Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin and R. G. Ingalls, eds.

RESOURCE MODELING IN DISCRETE–EVENT SIMULATION ENVIRONMENTS: A FIFTY–YEAR PERSPECTIVE

Charles M. Jenkins
Stephen V. Rice

201 Weir Hall
University of Mississippi
University, MS 38655, USA

ABSTRACT

Through 50 years of innovation, discrete–event simulation environments have offered similar approaches to modeling the resources that participate in simulations. These approaches involve "clients" and "servers" of varying activity levels that wait in queues of varying sophistication. While powerful enough for many applications, these models limit the complexity of the entities that may be represented. Analysis of more than thirty simulation environments provides the substrate for defining "levels" of modeling features from primitive foundations to advanced embellishment. This analysis not only supports comparison of existing resource models, but also informs the development of new approaches.

1 INTRODUCTION

The need for simulation modeling, driven by the demands of a world at war, arguably motivated the development of computer technology in the 1940s. The Ballistic Research Laboratory at the US Army Ordnance Department teamed with the University of Pennsylvania to expedite the creation of artillery range tables for a burgeoning stockpile of munitions. Their inability to meet the demand opened the door for the creation of the Electronic Numerical Integrator And Computer (ENIAC), which also drew the attention of theoreticians working on US atomic weapon programs (Akera 2002). That a plethora of models and approaches have arisen in the more than 50–year symbiosis between simulation and computing should surprise no one. This symbiosis has drawn the interest not only of countless doctoral students, but also of such leading lights as Donald Knuth and Harry Markowitz.

In his history of discrete–event programming languages, Nance notes that languages may be classified according to the world view (i.e., activity–, process–, or event–oriented) or according to chronological order of appearance. He divides the period 1955 to 1986 into five periods (Nance 1996):

- 1955–1960, the Period of Search. This period saw much experimentation with ways of simulating. The General Simulation Program (GSP), described in 1960, is credited as the first simulator.
- 1961–1965, the Advent. The "major" simulation programming languages (such as GPSS, SIMULA I, SIMSCRIPT, CSL, GASP, OPS, and DYNAMO) arose during this period. SIMULA, of course, would exert profound influence on the wider world of analysis and programming by introducing the object–oriented paradigm.
- 1966–1970, the Formative Period. During this period, rapidly improving hardware permitted clearer focus on consistent and clear concepts, as opposed to effective implementation, of simulation programming languages. New versions of GPSS, SIMULA, SIMSCRIPT, GASP, OPS, and CSL emerged.
- 1971–1978, the Expansion Period. This period saw major expansions of the capabilities and scope of such languages as the venerable GPSS, SIMSCRIPT, and GASP. Pritsker developed not only GASP II, but also GERT. Time–sharing systems and advances in terminal systems made possible interactive systems such as the Conversational Modeling Language (CML), designed not only as a simulation system but also as a language creation system.
- 1979–1986, The Period of Consolidation and Regeneration. This period saw the major players adapt their products to additional platforms, especially the personal computer, without changing the basic character of their simulation languages. SLAM and SIMAN arose during this period.

Though writing in 1996, Nance ends his discussion with languages extant in 1986. He claims that even ten years proves too short a vantage point to judge the historical significance of more recent languages. With unintended humor, he writes,

"The emergence of yet another popular general purpose language—Pascal—stimulated a repetition of history in the subsequent appearance of simulation packages based on the language" (Nance 1996). Today he would undoubtedly note the proliferation of packages for C++ and Java.

While their user interfaces have grown more powerful and, ostensibly, easier to use in the years since 1996, and though they may be written in object–oriented languages as opposed to procedural languages, today's off–the–shelf discrete–event–simulation (DES) environments rely on resource models that often differ only in degree from the models of the 1960s or 1970s. Entities in today's environments may have more user–defined attributes, and those attributes may be of more complicated types. The environments provide larger collections of ready–made resources, often focused on a particular niche, such as manufacturing. Resources may follow more complicated schedules and occupy finer states, and modern frameworks such as Möbius (The PERFORM Group 2008), Ptolemy (Lee 2008), and James (Uhrmacher and Schattenberg 1998) facilitate composing resource models from sub–models expressed in a variety of formalisms.

Real–world resources, of course, exhibit enormous diversity. We have proposed a typology, or multidimensional categorization, that describes resources based on who they are (their existential characteristics), when they provide their services (their availability), what they do and how well they do it (their utility), and how they do what they do (their implementation) (Jenkins and Rice 2007a). Building upon this typology, we have proposed a general resource model, expressed in the Unified Modeling Language, of active and engaged entities. The model treats clients and servers as equal partners that discover each other through registries and then negotiate and exercise contracts with one another (Jenkins and Rice 2007b).

While working on our models, we have analyzed more than 30 other simulation environments, with many more yet under analysis, that span almost 50 years of DES programming. As Nance noted, these environments take at least three forms: Some are "packages" welded onto existing languages, some are preprocessed into another existing language, and others stand alone (Nance 1996). Some of them faded away soon after their conference presentations, while others have lasted decades. While most of them support primarily discrete–event simulation, several of them offer support for continuous simulation, and a few support only continuous simulation. Though these environments represent many different perspectives, from instructional to proof–of–concept to experimental to commercial, they display remarkable continuity in their modeling of resources. While the environments themselves defy easy classification, we find that the features they offer, at least regarding resource modeling, may be described in levels of increasing scope and capability. At the lowest level, we find environments for the simple queuing models that every student simulationist has programmed. At the highest level, we find today's most capable environments, with their advanced queuing options, libraries of ready–made entities, and complex graphics.

2 LEVELS OF FEATURES

The sections that follow describe five levels of features in resource models:

- Level 1: Simple servers, simple clients, and FIFO queues
- Level 2: Simple decision making
- Level 3: Advanced service scheduling
- Level 4: Advanced queuing
- Level 5: Advanced entities

Each level encompasses a coherent set of concepts and relationships central to a single capability, such as decision making by entities or advanced queuing operations. The ordering of the levels reflects logical necessity and, to a smaller extent, developmental chronology, even though some of the more advanced concepts in upper levels actually appeared very early, at least in experimental or rudimentary forms. Few, if any, simulation environments operate entirely at one of these levels. Rather, they exhibit features of many levels. Environments embedded in or implemented as general–purpose programming languages, as well as environments that permit calls to external code, of course, permit the programmer to supply the features of all the levels, but this discussion will focus on the features natively supported by environments.

2.1 Level 1: Simple Servers, Simple Clients, and FIFO Queues

Simulation history and practice rest on the conceptual foundation of featureless client entities waiting in simple first–in–first–out (FIFO) queues for service from relatively featureless server entities. In Figure 1's depiction of this level, ovals with black outlines represent important concepts required for this level (and for later levels that build upon it). Within an environment, a concept may be implicit; for example, the environment may not expose the `Attributes` or states of an `Entity` to the modeler. The larger oval with a grey outline and labeled `Entity` encloses the concepts of `Active Entity` and `Passive Entity` to provide a way of reducing clutter when showing relationships common to both types of entity. Filled lines with labels connect concept ovals to depict the relationships among them. The relationships should be read in the direction of the arrowheads; for example, "`Entity travels Route`." Some relationships are bidirectional. Rectangles provide additional information about the concepts to which they are connected by dashed lines. An italicized heading introduces the type of information the rec-

tangle contains; for example, a simple queue has a FIFO discipline. Figures for subsequent levels follow the same semantics and build upon this figure.

