

THE VISSIM/DISCRETE EVENT MODELING ENVIRONMENT

Herb Schwetman

Mesquite Software, Inc.
4210 Spicewood Springs Rd., #201
Austin, TX 78759, U.S.A.

Arun Mulpur

Visual Solutions, Inc.
487 Groton Road
Westford, MA 01886, U.S.A.

ABSTRACT

VisSim/Discrete Event is a process-oriented, discrete event modeling toolkit based on the powerful and proven graphical interface provided by the underlying VisSim simulation environment. Analysts can use this toolkit to construct models of many different kinds of systems. Each model has one or more task graphs and a set of simulated resources. The task graphs specify the sequence of actions which characterize the behavior of entities in the real system. The comprehensive collection of task action blocks and simulated resources mean that many kinds of systems can be modeled.

1 INTRODUCTION

VisSim/Discrete Event is a modeling environment which simulation analysts can use to construct process-oriented models of complex systems. The environment employs a powerful graphical user interface which helps the analyst construct models and analyze the results. The easy-to-use interface and the comprehensive set of simulation objects mean that models tailored to specific systems can be readily constructed.

A discrete event simulation model typically consists of active entities and simulated resources, all constructed to model the behavior of a real system. In VisSim/Discrete Event, the active entities are called *tasks*, and the behavior of a task is specified by a *task graph*. A task graph actually specifies the behavior of set of similar tasks. A model must have at least one task graph, but a model can have many different task graphs, each specifying the behavior of a different set of tasks (or entities).

In many models, the tasks compete for access to (use of) a set of simulated resources, where each simulated resource corresponds to a real component in the real system. In a real system, there may be many different kinds of components. In VisSim/Discrete Event, there are only four different kinds of simulated resources.

These each serve as an abstraction of a large class of real components in a real system.

VisSim/Discrete Event is built on top of the well know VisSim continuous simulation environment [Visual Solutions 1996]. The CSIM18 simulation engine [Mesquite Software 1997, Schwetman 1996] supports the discrete event capabilities provided in the modeling environment. Many of the features of VisSim, such as data presentation blocks and arithmetic computation blocks are useful in constructing discrete event models.

This paper begins by summarizing some of the features of VisSim/Discrete Event. It then presents two examples, illustrating both the power and ease-of-use of this modeling environment. It also discusses using some of the facilities provided by VisSim.

2 TASK GRAPHS

A model in the VisSim/Discrete Environment consists of a number of simulation objects and one or more task graphs. Each task graph depicts the behavior of set of tasks (or processes). There are two kinds of task graphs: open graphs and closed graphs. In an open graph, a *taskSource* node generates new instances of the task; the task generation rate is controlled by the parameters of the node. Open graphs often contain a *taskSink* node, to collect (or absorb) completed tasks.

A closed graph contains a *sourceSink* node. A sourceSink node generates a fixed number of tasks at the beginning of the execution of the model. After this initial phase, tasks that enter this node are delayed for a period of time and then emerge as new tasks.

A task graph consists of a collection of blocks (or nodes) connected by arcs. These directed arcs (also called flexWires) show the paths which tasks follow as they move from one block to another. The blocks control the behavior of the tasks and allow the tasks to simulate the behavior of the entities in the real system being modeled. The kinds of blocks which control task behavior include:

- *taskDelay* - delay for a specified interval of simulated time
- *route* - choose a subsequent path based on branching probabilities
- *computeResult* - modify the values of variables
- *evaluateRelation* - test the value of a variable
- blocks which reference simulation objects

Figure 1 illustrates the specification and operation of a simple task graph. In Figure 1, the *taskSource* block repeatedly emits a new task. The interval between each new task is determined by attributes of the block which are specified in a dialog box. In the example, each new task first travels to (visits) the *taskDelay* block. The amount of simulated time spent at this block is determined, for each task, by attributes of the block; these attributes are again specified in a dialog box.

