Environments and Languages for Distributed Simulation

• The software implementation of the simulation is sometimes considered only as an afterthought by simulation designers.

• No project has the luxury of unlimited time.
  – While some might prefer to have a completely specified and refined model prior to beginning the software engineering process, time constraints alone make this an ill-advised strategy.
  – Many activities of simulation building and software engineering are complimentary.
  – Performing these activities sequentially rather than in parallel at best is wasting time and at worst increases the difficulty in maintaining consistency between the simulation model and the software implementation.

• Therefore, the software engineering process should start concurrently with the beginning of the simulation effort.
Software Implementation

• Unfortunately, a casual attitude towards software development may greatly complicate the verification and validation of the simulation.
  – Considerable time may be spent both debugging the software and fine tuning the model, so the choice of language platform and environment is important.
  – The software implementation is important since a validated model incorrectly implemented in software is not likely to produce useful data until the software errors are corrected.
  – Distributed simulation is of interest because of the larger, more interesting, problem domains which may be feasibly executed with the increased computing power distributed simulation makes available.
  – It is safe to presume that the power of distributed computing will most often be brought to bear on the most complex simulation applications. The software implementation of such systems is not trivial.
Choice of Environment

• Programmers, like everyone else, tend to prefer familiar environments to new environments.
  – The flexibility of software is such that there is rarely any one “right answer” when evaluating developmental environments and implementation languages.
  – However, this is a decision that will be lived with throughout the development, experimentation, and maintenance cycles of the simulation, hence it should not be taken lightly.

• As noted earlier in our definition of distributed systems, we are primarily interested in message passing as opposed to shared memory systems.
  – Message passing systems need only two basic functions added to the standard language support: send and receive [Karp 87].
Distributing Programs

• There are three general approaches to distributing programs.
• One is to build the needed instructions directly into the standard compiler, as in Ada 95.
• The second approach is to create special compiler versions which recognize inherent parallelism and add the necessary constructs at compile time.
  – The special Cray FORTRAN and C compilers would be examples of this strategy.
• The third approach is to use remote procedure calls (RPCs) or system calls to a distributed operating system to execute the necessary sends and receives.
Parallel Programming

- Few mainstream languages exist in which the standard versions support distributed programming [Volz 90].
- Most of the languages surveyed here will require RPCs or operating system support to be distributed.
- It is not our intent to detail parallel programming practices, this being a subject of several excellent textbooks already.
- For further reading on parallel and concurrent programming, we refer the reader to [Almasi 94] and [Andrews 91]
Simulation Languages

- There are many mature programming languages that were expressly designed to support simulation.
- Simulation languages typically provide specialized routines and libraries of special interest to simulation practitioners.
- As programming languages have advanced, it has become possible to add libraries of simulation routines to general-purpose languages.
- The continuing popularity and use of specialized simulation languages warrant their discussion, although they are not typically used in distributed implementations.
Specialized Simulation Languages

- [Graybeal 80] lists the following common services provided by specialized simulation languages:

1. Generating random variates.
3. Handling routines to simulate event executions.
5. Collecting data.
6. Summarizing and analyzing data.
7. Formulating and printing output.

How is this done in OPNET?
Most any system of interest will exhibit stochastic behavior.

- Typical stochastic elements might include interarrival times in a queuing system or equipment maintenance requirements in production systems.
- Good, minimal standard pseudorandom number generators exist.
- Programming a random number generator is a straightforward exercise. Unfortunately, many systems use poorly constructed random number generators [Park 88] that superficially appear to produce random numbers.
- It is subtly difficult to build a generator that produces (for practical purposes) infinite, statistically independent random numbers uniformly distributed. It is very important that the output of any random number generator be carefully checked using several of the well-known means.
Lehmer’s algorithm

- Lehmer’s algorithm is based on the careful choice of a modulus which is a large prime number and a positive integer multiplier.
- Random number generators based on this method are formally known as a prime modulus multiplicative linear congruential generator.
- For a detailed treatment of testing and validating random number generators, we refer the reader to [Pooch 93].
Time Management

• Time management is a major issue in simulation.
  – The two general methods are to use fixed-time increments or event-based clocks with variable-time increments.

• A specialized simulation language would be expected to support both methods of time management to drive a simulation.

• Execution of a scheduled event requires changes in system state by invoking the appropriate program module.