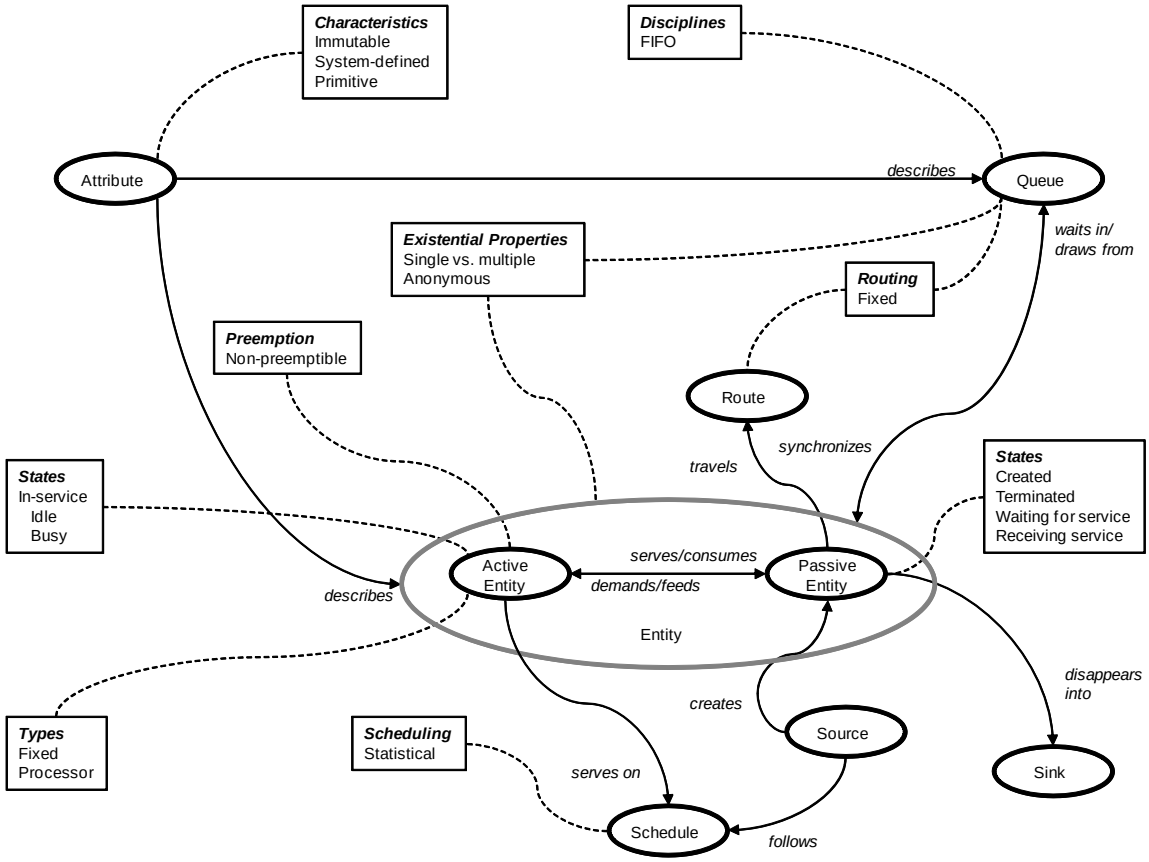

Figure 1: Level 1 encompasses concepts for simple queuing simulations

Systems simulated by Level 1 environments may be conceived in terms of active clients demanding service from passive servers or in terms of active servers serving passive clients. A single view, the latter one, will be used here for simplicity. In this view, an anonymous `Passive Entity` arises in a `Source` according to a stochastic `Schedule`. The `Passive Entity` travels a fixed `Route` to an anonymous `Active Entity`, where it may wait in a FIFO `Queue` for service. The `Active Entity`, typically a processor at a fixed location, eventually draws the `Passive Entity` from the `Queue` and serves it according to a stochastic `Schedule` governing the length of service. The `Passive Entity` then disappears into a sink, but the `Active Entity` persists until the end of the simulation. Entities and queues have only the `Attributes` required by the simulation environment, and entities occupy only simple states. For example, a `Passive Entity` may have a system–defined and system–managed attribute for waiting time, and will occupy one of four states—created, terminated, waiting for service, or receiving service.

While virtually any DES environment supports Level 1, the General Purpose Simulation System (GPSS) represents the best known example. (GPSS certainly possesses more advanced features, as well.) IBM introduced GPSS in 1961 as a simulation tool for non–programmers, and GPSS/H followed in 1977 (Wolverine undated). As late as 1996, Nance asserted that GPSS was still the most popular simulation programming language (Nance 1996). In a GPSS simulation, a transaction represents any entity that requires service—that is, simulation time—from a facility, one server possibly among others in a network of entities. A transaction is generated according to a random distribution, "seizes" a facility, holds it while the simulation clock advances according to a random distribution, "releases" the facility, and is eventually terminated (Dunning 1981, Schriber 1991). A facility has no explicit attributes and expresses no behaviors—it waits passively for transactions to appear and then marks time to simulate service. A facility that contains a transaction is busy; it is idle otherwise (Banks and Carson 1985). A storage represents a "room" in which multiple, identical servers reside. The capacity of a storage remains fixed throughout a simulation. A transaction "enters" a storage when one or more of its servers become available, holds those servers while the simulation clock advances, and then releases the servers and "leaves" the storage. If necessary, a transaction au-

tomatically enters a facility's or storage's queue and automatically departs the queue when its turn arrives. More than one server may draw from a single queue. A queue expresses no runtime behavior except statistics gathering (Dunning 1981, Schriber 1991). Simulationists working in Level 1 environments have necessarily shown great ingenuity in introducing behaviors not explicitly handled by the environment. For example, facility malfunction can be simulated in GPSS by creating a "goblin" transaction that ties up the server or makes it unavailable for a simulated time period (Banks and Carson 1985).

Other environments explicitly incorporate the features of Level 1. In SLAM II, for example, entities in a network model arise in `CREATE` nodes, wait in `QUEUE` nodes, receive service in `ACTIVITY` branches, and disappear into `TERMINATE` nodes. As with GPSS, a server fails when a specially created failure entity arrives to demand its service (Pritsker 1995). In a simple SIMAN simulation, an entity emerges from a `CREATE` block. It then enters a `SEIZE` block where it demands service from a resource. It waits, if necessary, in an implicit or explicit `QUEUE` block. After receiving the desired resource, it passes time in one or more `DELAY` blocks to simulate service time before relinquishing its resource in `RELEASE` blocks. Finally, it disappears into a `DISPOSE` block. The system tracks when an entity is created, when it enters a queue, how long it holds a resource, and the entity's current position in its sequence of processes. The server also tracks whether a server is busy or idle (among other states) (Pegden, Shannon, and Sadowski 1995). SIMSCRIPT II.5's temporary entities come into being and disappear through a simulation run (CACI 1987). On the other hand, the Control and Simulation Language (CSL), jointly developed by IBM United Kingdom Ltd. and Esso Petroleum Co. Ltd. and released in 1962, does not permit the dynamic creation and termination of entities (Buxton 1966).