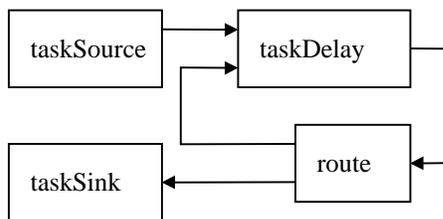


Figure 1: Task Graph

After experiencing a delay, each task next visits the *route* block. This *route* block has one input (for tasks) and two outputs for tasks. A probabilistic routing table routes incoming tasks to one of the output connectors at the block; the *i*-th entry in this table specifies the probability of exiting via the *i*-th output block connector. In the example, the task can either return to the *taskDelay* block (via connector 0) or move to the *taskSink* block (via connector 1). The *taskSink* block absorbs or terminates each task entering the block.

To summarize, a task graph specifies the behavior (sequence of actions) for each kind of task in a model. There can be many active tasks, all "following" the same task graph, and there can be several graphs in a single model.

3 SIMULATION OBJECTS

VisSim/Discrete Event provides four kinds of simulation objects:

- *serviceStations*,
- *storages*,
- *events*, and
- *mailboxes*.

A *serviceStation* consists of one or more servers and a single queue (for waiting tasks). Tasks access a

serviceStation by visiting either a *use* block or a *reserve* block. A task visiting a *use* block first reserves a server at the specified *serviceStation*; it then delays for an interval of simulated time and then releases the server. A task visiting a *reserve* block reserves a server at the specified *serviceStation* and then proceeds to other task action blocks; it will later visit a *release* or *releaseStack* block, to release the reserved server.

A *storage* consists of a collection of storage units (or tokens). A task can allocate a specified amount of storage (allocate a number of tokens); if the requested number of units is not available at the storage, the task will wait until a sufficient quantity of units has been deallocated by other tasks. A task allocates units of storage by visiting an *allocate* block; an *allocate* block "points to" or references a specific *storage* block. A task deallocates all of the storage units from a particular storage block by visiting a *deallocate* block.

An *event* is an object used to synchronize interactions between tasks. A task can wait for an event to occur by visiting a *wait* or *queue* block. Each of these blocks "points to" or references a specific *event* block. Another task can visit a *set* block ("pointing to" the same event block). This "set operation" changes the state of the event block, and allows all waiting tasks and one queued task at that event block to move on to other blocks.

A *mailbox* implements a means for communicating between tasks. A mailbox consists of two queues: one is a queue for unreceived messages and the other is for tasks waiting to receive a message. Both queues are FIFO (first-in, first-out) queues. A message is an integer-valued number. A task uses a *send* block to send a message to a mailbox. Another task can visit a *receive* block, to retrieve the next message from a mailbox. If there are no messages in the mailbox, the receiving mailbox waits until the next message arrives.

Many blocks have two exit connectors for tasks: one is labeled "pass" and is the path followed by tasks that "succeed" at the block; the other is labeled "fail" and is the path followed by tasks that "fail" at the block. An example of such a block is the *reserve* block: a maximum queue length can be specified for a *serviceStation*. If a task arrives at a *reserve* block and the queue length is at the specified limit, then that task will exit the *reserve* block as a failed task. If a task arrives at this *reserve* block and can join the queue; when it has obtained a server at the *serviceStation*, it will exit the *reserve* block via the "pass" connector. In addition to a maximum queue length condition, many types of blocks allow a "time limit" or time-out interval to be specified at a block.

