problem conservative mechanisms must address is to determine which event is save to process. If an LP contains an event $E_1$ with time stamp $T_1$ and it can determine that it is impossible to receive another event with time stamp smaller than $T_1$, then the LP can safely process event $E_1$ without a future violation of the local causality constraint. LP's containing no safe events must block. This can lead to deadlock situations if no appropriate precautions are taken.

Independently, Chandy and Misra [10], and Bryant [10] developed parallel discrete event simulation algorithms, where one statically specifies the links that indicate which process may communicate with which other processes. In order to determine when it is safe to process a message, it is required that messages from any process to any other process are transmitted in chronological order according their time stamps. Each link has a clock associated with it that is equal to either the time stamp of the message at the front of that link's queue or, if the queue is empty, the time of the last received message. The process repeatedly selects the link with the smallest clock and, if there is a message in that link's queue, updates its local clock to the link's clock and process the message. The order of event processing will be correct because all future messages received will have later time stamps than the local clock, since they will arrive in chronological order along each link. If the selected queue is empty, the process blocks. This is because the process may receive a message over this link with a time that is less than all the other input time stamps. Thus to insure correct chronology, the process is forced to wait for a message to update the clock on the link before the process can update its local clock. This protocol guarantees that each process will only process events in non-decreasing time stamp order, and thereby ensuring chronological integrity.

To summarize,

1. If $LP_i$ sends a message to $LP_j$ a link exists from $i$ to $j$.
2. A message contains an event and associated timestamp for that event.
3. If a message arrives from $LP_i$ in $LP_j$ it is stored by $LP_j$ in a buffer associated with the link from $LP_i$.
4. LP's send messages in strict chronological order (and are assumed to arrive at their destination in this strict order) - this guarantees that if $LP_j$ receives a message from $LP_i$ with timestamp $T$ that any other message received from $LP_i$ will have a larger timestamp.

The algorithm for each LP can now be summarized as

```
1. Look at all the buffers associated with links from other
   LP's, to find the event with the minimal timestamp.
2. Simulate up to this timestamp (it is guaranteed that no
   messages with events before this timestamp will arrive).
   Note that you handle all internal events on the event list
   and the event in the message with minimum timestamp.
3. If a buffer is empty, block and wait until a message
   arrives (this new message may have a timestamp before the
   messages in other non-empty buffers).
```

As an example, consider a $LP_i$ which has two links to other LP's, see Figure 12. On each link a number of events E have arrived. Furthermore, a number of internal events have been scheduled. The LVT starts at 1. In this starting situation, it is safe to handle events up to time 5, so, internal events $E_2$ and $E_4$ and external event $E_5$ are safe events and are handled. This then brings us to the situation, with LVT = 5, as shown in Figure 13. One new internal event has been scheduled. Link 1 is now empty, causing the LP to block and waiting for a new event to arrive at link 1, see Figure 14. An event with timestamp 12 arrived at link 1. Now it is safe to handle events up to time 11, etc.

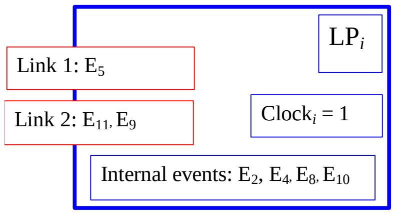

*Figure 12: An example of a conservative PDES, LP$_i$ has two links, with queued events E. Furthermore, the event list contains a number of internal events. The clock starts at 1, the subscript on the events their timestamp at which they are scheduled to be handled.*

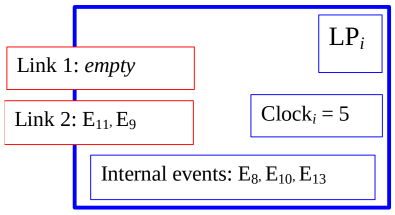

*Figure 13: Example of Figure 12 continued.*

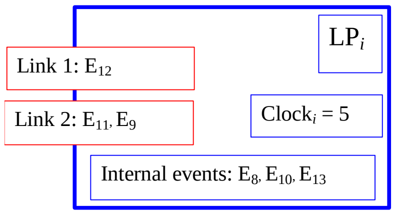

*Figure 14: Example of Figure 12 continued.*

Deadlock may occur if *all* LP’s have an empty link, e.g. if message traffic is low, then this situation readily occurs. Deadlock occurs when there is a cycle of blocked processes and each process is blocked due to another process in the cycle. For example consider the network of Figure 15. Each process is waiting on the incoming link containing the smallest clock value because the corresponding queue is empty. All three processes are blocked, even though there are event messages in other queues that are waiting to be processed.

