Proceedings of the 1988 Winter Simulation Conference
M. Abrams, P. Haigh, and J. Comfort (eds.)

The SIMPLE_1 simulation environment

Philip Cobbin
Sierra Simulations & Software
303 Esther Avenue
Campbell, California 95008
(408) 378-6374

Overview

SIMPLE_1 is a simulation language and integrated environment for IBM PC and compatible computers which blends visual and interactive modeling features with C, Pascal, and object oriented programming (OOP) concepts. SIMPLE_1 is a discrete and a continuous simulation language implemented as a modeling environment with integrated editor, documentation, diagnostics, and tutorials. The language has undergone continued refinement from introduction in 1985 and features user defined variables, functions, and an object oriented procedure concept called a process. A key element in the language is a rule based construct called a CONDITIONS block which defines rules for releasing entities from queues.

SIMPLE_1 supports modeling discrete and continuous systems world views using a network modeling orientation. Features of the language include the ability of the user to declare variables and statistics requirements, perform I/O operations on files and to animate simulation results in real time. Both text and bit-mapped images can be imported by models in the latest version of the language for animation and graphic display purposes. The language is reported to have a more natural syntax than other simulation languages[16], due in part to a more close resemblance to high-level programming languages and the free-format syntax of the language.

The body of a SIMPLE_1 model is composed of five sections: DECLARE, PRERUN, DISCRETE, CONTINUOUS, and POSTRUN. The DECLARE section is used to define key model variables such as entities, matrices, and so forth. The PRERUN and POSTRUN sections execute in a procedure like manner. SIMPLE_1 models typically employ the basic block types of the language to define discrete and continuous models. There are relatively few block types in SIMPLE_1 and this brevity of language concepts is due to the flexibility of the CONDITIONS block.

Discrete event aspects of a model are defined using an activity on node network structure. The Continuous aspects of the system model are described using algebraic state equations which define variables overtime via first order differential equations. The continuous aspects of the model are simulated using a Runge Kutta or other supported integration method with the step size assignable by the modeler.

The discrete aspects of the model are processed via an event scheduling mechanism to sequence the flow of entities through blocks in the network model. The discrete event processing algorithm evaluates and resolves interdependence conditions in the model prior to advancing to the next event.

An example of a MM1 queuing system model in SIMPLE_1 would look like:

```
DECLARE;
     GLOBALS:  TimeInSystem OBSERVE_STATS;
     ENTITIES: CUSTOMER(1);
END;
PRERUN;
     SET STOP_TIME := 1000;
END;
DISCRETE;
     CREATE,1,CUSTOMER,EXPON(5.0,1);
     SET CUSTOMER(1):=STIME;
WaitServer
     QUEUE,FIFO;
          CONDITIONS,NUM(Service)<1,
               WaitServer,,Service;
Service
     ACTIVITY EXPON(4.5,1);
     SET TimeInSystem:=STIME-CUSTOMER(1);
     KILL;
END;
CONTINUOUS; END;
POSTRUN;
     REPORT;
     STOP;
END;
```

Where the CREATE statement is generating the arrival stream and the QUEUE-CONDITIONS-ACTIVITY statement sequence is modeling the servicing of entities waiting in the queue. The CONDITIONS statement in this model is used to specify the conditions necessary for a customer to leave the queue and enter the service activity.

Cornerstone Concept: The CONDITIONS block:

The CONDITIONS block is used to define the state conditions required for entities to leave queues. The block is the cornerstone of the language and provides a unified queue release mechanism. The block functions somewhat analogous to a chameleon, in that a CONDITIONS block can be configured for a diversity of queue release constraints in much contrast to traditional discrete simulation languages. In a basic queue-server relationship a CONDITIONS block is used to associate a specific QUEUE with an ACTIVITY block. The CONDITIONS block is the principal

means for formation of groups of entities and is readily applied to modeling assembly constraints in manufacturing.

Notably absent in SIMPLE_1 is the concept of a resource for modeling complicated queue-server relationships. SIMPLE_1 does not employ resources because the CONDITIONS block is used to model simplistic and complex resource situations. Key system resources in SIMPLE_1 models are typically modeled as entities that are grouped with other entities while in use and SPLIT from the customer and routed to a QUEUE when the resource entity becomes idle. The advantage inherent in modeling resources as a separate entity type is the ability to model explicitly the decision making processes of the resource inclusive of the resources own attribute state. For example, the entry of passengers onto a bus is typically a function of the route assigned to a bus. Accordingly, modeling such a situation in SIMPLE_1 involves modeling decisions based on passenger attributes and bus system state variables.