The Simulation–Oriented Language (SOL) at its 1964 release married features of GPSS and SIMSCRIPT (Knuth and McNeley 1964a). As in GPSS, SOL simulations involve transactions that compete for facilities. SOL uses a store to represents a "space" of a specified capacity. A transaction attempts to enter a specified number of units of a store, suspends if necessary until all the specified space is available, occupies the space when it becomes available, and then leaves the store (Knuth and McNeley 1964b).

In Discrete Event Modelling on SIMULA (DEMOS), an environment influenced by GPSS and released in 1979, resources serve as pools of one or more identical elements. The simplest such resource, represented by the class `RES`, plays no active role in a simulation, but supports resource competition based upon mutual exclusion. An object's available count cannot grow beyond its initial value, but can rise and fall as other elements acquire and release its elements (Birtwistle 1979).

Released in 1984, AutoMod targets manufacturing and material transfer systems (Rohrer 1999). Its architecture features a process model in which active "loads" follow "processes," travel along "movement systems," use passive resources, and sometimes wait in queues. While loads typically represent customers or assemblies, they must also play the part of any active simulation component. They may be created according to a user specification or by the `create` or `clone` "action." Eventually, a load may `die`. A resource's capacity indicates the number of loads it can simultaneously serve. A process may set, increment, or decrement the capacity of a resource at any time. A load may attempt to get one or more units of one or more resources, may then wait for a specified period of time, and then may free any or all of its held resources (Brooks Automation 2003).

2.2 Level 2: Simple Decision Making

Level 2 adds features to Level 1 to support simple decision making, particularly in two areas. The first area involves `Queues`. On the one hand, a server entity may wish to choose among client entities waiting in a queue. On the other hand, a client entity may require the option to balk, that is, to choose not to enter a queue. The second area involves choosing which server entity to visit next. This area, like balking, represents dynamic choices about the `Route` an entity should take through a network. Making these runtime decisions requires, in particular, additional queue disciplines, such as last–in–first–out (LIFO), and additional information about entities. For example, entities participating in priority queues must have priority attributes. Allowing different entities to take different paths through the network requires the ability to record alternative paths for entities or groups of entities. Balking decisions, beyond the purely stochastic, may require information about tolerance for queue lengths. Granting additional attributes to entities provides individuality to the indistinguishable and anonymous entities of Level 1. Granting decision–making power to entities blurs the distinction between active and passive entities. Figure 2 depicts Level 2 features. Changes between Figure 1 and Figure 2 are marked in green, bolded text and with an asterisk.

As with Level 1, many simulation environments exhibit features of Level 2. A GPSS transaction may use numerically encoded attributes that can change as the transaction progresses through the network. Finding a long wait for a server it desires, a transaction can transfer to another point in the network or try another server. In addition, transactions may exhibit up to 128 levels of priority in their demand for a given server (Dunning 1981, Schriber 1991).

Each SOL transaction may receive a set of local variables that record its state, as well as a control strength that determines how it contends with other transactions for the privilege to use resources. Transactions may communicate with each other via global variables. In addition, each transaction follows a program that specifies, in Level 2 terms, its route through a network (Knuth and McNeley 1964b).

GASP IV, a product of the early 1970s, permits an entity to have 25 numeric attributes (Pritsker 1974). At its creation, a SLAM II entity receives a vector of numerical attributes that ride along with the entity and assume whatever meaning the programmer wants them to have. Attributes may be used, for example, to make decisions, to match similar entities, and to accumulate information over the life of an entity. They may undergo revision at ALTER nodes (Pritsker 1995).

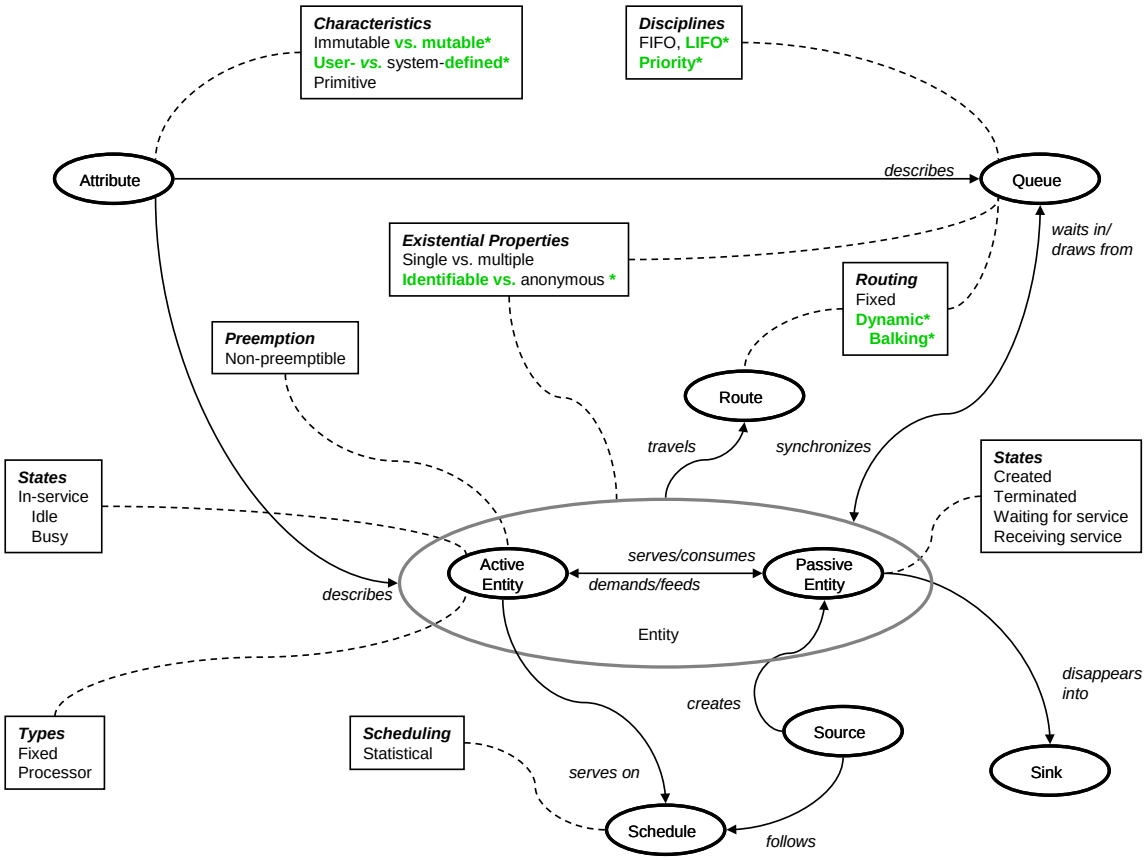

Figure 2: Level 2 adds to Level 1 features needed for simple decision making

SIMAN permits entities to visit stations in a model in different orders. In addition, the system provides numeric attributes to which the simulationist may ascribe any meaning required. These attributes may be changed as the entity flows through its process, and they may serve as the basis for decisions in the process, such as how long to use a resource or which path to take at a branch. When an entity demands service from a busy resource, it may wait in a queue governed by one of several disciplines, including FIFO, LIFO, and priority. In other cases, an entity may choose which of several resources it will demand or which of several queues it will enter, and a resource may sometimes choose which entity will be released from which queue, without regard for the queue's default discipline. An entity may balk, but if no balk path is specified, the entity disappears from the simulation (Pegden, Shannon, and Sadowski 1995)