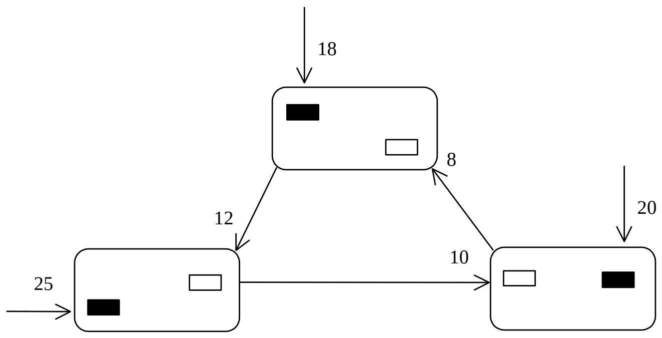

*Figure 15: An example of deadlock. (The numbers indicate time stamps.)*

Null messages are used to avoid deadlock. In this way LP’s inform each other of their LVT. This scheme requires that there is a strictly positive lower bound on the *lookahead* for at least one process in each cycle. Lookahead is defined to be the amount of time that a process can look into the future. In other words, if the local clock of the process is any time *T* and the process can predict all messages it will send with time stamps less than $T + L$, where $L$ is the lookahead. Thus, for a queueing network model, a strictly positive lower bound for the service time for some stations would be required. Intuitively, processes keep the clocks of their output links ahead of their local clocks by sending null messages.

A null message with time stamp $T_{null}$ from process $LP_A$ to $LP_B$, tells $LP_B$ that there will be no more messages from process $LP_A$ with time stamp less than $T_{null}$. Whenever a process finishes processing an event, it sends a null message on each of its output ports indicating the lower bound on the time stamp of the next outgoing message. The receiver of the null message can then compute new bounds on its outgoing links, send this information to its neighbors, and so on.

Chandy and Misra [11] also presented a two-phase scheme where the simulation proceeds until deadlocked, then the deadlock is detected and resolved. The mechanism is similar to that described above, except no null messages are created. Instead the computation is allowed to deadlock. The scheme involves a controller process to monitor for deadlock and control deadlock recovery. Deadlock detection mechanisms are described in [12]. The deadlock can be broken by the observation that the message with the smallest time stamp is always save to process; or, with use of a distributed computation, obtain a lower bound to enlarge the set of safe messages.

The degree to which processes can look ahead and predict future events; or more importantly, what will not happen in the simulated future critically determines the performance of conservative mechanisms. A process with lookahead $L$ can guarantee that no events, other than the ones that it can predict, will be generated up to time $Clock + L$. This may enable processes to safely process forthcoming messages that they have already received. Fujimoto describes lookahead quantitatively using a parameter called the lookahead ratio and presents empirical data to demonstrate the importance of exploiting lookahead to achieve good performance [13].

## D.4. Optimistic Methods

In optimistic methods one (optimistically) handles events on the event list and in the message buffers. If an event arrives with a timestamp smaller than the local virtual time (causality error !) some mechanism must be started to recover from this error. So, an LP's LVT may run ahead of the timestamp of events arriving on its incoming links and if errors are made in the chronology a procedure to recover is invoked. In contrast to conservative approaches, optimistic strategies need not determine when it is safe to proceed. Advantages of this approach are that it has a potentially larger speedup than conservative approaches and that the topology of possible interactions between processes need not be known.

An optimistic approach to distributed simulation called Time Warp was proposed by Jefferson [14]. Here virtual time is the same as the simulated time. The Local Virtual Time of a process is set to the minimum receive time of all unprocessed messages. Processes can execute events and proceed in local simulated time as long as they have any input at all. As a consequence, the local clock or LVT of a process may get ahead of its predecessors' LVTs, and it may receive an event message from a predecessor with time stamp smaller than its LVT, i.e., in the past of the process. If this happens the process *rolls back* in simulated time. The event causing the roll back is called a *straggler*, see Figure 16. Recovery is accomplished by undoing the effects of all events that have been processed prematurely by the process receiving the straggler.

The premature execution of an event will trigger two things that have to be rolled back: the state of the logical process and the event messages to other processes. Rolling back the state is accomplished by periodically saving the process state and restoring an old state on roll back, see Figure 17. Unsending a previously sent message is accomplished by sending an *anti-message* that annihilates the original when it reaches its destination, see Figure 18. Messages that are sent while the process is propagating forward in simulated time are called *positive messages*. If a process receives an anti-message that corresponds to a positive message that is still in the input queue, then the two will annihilate each other and the process will proceed, see Figure 19. If an anti-message arrives that corresponds to a positive message that is already processed, then the process has made an error and must also roll back. It sets its current state to the last state with simulated time earlier than the time stamp of the message, see Figure 20. A direct consequence of the roll back mechanism is that more anti-messages may be sent to other processes recursively, resulting in an avalanche of roll backs.

