

## **INSIDE DISCRETE-EVENT SIMULATION SOFTWARE: HOW IT WORKS AND WHY IT MATTERS**

Thomas J. Schriber

The University of Michigan  
Stephen M. Ross School of Business  
Ann Arbor, MI 48109-1234, USA

Daniel T. Brunner

Dan Brunner Associates  
P.O. Box 90343  
Indianapolis, IN 46290-0343 USA

Jeffrey S. Smith

Auburn University  
Industrial & Systems Engineering  
Auburn, AL 36849-5346, USA

### **ABSTRACT**

This paper provides simulation practitioners and consumers with a grounding in how discrete-event simulation software works. Topics include discrete-event systems; entities, resources, control elements and operations; simulation runs; entity states; entity lists; and their management. The implementations of these generic ideas in AutoMod, SLX, ExtendSim, and Simio are described. The paper concludes with several examples of “why it matters” for modelers to know how their simulation software works, including discussion of AutoMod, SLX, ExtendSim, Simio, Arena, ProModel, and GPSS/H.

### **1 INTRODUCTION**

A “black box” approach is often taken in teaching discrete-event simulation software. The external characteristics of the software are studied, but the foundation on which the software is based is touched on only briefly or totally ignored (for lack of time). Choices made in implementing the foundation and the impact of these choices on step-by-step model execution might not be studied at all. The modeler might then not be able to think things through when faced with developing good approaches for modeling complex situations, using interactive tools to come to an understanding of error conditions arising during model development, and using interactive tools to verify that complex system logic has been modeled correctly. The objective here, then, is to describe the logical underpinnings of discrete-event simulation and illustrate this material in terms of various implementations of discrete-event simulation software.

In Sections 2, 3 and 4 we comment on the nature of discrete-event simulation; basic simulation constructs such as entities, resources, control elements, and operations; and model execution. Section 5 discusses four specific implementations of entity management rules. Finally, Section 6 explores various aspects of “why it matters.” Throughout this paper, terms we define are **boldfaced** on first use. Tool-specific terms are Capitalized or, where appropriate, are spelled out in ALL CAPS.

This paper is a revised version of a similar paper that initially appeared in the 1996 Winter Simulation Conference Proceedings (Schriber and Brunner 1996). A version of the 1996 paper has been presented at the conference each year since 1996, with occasional updates, such as modifying the set of languages covered in detail. For example, the 1996 paper covered the entity-list management rules and “why it matters”

for SIMAN (Arena), ProModel, and GPSS/H. (A substantially expanded version of the 1996 paper containing figures, flow charts, and additional explanation can be found in Schriber and Brunner (1998)).

## 2 DISCRETE EVENT SIMULATION IN THE TRANSACTION FLOW WORLD VIEW

The “transaction-flow world view” often provides the basis for discrete-event simulation. In this world view, a system is visualized as consisting of discrete units of traffic that move (*flow*) from point to point in the system while competing with each other for the use of scarce (capacity-constrained) resources. The units of traffic are sometimes called “transactions,” giving rise to the phrase “transaction flow.” Numerous systems fit the preceding description. Included are manufacturing, material handling, service, health care, communication, and information processing systems, as well as other general queuing systems.

Fundamentally, a discrete-event simulation is one in which the state of a model changes only at a discrete, but possibly random, set of simulated time points, called event times. Two or more traffic units often have to be manipulated at one and the same time point. Such “simultaneous” movement of traffic is achieved by manipulating units of traffic *serially* at that time point. This leads to logical complexities because it raises questions about the *specific real-time order* in which two or more units of traffic are to be processed at a given *simulated* time.

The challenges faced by a *modeler* escalate for the *designer* of a modeling language. The designer must take the logical requirements of discrete-event simulation into account in a generalized way. Choices and tradeoffs exist. As a result, although discrete-event simulation languages are similar in broad terms, they can differ in subtle but important particulars. We seek to expand on this notion and provide specific examples in this paper. Before we do this, however, the following sections provide additional general simulation concepts.

## 3 ENTITIES, RESOURCES, CONTROL ELEMENTS, AND OPERATIONS

The term **entity** is used here to designate a unit of traffic (a “transaction”) within a model. Entities instigate and respond to **events**. An event is an instantaneous happening that changes the state of a model (or system). In a model of an order-filling system, for example, the arrival of an order (an event), might be simulated by bringing an entity into the model. There are two possible types of entities, here referred to as **external entities** and **internal entities**. External entities are those whose creation and movement is explicitly arranged for by the modeler (example: arrival of an order at an order-processing point). In contrast, internal entities are created and manipulated implicitly by the simulation software. For example, internal entities might be used in some languages to trigger simulated machine *failures* and implement machine *schedules*, whereas external entities might be used to simulate the *use* of machines for processing parts.

The term **resource** designates a system element that provides service (such as a drill, an automated guided vehicle, a worker, or space in an input buffer). Entities typically use resources (for example, a work-in-process entity claims space in an input buffer, then captures an automated guided vehicle to move the entity to the input buffer). Resources are usually capacity-limited, so entities compete for their use and sometimes must wait to use them, experiencing delay as a result (queueing). The term **control element** designates a construct that supports other types of delay or logical alternatives based on a system’s state. Control elements can take the form of switches, counters, user data values, and system data values built into the modeling tool. Complex conditions might be evaluated with arithmetic and/or Boolean expressions that examine the state of relevant control elements.

An **operation** is a step (or a series of steps) carried out by or on an entity while it moves through a system. The operations applicable to a ship at a harbor might include: arrive at the harbor; request a berth; capture a berth; request a tugboat; capture a tugboat; get pulled into the berth; free the tugboat; load cargo; request a tugboat; get pulled out of the berth; free the berth; get pulled into open water; free the tugboat; depart.