AutoMod's loads may have many system– or user–defined attributes, all of which may change during a simulation. A load's system–defined attributes include its priority, its color, and its type. User–defined attributes may be scalar or arrayed, up to four dimensions. Numeric types include not only such general types as `Integer` and `Real`, but also special types of `Acceleration`, `Distance`, `Time`, and `Velocity`. For these special types, AutoMod automatically converts among units of measure. Miscellaneous types include `Color` and `Location` (Brooks Automation 2003).

2.3 Level 3: Advanced Service Scheduling

Where Level 2 added features to enhance an entity's attributes and to permit simple decision making, Level 3 adds features to permit more complex service scheduling. Simulation environments operating at this level permit an entity to follow—that is, provide service—according to a schedule. In addition, they permit entities to fail, also according to a schedule. Thus an entity may occupy an additional out–of–service state that includes sub–states for failure, rest periods, and off–duty periods. Figure 3 depicts the features of Level 3. As before, asterisks and green, bolded text mark enhancements to the previous level.

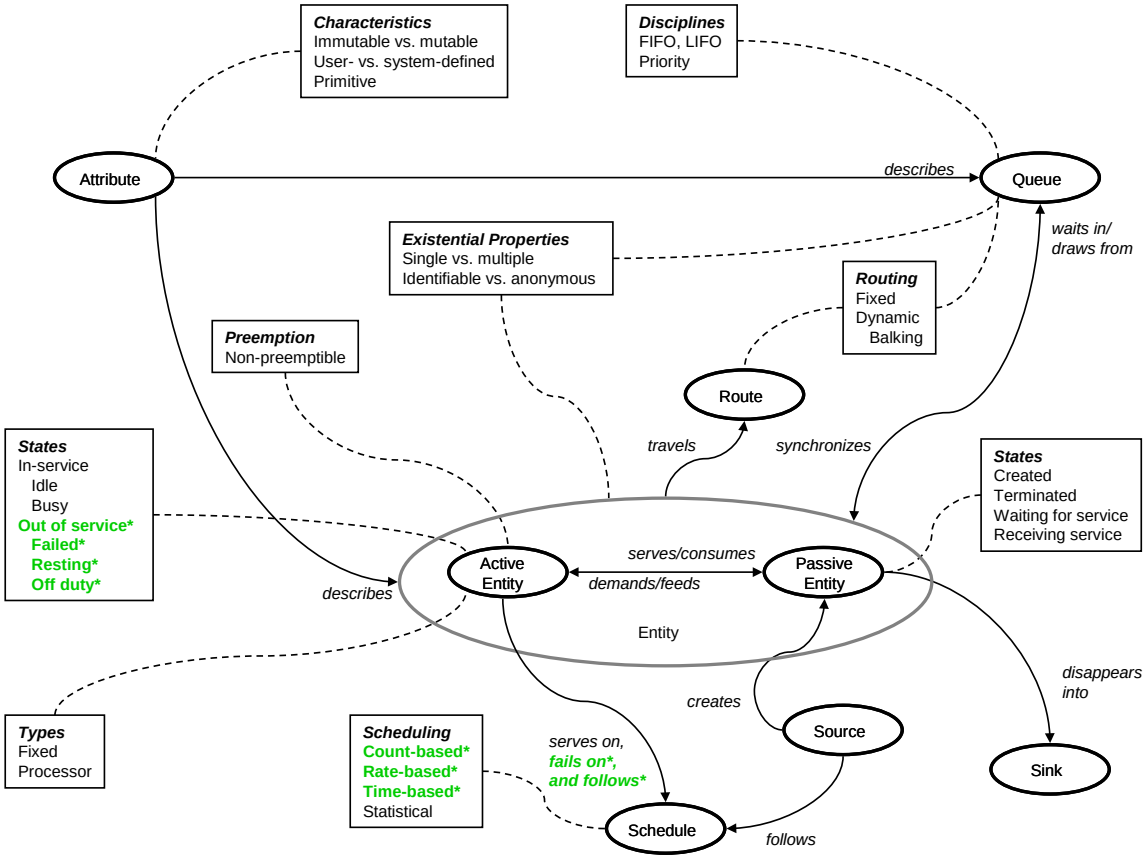

Figure 3: Level 3 adds to Level 2 features needed for advanced service scheduling

In GPSS, a facility can be made available or unavailable to simulate uptime and downtime, respectively, or any other meaning the programmer wishes. As noted earlier, however, server failure is simulated by the arrival of "goblin" transactions (Banks and Carson 1985).

In MODSIM an instance of `ResourceObj` models a resource of constrained capacity. The capacity of a resource may be incremented or decremented, and it changes as entities acquire and release its identical units. An entity may wait for a resource to give it one or more of its units and then later instruct the resource to take back those units (CACI 1992).

A SIMAN resource represents one or more identical, interchangeable units from which entities demand service. The capacity of a resource may be specified numerically, subject to updating throughout a simulation, and may vary according to a schedule expressed as a sequence of capacities and durations. The status may be drawn from one of SIMAN's four "autostates"—busy, idle, inactive, and failed—or from a user–defined set of named states, any of which may map onto autostates. The failure mode provides a rule governing when a resource fails. A resource may fail after serving a specified number of entities or after spending a specified amount of time in a particular state (e.g., busy) (Pegden, Shannon, and Sadowski 1995).

In AutoMod, the fundamental states of a resource (an entity required to complete a task) include "processing," "idle," and "down." The user may create new sub–states to further classify downtime as, for example, lunch time or break time. The availability of a resource may vary automatically according to a "cycle" that specifies when the system should bring up or take down the resource, or it may be taken down and brought up directly by a process (Brooks Automation 2003).

In ExtendSim, which was originally released in 1988, passive "items" undergo activities in `Activity` "blocks." These blocks hold an item for a specified processing time ("delay") that may follow a random distribution or that reflect a specified schedule, item attributes, or information input to the block as "value flows." They may even timeshare among many items to simulate multitasking. A single activity block may accept more than one item at a time. Once it accepts an item, an `Activity` block can be shut down, temporarily suspending the current item's processing. Shutdowns can simulate, for example, breakdowns and break periods and can occur on a scheduled or statistical basis, such as mean time to failure (Kraft 2003, Imagine That 2007).

