Research Idea: Filter-Based Discrete Event Simulation

Matthew J. Rutherford – mjr@cs.du.edu

November 7, 2008

1 Context

Discrete-Event Simulation (DES) is a powerful and efficient simulation paradigm used in many areas of science and engineering. Discrete-event simulations consist of active entities that contain the logic of the system (typically known as *processes*), and structured data elements known as *events* that represent the occurrence of an event at a particular process at a specific point in virtual time. State changes only occur in response to the occurrence of events, so there is no need, in a DES, to pick a time step and simulate the system at each time step.

There are many discrete-event technology platforms, some are specialized environments, some are libraries written in general purpose programming languages. In either case, scientists and engineers must, represent the state maintained by individual processes, "program" the logic of their processes, and design the structure of their events to match the problem they are trying to solve. Some subset of the state in entire simulation is sampled during the simulation execution; these data are subsequently analyzed, plotted and aggregated to aid in understanding of the problem, and to produce an answer to the research questions being considered.

One of the most appealing aspects of DES is its inherent efficiency: virtual time advances at whatever pace is most natural for the system being simulated, and there is no need to simulate the system at times for which there are no events occurring. The execution time of DES frameworks is critically important since scale and complexity of problems that can be solved is dictated by the time frame in which a simulation can be completed. Most simulation-based science and engineering is an iterative process in which the behavior and state of processes and the relevant events are adapted and enhanced as the understanding of a problem is better known. So, most problems are not simulated once, but rather repeated simulation runs are made with different values for initial conditions or parameters or to leverage randomness in the behavior of processes. For all of these reasons, the performance of DES engines is paramount, and many sophisticated techniques have been developed to parallelize and distributed the computation so that clusters and multicore machines can be brought to bear.

As with most software, there is often a trade off between simplicity, modularization and coherence on one hand and raw performance on the other. Many clean designs have been trampled in the search for improved performance. Similarly, programming sequential code is much more straightforward than programming a multi-threaded or otherwise parallel variant of the same software. This is a particularly unfortunate situation since many scientists and engineers who have need for high-performance simulation environments would like to be able to concentrate on their areas of specialty and not spend their time learning sophisticated programming and software engineering techniques or debugging and troubleshooting their code. Finally, the correctness of simulation software is paramount if the results and conclusions made from analysis of simulation output are to be used confidently. As simulation execution environments are made increasingly sophisticated to achieve high-performance, the testability of the logic is reduced.

2 Main Idea

Given that the use of DES is very important to science and engineering, that performance is king, and that the simplicity and coherence of the programming paradigm and environment are critical, we propose to develop a DES framework and environment that allows for sophisticated parallelization techniques while still keeping it simple to program.

There are several key points in the design of this environment that impact both the performance of the system, and also some aspects of the engineering of the software.

- **Processes as filters** – a filter is a simple program that reads its input from the standard input stream, and writes its output to the standard output stream. With process logic implemented in a filter paradigm, several things are now possible:
  - Total memory overhead in the system can be managed since the state of each process can be persisted in a database or on the filesystem and streamed into the relevant process as needed.
  - Since this is a fundamental feature of virtually every programming language, filters could be written in whichever languages make the most sense for the required functionality.
  - The logic of a particular process could even be implemented in several different languages partitioned by the attributes of the event that is triggering the processing.
  - Testing the logic of a process is as simple as sending inputs to the filter (e.g., through file redirection in a UNIX shell), storing the output of the filter into a file, and comparing the output with the known correct output (i.e., the *oracle*).
- **Separate maintenance of global state** – the execution engine would be responsible for reading the output of each filter and updating the global state of the simulation (in memory, in a file, or in a database) after the execution. This would enable some sophisticated speculative execution techniques in which filters were launched under optimistic assumptions about event scheduling (i.e., under the assumption that no events end up being scheduled during their next execution step), and if this assumption was found to be false, the output state of the filter could easily be discarded.
- **Varying execution engines** – the overall execution of the simulation can be managed in many different ways on a single uni processor system, on a multi- or manycore system, or on a parallel cluster of machines without having to change the implementation of the filters. For example, a relatively simple engine that forked a different process for each event scheduled at a particular time step could simply rely on the operating system scheduler to assign the computation to the available resources. A more sophisticated execution engine could leverage the resources of a cluster or the Grid, or. . . – all of this would be orthogonal to the development of logic by the end users, and would not even require recompilation (assuming a cluster had a homogeneous platform).

Other thoughts:

- The inputs to a filter would be: its identity (i.e., its simulation process id), the current virtual time, the identity of any other processes that it must know about, its current state, and the values associated with the event that is triggering the execution (the next item deals with the encoding of these data). The output would include the updated state of the process, the virtual time consumed, and any events scheduled by the process.
- The execution engine could be configured to pass the inputs and read the outputs from each filter in different ways. This could be global, or per-process. For the encoding types supported by the engine, libraries could be developed for popular programming languages to facilitate the development of parsing code in these languages (e.g., C, C++, Java, Python, Perl, . . .). This would also allow the transfer format to be both in ASCII or in binary as needed / desired.
- The persistence engine could be organized into experiments, and trials, and whatnot.

- Certain state items could be flagged as being collected, other state would be transient and could be purged from the global memory when appropriate.
- A particular simulation would be configured in a text-based file (maybe XML?) where the logic elements for each process type / or process instance could be specified, along with the topology of the simulation (i.e., how the processes are connected), and any initial state values for the processes.
- There might be need for a generative aspect to this, where events are described in XML or something and code to parse and generate the serialized event representations can be generated for all of the supported platforms.
- The engine would treat each filter as black-box and be able to use various performance measures to help debug bottlenecks and performance problems

3 Research Questions

- Can we achieve high-performance with this paradigm, what is the overhead imposed by copying the data streams, etc.
- Are the limitations imposed by the filter abstraction too stringent? Are they hard to program? Do non-programmers understand them?
- ...