## 4 OVERVIEW OF MODEL EXECUTION

A simulation project entails running **experiments**. Experiments are differentiated by the use of alternatives in a model's logic and/or input data. For example, alternative part-sequencing rules and/or varying the numbers of various types of machines (resources) might be tried in the model of a production system. Similarly, the number of loading and unloading berths in a harbor might be varied to assess the impact on system performance. Each experiment consists of one or more **replications** (trials). A replication is a simulation that uses the experiment's model logic and data but its own unique set of random numbers, and so produces unique statistical results that can be analyzed as part of a set of such replications (all of which are independent). A replication consists of initializing the model, running it until a run-ending condition is met, and reporting results. This "running it" phase is called a **run**.

During a run the simulation **clock** (an internally managed, stored data value) tracks the passage of *simulated time*. The clock advances (automatically) in discrete steps (typically of unequal size) during the run. After all possible actions have been taken at a given simulated time, the clock is advanced to the time of the next earliest event. Then the appropriate actions are carried out at this new simulated time, etc. In essence, the execution of a run therefore takes the form of a two-phase loop: "carry out all possible actions at the current simulated time," and then "advance the clock," with these two phases repeated again and again until a run-ending condition (usually a modeler-specified simulated time, number of processed entities, or some other condition) comes about. These two phases are respectively called the **entity movement phase** (EMP) and the **clock update phase** (CUP) here.

### 4.1 Entity States

As the simulation executes, entities migrate from state to state while moving through a model. The five states typically used are:

- Active** The state of the currently moving entity is the **active state**. By 'currently moving entity' we mean the entity that is currently executing the decision logic in the model. Only one entity moves at any instant of wall-clock time. This entity moves until it encounters a delay of one type or other. It then migrates to an alternative state.
- Ready** During an entity movement phase there may be more than one entity ready to move, and yet entities can only move (be in the active state) one-by-one. The **ready state** is the state of entities waiting to enter the active state during the current entity movement phase.
- Time-delayed** The **time-delayed state** is the state of entities waiting for a known future simulated time to be reached so that they can then (re)enter the ready state. A "part" entity is in a time-delayed state, for example, while waiting for the future simulated time at which an operation being performed on it will end.
- Condition-delayed** The **condition-delayed state** is the state of entities delayed until some specified condition comes about, e.g., a "part" entity might wait in the condition-delayed state until its turn comes to use a machine. Condition-delayed entities are transferred automatically from the condition-delayed state to the ready state when the specified conditions permit.
- Dormant** Sometimes it is desirable or necessary to put entities into a state from which no escape will be triggered automatically by changes in model conditions. We call this state the **dormant state**. Dormant-state entities rely on modeler-supplied logic to transfer them from the dormant state to the ready state. Job-ticket entities might be put into a dormant state, for example, until an operator entity decides which job-ticket to pull next, with consequent transfer of the job-ticket entity to the ready state. In this case, the specific conditions in effect when selecting the job ticket might not be known until the time comes for the operator entity to select a job ticket.

## 4.2 Entity Management Structures

In our generic model, we manage the entities by designating an Active Entity and a number of *entity lists*. The **active entity** (the **active-state entity**) moves nonstop until encountering an (attempted) step that causes it to migrate to another entity state (transfers it to another list) or removes it from the model. A ready-state entity then becomes the next active-state entity. Eventually there are no more ready-state entities at the current time. The EMP then ends and a CUP begins. The following lists are used to organize the entities.

**Current Events List** Entities in the ready state are kept in a single list we call the **current events list** (CEL). Entities migrate to the CEL from the future events list, delay lists, and user-managed lists (each described below). In addition, entities cloned from the active-state entity usually start their existence on the CEL. CEL entities are generally ranked in first-in, first-out (FIFO) order. Some software tools provide a built-in entity Priority attribute used to order Entities on the CEL by priority (with ties resolved using FIFO).

**Future Events List** Entities in the time-delayed state belong to a single list into which they are inserted at the beginning of their time-based delay. This list, called the **future events list** (FEL) here, is usually ranked by increasing entity move time. Move time is the simulated time at which an entity is scheduled to try to move again (the end of the treatment time for a patient in a clinic or the end of the processing time for a part in a manufacturing system, for example). At the time of entity insertion into the FEL, the entity's move time is calculated by adding the value of the simulation clock to the known (sampled) duration of the time-based delay. After an EMP is over, the CUP sets (advances) the clock's value to the move time of the FEL's highest ranked (smallest move time) entity. This entity is then transferred from the FEL to the CEL, migrating from the time-delayed state to the ready state and setting the stage for the next EMP to begin. The preceding statement assumes there are no other entities on the FEL whose move time matches the clock's updated value. In the case of move-time ties, some tools will transfer all the time-tied entities from the FEL to the CEL during a single CUP, whereas other tools take a "only one entity transfer per CUP" approach. Languages that provide internal entities usually use the FEL to support the timing requirements of these entities. (The FEL then includes internal and external entities).

**Delay Lists** Lists (there can be many) of entities in the condition-delayed state are called **delay lists**. These entities are waiting (e.g., waiting their turn to use a machine) until their delay is resolved so they can be transferred automatically into the ready state on the CEL. Delay lists, which are generally created automatically by the simulation software, are managed using one of two types of waiting. If a delay can be related easily to model events that might resolve the delay, then **related waiting** can be used to manage the delay list. For example, suppose a machine's status changes from busy to idle. In response, the software can automatically remove the next waiting entity from the appropriate delay list and put it in the ready state on the CEL. Related waiting is the prevalent approach used to manage conditional delays. If the delay condition is too complex to be related easily to events that might resolve it, **polled waiting** can be used. With polled waiting the software checks routinely to see if entities can be transferred from one or more delay lists to the ready state. Complex delay conditions for which polled waiting can be useful include Boolean combinations of state changes, e.g., a berth is empty and a tugboat is idle.

**User-Managed Lists** Lists (there can be many) of entities in the dormant state are called **user-managed lists**. The modeler must take steps to establish such lists and usually must provide the logic needed to transfer entities to and from the lists. Except for very simple one-line, one-server service points in a system, the underlying software has no way to know why entities have been put

into user-managed lists in the first place, and therefore has no plausible basis for automatically removing entities from such lists.