4 FIRST EXAMPLE

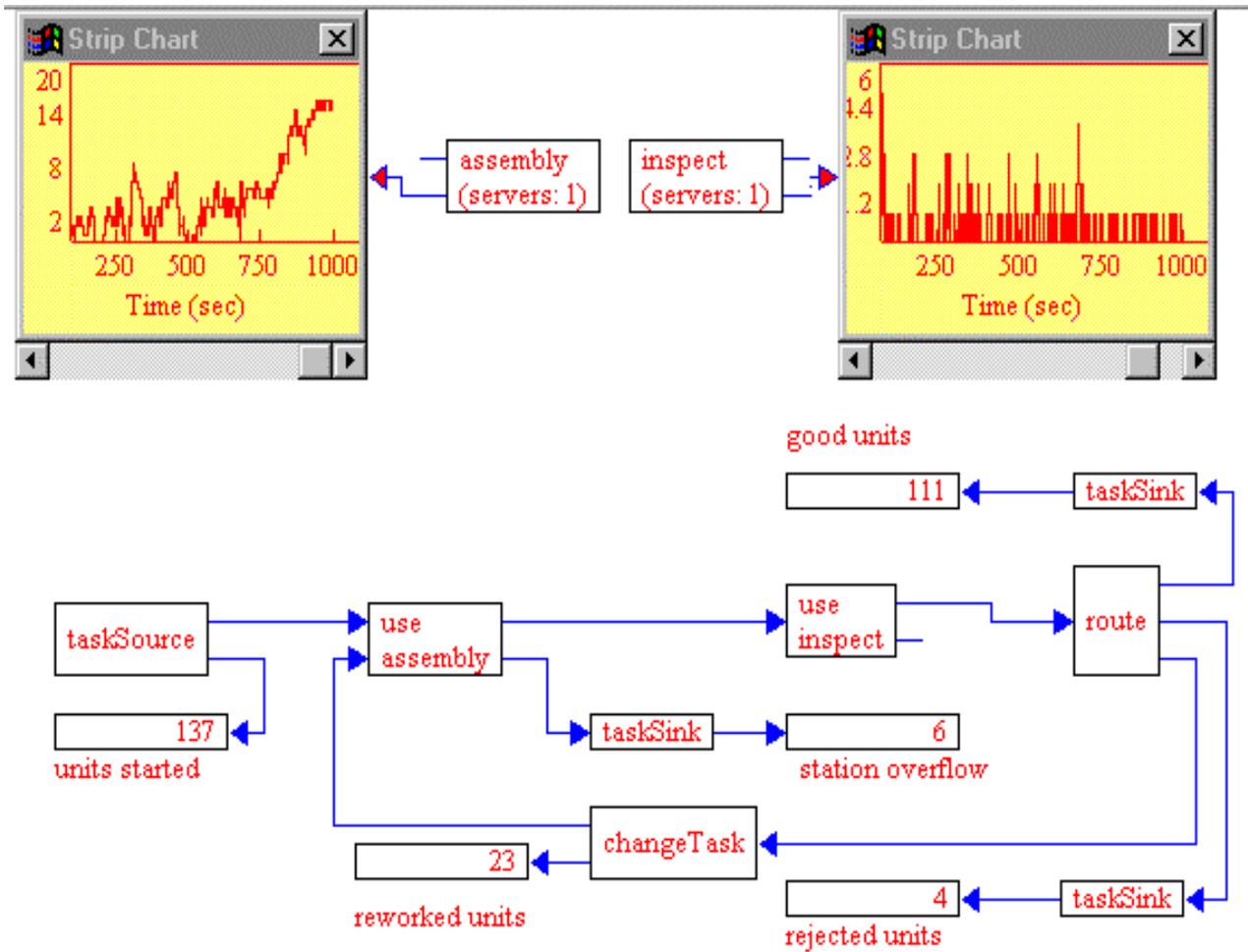


Figure 2: Assembly Station

Consider an assembly station where units arrive to be assembled. After assembly, the units then advance to an inspection station; units which pass the inspection leave the system; units which fail the inspection return to the assembly station, where they are reworked and then inspected again. Units which fail this inspection for the second time are discarded as rejects. The assembly station has a finite capacity for storing waiting units; the number of units at the station cannot exceed 15. When units arrive at the assembly station and the capacity will be exceeded, these units will be "lost".

In this example, the mean interarrival interval (for newly arriving units) is seven. The mean service interval at the assembly station is six units of time, and the mean service interval at the inspection station is two units of time. All intervals have negative exponential probability distributions. The probability of a unit passing the inspection is 0.8. A VisSim/Discrete Event version of this model appears in Figure 2.

The model in Figure 2 illustrates the use of several kinds of blocks. The two simulation objects are both serviceStations (the assembly station and the inspect station). The task graph begins with the source block on the left of the diagram. The parameters of the block specify the interarrival process for newly arriving units. The next block (the "use Assembly" block) represents the task activity associated with arriving at the station, waiting for access to the assembly "server", being assembled and then moving on as an assembled unit to the inspection station. The second input to the assembly station is the path used by units which failed the inspection and are being "reworked". The "fail" exit from the assembly station block is for units which cannot be held in the finite storage capacity of the station. These "lost" units go to a sink and the counter tallies the number of lost units. The parameters of the "use Assembly" block govern the service times at the block.