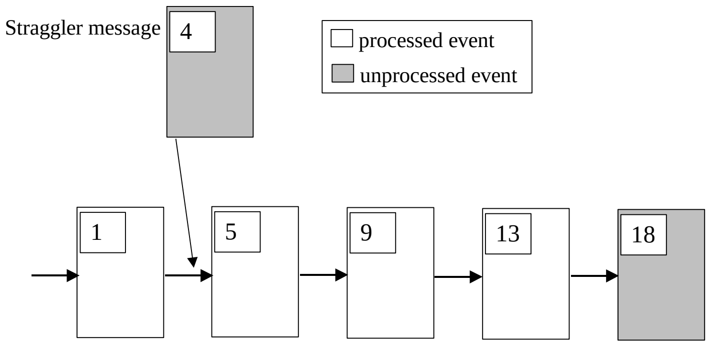

*Figure 16: Beginning of a roll back, a straggler arrives.*

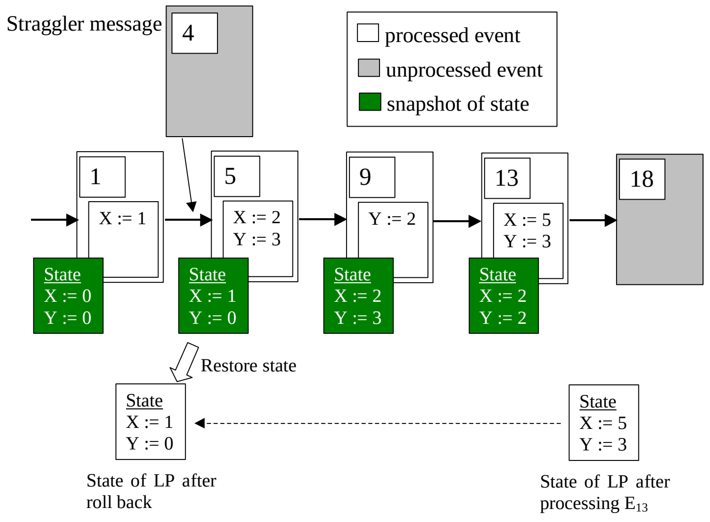

*Figure 17: State saving and recovery of the state at roll back.*

The Global Virtual Time (GVT) is the minimum of the LVTs for all the processes and the time stamps of all messages sent but unprocessed. No event with time stamp smaller than GVT will ever be rolled back, so storage used by such event (i.e., saved states) can be discarded.

The procedure just described is referred to as Time Warp with aggressive cancellation. An alternative is lazy cancellation, where anti-messages are not sent immediately after roll back. Here, the process resumes executing forward in simulated time from its new LVT, and when it procedures a message it compares it with the messages in its output queue. If the same message is recreated, then there is no need to cancel the message. An anti-message created at simulated time *T* is only sent after the process's clock sweeps past time *T* without regenerating the same message. Thus, under lazy cancellation a roll back at the successor process may be avoided. On the other hand, if messages are not reproduced, then roll backs at the successor processes will be required under both mechanisms, and they will occur sooner with aggressive cancellation.

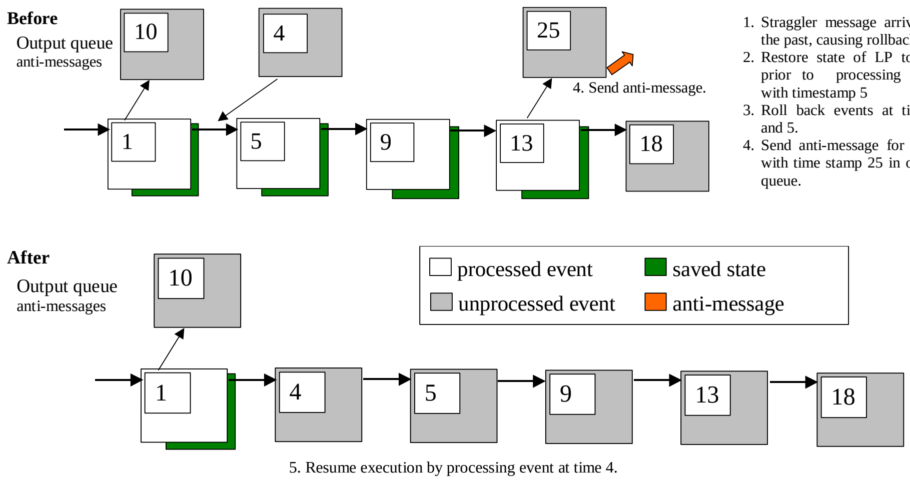
Before
Output queue
anti-messages
10
4
25
4. Send anti-message.
1
5
9
13
18
1. Straggler message arrives in the past, causing rollback
2. Restore state of LP to that prior to processing event with timestamp 5
3. Roll back events at time 9 and 5.
4. Send anti-message for event with time stamp 25 in output queue.
After
Output queue
anti-messages
10
processed event
saved state
unprocessed event
anti-message
1
4
5
9
13
18
5. Resume execution by processing event at time 4.