## 5 IMPLEMENTATION IN FOUR TOOLS

The tools chosen here for commentary on implementation particulars are AutoMod (Phillips 1997); SLX (Henriksen 2000; Schulze 2008); ExtendSim (Imagine That Incorporated 2015; Krahl 2012); and Simio (Kelton *et al.*, 2014). A previous version of this paper (Schriber and Brunner 1996) covered SIMAN (Arena) (Kelton *et al.*, 2014), ProModel (ProModel Corporation 2015), and GPSS/H (Henriksen and Crain 2000) in similar detail. We think these tools are representative, but they clearly are not exhaustive (see Swain 2013 for a recent biennial survey of the many available tools).

### 5.1 AutoMod

The current events list is named the Current Event List (CEL) in AutoMod (see Table 1). Cloned Loads, Loads leaving the FEL due to a clock update, and Loads ordered off Order Lists are placed immediately on the CEL. The insertion rule is to rank first by priority (priority is a built-in attribute of every Load) and then FIFO within priority. When the CEL becomes empty, the Condition Delay List (see below) is checked, and Loads may be transferred from there to the CEL. This continues until the CEL is empty and no more Loads can be transferred, at which point the EMP is over and a CUP is initiated.

Table 1: AutoMod Terminology.

<b>Generic Term</b>	<b>AutoMod Equivalent</b>
External Entity	Load
Internal Entity	Logical Load
Resource	Resource; Queue; Block
Control Element	Counter; Process Traffic Limit
Operation	Action
Current Events List	Current Event List
Future Events List	Future Event List
Delay List	Delay List; Condition Delay List; Load Ready List
User-Managed List	Order List

The AutoMod Future Event List (FEL) is like future event lists in other tools. Loads arrive on the FEL in the time-delayed state by executing a WAIT FOR statement. AutoMod allows the specification of time units (day, hr, min, sec) in a WAIT FOR statement. The AutoMod CUP removes multiple Loads from the FEL if they are tied for the earliest move time, inserting them one by one into their appropriate place on the CEL. There are also internal entities in AutoMod, called Logical Loads, that do things such as wait on the FEL to trigger scheduled shift breaks.

Delay Lists (DL's) are lists of Loads waiting to claim capacity provided by any one of five finite-capacity elements (Resource, Queue, Block, Counter, or Process Traffic Limit; see Table 1). Each finite capacity element within the model has a DL associated with it. The waiting that results in these five cases is related waiting. Whenever capacity is freed, one Load from the head of the element's DL is tentatively placed on the CEL (but a placeholder is left on the DL). When that Load is encountered during the EMP, it tries to claim the requested capacity. If it fails (for example because it wants two units but only one is free), it is returned to the DL in its original place (this is default behavior that the modeler can override. See Section 6.2 for more general discussion). Immediately after this evaluation, if there is still unused capacity,

the *next* Load (if any) on the relevant DL is placed on the CEL. Processing of the active Load then continues, followed by evaluation of this next tentatively placed Load. And so on, tentatively, for each next Load (if any) during the EMP.

For conditional waiting apart from the five cases described above, AutoMod has a WAIT UNTIL statement that results in polled waiting. WAIT UNTIL conditions can be compounded using Boolean operators. If a Load executes a WAIT UNTIL and the condition is false, the Load is placed on a single global AutoMod list called the Condition Delay List (CDL). After the CEL has been emptied, but before the simulation clock is updated, *all* Loads on the CDL are moved to the CEL (actually, the CDL “becomes” the CEL) *if* there has been a state change for at least one element of the same general type (e.g. Queue) for which *any* Load on the CDL is waiting (this mechanism is primarily “polled,” where the polling process is triggered by a change in state of at least one element of the same general type).

If the CEL is now non-empty, the EMP resumes. If the condition for which a CEL Load is waiting is not yet satisfied, AutoMod moves that Load from the CEL back to the CDL. The CDL in some cases may be emptied multiple times during one EMP until eventually the CEL has been emptied without having triggered a state change related to any Load on the CDL. A CUP then occurs. Because of the potential for repetitive list migration with WAIT UNTIL, AutoMod’s vendor encourages the use of Order Lists or other explicit control mechanisms to manage complex waiting.

AutoMod implements the dormant state with Order Lists, which are user-managed lists of Loads. After a Load puts itself onto an Order List (by executing a WAIT TO BE ORDERED Action), it can only be removed by another Load (or another active model element such as a Vehicle) that executes an ORDER Action. An ORDER Action may specify a quantity of Loads, or a condition that must be satisfied for a given Load if that Load is to be ordered, or both. Loads successfully ordered are placed immediately on the CEL (one at a time according to how they were chosen from the Order List, and ranked on the CEL by priority, with priority ties resolved FIFO). Order Lists can achieve performance improvements over CDL waiting because Order Lists are never scanned except on explicit request. AutoMod Order Lists offer several interesting wrinkles, including: the ability for an ordering Load to place a back order if the ORDER quantity is not satisfied; the ability for a Load on an Order List to be ordered to *continue* to the next Action instead of to a Process (this feature is useful for control handshaking); and the ability to have a function called for each Load on the Order List (by using the ORDER...SATISFYING Action).

AutoMod has several material handling constructs that are integrated with Load movement. For vehicle systems there are three other types of lists (not included in Table 1). Loads on Load Ready Lists (LRL) (one list per vehicle system) are waiting to be picked up by a vehicle. Loads claimed (but not yet picked up) by a vehicle reside on the vehicle’s Vehicle Claim List (VCL). Claimed loads that have been picked up reside on the vehicle’s Vehicle Onboard List (VOL). The vehicle then becomes the active “load” and moves among AutoMod’s lists (FEL, CEL, and possibly DL’s) rather than the Loads themselves.