2.4 Level 4: Advanced Queuing

Where Levels 2 and 3 enriched the representation and behavior of clients and servers, Level 4 enhances the representation and capabilities of queues. Clients may not only assert a priority when demanding service, but they may also preempt clients of lower priority. Interrupted clients may or may not be able to resume service later. Clients may choose to wait in a queue until a Boolean condition tests `true` or only for a specified length of simulated time. Servers may need to draw clients from queues according to more general disciplines such as longest–waiting–time, least–remaining–service–time, or lowest–setup–cost. Producer and consumer entities may need to synchronize their interaction through bins of products, and other entities may wish to submit themselves as slaves to master entities. Finally, entities may wish to combine. Once combined, they may lose or retain their individual identity. Figure 4 depicts the features added to Level 3 to support these enhanced forms of collaboration and competition, with green and bolding again representing new features.

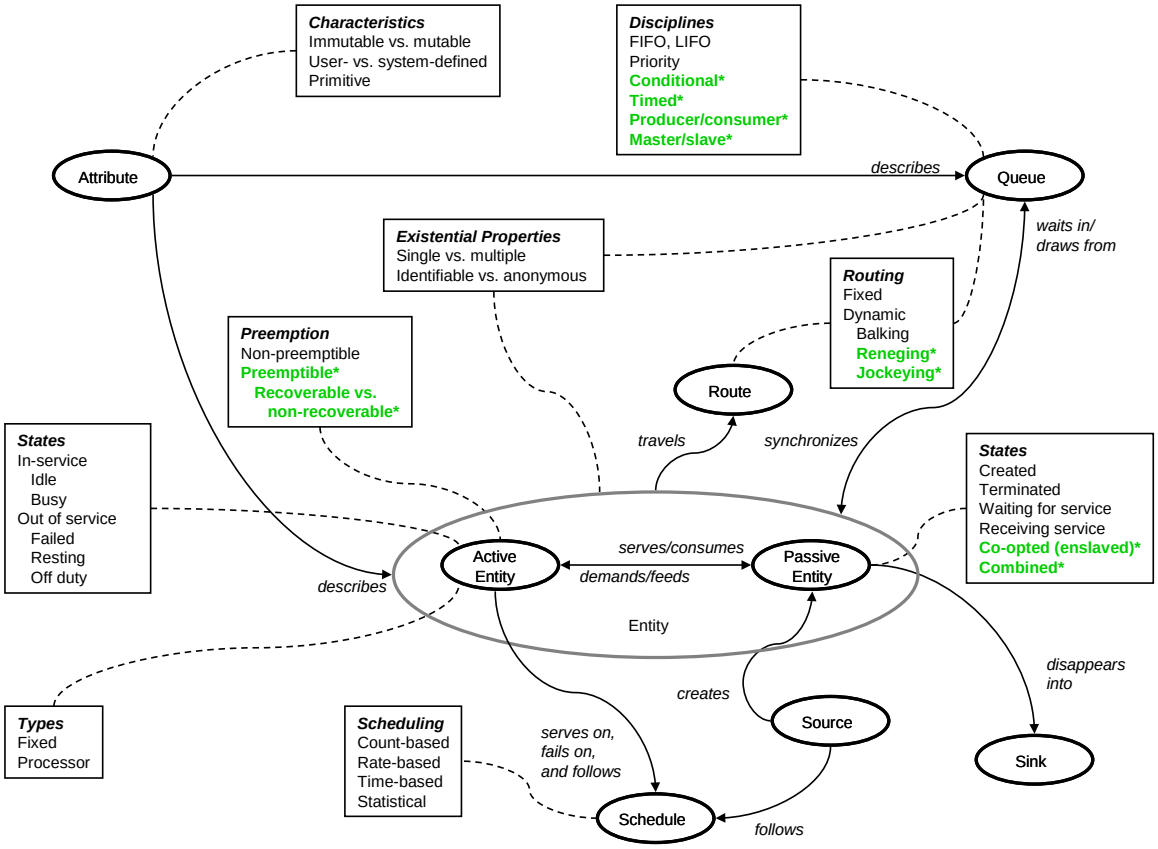

Figure 4: Level 4 adds to Level 3 features needed for advanced queuing

In DEMOS, a `BIN` facilitates simulations involving producers and consumers. Though a `BIN` has an initial quantity of identical elements, its count can grow as producers create new elements. A consumer may take one or more elements from a `BIN`, and a producer may give one or more elements to a `BIN`. If necessary, a `BIN` maintains a queue of waiting consumers. In more complicated situations, where entities must cooperate very closely with each other, one object serves as a master to one or more slave objects. Truck entities, for example, submit themselves as slaves to be transported across a body of water by a master ferry entity. This cooperation occurs through the agency of a queue represented by the class `WAITQ`. A prospective slave entity waits in the queue to be "co–opted" by a master entity, which may simply take the first waiting slave or select a slave that meets a set of criteria. An entity that has not been co–opted is "available." DEMOS also provides a conditional queue modeled by the `CONDQ` class. An entity may submit to a waiting queue, only to be released when a condition tests `true`. Each entity waiting in the queue may specify a different condition. The queue accepts signals indicating that it should look for a waiting entity or possibly all waiting entities whose conditions now test `true` (Birtwistle 1979).

Psim, an environment developed for instructional purposes in the late 1990s, closely resembles DEMOS in its handling of queues. First, the `bin` class represents a container to which a producer process may give one or more identical resources and from which a consumer process may take and consume one or more resources. As with the `bin` resource, a process that

attempts to take more resources than the `bin` has available will wait in a hidden priority queue. Second, processes cooperate in accomplishing a task when one or more processes submit themselves as slaves to a dominant, master process. The prospective slaves wait in a `waitq` object for a master to come along, and a master co–opts waiting slaves. If necessary, the prospective master enters the `waitq` object's master queue until sufficient slaves appear. Third, a process can use a `condq` object to `waituntil` a condition tests `true`. Another process can signal the conditional queue that the state of the system may have changed enough to cause the condition to test `true`. Fourth, a process can attempt to interrupt another process by sending it a `p_interrupt` message. Depending on the interrupting process' relative priority, the interrupted process may then relinquish resources or take other actions (Garrido 1999).

SLAM II entities, including the specially–created failure entities, may preempt lower priority entities' use of single–server activities. Entities may gather at an `ACCUMULATE` node until a condition is reached (a specified count of entities or a specified sum of entity attributes, for example), and then a single entity leaves the node with the attributes of one or more of the original entities. Likewise, entities gather at a `BATCH` node until a condition is satisfied or until a marker entity with a negative attribute value arrives. A single batch entity then leaves the node, but the individual entities can be later recovered at an `UNBATCH` node (Pritsker 1995).

