Proceedings of the 2007 Winter Simulation Conference
S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds.

SIMIO: A NEW SIMULATION SYSTEM BASED ON INTELLIGENT OBJECTS

C. Dennis Pegden

Simio Corporation
PO Box 34
Sewickley, PA 15143, USA

ABSTRACT

Over the 50 year history of discrete event simulation the growth in applications has been facilitated by some key advances in modeling that have simplified the process of building, running, analyzing, and viewing models. Three important advances have been: the modeling paradigm shift from an event to a process orientation, the shift from programming to graphical model building, and the emergence of 2D/3D animation for analyzing and viewing model execution. These key advances were made 25 years ago and provided the foundation for the current set of modeling tools that are in wide use today. The past 25 years has been a period of evolutionary improvements with no significant advances in the core approach to modeling. The tools that are in use today are really just refined versions of what existed 25 years ago. This paper describes a new modeling system – Simio™- that is a departure from the design of existing modeling tools with the aim of improving the activity of model building. Simio is designed to simplify model building by promoting a modeling paradigm shift from the process orientation to an object orientation.

1. MODELING PARADIGMS

In the early days of discrete event simulation the dominant modeling paradigm was the event orientation implemented by tools such as Simscript (Markowitz, et .al 1962.) and GASP (Pritsker, 1967). In this modeling paradigm the system is viewed as a series of instantaneous events that change the state of the system. The modeler defines the events in the system and models the state changes that take place when those events occur. This approach to modeling is very flexible and efficient, but is also a relatively abstract representation of the system. As a result many people found modeling using an event orientation to be difficult.

In the 80's the process orientation displaced the event orientation as the dominant approach to discrete event simulation. In the process view we describe the movement of passive entities through the system as a process flow. The process flow is described by a series of process steps (e.g. Seize, Delay, Release) that model the state changes that take place in the system. This approach dates back to the 1960's with the introduction of GPSS (Gordon, 1960) and provided a more natural way to describe the system. However because of many practical issues with the original GPSS (e.g. an integer clock and slow execution) it did not become the dominant approach until improved versions of GPSS (Henriksen, 76) along with newer process languages such as SLAM(Pegden/Pritsker, 79) and SIMAN (Pegden, 82) became widely used in the 80's.

During the 80's and 90's graphical modeling and animation also emerged as key features in simulation modeling tools. Graphical model building simplified the process of building process models, and graphical animation dramatically improved the viewing and validation of simulation results. The introduction of Microsoft Windows made it possible to build improved graphical user interfaces and a number of new graphically based tools emerged (e.g. ProModel and Witness).

Another conceptual advance that occurred during this time was the introduction of hierarchical process modeling tools that supported the notion of domain specific process libraries. The basic concept here is to allow users to create new process steps by combining existing process steps. The widely used Arena modeling system (Pegden/Davis, 1992) is a good example of this capability.

Since the wide spread shift to a graphics-based process orientation there have been refinements and improvements in the tools, but no real advances in the underlying framework. The vast majority of discrete event models continue to be built using the same process orientation that has been widely used for the past 25 years.

Although a process orientation has proven to be very effective in practice, an object orientation provides an attractive alternative modeling paradigm that has the potential to be more natural and easier to use. In an object orientation we model the system by describing the objects that make up the system. For example we model a factory

by describing the workers, machines, conveyors, robots, and other objects that make up the system. The system behavior emerges from the interaction of these objects.

Although a number of products have been introduced to support an object orientation, to date they have been largely shunned by practitioners who have elected to stick with the process orientation. A big reason for this is that while the underlying modeling paradigm might be simpler and less abstract, the specific implementation may be difficult to learn and use (e.g. require programming), or slow in execution. This is no different than the challenges faced by the process orientation in unseating the event orientation. Although the first process modeling tool (GPSS) was introduced in 1961, it took 25 years before the process orientation was developed to the point that practitioners were persuaded to make the paradigm shift.