## 5.2 SLX

SLX is a hierarchical language in which the built-in primitives are at a lower level than most other simulation languages, facilitating user (or developer) definition of the behavior of many system elements. This design philosophy allows the SLX user (or developer) to create higher-level modeling tools whose constructs have precisely defined modifiable behavior. Equivalents for the generic terms for users of low-level SLX are given in Table 2. For example, SLX uses Control Variables to act as Control Elements. The “control” modifier can be attached to a Variable of any data type (integer, real, string, etc.). A Control Variable can be global, or it can be a local Variable declared in an Object’s Class definition. (A Class-declared Variable is an attribute in other tools.)

SLX has two types of Objects: Active and Passive. The two are distinguished by the presence of actions – executable Statements – in an Active Object’s Class definition (even without actions, Passive Objects are useful in their own right, functioning as user-defined complex data structures). Table 3 shows how higher-level SLX-based tools might exploit the definitional capabilities of SLX.

Table 2: SLX Terminology (Low-level).

<b>Generic Term</b>	<b>SLX Equivalent</b>
External Entity	Active Object and its Puck(s)
Internal Entity	none
Resource	Control Variable
Control Element	Control Variable
Operation	Statement
Current Events List	Current Events Chain
Future Events List	Future Events Chain
Delay List	Delay List
User-Managed List	Set (see below)

Table 3: Tools Based on SLX.

<b>Generic Term</b>	<b>SLX Equivalent</b>
Resource	Active or Passive Object
Control Element	Active or Passive Object
Operation	User-defined Statement
Delay List	User-defined (based on Set)
User-Managed List	User-defined (based on Set)

The current events list is named the Current Events Chain (CEC) in SLX. The members of the CEC are given the interesting name Puck. What is a Puck? SLX dissociates the concept of an Active Object (with its associated local data) from a Puck, which is the “moving entity” that executes the actions, carries its own entity scheduling data, and migrates from list to list. The effect of this dissociation is that a single Object can “own” more than one Puck. All Pucks owned by a single Object share the Object’s local data (attributes). For example, one application of this “local parallelism” feature (as compared with the “global parallelism” offered by “clone” or “split” actions in other languages) is the use of a second Puck to simulate a balk time while the original Puck is waiting for some condition. (If the condition comes about before the balk time has elapsed, no balking occurs; otherwise, balking does occur.)

Activating a new Object creates one Puck and launches that Puck into action. In many cases no additional Pucks are ever created for that Object, and the combination of an Active Object and its Puck forms the equivalent of an entity (passive Objects have no actions and therefore own no Pucks). Newly activated Pucks, Pucks leaving the FEC due to a clock update, and reactivated Pucks (see below) are placed on the CEC, ranked FIFO by priority. The CEC is empty when an EMP ends.

The SLX Future Events Chain (FEC) is like future events lists in other tools. Pucks arrive on the FEC in the time-delayed state by executing an ADVANCE statement. The SLX CUP will remove multiple Pucks from the FEC if they are tied for the earliest move time, inserting them one by one into their appropriate place on the CEC. Because the SLX kernel functionality does not include downtimes or even repetitive Puck generation (scheduled arrivals), all activity on the SLX FEC unfolds as specified by the developer of the SLX model. More generally, if a user is using a model (or is using a model builder) that contains higher-level primitives defined by a developer, chances are that all kinds of things are going on behind the scenes, hidden from the higher-level user’s view.

Delay Lists (DL’s) in SLX are lists of Pucks waiting (via WAIT UNTIL) for state changes in any combination of Control Variables and the simulation clock value. A Puck waiting for a compound condition involving two or more Control Variables is listed on more than one DL. All higher-level constructs defined by developers can use this mechanism. Each Control Variable (which may be a local Variable, in which case there is one for each Object in the Class) has a separate DL associated with it.

A DL is ranked by order of insertion. All pucks on a DL are removed whenever the associated Control Variable changes value and are inserted one at a time into the CEC. Removed Pucks that are waiting for compound conditions are also tentatively removed from each of the other Delay Lists to which they belong. As these Pucks are encountered on the CEC during the EMP, those failing to pass their WAIT UNTIL are returned to the Delay List(s) for those Control Variables still contributing to the falseness of the condition. For conditions that include a clock reference, the Puck is inserted if necessary into the FEC, subject to early removal from the FEC if the condition becomes true due to other Control Variable changes. This low-level related waiting mechanism based on Control Variables is the default SLX approach to modeling all types of simple or compound condition-delayed states.

SLX handles the dormant state in a unique way. Instead of moving the Puck from the active state to a user-managed list and suspending it, all in the same operation, SLX breaks this operation into two pieces. First, the Puck usually joins a Set. But joining a Set does not automatically suspend the Puck. A Puck can belong to any number of Sets. Set membership merely provides any other Puck with access to the member Pucks. To go into the dormant state, a Puck executes a WAIT statement. It then is suspended indefinitely, outside of any particular list, until another Puck identifies the waiting Puck and executes a REACTIVATE statement for it. Often this other Puck is scanning a Set to find the Puck to REACTIVATE, but a Set is not exactly the same as a user-managed list in our terminology. A dormant-state Puck might be a member of no Sets (as long as a pointer to it has been stored somewhere) or of one or more Sets. An SLX developer can easily define a user-managed list construct, using Sets, WAIT, and REACTIVATE as building blocks, that mimics those of other languages or offers its own unique features.

### 5.3 ExtendSim

ExtendSim (originally named Extend) uses a message-based architecture for discrete-event simulation. Various types of messages are used to schedule events, propel Items (Entities) through a model, enforce the logic incorporated into a model, and force computation. The senders and receivers of messages are Blocks (Operations), including the Executive Block (master controller). In ExtendSim, it is *Block* execution that is scheduled (when a Block executes, for example, this can trigger the sending of messages back and forth among Blocks, with the effect of moving an Item along its Block-based path in a model). Table 4 summarizes ExtendSim equivalents for the terms introduced in the earlier generic discussion.

Table 4: ExtendSim Terminology.