A SIMAN queue may be governed not only by the usual FIFO, LIFO, and priority disciplines, but also by low– or high–value–first based on a specified attribute. A resource may choose which entity will be released from which queue, without regard for the queue's default discipline. An entity may not only balk (refuse to wait for service), but also jockey (move between queues) and renege (leave a queue before receiving service). Entities may accumulate in a resource's queue until a required number have arrived, at which point they leave the queue and receive service simultaneously. Similarly, a block may aggregate entities drawn from one or more parallel queues into a temporary set (e.g., a pallet) or into a permanent set (an assembly or blend) whose attributes reflect those of the first or last entity combined or of the sum or product of all the attributes of all the entities. While the original entities may be recovered from a temporary set, the original elements of a permanent set cease to be—only the original number of entities may be recovered, and they will all have the same properties as the permanent set had. On the other hand, a single entity that requires concurrent service from multiple resources can create clones that flow down different branches in the network (Pegden, Shannon, and Sadowski 1995)

In SIMUL8, released in 1994, passive "work items" wait in a special queue called a "bin" that may prioritize items accord to their "labels," their "startup loads," and their remaining shelf–life. A tank queue contains liquid products and fires triggers when it is full or empty or as it nears full or empty. A server entity called a work center may receive a work item as soon as it arrives; receive only items that have expired in a queue; collect the youngest or oldest item in a queue; wait for a work item with a particular label value; wait until specified numbers of work items of specified types arrive; group arriving items into a single item; cycle its attention among competing sources; and so forth (Hauge and Paige 2004).

Other environments offer similar features. For example, a MODSIM entity may assert a priority demand for a `ResourceObj`, potentially preempting another entity's use of the resource, or it may limit its wait, or it may do both (CACI 1992). An SOL transaction asserting higher control strength may preempt a transaction that asserted lower control strength. Once it takes control of a facility, a transaction may wait until some condition tests true (Knuth and McNeley 1964b). An AutoMod `Activity` block can be preempted, terminating the current item's processing, or shut down, temporarily suspending the current item's processing. `Queue` blocks may enforce disciplines that depend on item attributes (Imagine That 2007). FreeSML, a 1995 product, permits entities to be removed from a queue based upon their current priority, as opposed to their priority upon entry to the queue (DiLeo 2005).

2.5 Level 5: Advanced Entities

Building largely upon the features of lower levels, Level 5 provides advanced entities with richer attributes, capable of providing specialized service, and occupying finer states. Advanced servers, for example, can calculate their cost of service more accurately by distinguishing the costs to travel to the work site, set up, process, and tear down after service. The impact of a failure can be more accurately simulated by modeling the wait for a repairman and the actual repair time. Many environments provide these advanced entities as pre–built resources, including various forms of transportation devices that simulate, for example, conveyor belts and automatic guided vehicles (AGVs). In addition, the distinction between active and passive entities, first blurred by the features of Level 2, in many cases disappear as both clients and servers take more active roles in a simulation.

Many environments, especially those embedded in or implemented as general–purpose programming languages, permit entities to take on advanced attributes. For example, SIMSCRIPT (CACI 1987), SIMULA (Franta 1977), DEMOS (based on SIMULA, Birtwistle 1979), Sim++ (based on C++, Lomow and Baezner 1990), SIMOD (based on Modula–2, L'Ecuyer and Giroux 1987), and Psim–J (based on Java, Garrido 2001) allow attributes of any type supported by the base language. AutoMod allows a user to incorporate C–language structs (Brooks Automation 2003). ExtendSim permits attributes that represent keys into the ExtendSim database (Imagine That 2007). ROSS, an early–1980s product of The Rand Company, permits enti-

ties to gain or lose attributes, which may be numeric, non–numeric, or list–based (McArthur, Klahr, and Narain 1984; McArthur, Klahr, and Narain 1985).