*Figure 18: Sending of an anti-message.*

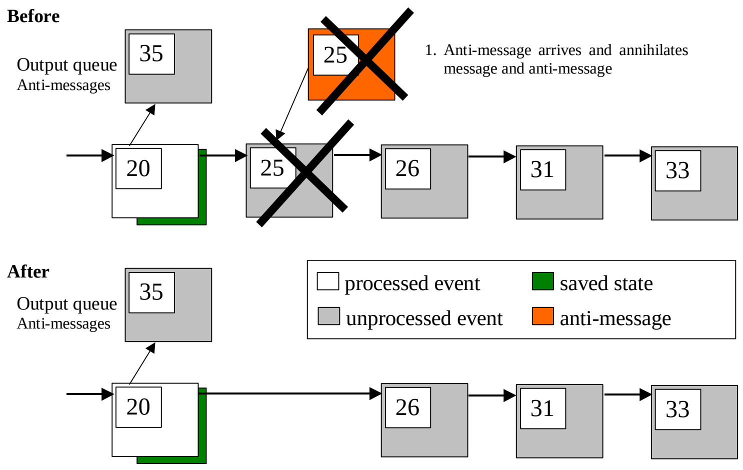

*Figure 19: Arrival of anti-message, annihilating a message still in the event list.*

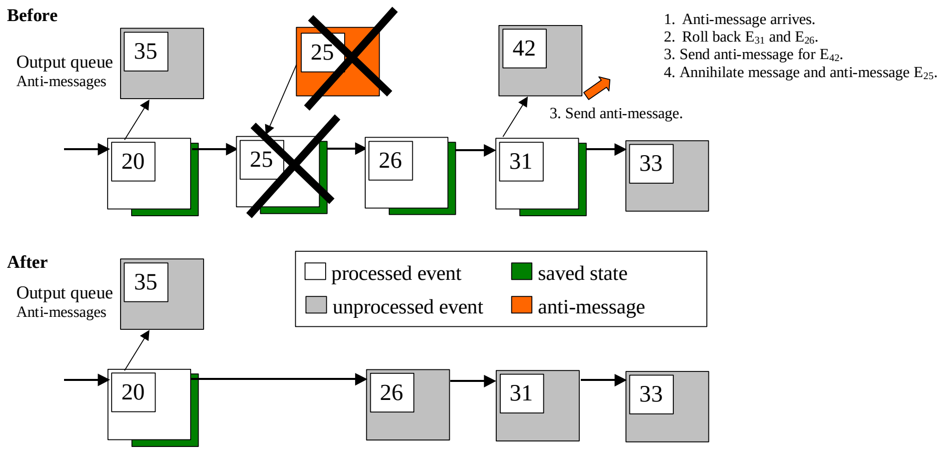

*Figure 20: Arrival of anti-message, annihilating a message that was already processed, thus triggering another roll back.*

Depending on the application, lazy cancellation may either improve or degrade performance. States may be saved less frequently at the expense of greater overhead for roll back. As a consequence, lazy cancellation requires more memory than aggressive cancellation.

Conservative methods offer good potential for certain classes of problems. A major drawback, however, is that they cannot fully exploit the parallelism available in the simulation application. If it is possible that event $E_A$ might affect $E_B$ either directly or indirectly, conservative approaches must execute $E_A$ and $E_B$ sequentially. If the simulation is such that $E_A$ seldom affect $E_B$ these events could have been processed concurrently most of the time. As a consequence, conservative algorithms heavily rely on lookahead to achieve good performance.

Optimistic methods offer the greatest potential as a general purpose simulation mechanism. A critical question faced by optimistic approaches is whether the system will spent most of its time on executing incorrect computations and rolling them back, at the expense of correct computations. An intuitive explanation why the behaviour tends to be stable is that incorrect computations can only be initiated by a premature execution of a correct event. This premature execution, and subsequent incorrect computations, are by definition in the simulated time future of the correct, straggler computation. Also, the further the incorrect computation spreads the further it moves into the simulated time future, thus lowering its priority for execution. Preference is always given to computations containing smaller time stamps. The incorrect computation will be slowed down, allowing the error detection and correction mechanism to correct before too much damage has been done.

A more serious problem with the optimistic mechanisms is the need to periodically save the state of each logical process. This limits the effectiveness of the optimistic mechanisms to applications where the amount of computation, required to process an event, is significantly larger than the cost of saving the state vector.

The type of application, or classes of applications, is important when determining an appropriate approach to distributed simulation. For dynamic topology systems and systems with irregular interactions, Time Warp methods are preferred over conservative methods, especially if state-saving overheads do not dominate. On the other hand, if the application has good lookahead properties, conservative algorithms can exploit the special structure within a fixed topology system. If the application has both poor lookahead and large state-saving overheads all existing parallel discrete event simulation approaches will have trouble obtaining good performance, even if the application has a considerable amount of parallelism.