• Not surprisingly, all three of the simulation languages profiled in the textbook have built-in time management routines.
Simulation Time Management

• Simulation time can be managed by fixed-increment time advance (periodic scan) or variable-increment time advance (event scan).
• Fixed-increment schemes are generally easier to implement in a distributed system where clock synchronization is an issue.
• Ordering events that occur during each time interval may be problematic.
• [Law 91] observes that fixed-increment time advance is rarely used in discrete event simulation models when the times between successive events vary greatly.
• Variable-increment time advance schemes rely on the occurrence of events to determine when to update the clock.
Event Execution

• Events are scheduled for execution.
• As an event is executed, the system must be updated to reflect any state changes caused by the event.
• State changes are effected by the program, often by calling a module such as `decrement_queue` or `fire_mission(parameter list)`.
• The complexity and number of the required state changes will determine the complexity of the supporting module.
• Simple queue updates may only require a few lines of code.
  – The aftermath of a simulated artillery engagement may require several non-trivial routines to update ammunition counts and to effect state changes in targets.
Many models require efficient queue management.
- Whether the model is the classic queuing theory example of people waiting in line at a fast-food restaurant or a series of targets waiting to be “serviced” by attack aircraft, many systems involve competition for limited resources.

The representation and manipulation of waiting lines can be accomplished in many ways.
- A straightforward way of representing queues is by a list because the primary operations in queue management are the addition and deletion of queue elements.
- The use of pointers and a list structure is one easy way to handle queues.
- Considerable research and development have centered around new efficient list processing methods that include binary trees, indexed lists, multi–level indexed lists, and partitioned lists.
- The implementation of these techniques in a simulation model often results in an order of magnitude reduction in computation time.
- Thus a language with efficient list-processing capability offers a significant advantage in simulation.
Parameters and Data Collection

- Many simulation models are implemented to assess the effect on the system of varying certain conditions or parameters.
- This measurement and comparison requires the collection of data.
- There are two distinct philosophies for collecting data within a simulation: the classical approach and the total approach.
  - In the classical approach we define precisely what information and statistics are to be collected prior to the simulation and use effective methods for collecting and calculating those within the simulation.
  - In the total approach a database is created during the course of the simulation which consists of all the data and information that can be collected. An inquiry mechanism is used to extract any desired information from this database after the simulation.
• There are many criteria for selecting a programming language.

  – Specialized simulation languages offer significant advantages in simulation programming.
  – Special-purpose simulation languages were developed (beginning in the late 1950’s) because many simulation projects needed similar functions across various applications.
  – Many, though not all, were derivatives of FORTRAN.
  – The simulation languages shown at left are all more than thirty years old but are still being modified and used.
**Simula 67**

- The object-oriented paradigm is rooted in SIMULA.
- The SIMULA language is closely associated with discrete-event simulation, and it is a mature procedural language.
- SIMULA is often best recognized for its simulation features, which is why we categorize it as a simulation language.
  - However, SIMULA may also be used for non-simulation applications. SIMULA is based on the ALGOL programming language [Kreutzer 86].
- Although first developed thirty years ago, SIMULA is still in use, primarily in Europe and Australia [Sadiku 95].
- Wegner listed the development of SIMULA as one of the major milestones in the evolution of programming languages [Wegner 87].
- The implementation of the SIMULA class language construct was an important improvement over the ALGOL block language construct.
Coroutines

- **Coroutines** in SIMULA are implemented in class structures and is one way to **control concurrent execution**.
- The developers of SIMULA pioneered the concept of mutual control between procedures known as coroutines.

- Control is passed from one coroutine to another by means of a `RESUME` statement.
  - Coroutine A executes for some time and then passes control to coroutine B.
  - Later, coroutine B executes `RESUME A` and control returns to coroutine A.
- Order of execution is determined by the coroutines themselves.
- Coroutines do not support **true parallel processing** but rather the execution of one process at a time.
- Synchronization is implicit since only one coroutine executes at a time.
SIMULA Summary [Kerr 89]

- Conventional general-purpose ALGOL-style algorithmic capability.
- Object-oriented programming (classes) encompassing encapsulation, inheritance, information hiding, autonomous activity, and strong typing supporting the concepts of modularity, generalization, specialization, abstraction, polymorphism, and pseudo-parallelism.
- Basic features for manipulating text strings.
- File concept supporting sequential and direct access methods for byte- and record-structured files.
- Large repertoire of utility functions.
- Features supporting two-way linked lists. More complicated list structures such as trees and lattices are easily constructed from the basic class facilities.
- Features supporting discrete event simulation in various styles including the object-oriented process view.
SIMSCRIPT

- SIMSCRIPT was developed by the RAND Corporation in 1962 by Markowitz, Karr, and Hausner [Markowitz 63].
- The most current version of SIMSCRIPT is SIMSCRIPT II.5 which is proprietary.
- Originally, SIMSCRIPT was implemented as a FORTRAN preprocessor, but beginning with SIMSCRIPT I.5 in 1964 this was no longer the case.
- SIMSCRIPT executable files are created through the use of SIMSCRIPT compiler.
SIMSCRIPT Summary

- SIMSCRIPT was designed as a higher-order programming system [Kreutzer 86]. Thus, it may be used for applications other than simulation, as well as for a broad range of simulation applications.