Generic Term	ExtendSim Equivalent
External Entity	Item
Internal Entity	Blocks/Messages
Resource	Block; Resource; Item; Resource Pool; Advanced Resources
Control Element	Block
Operation	Block
Current Events List	Next Times Array and Current Events Array
Future Events List	Time Array; Linked-lists within Blocks
Delay List	List of Items Resident in a Pre-Programmed Block
User-Managed List	List of Items Resident in a User-Programmed Block; Database Tables; Dormant Items in a Queue Equation Block



A Block is the basic modeling construct in ExtendSim. Each Block has an icon, message-passing connectors, dialog capability, and behavior-defining code. Residence Blocks can hold Items while simulated time goes by, whereas Passing Blocks cannot (items go through Passing Blocks in zero simulated time). Models can be constructed by selecting pre-programmed Blocks from ExtendSim's Block libraries. The modeler can also modify the source code given for library Blocks (all Blocks in the base version of ExtendSim are open source). Finally, the modeler can create customized Blocks from scratch (user-programmed Blocks) using development tools that ExtendSim provides.

ExtendSim uses a Time Array to schedule future Block executions. For a given model, the Time Array contains one or more elements for each Block. A Time Array element records the future time for which execution of that Block has been scheduled. The potential for a Block to have more than one Time Array element is an enhancement in Version 7 of the language (Version 8 is the current version). This feature can be useful when a Block has multiple, dissimilar events, as for example in conveyor modeling.

Blocks not currently scheduled for future execution are temporarily "blacked out" by recording arbitrarily large time values for them in the Time Array. Residence Blocks that can hold multiple Items manage the corresponding event times internally, with only the earliest of the Block's event times kept in the Time Array. This is a two-stage event list, as Blocks can contain optimized linked-lists that serve as their own future event lists. Block execution can result in scheduling future Block executions. For example, if messages are passed that result in an Item entering a unit-capacity Residence Block designed to hold the Item until a sampled amount of simulated time has elapsed, then the Time Array entry for that Block will have its value set accordingly.

The number of Blocks in a given model is constant, which means the Time Array is of fixed and relatively small size. Because of its small size, the Time Array is searched to find the imminent event time; it is not kept in sorted order. This makes it straightforward for a Block to change its event time because no searching of the event list is required.

The Next Times Array is used to manage the execution of Blocks whose execution has been scheduled via the Time Array. The Next Times Array is populated just prior to a Block Execution Phase (ExtendSim's equivalent of an EMP) as follows. At each CUP, the Time Array is searched to find the earliest future time at which a Block execution has been scheduled. Identifiers for the corresponding Block (or Blocks, in case of time ties) is (or are) then put into the Next Times Array. The Block Execution Phase (BEP) then begins, with the Executive messaging the most highly qualified Block in the Next Times array to start its execution.

The Current Events Array is used to manage the *resumption* of execution of Blocks whose execution has been temporarily suspended during the course of a Block Execution Phase. For example, suppose a Block sends a message, and the receiving Block replies (returns control) immediately to the sending Block (even though the receiving Block still has to do additional processing at the simulated time in question). In this case, the receiving Block's identifier is added to the Current Events Array. When the sending Block is finished executing, the Executive sends a message to the most highly qualified Block in the Current Events Array to resume its execution. Eventually, the Current Events Array becomes empty. Then the Executive turns again to the Next Times Array, sending a message to the most highly qualified Block to start executing.

During a Block Execution Phase, Blocks can schedule themselves to be executed at the *current* simulated time (that is, during the ongoing BEP). The Current Events Array comes into play here, too, to manage the execution of Blocks in such cases. For example, if a capacity-constrained Block becomes non-full as a result of some other Block's execution, the non-full Block puts its identifier into the Current Events Array. The Executive will later (but at the same simulated time) send a message to the Block to start executing. The Block will then try to pull into itself Items (if any) that have been waiting to enter the Block (in ExtendSim, Items can be both pulled and pushed through a model). When the Current Events Array and the Next Times Array both become empty, this brings ExtendSim's Block Execution Phase to an end. Then the next CUP and BEP take place, repeating until a simulation-ending condition is satisfied.

Delay lists consist of Items delayed in Residence Blocks, waiting their turn to be pulled or pushed into their next Block(s). Message passing is used to accomplish the pulling and pushing when model conditions permit. ExtendSim provides related-waiting management of these Items based on user-specified first-in, first-out (FIFO), last-in, first out (LIFO), Priority, Attribute, Reneging, Matching, and Equation-based alternatives. Waiting for the resolution of compound conditions is normally achieved in ExtendSim by appropriately combining Blocks and exploiting ExtendSim's message-based architecture. We view this as a form of related waiting, because it is a change in an underlying value that triggers a re-evaluation of the condition that brought about the waiting in the first place.

Because of ExtendSim's messaging architecture, polled waiting is generally not necessary. A message is sent when a value changes and any conditions are evaluated at that moment. Waiting for a clock based event can be achieved by using a Block that schedules events, e.g., Shift; Lookup Table; Equation. These Blocks send a message at scheduled times. Polled waiting is available, however, with use of the Gate Block and selection of its "Check demand at each event" option.

The modeler can work with User-Programmed Blocks to create and manage lists of the modeler's own design. The code for custom Blocks can be written to achieve the modeler's objectives in this regard, just as the code for ExtendSim's pre-programmed Blocks has been written to specify the behavior of those Blocks. ExtendSim provides functions that can be used by Blocks to share lists (arrays) with other Blocks, further supporting customized list management in models.

## **5.4 Simio**

Simio is an object-oriented language in which all model constructs are Simio Objects derived from the same base object. Most Simio users will primarily use objects from the Simio Standard Library for constructing models. Entities, Resources, Servers, Workstations, Sources, Sinks, Nodes, and Connectors are commonly used Objects from the Standard Library. Underneath, Simio is based on Processes that are comprised of individual Process Steps executed by Tokens. Standard Library objects' behaviors are implemented (by the object designer) using processes. While most models will be built using these high-level Objects, users (and developers) have complete access to Simio processes and can develop complete models as well as reusable Objects using Simio processes. In addition, users can augment the behavior of existing Objects using Add-on Processes and/or by Subclassing existing Objects. Simio models themselves are actually Objects that can be embedded in other models. The object structure of Simio facilitates the development of custom Object libraries that can simplify the modeling process in different application domains.