## E. APPENDIX

In this section we refresh your memory on discrete and continuous random variables, expectation and variance, and a number of probability distribution functions with some emphasis on those distributions that are relevant for the Poisson process.

### E.1. Elements of Probability

If we do an experiment of which the outcome is not determined in advance, for example the throwing of a dice or flipping of a coin, we may try to use stochastic techniques to describe the system in question. If we do a simulation experiment where we try to mimic a specific stochastic system we need random variables in order to be able to do so.

#### Distributions: Discrete and Continuous

We speak of discrete random variables if the outcome $X$ of an experiment can take a finite or at most countable number of possible values. For example you may think of flipping one coin. The outcome $X$ of this experiment is restricted to heads or tails. In this case we say that such a system can be described by a *probability mass function* $p(x)$:

$$p(x) = P[X = x] ,$$

which denotes the probability that the outcome of the experiment $X$ is equal to $x$. For example in case of the coin flipping experiment:

$$p(heads) = P[X = head] = 1/2 .$$

In general $X$ is a discrete random variable if it can take on only a finite or countable set of possible values $x_1, x_2, \ldots, x_n \ldots$ Since $X$ must take on one of these values we have:

$$\sum_{i=0}^{\infty} p(x_i) = 1 .$$

This means that the probability that the outcome of the experiment lies in the set of outcomes $x_1, x_2, \ldots, x_n \ldots$ is equal to unity.

On the other hand if the random variable $X$ can take on a continuous range of values we will have to describe it by means of a *cumulative distribution function (cdf)*. From this function one can derive the probability that $X$ will lie in a specific continuous range of values. The function

$$F(x) = P[X < x]$$

describes the probability that $X$ takes on a value that is less than or equal to $x$. Analogous to the probability mass function in the discrete case we define in the continuous case the *probability density function* $f(x)$ (*pdf*):

$$f(x) = \frac{dF(x)}{dx} .$$

This density function has to be normalized to unity. This means that

$$\int_{-\infty}^{\infty} f(x)dx = 1 ,$$

which says that the outcome of the experiment $X$ lies in the total range of possible values of $x$ with probability 1. Note that $f(x)\mathrm{d}x$ is the probability that the random variable $X$ is in the interval $[x, x+\mathrm{d}x]$.

As an example, suppose lifetime of a part in a machine is given by $X$, where $X \geq 0$. The *pdf* (in years) is given by

$$f(x) = \tfrac{1}{2} e^{-x/2} .$$

Therefore, the probability that the lifetime is between 2 and 3 years is

$$P(2 \leq X \leq 3) = \tfrac{1}{2} \int_{2}^{3} e^{-x/2} dx = e^{-3/2} - e^{-1} = 0.145 .$$

## Expectation: Discrete and Continuous

The expected value or *expectation* of $X$, also called the mean of $X$ is denoted by $E[X]$, $<X>$, or $\mu$. For a discrete random variable it is defined by:

$$E[X] = \sum_i x_i\, p(x_i) .$$

As an example you may think of the coin flipping experiment. Say that we denote heads by 0 and tails by 1. Both events occur with probability 1/2. Therefore the expectation value of this experiment:

$$E[X] = 0 * P[X = 0] + 1 * P[X = 1] = 0 * \frac{1}{2} + 1 * \frac{1}{2} = \frac{1}{2}.$$

In an analog way for a continuous random variable we can calculate the expectation using the probability density function for *X*.

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx .$$

If we are interested in the random variable *g*(*X*), where *g* is some given function we simply write

$$E[g(X)] = \sum_i g(x_i) p(x_i)$$

for the discrete case and

$$E[g(X)] = \int_{-\infty}^{\infty} g(x) f(x) dx$$

for the continuous case.

The expectation is a linear operation in the sense that for any two random variables $X_1$ and $X_2$:

$$E[X_1 + X_2] = E[X_1] + E[X_2]$$

which generalizes to:

$$E[\sum_i X_i] = \sum_i E[X_i]. \qquad [1]$$

Equation 1 holds for the continuous as well as the discrete case.

## Variance: Discrete and Continuous

The expectation value *E*[*X*] of the random variable *X*, is a weighted average of the possible values of *X*. However, it doesn't yield any information about the variation of these values. One way of measuring this variation is to consider the average value of the square of the difference between *X* and *E*[*X*].