The CONDITIONS block has proved to be a flexible and powerful construct of the language. The block ties together SIMPLE_1's basic set of discrete simulation primitives namely the QUEUE and the ACTIVITY block. A MONITOR block construct operates in a similar fashion as the CONDITIONS block by monitoring specific aspects of the simulation and is typically used to drive real time animation of simulation results. As changes occur in the model during an event the affected MONITORs are executed prior to advancing to the next event. For example when an INSPECT activity is started or completed a MONITOR is used to detect the change in state of the activity. As a side affect of the INSPECT activity changing state a MONITOR block would be used to update information on the screen to show the current number of INSPECT activities in progress. MONITOR statements in general are used for driving model animation and in tandem with CONDITIONS block for calculation of decision making variables.

The CONDITIONS block concept also supports building models in stages. In most situations you start off modeling the main processes and add embellishments to capture additional constraints on system operation. When modeling assembly of a product one can start by modeling the basic process sequence and add part queues later to capture the affects of assembly constraints on the overall efficiency of the system. In addition, blocking, and other constraints on system operation can typically be added in stages without a major restructuring of the model.

Groups of Entities:

The CONDITIONS block is used to release entities from queues. Entities are a dynamic type with each entity type defined by a unique identifier i.e. TV, ForkTruck. The CONDITIONS block can release multiple QUEUES as a set and organize the released entities into a group of entities. Entity groups in turn are viewed as traveling together and can share attributes. Constructs in the language allow further processing on groups of entities to SPLIT them up, CLONE the groups, or REGROUP their relative ordering.

Manipulation of entity attributes by their unique name simplifies referencing attributes and improves the self documentation of models. When entities are organized into groups the ^ operator is used in referencing individual entities. For example, If televisions go by the name TV and each has four attributes then

```
TV(3)^5
```

would reference the third attribute of the fifth TV in a group of TVs.

The entity grouping feature of the language is particularly suited for modeling assembly operations in manufacturing and complex resource management situations.

OOP concepts & SIMPLE_1

Object oriented programming languages ala Smalltalk are based on the Simula simulation language and emphasize message sending among objects to accomplish programming tasks. Fundamental properties of OOP languages are: abstraction, encapsulation, inheritance, and polymorphism. SIMPLE_1 was developed independently from the OOP community. SIMPLE_1 is not a pure OOP language but it does possesses OOP like characteristics. The language supports the application of OOP methods while differing from OOP languages such as Smalltalk by being more closely related to high level programming languages. SIMPLE_1 differs from the OOP constructs by requiring a less rigorous adherence to polymorphism and encapsulation. The implementation of SIMPLE_1 has evolved to require less run time binding of variables to improve execution speed with current development efforts migrating to that of a compiled language. The CONDITIONS block in particular maximizes binding of event dependencies at compile time for efficient evaluation of queue release conditions at run time. SIMPLE_1 is in essence a blending of Simulation concepts with block oriented procedural languages such as C and Pascal. The four fundamental concepts of OOP languages do however form a good outline for surveying SIMPLE_1 concepts.

Abstraction:

Simulation languages emphasize abstraction in their constructs and SIMPLE_1 uses abstraction for modeling the behavior of entities in a system primarily with QUEUE and ACTIVITY blocks. Message passing in models is accomplished by one of two mechanisms. Organizing entities into groups is used to share/transfer information and the CONDITIONS block is used to both define rules for

releasing entities from queues and for defining monitoring conditions (a kind of message) between modeling elements.

A simple example illustrates abstraction. The SIMPLE_1 code fragment:

```
          CREATE,1,customer, 45;
WaitForTellerIn QUEUE,FIFO;
                CONDITIONS,NUM(TellerService)<1,
                WaitForTellerIn,,TellerService;
TellerService   ACTIVITY  40;
```

Describes a simple queuing situation. As a representation of the system the model creates customer entities and sends them to a queue to wait for an available teller to conduct a banking transaction. Embellishing the fragment with comments within braces {} the code can be interpreted as:

```
CREATE,1,customer,{every} 45;{minutes}
WaitForTellerIn QUEUE,FIFO; {until the}
CONDITIONS, NUM(TellerService) < 1,
{are true at which time remove the customer
from } WaitForTellerIn,, {and send them to
the activity} TellerService; {where }
TellerService {is an } ACTIVITY{taking} 40;
{time minutes to complete}
```

The CONDITIONS statement is used here as a message manager to monitor the state of both the WaitForTellerIn queue and the TellerService activity. As the queue and/or activity blocks change state the CONDITIONS block is informed to trigger the next release from the queue. Encapsulation is occurring here in that the CONDITIONS block is an object performing a queue release function without the modeler having to operate on event list structures or other low level simulation details.