The "use Inspection" block represents the task activities associated with waiting for inspection and then being inspected by the inspector server. The parameters of this block control the inspection service time.

The route block routes tasks using the probability of passing or failing the inspection to select the output connector for the block taken by a task. The requirement that units which fail the inspection for the second time are rejected is handled using the "class attribute" for tasks. There can be two routing tables in the route block: one table for class 1 tasks and another table for class 2 blocks. In this example, class 1 tasks are those being assembled and inspected for the first time and class 2 tasks are those being reworked and inspected for the second time.

On leaving the route block, passed units exit the system via the upper sink block; the counter for this sink tallies the number of assembled units which passed the inspection. Failed class 1 tasks move to the *changeTask* block where the class attribute for the task is changed from 1 to 2. These tasks then move back to the "use Assembly" block for rework. Failed class 2 blocks leave the route block and move to a sink block; the counter for this sink block tallies the number of rejected units. The numbers in the lost, passed and rejected counters can be used to estimate the performance of this system.

This model can be enhanced in many ways. Different assembly times for newly arriving units and rework units can be specified by introducing an additional "use Assembly" block for the rework units. Introducing simulated travel times for units moving between stations can be accomplished by inserting *hold* blocks between pairs of blocks. The impact of having multiple assembly servers can be modeled by changing the number of servers as the "Assembly" (serviceStation) block.

5 SECOND EXAMPLE

As a second example, consider a model of a computer system processing computational tasks. Each task consists of a sequence of intervals in which both the CPU and one of three disk drives is accessed. These accesses occur simultaneously, but the task waits until both of these activities complete before advancing to the next interval. The number of intervals is uniformly distributed between 1 and 10.

Figure 3 gives the diagram for a model of this system. In this model, each arriving task needs a local variable, to hold the number of intervals (called count) for the task. The *fork* node spawns a sub-task, to perform the disk access, while the main task performs the CPU access. The CPU is represented by a single server serviceStation, and the disk is represented by a three-