If *X* is a random variable with mean *E*[*X*], then the variance of *X*, denoted by *V*[*X*] or $\sigma^2$ is defined by:

$$V[X] = E[(X - E[X])^2].$$

It can simply be derived that:

$$V[X] = E[X^2] - E[X]^2. \qquad [2]$$

The covariance of two random variables *X* and *Y*, denoted *Cov*[*X*, *Y*] is defined by:

$$Cov[X,Y] = E[XY] - E[X]E[Y]. \qquad [3]$$

The covariance gives an idea of the correlation between two variables *X* and *Y*. What can be understood by "correlation" between two variables can be see as follows. As in the coin flipping experiment, assume you flip *n* times the same coin. Set *X* to be the stochastic variable that describes the number of heads that will come out. Simultaneously let *Y* be the variable that describes the number of times that tails will come out. It is clear that the outcomes of the simultaneous experiments *X* and *Y* depend on each other. Let *n* = 1, heads = 0 and tails = 1. Then

$$E[X] = \frac{1}{2},\ \ E[Y] = \frac{1}{2},\ \text{ and }\ E[XY] = 0\ \text{ and thus } Cov[X,Y] = -\frac{1}{4}.$$

This is an example of anti-correlation. In other words: if *X* has a high value consequently *Y* must have a low value. If *X* and *Y* are independent *Cov*[*X*, *Y*] = 0. Given formulas 2 and 3 we can derive furthermore that

$$V[X+Y] = V[X] + V[Y] + 2Cov[X,Y]. \qquad [4]$$

If now *X* and *Y* are independent random variables then

$$Cov[X,Y] = 0 \text{ and thus } V[X+Y] = V[X] + V[Y].$$

Finally note that the standard deviation $\sigma$, defined as

$$\sigma = \sqrt{V[X]}$$

has the same unit as the expectation.

## E.2. The Poisson Process

Consider random arrival events, such as customers entering a shop or the breakdown of parts in a large machine. Define a counting function $N(t)$ that counts the total number of events that arrived in the time interval $[0, t]$, $t \geq 0$. $N(t)$ is the observation of a random variable that assumes the values 0, 1, 2, … Finally assume that the probability that an arrival occurs in $[t,t+dt]$ equals $\lambda dt$.

This arrival process, or counting process $\{N(t), t \geq 0\}$ is a Poisson process with mean rate $\lambda$ if

- arrivals occur one at a time;
- arrivals are completely random; number of arrivals in an interval $[t,t+s]$ depends only on interval length *s* (so, e.g. no ‘rush hours’);
- no correlation exists between non-overlapping time intervals.

Now introduce $p_n(t)$ the probability that $N(t) = n$, i.e.

$$P(N(t) = n) = p_n(t).$$

This is the famous *Poisson distribution*, which takes the following form:

$$p_n(t) = \frac{(\lambda t)^n e^{-\lambda t}}{n!},\ \ t \geq 0\ \text{ and }\ n = 0,1,2\cdots. \qquad [5]$$

**Proof of equation [5]**

$p_n(t+\delta t)$ is the sum of independent, compound probabilities that there were *n* arrivals at time *t* and no new arrivals in the interval $[t, t+\delta t]$, and that there were *n* -1 arrivals at time *t*, and one arrival in the interval $[t, t+\delta t]$, and so on. So, for $n \geq 1$

$$p_n(t+\delta t) = p_n(t)(1-\lambda\delta t) + p_{n-1}(t)\lambda\delta t + \text{h.o.t. in } \lambda\delta t\,,$$
$$\frac{p_n(t+\delta t)-p_n(t)}{\delta t} = \lambda p_{n-1}(t) - \lambda p_n(t) + \frac{\text{h.o.t. in } \lambda\delta t}{\delta t}\quad,$$

with h.o.t. meaning higher order terms. Taking the limit of $\delta t \to 0$ we find

$$\frac{dp_n(t)}{dt} = \lambda p_{n-1}(t) - \lambda p_n(t)\,. \qquad [6]$$

For $n = 0$ this reduces to $p_0(t+\delta t) = p_0(t)(1-\lambda\delta t)$ or

$$\frac{dp_0(t)}{dt} = -\lambda p_0(t) \qquad [7]$$

Solving Equation [7], with initial condition $p_0(0) = 1$, we find $p_0(t) = e^{-\lambda t}$. Next, we solve Equation [6]:

$$p_n(0) = 0,\quad n \geq 1$$
$$\frac{dp_1(t)}{dt} = \lambda p_0 - \lambda p_1,\ \text{so}\ \ p_1(t) = (\lambda t)e^{-\lambda t}\,,$$
$$\frac{dp_2(t)}{dt} = \lambda p_1 - \lambda p_2,\ \text{so}\ \ p_2(t) = \frac{(\lambda t)^2}{2!}e^{-\lambda t}\,,$$
$$\vdots$$
$$p_n(t) = \frac{(\lambda t)^n}{n!}e^{-\lambda t}\,.$$

This concludes the proof.