This paper describes Simio – a new simulation modeling tool that is designed to make the object orientation easy to use and efficient to execute. Although Simio incorporates a number of innovative features in pursuit of this goal, only time will tell if this tool has bridged the many practical issues that must be addressed to trigger a widespread paradigm shift in the way practitioners build models.

The tool is designed from the ground up to support the object modeling paradigm; however it also supports the seamless use of multiple modeling paradigms including a process orientation and event orientation. It also fully supports both discrete and continuous systems, along with large scale applications based on agent-based modeling. These modeling paradigms can be freely mixed within a single model.

2. THE SIMIO OBJECT PARADIGM

Simio is a ***sim***ulation modeling framework based on ***i***ntelligent ***o***bjects. The intelligent objects are built by modelers and then may be reused in multiple modeling projects. Objects can be stored in libraries and easily shared. A beginning modeler may prefer to use pre-built objects from libraries, however the system is designed to make it easy for even beginning modelers to build their own intelligent objects for use in building hierarchical models.

An object might be a machine, robot, airplane, customer, doctor, tank, bus, ship, or any other thing that you might encounter in your system. A model is built by combining objects that represent the physical components of the system. A Simio model looks like the real system. The model logic and animation is built as a single step.

An object may be animated to reflect the changing state of the object. For example a forklift truck raises and lowers its lift, a robot opens and closes its gripper, and a battle tank turns its turret. The animated model provides a moving picture of the system in operation.

Objects are built using the concepts of object-orientation. However unlike other object oriented simulation systems, the process of building an object is very simple and completely graphical. There is no need to write programming code to create new objects.

The activity of building an object in Simio is identical to the activity of building a model – in fact there is no difference between an object and a model. This concept is referred to as the equivalence principle and is central to the design of Simio. Whenever you build a model it is by definition an object that can be instantiated into another model. For example, if you combine two machines and a robot into a model of a work cell, the work cell model is itself an object that can then be instantiated any number of times into other models. The work cell is an object just like the machines and robot are objects. In Simio there is no way to separate the idea of building a model from the concept of building an object. Every model that is built in Simio is automatically a building block that can be used in building higher level models.

3. THE OBJECT ORIENTED FOUNDATION

Many popular programming languages such as C++, C#, and Java are all built around the basic principles of object oriented programming (OOP). In this programming paradigm software is constructed as a collection of cooperating *objects* that are instantiated from *classes*. These classes are designed using the core principles of *abstraction*, *encapsulation*, *polymorphism*, *inheritance*, and *composition*.

The abstraction principle can be summarized as focusing on the essential. The basic principle is to make the classes structure as simple as possible.

The encapsulation principle specifies that only the object can change its state. Encapsulation seals the implementation of the object class from the outside world.

Polymorphism provides a consistent method for messages to trigger object actions. Each object class decides how to respond to a specific message.

Inheritance is a key concept that allows new object classes to be derived from existing object classes: this is sometimes referred to as the "is-a" relationship. This is also referred to as *sub-classing* since we are creating a more specialized class of an object. Sub-classing typically allows the object behavior to be extended with new logic, and also modified by overriding some of the existing logic.

Composition allows new object classes to be built by combining existing object classes: this is sometimes referred to as the "has-a" relationship. Objects become building blocks for creating higher level objects.

It is interesting to note that the roots of these ideas date back to the early 1960's with the Simula 67 simulation modeling tool. This modeling tool was created by

Kristen Nygaard and Ole-Johan Dahl (1962) of the Norwegian Computing Center in Oslo to model the behavior of ships. They introduced the basic concepts of creating classes of objects that own their data and behavior, and could be instantiated into other objects. This was the birth of hierarchical modeling and object-oriented programming.

Many people assume that object-oriented programming concepts were developed within the programming world; however this was not the case. These principles were developed for building simulation models, and then adopted by the programming world.

Although the simulation world created the original object-oriented concepts, it has yet to produce an object-oriented modeling framework that practitioners have widely judged to be useful. Although there have been a number of attempts to provide such a framework, in the end practitioners have for the most part stuck to their proven process orientation for modeling. One of the big reasons for this is that most past attempts have simply been object-oriented programming libraries that require the user to step back in time 25 years and again code their models in a programming language.