Equivalents for the generic terms for Simio users are given in Table 5. Note that any Simio object can act as a resource by setting the "Resource Object" property to True. The Resource object from the Standard Library is analogous to a resource in the context of this paper. The relationship between Simio entity objects and tokens is similar in concept to the relationship between the active object and pucks in SLX. In particular, as objects interact (e.g., an Entity object moving through a Server object to queue for and access a finite-capacity resource), one or more tokens associated with these objects are executing the processes (sequences of steps) that comprise the objects' behavior. The capability of having multiple tokens executing different processes on behalf of the same object provides modeling flexibility.

Object Properties and States serve as Simio control elements. Properties and States are both attributes of Objects. Properties are set at the start of a run and do not change during the run, whereas States can be set and changed at any time during a model run. Properties are typically used during model construction and experimentation and States are typically used during a model run to control flow and/or track and report statistics. Events are a bit more abstract in Simio than in some other packages. In Simio, events are characterized by an execution time (the simulation time when the event occurs), a reference to a procedure call (to be executed when the event occurs), and a reference to an Object (which provides additional data to the procedure, when necessary). An event's referenced Object may or may not be a user-visible Entity or associated Token. For events associated with Entity movement, the event Object refers to a Token associated with the Entity and the event procedure executes a process step (called a "token arrived to a step")

event). Most events that occur during a model run will be of this type. Examples of other, internal events include end-of-run events, States crossing a threshold events, Entity collision events, etc. Simio also supports user-defined events that can be fired, waited for, and used to trigger processes.

Table 5: Simio Terminology.

Generic Term	Simio Equivalent
External Entity	Entity/Token
Internal Entity	Token
Resource	Object (Resource, for example)
Control Element	States (Model states, Entity states, Resource states, for example)
Operation	Step
Current Events List	CEL – Current Event List
Future Events List	FEH – Future Event Heap
Delay List	Queue/Allocation Queue
User-Managed List	Storage Element

During model execution, once the CUP occurs and Simio enters the EMP, all events scheduled for the current simulation time are removed from the Future Event Heap (FEH) and are placed on the Current Event List (CEL). The heap data structure is used for future events because of its computational efficiency. Simio executes all of the events on the CEL (sequentially) before advancing the simulation time to the next event time. Events on the CEL are prioritized by event Type (normal, early, and late). Event procedures can create other events (future events as well as events scheduled for the current simulation time), update the system state, trigger the recording of statistics, and terminate the model run.

Allocation Queues are lists of entities waiting for resources. When a resource becomes available, Simio goes through a standard reallocation process whereby the “first” Entity in the Allocation Queue (determined by the queue ranking rule) will be placed at the front of the CEL and the Entity that released the resource will be processed through steps until it reaches a delay (either a time delay, a condition delay, or a user-specified storage). Under this process, the last releasing Entity is processed until it reaches a delay and the last Entity that was allocated resource capacity is the first to begin processing (at the same simulation time). Simio implements the dormant state using Storage or Station elements. A Storage element implements a *logical* queue where Objects can be “placed” by the user for later removal. Storage elements have no physical location and a given Object can simultaneously reside in multiple Storages. A Station element, on the other hand, defines a capacity-constrained physical location where Entity Objects can be stored for later removal and processing.

## 6 WHY IT MATTERS

In Sections 6.1-6.5 we describe situations that reveal some practical differences in implementation particulars among Arena (which uses the SIMAN language and processor), ProModel, GPSS/H, AutoMod, SLX, ExtendSim, and Simio. None of the alternative approaches mentioned is either intrinsically “right” or “wrong.” The modeler simply must be aware of the alternative in effect in the simulation software being used and work with it to produce the desired outcome. Otherwise, it is possible to mis-model a situation and perhaps not become aware of it. In Section 6.6, we comment on how knowledge of software internals is needed to make effective use of model checkout tools. Finally, in Section 6.7, we point out that knowledge of internals aids in understanding performance monitoring.

## 6.1 Trying to Re-capture a Resource Immediately

Suppose a job in a flexible job shop releases a machine (for which other jobs are waiting), then, as what will be its next step, decides to re-capture that machine. Will the job re-capture the machine immediately, or will a waiting job (even whether less qualified or equally qualified) capture the machine instead?

Of interest here is the order of events following the releasing of a resource. There are at least three alternatives: (1) Coupled with the resource-release event is the *immediate* choosing of the next user of the resource, without the releasing entity having yet carried out its next step; (2) The choosing of the next resource user is *deferred* until the releasing entity has potentially become a contender; and (3) Without paying heed to other contenders, the releasing entity recaptures the idle resource immediately. Arena, ExtendSim, and Simio (the default behavior) implement (1). ProModel implements (2). GPSS/H and AutoMod implement (3) by default. In SLX, using a Control Variable as the resource state, the result is also (3). (In some of the tools, modelers can implement higher-level resource constructs or use additional instructions so that models behave according to the modeler's choice in this regard.)

## 6.2 The First in Line is Still Delayed

Suppose two condition-delayed entities are waiting in a delay list because no units of a particular resource are idle. Suppose the first entity needs *two* units of the resource, whereas the second entity only needs *one* unit. Now assume that one unit of the resource becomes idle. The needs of the first list entity cannot yet be satisfied, but the needs of the second entity can. What will happen?

There are at least three possible alternatives: (1) Neither entity claims the idle resource unit; (2) The first entity claims the one idle resource unit and waits for a second unit; and (3) The second entity claims the idle resource unit and migrates to the ready state. As in Section 6.1, each of these alternatives comes into play in the tools considered here. Arena (SEIZE) and ProModel (GET or USE) implement (1) and (2) respectively, by default. AutoMod (GET or USE), GPSS/H (ENTER or TEST) and SLX (WAIT UNTIL on a Control Variable) implement (3) by default. ExtendSim also implements (3) by default. But ExtendSim gives the modeler the choice of locally implementing (1) for resources specified by the modeler. The modeler does this by checking an "Only allocate resource pool to the highest ranked Item" option for each such resource. Simio allows the modeler to select option (1) or (2) by using the Number of Objects property and the "repeating resource seizes" option with the Seize step.