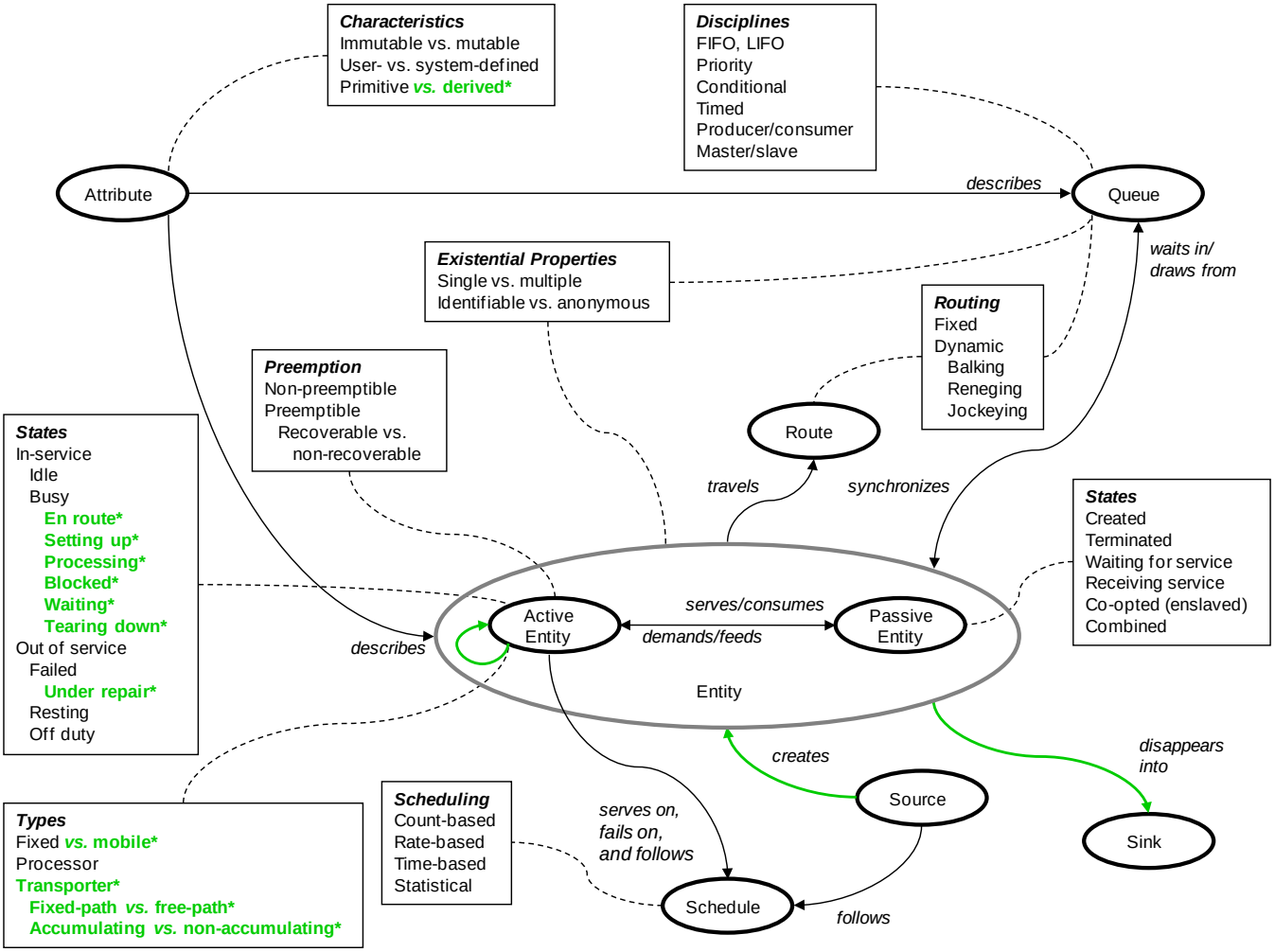

Figure 5: Level 5 adds to Level 4 features needed for advanced entities

Besides its entities and resources, SIMAN provides four other constructs that serve to convey entities between stations. First, a `ROUTE` block simply transports entities to a specified station or to the next station in the entity's station sequence with a specified delay, but with no provision for congestion or competition for conveyance. Second, a conveyor moves entities along a fixed path with fixed locations for loading and unloading. An entity must request a quantity of "space" on the conveyor, wait until the space becomes available, disengage the conveyor from its route for loading, ride the conveyor, disengage the conveyor for unloading, and then exit the conveyor. Third, a free–path transporter represents a conveyance, perhaps a forklift, that moves an entity to a specified station in its station set at a specified velocity. An entity must request the transporter, wait for the transporter to travel at its defined speed from its current location to the entity's current station, wait for loading, wait for conveyance at a specified velocity, wait for unloading, and then release the transporter. Instead of waiting passively at a station for a transporter, an entity can acquire a transporter ahead of its anticipated need and then send the transporter on ahead to wait for the entity's arrival. Fourth, an AGV moves entities throughout a network of intersections—some mapped to stations—connected by links of one or more zones of equal length. Unlike free–path transporters, however, AGVs experience congestion as they compete for the same zones, links, or intersections, but they can be redirected around obstructions. A conveyor, a free–path transporter, or an AGV may experience failure and undergo repair (Pegden, Shannon, and Sadowski 1995).

AutoMod provides several abstractions of the equipment used to store and move materials. First, a conveyor moves loads through sections connected by transfers. Loads enter conveyor systems at stations. As they travel, they may interact with photoeyes, thereby triggering code to control movement. `Motors` power conveyors and may be taken up or brought down.

Second, a load may ride in a carrier suspended from a `dog` in an overhead power–and–free system, or it may be pushed along by `dogs` in towline chains running along the floor. Power–and–free systems and towlines move at a constant velocity, and the `dogs` follow a regular spacing along the system. Third, `materials` may occupy `tanks` and move about through `tank transfers` visualized as `pipes`. Tanks may have associated triggers that respond to the level of material. Fourth, in path mover systems, `vehicles` pick up loads at `control points`, transport them along `guide paths` connected by `transfers`, and then set them down again at `control points`. AutoMod provides facilities for scheduling and routing vehicles and for avoiding collisions. Fifth, Automated Storage and Retrieval Systems (AS/RS) represent aisles of storage racks, along with the Storage/Retrieval Machines (S/RMs) that store and retrieve loads. Each aisle contains a grid of bins arranged in horizontal `tiers` and vertical `bays`. Finally, a bridge crane moves between pickup and delivery zones on overhead rails (Brooks Automation 2003).

In Flexsim, first released in 2003 (Flexsim undated), a host of active components may be divided into immobile "fixed resources" and mobile "task executers." Fixed resources include sources, queues, processors, sinks, conveyors, combiners, separators, multiprocessors, and other types. Processors may occupy one of several states—idle, in setup, processing, blocked (waiting for a downstream object to take up its released flow items), waiting for an operator or a transport, and down. Combiners and separators represent special processors for grouping and ungrouping flow items. Besides the states of a basic processor, a combiner may also occupy a collecting state while it waits for flow items to appear. Conveyors move one or more flow items along a fixed path. An accumulating conveyor simulates a roller conveyor that allows flow items to stack up against each other when the lead flow item is blocked. A non–accumulating conveyor simulates a belt conveyor in which all onboard flow items must stop if the lead item stops. A conveyor may be empty, conveying, blocked, or waiting for transport. It may have, furthermore, photoeyes that trigger state changes as flow items pass in front of them (Flexsim 2008)

Flexsim's task executors move about the simulated workflow, incurring travel time and, if necessary, detecting collisions as they go. They load and unload flow items, and they serve as shared resources to fixed resources. They have attributes such as capacity, top speed, acceleration, and deceleration. A special object called a dispatcher may be employed to direct the activities of task executors for a particular object. Operators represent human task executors, complete with animated human icons, who may be called to an object, perhaps to perform maintenance, and who stay with that object until released. Transporters, animated as forklifts, carry flow items between objects. Cranes and elevators resemble transporters, but with different animations. Finally, robots represent a form of task executor that moves flow items not by moving its base, but by moving its arms (Flexsim 2008).

3 LEVEL 6

Our survey of resources revealed great variability in the attributes and behaviors of real–world entities likely to appear in simulations (Jenkins and Rice 2007a). Our survey of simulation environments, on the other hand, showed remarkable similarity in how those resources are modeled. The designers of these environments have focused on certain attributes and behaviors, no doubt capturing the ones most salient in their target applications. While they have chosen different names for the same concept—work item, load, transaction, etc.—they have nonetheless built environments capable of modeling similar types of systems. The longevity of many of these environments in the marketplace attests to the usefulness of their models.

As we have shown, the features of these environments may be described in levels, from the features required for rudimentary single–server–single–queue simulations to the complex, pre–built entities of modern, GUI–based simulation platforms. As noted earlier, few environments may be pigeon–holed into a single level, for most environments exhibit features of multiple levels. It may be more useful and practical to speak of a given feature as characteristic of a certain level or to debate which features belong in different levels. From our perspective, the more interesting question is: What should Level 6 look like?

Prior work convinces us that many simulations, perhaps only because of constraints imposed by available simulation environments, unfortunately treat certain entities, whether clients or servers, as unrealistically passive. Real–world entities often exhibit a complex give–and–take that resembles a dance among partners. Our own first attempt at a general resource model attempted to remove the distinction between clients and servers. We considered a resource to be any potentially active entity in a simulation that has something of value to offer another entity (Jenkins and Rice 2007b).

We assert that Level 6 features should, when appropriate, remove the distinction between client and server—between active and passive entities—and therefore permit representation of more realistic and complex interactions. For example, resources will naturally search for and discover each other, perhaps through registries, before negotiating agreements to exchange tokens of mutually–agreed value. Where appropriate, resources will have the information and capability to seek their own interests. Such resources would, of course, have much in common with the agents described by Swaminathan, Smith, and Sadeh (1998), by Uhrmacher and Schattenberg (1998), and by many others. Indeed this line of investigation has led to a conviction that even the environment in which resources or agents live must be treated as a dynamic part of the model (Helleboogh et al. 2007).

We are experimenting with conceptual resource models that describe the richer attributes and behaviors we envision. These models will be expressed as layers of domain–specific languages, with the lowest layer compiling to a general–purpose programming language such as Java or SIMSCRIPT III. The present work guides our experiments in two ways. First, it reveals what has proven useful through 50 years of simulation experience. Second, it reveals how simulationists are accustomed to thinking and working. To be successful, we must not only accommodate capabilities in Level 6, we must provide them in a way that feels natural to simulationists.

REFERENCES

Akera, A. 2002. The early computers. In *From 0 to 1: An authoritative history of modern computing*, eds. A. Akera and F. Nebeker, 63–75. Oxford: Oxford University Press.

Banks, J. and Carson, J. S. 1985. Process–interaction simulation languages. In *Simulation* 44:225–235.