4. THE SIMIO OBJECT FRAMEWORK

The Simio object framework is built on the same basic principles as object oriented programming languages; however these principles are applied within a modeling framework and not a programming framework. For example the Microsoft development team that designed C# applied these basic principles in the design of that programming language. Although these same principles drive the design of Simio, the result is not a programming language, but rather a modeling system. This distinction is important in understanding the design of Simio.

Simio is not simply a simulation modeling tool that is programmed in an OOP language (although it is programmed in C#). Likewise it is not simple a set of classes available in an OOP language such as Java or C++ that are useful for building simulation models. Simio is a graphical modeling framework to support the construction of simulation models that is designed around the basic object oriented principles. For example when you create an object such as a "machine" in Simio, the principle of inheritance allows you to create a new class of machines that inherits the base behavior of a "machine", but this behavior can be modified (overridden) and extended. Whereas in a programming language we extend or override behavior by writing methods in a programming language, in Simio we extend or override behavior by adding and overriding graphically defined process models.

This distinction between object oriented modeling and object oriented programming is crucial. With Simio the skills required to define and add new objects to the system are modeling skills, not programming skills.

5. THE ANATOMY OF AN OBJECT (MODEL)

When you create a model in Simio, you are creating an object class from which multiple instances can be created. This process is referred to as instantiation.

When you instantiate an object into a model, you may specify *properties* of the object that govern the behavior of this specific instance of this object. For example the properties for a machine might include the setup, processing, and teardown time, along with a bill or materials and a operator required during the setup. The creator of the object decides on the number and the meaning of the properties. The properties in Simio are strongly typed and can represent numeric values, Booleans, strings, object references, dates and times, etc. Since any model that you build is by definition an object, you have the opportunity to parameterize your model through properties as well.

It should be noted that instantiating a model is not the same as copying or cloning the model. When a model is used as a building block in the construction of other models it may be instantiated many times in many different models. The model instance simply holds a reference to the one model definition that is used over and over again. The instance also holds the property values since these are unique to each instance. However the model logic is shared by all instances. Regardless of how many instances are created, there is only one class definition of the object, and each instance refers back to this single definition. Each instance holds the properties that are unique to that instance, but looks back to the definition to get its underlying behavior. If the behavior in the definition is changed then all instances automatically make use of this new behavior.

In addition to properties, objects also have *states*. States are also strongly typed but always map to a numeric value. For example the Booleans true and false map to 1 and 0, and an enumerated list of state names map to the list index position (0, 1, …., N) in the list. A state changes as a result of the execution of the logic inside the object. Properties can be thought of as inputs to an object, and states can be thought of as output responses that change throughout the execution of the object logic. A state might represent a count of completed parts, the status of a machine selected from an enumerated state list, the temperature of an ingot heating in a furnace, the level of oil in a ship being filled, or the accumulation level on a conveyor belt.

There are two basic types of states: discrete and continuous. A discrete state is a value that only changes at event times (customer arrival, machine breakdown, etc.)

A continuous state (e.g. tank level, position of a cart, etc.) has a value that changes continuously over time.

6. THREE OBJECT TIERS

One of the important and unique internal design features of Simio is the use of a three tier object structure that separates an object into an object definition, object instance, and object realization. The object definition specifies the object behavior and it is shared by all instances of the object. An object instance is simply an instance of that object within a parent object definition (e.g. a lathe machine instance is placed inside a work cell definition). The object instance defines the property values for each individual instance of the object and this instance data is in turn shared by all object realizations.

The object realization is used to represent a specific realization of an instance within an expanded model hierarchy. For example, each time a new work cell instance is placed in a parent object definition (e.g. a production line) it creates the need for a new realization for the embedded lathe. Although the work cell definition is built from a single lathe instance, this single lathe instance cannot hold the state values corresponding to multiple lathe realizations that result from multiple instances of the work cell. The object realizations provide the mechanism for holding this hierarchical state information in a very compact form. The object realizations are only created during model execution and hold only the model state variables and a reference to their parent object instance. This is a highly efficient structure that is crucial for large scale applications such as agent-based models that can have many thousands of object realizations.