Recalling that $\alpha e^{\alpha} = \sum_{x=0}^{\infty}\frac{\alpha^x}{(x-1)!}$ and $\alpha(1+\alpha)e^{\alpha} = \sum_{x=0}^{\infty}\frac{x\alpha^x}{(x-1)!}$ we can easily derive the expectation and variance for the Poisson distribution. Writing the Poisson distribution as $p(x) = \frac{\alpha^x e^{-\alpha}}{x!}, \alpha = (\lambda t), x = 0,1,2,\cdots$ we immediately find

$$E[X] = \sum_{x=0}^{\infty}\frac{x\alpha^x e^{-\alpha}}{x!} = e^{-\alpha}\sum_{x=0}^{\infty}\frac{\alpha^x}{(x-1)!} = e^{-\alpha}\alpha e^{\alpha} = \alpha\,,$$
$$E[X^2] = \sum_{x=0}^{\infty}\frac{x^2\alpha^x e^{-\alpha}}{x!} = e^{-\alpha}\sum_{x=0}^{\infty}\frac{x\alpha^x}{(x-1)!} = e^{-\alpha}\alpha(1+\alpha)e^{\alpha} = \alpha(1+\alpha)\,,$$
$$V[X] = \alpha(1+\alpha) - \alpha^2 = \alpha\,.$$

So, both the mean and varaince of the Poisson distribution are equal to $\alpha = \lambda t$.

So far we considered the counting process $N(t)$ which is a discrete random process that counts the arrivals in a Poisson process in the interval $[0, t]$. Directly related to this, and of great practical relevance for Discrete Event Simulations, is the distribution of times between arrivals. More specifically, consider the actual times of arrival:

let the first arrival be at $t = A_1$,

let the second arrival be at $t = A_1 + A_2$, etc.

$A_1$, $A_2$,… are successive inter arrival times. The inter arrival times are a continuous random variable. We now seek the probability density function for this random variable. Since the first arrival occurs after time $t$ if and only if there are no arrivals in $[0, t]$, we immediately conclude that

$$P(A_1 > t) = P(N(t) = 0) = e^{-\lambda t} .$$

Thus, the probability that the first arrival will occur in [0, *t*] is then given by

$$P(A_1 < t) = 1 - e^{-\lambda t} .$$

This is the cumulative distribution function of the probability distribution function that we are looking for. First consider the exponential distribution

$$p(x) = \lambda e^{-\lambda x}, \quad x \geq 0,$$

$$E[X] = \lambda \int_0^\infty x e^{-\lambda x} dx = -x e^{-\lambda x} \Big|_0^\infty + \int_0^\infty e^{-\lambda x} dx = 1/\lambda,$$

$$E[X^2] = \lambda \int_0^\infty x^2 e^{-\lambda x} dx = -x^2 e^{-\lambda x} \Big|_0^\infty + 2/\lambda \int_0^\infty \lambda x e^{-\lambda x} dx = 2/\lambda^2 ,$$

$$V[X] = \frac{2}{\lambda^2} - \frac{1}{\lambda^2} = \frac{1}{\lambda^2}.$$

The cumulative distribution function of the exponential distribution equals

$$F(x) = P(X \leq x) = \int_0^x \lambda e^{-\lambda t} dt = 1 - e^{-\lambda x} .$$

This is exactly the cumulative distribution function of the inter arrival times of a Poisson process with mean rate $\lambda$. Hence, $A_1$, and for that matter all inter arrival times, are exponentially distributed with mean 1/ $\lambda$.

As an example, suppose that in a factory a repair is needed with a mean of $\alpha$ = 2 per day. The occurrence of failures is known to be a Poisson process. With these assumptions, the probability of 3 repairs in the next day will be p(3) = (e-2 23)/3! = 0.18. The probability of 2 or more repairs in the next day will be P(2 or more) = 1 - p(0) - p(1) = 0.594.

## E.3. Other distributions

We end with some other useful discrete and continuous distributions. The question how to choose a specific distribution to model a stochastic process is far from trivial, and will be based on (previous) knowledge of the system, on actual experiments, or on by treating a specific choice as a hypothesis to be tested using statistical methods (e.g. the chi-squared test or Kolmogorov-Smirnov test).

Consider the *Bernoulli process*, which is a trial with two possible outcomes, succes ($x_i$ = 1) or failure ($x_i$ = 0). The *Bernoulli distribution* is the probability mass function for the discrete random variable describing the outcome of the Bernouilli process. The probability mass function and expectation and variance are

$$p(x) = \begin{cases} p, & x = 1, \\ 1 - p = q, & x = 0, \\ 0, & \text{otherwise}, \end{cases}$$

$$E[X] = p,$$

$$V[X] = p(1 - p).$$

Consider again the Bernoulli process. *X* is now the number of trials to achieve the first success. This gives the Geometric distribution, whith the following probability mass function and expectation and variance:

$$p(x) = q^{x-1}p, \qquad x = 1, 2, 3, \cdots,$$
$$E[X] = \frac{1}{p},$$
$$V[X] = \frac{q}{p^2}.$$