## 6.3 Yielding Control Temporarily

Suppose the active entity wants to give control to one or more Ready-State entities, but then needs to become the active entity again before the simulation clock has been advanced. This scenario might come into play, for example, if the active entity has opened a switch permitting a set of other entities to move past a point in the model, and then needs to re-close the switch after the forward movement of the other entities has been accomplished. As an example, consider a case where a group of identically flavored cartons of ice cream is to be transferred from an accumulation point to a conveyor leading to a one-flavor-per-box packing operation.

In Arena, the effect can be accomplished *approximately* with a DELAY that puts the active entity into a time-delayed state for an arbitrarily short *but non-zero* simulated time. In ProModel, "WAIT 0" can be used to put the active entity back on the FEC. The entity will be returned later (wall-clock time, but at the same simulated time) by the CUP to the active state. In GPSS/H, the active Xact (Transaction) can execute a YIELD Block to migrate immediately from the active state to the ready state (positioned last in its priority class) and force a restart of the CEC scan. Higher-ranked CEC Xacts are then given a chance to become active before the yielding Xact becomes active again at the same simulated time.

SLX (YIELD), AutoMod (Wait For 0), and Simio (Delay step with Math.Epsilon) all provide solutions in which the active Puck (SLX), Load (AutoMod), or Token (Simio) is shifted to the back of its priority class on the CEC, where it waits to become the active entity again before the clock has been advanced.

In ExtendSim, “yield and then eventually resume” is part of the architecture. A message is sent out through the appropriate Block connector when an Item moves into or out of a Block. This message propagates to other connected Blocks, perhaps changing system status or moving Items from one Block to another as a result. When the originating Block eventually receives the reply, it continues processing the original Item.

#### 6.4 Conditions Involving the Clock

Every language provides a time-delay capability for FEL waiting. This works well when an entity needs to wait until a known clock value has been reached. But what if an entity needs to wait for a compound condition involving the clock, such as “wait until a specified input buffer is empty *or* it is exactly 5:00 PM?” A typical approach to this is to clone a dummy (“shadow”) entity to do the time-based waiting. Management of such dummy entities can be cumbersome, particularly for very complex rules. ProModel does not use polled waiting, so a dummy entity would be the best approach available. (Otherwise, the condition would not be checked until the other component of the compound condition had a value change.) ExtendSim also does not use polled waiting, so a similar situation applies for ExtendSim, and any Block that can schedule events can be used. With Simio, entity objects can have multiple associated tokens, each waiting for a different component of the compound condition. This is similar in concept to using a dummy entity, but does not require an additional entity.

When a polled waiting mechanism is present, if a single entity tries to wait for a compound condition involving the clock, an interesting problem can arise. This is because the next polling time may not match the target clock time. Arena and AutoMod detect the truth of compound conditions via their end-of-EMP polling mechanisms. GPSS/H also detects the truth via its version of polled waiting (refusal-mode TEST). But in the absence of a something that waits on the FEL until exactly 5:00 PM (i.e., the approach recommended above for ProModel, ExtendSim, and Simio), all three of those tools are subject to the possibility that the first EMP that finds the condition true occurs when the clock has a value *greater* than 5:00 PM. This can be problematic if the exactness of 5:00 PM is important.

SLX recognizes the clock as a related wait-until target. A WAIT UNTIL using a future clock value in a way that contributes to the falseness of the condition will cause the Puck to be scheduled onto the FEL to force an EMP at the precise time referenced. This solves the greater-than-the-desired-time problem. Note that this Puck may also be waiting on one or more delay lists.

#### 6.5 Mixed-Mode Waiting

Suppose many entities are waiting to capture a particular resource, while a user-created controller entity is waiting for the compound condition “the shift status is ‘off-shift’ *and* number waiting is less than six *and* resource is not currently in use” to take some action (such as shutting the resource down, in languages that allow user-defined entities to shut down resources; or displaying a status message). How can we guarantee that the controller entity will be able to “cut in front” of the waiting entities at the appropriate simulated time (before the idle resource is recaptured)?

One way to handle this would be through entity priorities in languages that offer this feature. However, as described below, that might not work even if the controller has relatively high priority. The key issue is the method used to implement the waiting. If it is “related” for the entities waiting to capture the resource and “polled” for the controller entity waiting for the compound condition (this is what we mean by the term “mixed-mode waiting”), things can get complicated. Every time the resource becomes free, a new entity will be selected from a delay list immediately in Arena and via the CEL in AutoMod, in both cases preceding the end-of-EMP checking for polled wait conditions (and thereby ignoring the entity priority of the

controller). There are ways to work around this if desired, such as using a different type of operation to force a polled wait for entities wishing to use the resource.

In GPSS/H, using a high-priority controller Xact at a refusal-mode TEST Block, the controller waits at the front of the CEC. The Facility RELEASE triggers a scan restart and the controller does its job. In ProModel there is no polled waiting but there can be related waiting on compound conditions involving Variables. Variables would have to be defined and manipulated for each element of the Boolean condition and, to assure equal competition, the entities waiting to capture the resource might also have to use WAIT UNTIL instead of GET or USE. Another possibility with ProModel would be to have the entity that frees the resource do some state-checking right away (in effect becoming a surrogate for the controller). This is possible because of the deferred-selection method used by ProModel (see Section 6.2).

In the related waiting of SLX, a Puck awaiting a compound condition will be registered on the delay lists of those (and only those) Control Variables that are contributing to the falseness of the condition. The SLX architecture (in which only global or local Control Variables and the clock can be referenced in any sort of conditional wait at the lowest level) assures that there will already be Variables underlying the state changes being monitored. The modeler defines them as Control Variables.