- SIMSCRIPT programs consist of three parts:
  1. Preamble:
     - Block where global variables are defined.
     - Precedes main program.
     - Contains no executable statements.
  2. Main Program
     - Block where program execution begins.
     - Mandatory element for all SIMSCRIPT programs.
  3. Event Routines
     - Subroutines written for each event.
     - Executed when simulated event is scheduled.
GPSS

• The General Purpose Simulation System (GPSS) is a non-procedural language designed to model queueing systems. GPSS was developed in 1961 by [Gordon 78] and was designed for the express purpose of simulating the operation of discrete systems.
  – Interpreted and compiled versions of GPSS exist.

• GPSS provides the analyst with predefined blocks to construct queueing-based simulation models.
  – This may be viewed as advantageous in terms of easing the programming skills needed to use GPSS, but this ease comes at the cost of flexibility.
  – The analyst lacks the means to significantly change the predefined blocks provided by GPSS.
The simulated system is represented by a set of blocks connected by lines.

Each block represents some activity, and each line represents a path to the next activity as shown left.

Each block symbol is unique, thus providing a ready interpretation of the block diagrams.

For a more complete description of GPSS, see [Schriber 91].
FORTRAN

• Introduced in 1957, FORTRAN (for IBM Mathematical FORmula TRANslation System) was the first high-level general-purpose language to be standardized and is still in use more than forty years after it was first specified.
  – [Wegner 87] calls FORTRAN “the single most important milestone in the development of programming languages.”
  – FORTRAN has been widely used in simulation and is well surveyed in the literature [Graybeal 80][Pooch 93][Law 91].
  – FORTRAN remains the programming language of choice for many segments of the scientific community.
  – A large amount of simulation software has been written and is being maintained in FORTRAN. FORTRAN is likely to continue in use through the 21st century.
  – One reason for FORTRAN’s staying power is the rich set of scientific program libraries that have been thoroughly verified and validated.
Ada 95 Overview

• In 1975, the U.S. Department of Defense (DoD) began searching for a standard programming language that would support specific military requirements such as embedded real-time programming as well as general-purpose requirements.
• In 1983, Ada was designated as military standard MIL-STD 1815a, which also became the ANSI standard.
• The most recent revision is ANSI/ISO/IEC-8562:1995, popularly called Ada 95.
• Many of the programming language innovations pioneered in Ada are now taken for granted.
• Interest in Ada has not been limited to military applications. It is also used in many commercial sectors, particularly those areas requiring high reliability.
• Ada was designed from the beginning to strongly support software engineering principles. The design of Ada stresses the ease of reading code over the ease of writing code, thus greatly contributing to maintainability [Naiditch 95].
Ada and Event-Driven Simulation

- When a procedural language not specifically designed for simulation is used, one of two approaches may be used to build the simulation: event-driven simulation or time-driven simulation [Cook 93].
  - Event-driven simulation is desirable from a design, implementation, and verification/validation standpoint; however, time-driven simulation is often easier to code.
  - Time-driven simulation requires the system to perform a check of all possible events for each logical time tick. Although very easy to implement, this type of implementation typically leads to large, monolithic programs that are very difficult to verify.
Event-driven simulation

- Event-driven simulation allows closer modeling or abstraction of the real world.
- Each event from the real world that is being simulated is represented as a separate event within the program code.
- The scheduling and handling of these events, however, is difficult to program.
  - Often, an interrupt-driven implementation is the only alternative.
  - The resulting operating system overhead can be costly.
  - In addition, the scheduling and handling of interrupts requires coding at the operating system level.
  - This is difficult to program and even more difficult to test.
  - Minor changes to an interrupt-driven program can cause major changes in performance.
  - Because of the difficulty of using general purpose languages for event-driven simulations, specific simulation languages, such as Simscript or Simula, are often used.
Ada Tasking

- Ada, however, has constructs that allow event-driven simulations without resorting to operating system interrupts.
- The mechanism that allows this is the tasking model. The use of the tasking model in Ada to model the real world could best be called model-driven simulation.
- Ada easily allows each external event generator to be modeled as a separate task [Hamilton 95].
- In this manner, each can be modeled as a task call or an accept.
- Adequate timing features in Ada 83 exist, permitting tasks to use physical timing for event simulation.
- Each task can share common data with other tasks. This feature permits events to modify the state of the system without performing a task call or accept.
- In this manner, multiple tasks, each representing an event generator, can modify the system state quickly.
C/C++