7. THREE WAYS TO BUILD OBJECT DEFINITIONS

The previous example in which we defined a new object definition (work cell) by combining other objects (machines and a robot) is one example of how we can create object definitions in Simio. This type of object is called a *composed object* because we create this object by combining two or more *component* objects. This object building approach is fully hierarchical, i.e. a composed object can be used as a component object in building higher level objects. This is only one way of building objects in Simio - there are two other important methods.

The most basic method for creating objects in Simio is by defining the logical processes that alter their state in response to events. For example, a machine object might be built by defining the processes that alter the machine state as events occur such as part arrival, tool breakdown, etc. This type of modeling is similar to the process modeling done in traditional modeling systems in use today such as Arena or GPSS. An object that is defined by describing its native processes is called a *base object*. A base object can in turn be used as a component object for building higher level objects.

The final method for building objects in Simio is based on the concept of inheritance. In this case we create an object from an existing object by overriding (i.e. replacing) one or more processes within the object, or adding additional processes to extend its behavior. In other words we start with an object that is almost what we want, and then we modify and extend it as necessary to make it serve our own purpose. For example we might build a specialized drill object from a generalized machine object by adding additional processes to handle the failure and replacement of the drill bit. An object that is built in this way is referred to as a *derived object* because it is sub-classed from an existing object.

Regardless which method is used to create an object, once created it is used in exactly the same way. An object can be instantiated any number of times into a model. You simply select the object of interest and place it (instantiate it) into your model

8. OBJECT CLASS

There are four basic classes of objects in Simio. These four classes of objects provide a starting point for creating intelligent objects within Simio. By default, all four of these classes of objects have very little native intelligence, but all have the ability to gain intelligence. You build intelligent versions of these objects by modeling their behavior as a collection of event driven processes.

The first class is the most basic and is simply referred to as an object. An object has a fixed location in the model and is used to represent the things in your system that do not move from one location to another. Objects are used to represent stationary equipment such as machines, fueling stations, etc.

Objects are also typically used for developing agent-based models. This modeling view is useful for studying systems that are composed of many independently acting intelligent objects that interact with each other and in so doing create the overall system behavior. Examples of applications include market acceptance of a new product or service, or population growth of competing species within an environment.

The second class of object in Simio is an entity. An entity is sub-classed from the object class and has one important added behavior. Entities can move through the system from object to object over links. Examples of entities include customers in a service system, work pieces in a manufacturing system, ships in a transportation system, tanks in a combat system, and doctors, nurses, and patients in a health delivery system.

Note that in traditional modeling systems such as GPSS or Arena the entities are passive and are acted upon by the model processes. However in Simio the entities can have intelligence and control their own behavior.

The third class of object in Simio is a link and is subclassed from the object class. A link defines a pathway for entity movement between objects. Links can be combined together into complex networks. Although the base link has little intelligence we can add behavior to a link to allow it to model unconstrained flow, congested traffic flow, or complex material handling systems such as accumulating conveyors or power and free systems.

The fourth class of object is a transporter and is subclassed from the entity class. A transporter is an entity that has the added capability to pickup, carry, and drop off one or more other entities. By default transporters have none of this behavior, but by adding model logic to this class we can create a wide range of transporter behaviors. A transporter can model a taxi cab, bus, AGV, subway car, forklift truck, or any other object that has the ability to carry other entities from one location to another.

A key feature of Simio is the ability to create a wide range of object behaviors from these four basic classes. The Simio modeling framework is application domain neutral – i.e. these four basic classes are not specific to manufacturing, service systems, healthcare, military, etc.. However it is easy to build application focused libraries comprised of intelligent objects from these four classes designed for specific application.. For example it is relatively simple to build an object (in this case a link) that represents a complex accumulating conveyor for use in manufacturing applications. The design philosophy of Simio directs that this type of domain specific logic belongs in the objects that are built by users, and not programmed into the core system.