Birtwistle, G. M. 1979. *A system for discrete event modelling on Simula.* New York: Springer–Verlag New York, Inc.

Brooks Automation, Inc. 2003. *AutoMod User's Manual Volumes 1 and 2.* Salt Lake City, UT: Brooks Automation. Available via <www7.informatik.uni-erlangen.de/~heindl/teaching/ws04/sm1/f/exnotes/Automod11/amvol1and2.pdf> [accessed December 12, 2008].

Buxton, J. N. 1966. Writing simulations in CSL. In *The Computer Journal* 9: 137–143.

CACI, Inc.–Federal. 1987. *SIMSCRIPT II.5 programming language*, Los Angeles: CACI, Inc.

CACI Products Company. 1992. *MODSIM II® reference manual*, La Jolla, California: CACI Products Company.

DiLeo, J. J. 2005. Delivering on the open–source simulation language promise. In *Proceedings of the 2005 Winter Simulation Conference*, eds. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 2513–2523. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Dunning, K. A. 1981. *Getting started in GPSS: General Purpose Simulation System*, San Jose, California: Engineering Press, Inc.

Flexsim Software Products, Inc. Undated. About Flexsim Software Products, Inc. Available via <www.flexsim.com/company/about> [accessed September 17 2008].

Flexsim Software Products, Inc. 2008. *Flexsim User Manual.* Electronic manual provided with demonstration version of Flexsim. Orem, Utah: Flexsim Software Products, Inc. Available via <www.flexsim.com/downloads/freetrial/> [accessed September 17, 2008].

Franta, W. R. 1977. *The process view of simulation.* New York: Elsevier North–Holland.

Garrido, J. M. 1999. *Practical process simulation using object–oriented techniques and C++.* Boston: Artech House.

Garrido, J. M. 2001. *Object–oriented discrete–event simulation with Java: A practical approach.* New York: Kluwer Academic/Plenum Publishers.

Hauge, J. W. and Paige, K. N. 2004. *Learning SIMUL8: The complete guide.* 2nd ed. Bellingham, WA: Plain Vu Publishers.

Helleboogh, A., Vizzari, G., Uhrmacher, A., and Michel, F. 2007. Modeling dynamic environments in multi–agent simulation. *Autonomous Agents and Multiagent Systems.* 14:87–116.

Imagine That, Inc. 2007. *ExtendSim User Guide.* San Jose, California: Imagine That, Inc. Available via <www.extendsim.com/support_manuals.html> [accessed December 23, 2008].

Jenkins, C. M. and Rice, S. V. 2007a. A typology for resource profiling and modeling. In *Proceedings of the 40th Annual Simulation Conference*, 194–203. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Jenkins, C. M. and Rice, S. V. 2007b. A general model of resources using the Unified Modeling Language. In *Proceedings of the Huntsville Simulation Conference 2007*. San Diego, California: The Society for Modeling and Simulation International.

McArthur, D., Klahr, P., and Narain, S. 1984. *ROSS: An object–oriented language for constructing simulations*. Rand Note R–3160–AF, the Rand Publication Series. Santa Monica, California: The Rand Corporation.

McArthur, D., Klahr, P., and Narain, S. 1985. *The ROSS language manual*. Rand Note N–1854–1–F, the Rand Publication Series. Santa Monica, California: The Rand Corporation.

Knuth, D. E. and McNeley, J. L. 1964a. SOL—A symbolic language for general–purpose systems simulation. *IEEE transactions on electronic computers* EC–13: 401–408.

Knuth, D. E. and McNeley, J. L. 1964b. A formal definition of SOL. In *IEEE transactions on electronic computers* EC–13: 409–414.

Krahl, D. 2003. Extend: An interactive simulation tool. In *Proceedings of the 2003 Winter Simulation Conference*, eds. S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, 188–196. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

L'Ecuyer, P. and Giroux, N. 1987. A process–oriented simulation package based on Modula–2. In *Proceedings of the 1987 Winter Simulation Conference*, eds. L. Thesen, H. Grant, and W. D. Kelton, 165–173. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Lee, E.A. 2008. Introduction. In *Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)*, Technical Report No. UCB/EECS-2007-7. Document Version 7.0 for use with Ptolemy II 7.0, eds. C. Brooks, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, 1–46. Department of Electrical Engineering and Computer Sciences. University of California, Berkeley. Available via <www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-28.pdf> [accessed June 9, 2009].

Lomow, G. and Baezner, D. 1990. A tutorial introduction to object–oriented simulation and Sim++. In *Proceedings of the 1990 Winter Simulation Conference*, eds. O. Balci, R. P. Sadowski, and R. E. Nance, 149–153. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Nance, R. E. 1996. A history of discrete event simulation programming languages. In *History of programming languages—II*, eds. T. J. Bergin and R. G. Gibson, 369–427. New York: Association for Computing Machinery.

Pegden, C. D., Shannon, R. E., and Sadowski, R. P. 1995. *Introduction to simulation using SIMAN*. 2nd ed. New York: McGraw–Hill, Inc.

Pritsker, A. A. B. 1974. *The GASP IV simulation language*. New York: John Wiley & Sons.

Pritsker, A. A. B. 1995. *Introduction to simulation and SLAM II*. 4th ed. New York: John Wiley & Sons.

PERFORM Group, The. 2008. *Möbius Manual Version 2.2.1*. University of Illinois at Urbana–Champaign. Available via <www.mobius.uiuc.edu/manual/MobiusManual.pdf> [accessed June 10, 2009].

Rohrer, M. 1999. AutoMod product suite tutorial (AutoMod, Simulator, AutoStat) by AutoSimulations. In *Proceedings of the 1999 Winter Simulation Conference*, eds. P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, 1:220–226. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Schriber, T. J. 1991. *An introduction to simulation using GPSS/H*. New York: John Wiley & Sons.

Swaminathan, J.M., Smith, S.F., and Sadeh, N.M. 1998. Modeling supply chain dynamics: A multiagent approach. *Decision Sciences*. 29:607–632.

Uhrmacher A. and Schattenberg, B. 1998. Agents in discrete event simulation. In *Proceedings of the European Symposium on Simulation*, 8 pages. Available via <citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.5.4781&rep=rep1&type=pdf> [accessed June 22, 2009].

Wolverine Software Corporation. Undated. GPSS/H overview. Available via <www.wolverinesoftware.com/GPSSHOverview.htm> [accessed August 3, 2008].

AUTHOR BIOGRAPHIES

CHARLES M. JENKINS is a PhD candidate in the Department of Computer and Information Science at the University of Mississippi. Besides more than five years' teaching experience, he has worked in computing in the petroleum industry for more than twenty years. His email is <cmjenkin@olemiss.edu>.

STEPHEN V. RICE is an Assistant Professor in the Department of Computer and Information Science at the University of Mississippi. His career in computer science spans 31 years. He co–invented the MODSIM object-oriented simulation programming language in the late 1980s and wrote the first MODSIM compiler. In recent years, he led the design of the SIMSCRIPT III simulation programming language, which adds object-oriented features to the SIMSCRIPT II.5 language. His email is <rice@cs.olemiss.edu>.