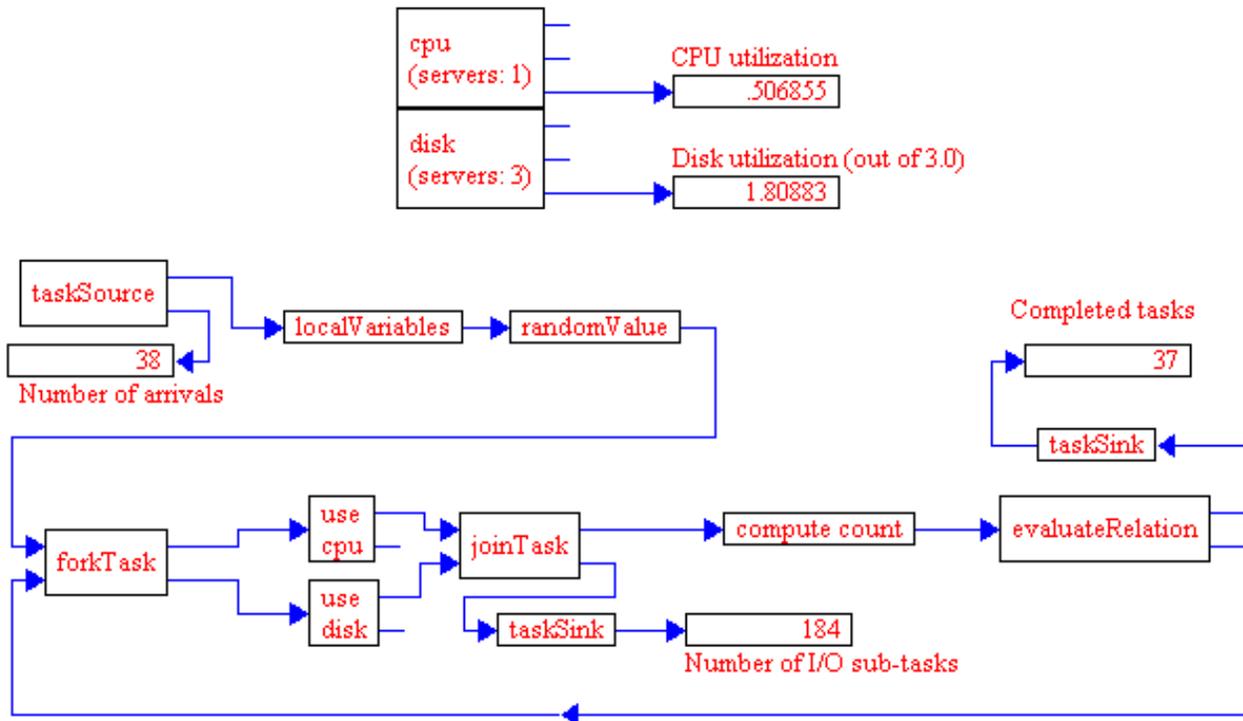


Figure 3: Computer System

server serviceStation. The *join* block forces the “parent” task to wait until the “child” task (the sub-task) has passed through. This is the synchronization that is required. The *compute* count and the *evaluateRelation* blocks decrement the “count” variable and then test, to see whether the task should perform another interval or exit as a completed task.

This kind of model is useful in determining the number disk servers and the CPU speed required to “keep up” with the arriving tasks. More complex models, with more resources and more complex behavior patterns can be easily constructed using this model as a starting point.

6 OTHER FEATURES

The underlying VisSim continuous simulation environment contains many features which are useful to VisSim/Discrete Event models. For example, *display blocks* and *strip charts* are often used to present data such as number of completed tasks at a block, server utilization at a serviceStation and queue length at an event. Examples of these blocks appeared in the two preceding examples. The comprehensive arithmetic computation blocks can be used to derive statistics useful to the model user. An example of this would be to calculate the percentage of arriving tasks which complete “successfully”.

In some applications, it is convenient to combine continuous and discrete event simulation techniques to construct a more accurate representation of the underlying system. An example of this would be an application in which the behavior of a component is best described by a time-dependent equation. The VisSim/Discrete Event *detector* block allows a task to wait until a value crosses a threshold value.

VisSim also has an animation block, which lets a model display some aspect of its behavior in a more visual fashion. This animation block lets an model select and display one of 16 bit-mapped images. This can be very helpful in display the changing state of a component within the model.

7 SUMMARY

VisSim/Discrete Event is a powerful simulation modeling environment with an easy-to-use and consistent graphical interface. The environment supports the task (or process) oriented approach to implementing discrete-event simulation models. The collection of simulation objects and task control blocks is used to construct models of many different kinds of systems. The examples above demonstrate just a few of the features available to the analyst building and using these models.

ACKNOWLEDGEMENTS

CSIM is copyrighted by Microelectronics and Computer Technology Corporation (MCC). CSIM18 is supported and marketed by Mesquite Software, Inc. under license from MCC. VisSim and felxWires are trademarks of Visual Solutions, Inc. VisSim/Discrete Event was developed by Mesquite Software, Inc. and is marketed under a joint arrangement with Visual Solutions, Inc.

REFERENCES

- Mesquite Software, Inc. 1997. *User's Guide, CSIM18 Simulation Engine*. Austin, TX.
- Schwetman, H. 1996. CSIM18 - The Simulation Engine. In *Proceedings of the 1996 Winter Simulation Conference*. ed. J. Charnes, D. Morrice, D. Brunner, and J. Swain, 517 - 521. San Diego, CA.
- Visual Solutions, Inc. 1995, *VisSim user's guide*, Version 2.0, Westford, MA.
- Visual Solutions, Inc. 1996. *VisSim/Discrete Event user's guide*, Version 1.0. Westford, MA.

AUTHOR BIOGRAPHIES

HERB SCHWETMAN is founder and president of Mesquite Software, Inc. Prior to founding Mesquite Software in 1994, he was a Senior Member of the Technical Staff at MCC from 1984 until 1994. From 1972 until 1984, he was a Professor of Computer Sciences at Purdue University. He received his Ph.D. in Computer Science from The University of Texas at Austin in 1970. He has been involved in research into system modeling and simulation as applied to computer systems since 1968.

ARUN MULPUR is a product manager at Visual Solutions, Inc., which he joined in 1994. He received his Eng. D. in Electrical Engineering from the University of Massachusetts, Lowell, in 1994.