9. CREATING INTELLIGENT OBJECT WITH PROCESSES THREE WAYS TO BUILD OBJECT DEFINITIONS

Modeling in Simio begins with base objects – it is the foundation on which higher level objects are built. A base object in Simio is an object, entity, link, transporter, agent, or group that has intelligence added by one or more processes. Processes give an object its intelligence by defining the logic that is executed in response to events.

Each process is a sequence of process steps that is triggered by an event and is executed by a *token*. A process always begins with a single Begin step, and ends with a single End step. A token is released by the Begin step and is simply a thread of execution (similar to entities in Arena). A token may have properties (input parameters) and states (runtime changeable values) that control the execution of the process steps. You can define your own classes of tokens that have different combinations of properties and states.

The modeling power of Simio comes from the set of events that are automatically triggered for the four basic classes of objects, along with the process steps that are available to model state changes that occur in response to these events. Fully mastering the art of building intelligent objects involves learning the events and the collection of available process steps, along with the knowledge and experience of how to combine these steps to represent complex logic.

Each step in Simio models a simple process such as holding the token for a time delay, seizing/releasing of a resource, waiting for an event to occur, assigning a new value to a state, or deciding between alternate flow paths. Some steps (e.g. *Delay*) are general purpose steps that are useful in modeling objects, links, entities, transporters, agents, and groups. Other steps are only useful for specific objects. For example, the *Pickup* and *Deliver* steps are only useful for adding intelligence to transporters and the *Enter* and *Exit* steps are only useful in adding intelligence to Links.

Each object class has its own set of events. For example, a link provides events that fire when entities enter and leave the link, merge fully onto the link, collide with or separate from other entities that reside on the link, move within a specified range of another entity, etc. By providing model logic to respond to these events we can completely control the movement of entities across the link. For example, to add accumulation logic to the link we simply write a small process that its triggered when an entity collides with the entity it is following, and reassigns the speed of the entity to match the speed of the entity that it is following.

The process steps that are used to define the underlying logic for an object are stateless – i.e. they have properties (input parameters) but no states (output responses). This is important because this means that a single copy of the process can be held by the object class definition, and shared by an arbitrary number of object instances. If the process logic is changed, this fact is automatically reflected by all instances of the object.

The states for an object instance are held in *elements*. Elements define the dynamic components of an object and may have both properties (input parameters) and states (runtime changeable values). Within an object the tokens may execute steps that change the states of the elements that are owned by the object.

An example of an element is the station that defines a location within an object. Stations are also used to define entry and exit points into and out of an object. Entities can transfer into and out of stations (using the Transfer step), and a station maintains a queue of entities currently in the station as well as entities waiting to transfer into the station. A station has a capacity that limits transfers into a

station. Hence an entity arriving to an object over a link can only exit the link and enter the object if the entry station for the object has capacity available.

10. TRIGGERING PROCESSES WITH EVENTS

There are several different ways to trigger the execution of a process; however one of the most common ways is by a triggering event. A triggering event is simply an event that "triggers" the Begin step in the process to send out a new token. A triggering event can be one of four basic types: time event, logic event, change event, or cross event.

A time event is a convenient way to generate random arrivals to a process. A time event is fired automatically according to a specified time pattern. This time pattern can be either a stationary or a non-stationary pattern. In the case of a stationary pattern the properties specify the time of the first event, the time between each successive event, and the maximum number of events to fire. These parameters can be constant values (e.g. every 5 minutes), or random values (e.g. a sample from an exponential distribution). In the case of a non-stationary pattern the properties specify a repeating pattern cycle that varies over the time of day and the day of week (or any appropriate cycle). This is useful for modeling time-dependent customer demand.

A logic event is used to trigger processes based on a logical occurrence. Here the event is being fired by the underlying logic of the model as opposed to some specified time pattern. A typical example is the station's *TransferIn* event. This is a logic event that is fired whenever an entity is transferred into a station that is owned by the object. This is the standard way for triggering process logic to respond to an entity arrival to an object.