Again consider the Bernoulli process. $X$ is now the number of successes in $n$ Bernoulli trials. This gives the Binomial distribution. The probability to obtain $x$ successes in $n$ trials is $p^x q^{n-x}$. There are a total of $\binom{n}{x}$ possible experiments of length $n$ with $x$ successes. Therefore, the probability mass function for the Binomial distribution is

$$p(x) = \binom{n}{x} p^x q^{n-x} \qquad x = 0, 1, 2, \cdots .$$

The Binomial random variable $X$ can be considered as the sum of $n$ Bernoulli processes $X_i$, $X = X_1 + X_2 + \ldots + X_n$. Therefore one immediately finds for the expectation and variance of the Binomial distribution

$$E[X] = p + p + \cdots p = np,$$
$$V[X] = pq + pq + \cdots pq = npq.$$

It is possible to derive the Poisson distribution from the Binomial distribution by taking a special limit. Let $n \to \infty$ and $p \to 0$, such that $np = \lambda$, a nonzero finite constant. So, we have an infinite large pool ($n \to \infty$), with very small chance of success ($p \to 0$), but with a finite constant rate $\lambda$. With these definitions we then find

$$P(X = k) = \binom{n}{k} p^k q^{n-k} = \frac{n!}{k!(n-k)!}\left(\frac{\lambda}{n}\right)^k \left(1 - \frac{\lambda}{n}\right)^{n-k}$$
$$= \frac{n(n-1)\cdots(n-k+1)}{n^k}\frac{\lambda^k}{k!}\left(1 - \frac{\lambda}{n}\right)^{-k}\left(1 - \frac{\lambda}{n}\right)^n .$$

In the limit $\lim\limits_{n\to\infty}\left(1 - \frac{\lambda}{n}\right)^n = e^{-\lambda}$, so $P(X = k) \to \frac{\lambda^k e^{-\lambda}}{k!}$, $k = 0, 1, \cdots$, which is the Poisson distribution (Equation [5]).

Finally we consider three important distributions for continuous random variables. A continues random variable is uniformly distributed over the interval $[a, b]$ if its probability density function is given by

$$f(x) = \begin{cases} \dfrac{1}{b-a}, & a \le x \le b \\ 0, & \text{otherwise} \end{cases}$$

with

$$E[X] = \frac{a+b}{2},$$
$$V[X] = \frac{(b-a)^2}{12}.$$

A random variable $X$ with mean $\mu$ and variance $\sigma^2$ is normal distributed if its probability density function equals

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[ -\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2 \right].$$

The normal distribution is widely used and is forms the basis of the central limit theorem (see e.g in the lecture 'stochastic simulation', [7]). The normal distribution can be derived from the binomial distribution in the limit of $n \to \infty$, $p$ is constant, and $k/n \to p$ (proof not shown).

If a process consists of $k$ sub-processes and the time needed for each sub-process is exponentially distributed, then the time for the total process is Erlang distributed, with probability density function

$$p(x) = \frac{(\alpha x)^{k-1}}{(k-1)!}\alpha e^{-\alpha x}, \quad x \ge 0,\ k = 1, 2, 3, \cdots,$$
$$E[X] = \frac{k}{\alpha},$$
$$V[X] = \frac{k}{\alpha^2}.$$

## F. References

1. Hoekstra, A.G.: Introduction Parallel Computing. Faculty of Science, University of Amsterdam, The Netherlands, (2001)
2. Sloot, P.M.A.: Simulation and Modelling. Faculty of Science, University of Amsterdam, The Netherlands, (2002)
3. Siegler, B.: Theory of Modeling and Simulation. (1976)
4. Cellier, F.: Continuous System Modeling. (1990)
5. Minski, M.: Models, Mind, Machines. (1965)
6. Korn, G.,Wait, J.: Digital Continuous System Simulation. (1978)
7. Sloot, P.M.A.: Computational Physics: Stochastic Simulation. Faculty of Science, University of Amsterdam, The Netherlands, (2002)
8. Hooper, J.W.: Strategy related characteristics of discrete event languages and models. Simulation **46** (1986) 153-159
9. Banks, J., Carson, J.S.,Nelson, B.L.: Discrete-Event System Simulation. Prentice Hall, (1999)
10. Chandy, K.M.,Misra, J.: Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. IEEE Transactions on Software Engineering **SE-5** (1979) 440-452
11. Chandy, K.M.,Misra, J.: Asynchronous distributed simulation via a sequence of parallel computations. Communications of the ACM **24** (1981) 198-205
12. Misra, J.: Distributed discrete event simulation. ACM Computing Surveys **18** (1986) 39-65
13. Fujimoto, R.M.: Performance measurements of distributed simulation strategies. Trans. Soc. Comput.Sim. **6** (1989) 89-132
14. Jefferson, D.R.: Virtual Time. ACM Transactions on Programming Languages and Systems **7** (1985) 404-425