- C++ was developed because its author wanted to write some event-driven simulations for which SIMULA67 would have been ideal, except for efficiency considerations [Stroustrup 91].
- C++ is based most closely on the C language.
- C is described non-pejoratively as a relatively low-level language [Kernighan 88].
- Stroustrup attached great importance to maintaining compatibility with C, which prevented him from cleaning up C syntax [Stroustrup 91].
  - Given the large amount of C code in use today, it is hard to criticize the decision for C++ to maintain backward compatibility with C.
  - In direct contrast to Ada, C is not a strongly typed language. There are advantages and disadvantages to a weakly typed language.
  - The essential point is that users of C/C++ need to be aware that C/C++ is a weakly typed language and defensive programming is appropriate.
  - As with most programming language design issues, trade-offs are involved.
Random Number Generators [Park 88]

• Random number generation is a concept of fundamental importance in many areas of computer science.
  – Knuth – The Art of Computer Programming
• Park and Miller establish a minimal standard for a multiplicative linear congruential generator.
  – multiplier 16807
  – prime modulus $2^{31}-1$
  – modulus or remainder operator.
  – Review
    • The % operator (Java) returns the remainder of two numbers. For instance 10 % 3 is 1 because 10 divided by 3 leaves a remainder of 1.
• Porting this RNG (or any other RNG) is not as easy as it seems.
Minimal Standards

- Easy to hack a procedure that will produce a strange, unpredictable sequence of numbers.
- This is NOT the same as producing a virtually infinite sequence of statistically independent random numbers, uniformly distributed between 0 and 1.
- Lehmer’s algorithm, correctly implemented, represents a good minimal standard to evaluate other RNGs.
  1. If the algorithm parameters are chosen correctly and
  2. If the software implementation is correct
Lehmer’s Algorithm [Park 88]

1. modulus: m – a large **prime** number
   – prime prevents generation to zero sequences
2. multiplier a – an integer in the range of 2, 3... m-1 and the subsequent generation of the integer sequence $z_1, z_2, z_3$ via the interactive equation --
3. $z_{n+1} = f(z_n)$ for $n = 1, 2.....$
   where the generating function $f()$ is defined for all $z$ in $1,2,......, m-1$.
4. $f(z) = az \mod m$.
5. $u_n = z_n/m$ for $n = 1,2$
   sequence of z’s must be initialized by choosing an initial seed $z_1$ from 1, 2...m-1
Picking a Seed / Multiplier

• Old lore: “Pick an initial seed with at least five digits, the last digit being odd.”
• Unnecessary and unacceptable for a minimal standard RNG.
  – All initial seeds between 1 and 2147483646 \( (2^{31}) \) equally valid.
  – Original authors suggest coding an initialize function:
    • which prompts for an initial value of seed and forces it to a value between 1 and \( 2^{31} \) (good!)
    • also suggest using individual SSNs as unique identifiers (bad!)
• \( a = 7^5 = 16807 \) largely because \( f(z) = 16807z \mod 2147483646 \) is a full period generating function
Example Bad Generator

- \( f(z) = (9806z + 1) \mod 131071 \)
- 131071 is the Mersenne Prime \( 2^{17} - 1 \)
- \( f(37911) = 37911 \)
- as long as 37911 is NOT used as a seed, works well, period of almost 131070.
Summary

• Fifteen years ago the distinction between simulation languages and general-purpose languages was clear.
• Simulation languages such as SIMSCRIPT and GPSS as well as SLAM and GASP offered important simulation features that had to be explicitly and tediously developed in a general-purpose language.
• As the software engineering discipline began to mature in the 1980s, software reuse received more attention. FORTRAN had long benefited from important software libraries to support scientific and mathematical programming.
• The development of simulation libraries for C++ [Fishwick 95] and Ada 95 [Tindell 94] are giving general-purpose languages the same capabilities as simulation languages while preserving the greater flexibility of the general-purpose languages.
Conclusion

• Language choice is often a matter of taste or driven by external requirements.
• Perhaps the single most consistent factor is the existence of legacy software and the familiarity and preferences of the programming staff.
• The existing software libraries written in FORTRAN ensure that FORTRAN will continue in use for many more years to come.
• Newer simulations in Ada 95, C++ and other languages.
• However, there is a wide body of experience and legacy code invested in the older simulation languages so we can expect them to remain in service for some time to come.