A change event occurs when ever a specified discrete state variable changes value (e.g. a queue length changing). A change event is defined by simply specifying a discrete state variable of interest. The change event is fired whenever the value of this state variable changes.

A cross event is used to monitor continuous state variables. Since a continuous state variable is constantly changing, a change event is not meaningful. Instead we define a cross event that is fired whenever the state variable crosses a specified threshold in either a *positive*, *negative*, or *either* direction. For example we might use a cross event to trigger a process whenever the tank level reaches full (positive cross with maximum tank level) or empty (negative cross with 0). Although cross events are very useful with continuous state variables, they may also be used with discrete state variables.

11. DECISION PROCESSES

Most processes in Simio span time – i.e. simulated time advances from the point in time when a token is first released from the Begin step until it arrives to the End step. This time delay may be caused by explicit delays at the Delay step (e.g. delay for 2 minutes), or by queuing delays at constrained steps (e.g. the Seize step).

Processes that may span time are referred to as standard processes. However there is a special type of process in Simio that is referred to as a decision process. A decision process executes in zero time and is used to make a decision about a specific action. For example, when a transporter arrives to a transfer station and decides to pick up an entity, it triggers a decision process owned by the entity that can decide to accept or reject the pickup. Since decision processes must always execute in zero time, steps that execute over time (e.g. Delay, Seize, Wait, Allocate, etc.) are not allowed in decision processes.

12. FINITE CAPACITY SCHEDULING

Although simulation has traditionally been applied to the design problem, it can also be used on an operational basis to generate production schedules for the factory floor. When used in this mode, simulation is a Finite Capacity Scheduler (FCS) and provides an alternative to other FCS methods such as optimization algorithms and job-at-a-time sequencers. However simulation based FCS has a number of important advantages (e.g. speed of execution and flexible scheduling logic) that make it a powerful solution for scheduling applications

Simulation provides a simple yet flexible method for generating a finite capacity schedule for the factory floor. The basic approach with simulation-based scheduling is to run the factory model using the starting state of the factory and the set of planned orders to be produced. Decision rules are incorporated into the model to make job selection, resource selection, and routing decisions. The simulation constructs a schedule by simulating the flow of work through the facility and making "smart" decisions based on the scheduling rules specified. The simulation results are typically displayed as jobs loaded on interactive Gantt chart that can be further manipulated by the user. There are a large number of rules that can be applied within a simulation model to generate different types of schedules focused on measures such as maximizing throughput, maintaining high utilization on a bottleneck, minimizing changeovers, or meeting specified due dates.

Because of the special requirements imposed by scheduling applications (e.g. the need for specialized decision rules and the need to view the results in the form of an interactive Gantt chart), simulation-based scheduling

applications have typically employed specialized simulators specifically designed for this application area. The problem with this approach is that the specialized simulators have built-in, data-driven factory models that cannot be altered or changed to fit the application. In many cases this built-in model is an overly simplified view of the complexities of the production floor. .This *one model fits all* approach severely limits the range of applications for these tools. Some production processes can be adequately represented by this fixed model, but many others cannot.

Simio takes a different approach by allowing the factory model to be defined using the full general-purpose modeling power of the tool. Hence the range of applications is no longer restricted by a fixed built-in model that cannot be altered or changed between applications. The complexities of the production process can be fully captured by the user-built Simio model. This not only includes the logic within each work center, but also the material handling required to move jobs between work centers.

The specialized requirements for FCS applications are addressed by incorporating features into Simio to specifically support the needs of FCS. These features include the support for externally defined job data sets along with very flexible modeling of resources and materials. Although these features are specifically designed to unleash the full modeling power of Simio for FCS applications, they are also useful in general modeling applications.