As with ProModel and SLX, ExtendSim would use related waiting to detect and immediately respond to a change in the compound condition. Because the state change of the Resource is immediately broadcast as a message before the Resource is reallocated, this message can be used to control the sequence and logic of Item selection. A Resource is not allocated unless there is both downstream space for an Item and Resource availability, so blocking the output path from a Queue will prevent Resource allocation.

In Simio, Resource Objects can have their own logic implemented using Add-on Processes, so there is no need to use “controller entities” to implement the control mechanism. In this case, The “Off-Shift” and “Released” add-on processes can be used to check for the complex condition involving the shift status, queue size, and Resource status. Note that this is equivalent to related waiting in that the condition will only be checked when the resource goes off-shift and/or when a unit of capacity is released.

## **6.6 Interactive Model Verification**

We now comment briefly on why a detailed understanding of “how simulation software works” supports interactive probing of simulation-model behavior. In general, simulation models can be run interactively or in batch mode. Interactive runs are of use when checking out (verifying) model logic during model building and when troubleshooting a model if execution errors occur. Batch mode is then used to make production runs.

Interactive runs put a magnifying glass on a simulation while it executes. The modeler can follow the active entity step by step and display the current and future events lists and the delay and user-managed lists as well as other aspects of the model. These activities yield valuable insights into model behavior for the modeler who knows the underlying concepts. Without such knowledge, the modeler might not take full advantage of the software’s interactive tools or, worse yet, might not even use the tools.

## **6.7 Performance Issues**

Simulation experiments can consume substantial amounts of computer time. Other things equal (including the model builder’s skill), computer-time requirements depend on the design and implementation of the software used to build models. This dependency can be understood with knowledge of “how simulation software works.” For example, consider user-managed lists vs. related waiting in models in which large numbers of entities contend for a resource. Performance is an important enough issue to motivate some simulation software (e.g., ExtendSim; SLX; and Simio) to supply performance profilers which, for example, can produce histograms showing where CPU time is spent during model execution.

## ACKNOWLEDGMENTS

Much of the information in this paper was provided by vendor personnel. The authors acknowledge the support of Deb Sadowski; Vivek Bapat; Charles Harrell (ProModel); Kenneth Farnsworth and Tyler Phillips (AutoMod); Robert C. Crain and James O. Henriksen (GPSS/H and SLX); David Krahl (ExtendSim); and David T. Sturrock and C. Dennis Pegden (both originally with Arena/SIMAN, and now with Simio).

## REFERENCES

- Henriksen, J. O. 2000. "SLX: The X is for Extensibility." *Proceedings of the 2000 Winter Simulation Conference*, edited by J. A. Joines, R. Barton, K. Kang, and P. Fishwick, 183-190, Piscataway, NJ: IEEE.
- Henriksen, J. O., and R. C. Crain. 2000. "GPSS/H: A 23-year retrospective view." *Proceedings of the 2000 Winter Simulation Conference*, edited by J. A. Joines, R. Barton, K. Kang, and P. Fishwick, 177-182, Piscataway, NJ: IEEE.
- Imagine That Incorporated. 2015. *ExtendSim 9.2 User Guide*. San Jose, California. Imagine That Inc.
- Kelton, D., J. S. Smith and D. T. Sturrock. 2014. *Simio & Simulation: Modeling, Analysis, Applications*, 3<sup>rd</sup> Ed. Sewickley, PA: Simio, LLC.
- Kelton, D., R. Sadowski, and N. Zupick. 2014. *Simulation with Arena*, 6<sup>th</sup> Ed. New York, NY: McGraw Hill.
- Krahl, D. 2012. "ExtendSim: A History of Innovation." *Proceedings of the 2012 Winter Simulation Conference*, edited by C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. Uhrmacher.
- Phillips, T. 1997. "Know your AutoMod Current Events." In *AutoFlash* 10(7). Bountiful, Utah: AutoSimulations, Inc.
- ProModel Corporation. 2015. *ProModel Version 9.2 User Guide*. Orem, UT: ProModel Corporation.
- Schriber, T. J., and D. T. Brunner. 1996. "Inside Simulation Software: How It Works and Why It Matters." *Proceedings of the 1996 Winter Simulation Conference*, edited by J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, 23-30. Piscataway, NJ: IEEE.
- Schriber, T. J. and D. T. Brunner. 1998. "How Discrete-Event Simulation Software Works." Chapter 24 in *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*, ed. J. Banks. New York, NY: John Wiley & Sons.
- Schulze, T. 2008. *Simulation Needs SLX*. Annandale, Virginia: Wolverine Software.
- Swain, J. J. 2015. "Simulation Software Survey." *OR/MS Today* 42(5). (This is the 10<sup>th</sup> biennial survey.)

## AUTHOR BIOGRAPHIES

**THOMAS J. SCHRIBER** is a Professor of Technology and Operations in the Stephen M. Ross School of Business at the University of Michigan. He is a recipient of the INFORMS Simulation Society's Lifetime Professional Achievement Award, and is a 'Pioneer of Simulation' (<http://d.lib.ncsu.edu/computer-simulation>). Email: [schriber@umich.edu](mailto:schriber@umich.edu).

**DANIEL T. BRUNNER** is a Managing Partner at Dan Brunner Associates LLC. He holds a BSEE from Purdue University and an MBA from the University of Michigan. He has served as a WSC Business Chair (1992), General Chair (1996) and Transportation Applications Track Coordinator (1998). He can be reached at: [dan.brunner@danbrunner.com](mailto:dan.brunner@danbrunner.com).

**JEFFREY S. SMITH** is the Joe W. Forehand Professor of Industrial and Systems Engineering at Auburn University. He has served as the WSC Business Chair (2010) and General Chair (2004) and is currently on the WSC Board of Directors. He has a BIE from Auburn University and a MS and PhD (both in Industrial Engineering) from Penn State University. His email and web addresses are: [jsmith@auburn.edu](mailto:jsmith@auburn.edu) and <http://jsmith.co>.