Encapsulation:

In SIMPLE_1 the fundamental modeling unit in discrete systems is the entity. Entities in SIMPLE_1 have attributes and form a packet of data. In an OOP sense a SIMPLE_1 entity by itself is not an object. Rather an object is a combination of entities and a network of discrete processes. A current developments in SIMPLE_1 is the encapsulation of discrete and continuous processes in a procedure like structure. Unlike procedures however a SIMPLE_1 process can have the return from the process delayed by interactions with QUEUE and ACTIVITY blocks. The process construct allows extending model abstractions and packaging the implementation. In a manufacturing system context a process model of a manufacturing process might be:

```
CREATE,BatchSize,Castings,24;
       {setup}                {run}
Turn,UNIFORM(3.0,5.0,1),NORMAL(21.5,3,1);
Mill,EXPON(15.2,1)     ,NORMAL(45.0,60.0,1);
ExitSystem;
```

Where details of how the TURN operation are implemented is defined by a SIMPLE_1 process. A process for the Turn operation defines the activities associated with the operation and can include ACTIVITY, QUEUE, and CONDITIONS statements to define a sub-model for the Turn operation. An example process is:

```
PROCESS Turn,Setup,Run;

 ENTRY_POINT LatheWip;
 DECLARE;
     GLOBALS: NumLth;
 END;
 PRERUN;
     SET NumLth := 3;
 END;
 DISCRETE;
 LatheWip    QUEUE,FIFO;
             CONDITIONS,
               NUM(LSetup)+NUM(LRun) < NumLth,
                        LatheWip,,LatheSetup;
 LSetup      ACTIVITY Setup;
 LRun        ACTIVITY Run;
 END;
 CONTINUOUS; END;
 POSTRUN; END;

END_PROCESS;
```

Where the calling entity references Turn which results in the entity beginning processing at the entry point labeled LatheWip which is a QUEUE.

The PROCESS construct is a current development in the language which allows the ACTIVITY, QUEUE, and CONDITIONS blocks to be used for developing libraries of detailed low level sub-models that are then used to construct other models at a high level of abstraction.

Inheritance:

Identifiers are globally scoped in SIMPLE_1 thus diverging markedly from the inheritance principle of OOP. In addition the related "need to know" and data hiding concepts are counter to the thrust of the language. SIMPLE_1 as a simulation language maximizes the communication among variables in the model. Data hiding and inheritance are concepts driven by the practical needs to control large programming projects and to some extent the "need to know" constraints of military systems applications.

Data hiding inhibits Statistics collection on model behavior. When variables are declared key words are used to specify automatic collection of time persistent and observational statistics. The language supports statistics collection for scalars and matrices; Data hiding would complicate the referencing and reporting of these statistics.

Polymorphism:

The language supports Polymorphous message passing by employing entities to pass messages via their attributes. The mechanism in outline form involves assigning the message information to an entity which in turn is combined with the receiver entity, or entity

group. The transmission of the message is accomplished by joining the messenger entity with a target receiver group with a CONDITIONS statement. Orders for example might be transmitted to a "RED" field commander by the code fragment:

```
SET ORDERS(Immediate) := Patrol:
    ORDERS(Area)      := Sector(5);
SendOrders
  QUEUE,FIFO;
      CONDITIONS,,SendOrders,,RedCommands:
                  NextRedMsg,,RedCommands;
```

When the RedCommands block is processed the receiver entity will have access to the ORDERS entities information. The actions taken by a Red commander to the Patrol command may be quite different from those of a Blue commander. Note that the Polymorphous property is supported here in a fashion similar to implementations with C or Pascal. The context specific actions on messages is supported by SIMPLE_1 versus being a requirement in the language.