A Simio job data set allows a list of jobs to be externally defined for processing by the simulation model. The jobs are defined in a data set containing one or more tables, with relations defined between table columns. The specific schema for holding the job data is arbitrary and can be user defined to match the data schema for the manufacturing data (e.g. an ERP system). The job data typically includes release and due date, job routings, setup and processing times, material requirements, as well as other properties that are relevant to the system of interest.. The objects in Simio can directly reference values specified in the job data set (e.g. processing time) without knowing the schema that was implemented to store the data..

The Resource element in Simio provides direct support for modeling complex resource selection and dynamic routing logic. Resources in Simio have unit capacity and can be seized, released, and preempted by entities (jobs). Resources can follow work shifts that can alter the time spent by a job being processed by the resource. Resources can also model complex changeover logic for jobs that utilize the resource (e.g. attribute dependent or sequence-dependent changeovers). Resources can be placed in multiple lists, and selection of a resource from a list can be based on flexible rules such as minimum changeover time or longest idle time. Jobs can also be dynamically routed between objects based on the state of a resource (e.g. a machine) that is owned by each object. Resources also support very flexible rules (earliest due date, least remaining slack, critical ratio, etc) for selecting between competing jobs that are waiting to seize the resource. Finally the job usage history for resources can be displayed on an interactive Gantt chart.

The Materials element in Simio provides direct support to model things that can be consumed and produced during the execution of the model. Materials can also be defined hierarchically to model a traditional Bill of Materials (BOM) for manufacturing applications. Hence a manufacturing step can be modeled as the consumption of a specific list of materials within the hierarchical BOM.

13. SUMMARY

Simio is a new modeling framework based on the core principles of object oriented modeling. It is unique in the following ways:

1. The Simio framework is a graphical object-oriented modeling framework as opposed to simply a set of classes in an object-oriented programming language that are useful for simulation modeling. The graphical modeling framework of Simio fully supports the core principles of object oriented modeling without requiring programming skills to add new objects to the system.
2. The Simio framework is domain neutral, and allows objects to be built that support many different application areas. The process modeling features in Simio make it possible to create new objects with complex behavior.
3. The Simio framework supports multiple modeling paradigms. The framework supports the modeling of both discrete and continuous systems, and supports an event, process, object, and agent modeling view.
4. The Simio framework provides specialized features to directly support applications in finite capacity scheduling that fully leverage the general modeling capabilities of Simio.

REFERENCES

Gordon, Geoffrey, 1960 October 25. *A general purpose systems simulator. (Unpublished manual.)* White Plains, N.Y.: IBM. Corp. ASDD Commercial Dept.

Henriksen, J. O. 1976. Building a better GPSS: a 3:1 enhancement. In *Proceedings of the 1975 Winter Simulation Conference*, 465-469. New Jersey: AFIPS Press

Markowitz, H., Hausner, B., and Karr, H. *SIMSCRIPT: A simulation programming language*, Prentice Hall, Englewood Cliffs, N. J. 1962

Nygaard, K and O-J Dahl, . SIMULA -- An Extension of ALGOL to the Description of Discrete-Event Networks, presented at the *Second International Conference on Information Processing* (1962)

Pegden, C. D. and A. A. B. Pritsker (1979). SLAM: Simulation Language for Alternatives Modeling. *Simulation*, Vol. 33, No. 5.

Pegden, C. D. (1982). *Introduction to SIMAN*. Systems Modeling Corporation.

Pegden, C. D. and D. A. Davis (1992) Arena: a SIMAN/Cinema-based Hierarchical Modeling System*;* In *Proceedings of the 1975 Winter Simulation Conference* 390-399

Pritsker, A. A. B. 1967. *GASP H User's Manual*. Arizona State University.

AUTHOR BIOGRAPHIES

C. Dennis Pegden was the founder and CEO of Systems Modeling Corporation, now part of Rockwell Software, Inc. He has held faculty positions at the University of Alabama in Huntsville and The Pennsylvania State University. He led in the development of the SLAM, SIMAN, Arena, and Simio simulation tools. He is the author/co-author of three textbooks in simulation and has published papers in a number of fields including mathematical programming, queuing, computer arithmetic, scheduling, and simulation. . His email is cdpegden@simio.biz.