Applications of SIMPLE_1:

SIMPLE_1 has been applied in manufacturing, academia, and by the United States Military. Applications to date have ranged from manufacturing systems, robotics justification, health care systems, emergency planning, and analysis of logistic support systems. SIMPLE_1 has been used to plan for future manufacturing systems, as well as a tool for scheduling current systems. Factory scheduling applications have been developed with the simulations in one case interfacing with shop floor data collection systems to trace drive the simulation via historical bar code data. CIM applications include simulations of automated circuit board assembly and automated material handling systems for defense-aerospace application in addition to cellular manufacturing investigations[9]. The language has not been limited solely to manufacturing with health care applications reported by Smith[12]. A number of students throughout the United States have employed the language in support of their graduate work and the language is currently being used in simulation courses at a number of universities. The language has been employed by Starr et al [13] to investigate schedule recovery strategies and currently the software is being used in applied research for scheduling an electronic assembly system in a Midwestern aerospace site. Inspection issues relative to FMS systems was investigated using SIMPLE_1 according to Hauck [10]. SIMPLE_1 was developed to model complex logistics issues and one of the first applications of the language was an investigation of logistics support for avionics equipment by Bottomley[1].

SIMPLE_1 References

[1] Bottomley, Larry D. Capt. USAF, "Station Loading on the DATSA (Depot Automated Test Station for Avionics)", unpublished Masters Thesis, Air Force Institute of Technology, Wright-Patterson AFB, OH, 1986.

[2] Cobbin, Philip, "SIMPLE_1: A simulation environment for the IBM PC", Modeling and Simulation on Microcomputers, Claude, C. Barnett, Editor, Society for Computer Simulation, La Jolla, 1986, pp 243-248.

[3] Cobbin, Philip, "Applying SIMPLE_1 to manufacturing systems", Summer Computer Simulation Conference, July 28-30 1986, Reno, Nevada, Roy Crosbie and Paul Luker, Editors, Society for Computer Simulation, La Jolla, pp 724-730.

[4] Cobbin, Philip, "A Tutorial on the SIMPLE_1 simulation environment", Winter Simulation Conference proceedings, December 1976, Washington D.C. pp 168-177.

[5] Cobbin, Philip, " Modeling tote stacker operation as a WIP storage device", Winter Simulation Conference proceedings, December 1986, Washington D.C. pp 597-605.

[6] Cobbin, Philip, "SIMPLE_1: Follow-on developments in the life of a micro-based simulation language", Modeling and Simulation on Microcomputers, Paul F. Hogan Editor, Society for Computer Simulation, La Jolla, 1987, pp 29-32.

[7] DiBiase, Debra, "The cash flow simulator: A microcomputer based model", Modeling and Simulation on Microcomputers, Paul F. Hogan Editor, Society for Computer Simulation, La Jolla, 1987, pp 101-103.

[8] DiBiase, Debra, "The inventory simulator: A microcomputer based inventory model", Modeling and Simulation on Microcomputers, Paul F. Hogan Editor, Society for Computer Simulation, La Jolla, 1987, pp 104-106.

[9] Dooley, Starr, Vig & Mahmoodi, "Cellular Manufacturing Project Workshop", CIM Consortium University of Minnesota, May 1988.

[10] Hauck, Warren Stephen, "A study of heuristics for inspection location in flexible manufacturing systems", unpublished Masters Thesis, The University of Iowa, Iowa City Iowa, 1987.

[11] Sierra Simulations & Software: SIMPLE_1 User's guide and reference manual, 1985.

[12] Smith, Philip E. "Simulation as a valuable tool: A proposal to combine admitting and outpatient registration" Midwest Regional Conference of the Health-care Information Management Systems Society, 1987.

[13] Starr, Patrick, Skrien,Douglas, and Meyer, Robert,"Simulating schedule recovery strategies in manufacturing assembly operations" Winter Simulation Conference proceedings, December 1986, Washington D.C. pp 694-699.

[14] Thinnes, Keren, M., "Simulation of Printed Circuit Board Manufacturing" unpublished Masters Thesis, The University of Iowa, 1987.

[15] "Introduction to SIMPLE_1", Video tape lecture series developed at The University of Idaho, Moscow, Idaho, 1988.

[16] Van Houten, Keren, "Simulation Languages for PCs take different approaches" IEEE Software, January 1988 pp 91-94.

Authors´ Biography

Philip Cobbin is the owner of Sierra Simulations & Software and is the developer of SIMPLE_1. In addition to simulation software development Phil has taught undergraduate and graduate level simulation courses and consults on the application of simulation. He holds a Master of Science in Industrial Engineering from Purdue University and a Bachelor of Science in Industrial Engineering and Operations Research from the University of Massachusetts at Amherst. Phil is a native of Los Angeles and has been previously employed by the General Products Division of the International Business Machines corporation (IBM) performing simulation modeling and material handling engineering activities.