Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

SIMPLE_1: THE LANGUAGE & MODELING ENVIRONMENT

Philip Cobbin
Sierra Simulations & Software
RR#2 Box 918B
Canaan, New Hampshire 03741
(603) 523-9645

Abstract

SIMPLE_1: (SIMulation Program for Logistics Engineering) 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 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, procedures, 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.

Introduction

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 [Van Houten 1988], due in part to a more close resemblance to high-level programming languages and the free-format syntax of the language. The current version 4.0 release is more than two and half times faster than the previous release. With language constructs principally developed the performance aspects of SIMPLE_1 have been refined.

Discrete system models are defined via networks used to define the flow of events for entities. A key concept in SIMPLE_1 is the ability to organize entities into groups which travel together and retain their unique characteristics. In addition, groups of entities can be assembled as a collection of different types. The entity grouping feature of the language is particularly suited for modeling complex resource management situations typically encountered in logistics, health care and industrial systems.

Integrated Modeling Environment

The implementation of SIMPLE_1 is organized into a set of modules accessed by a pull down menu system. The editor is an integral part of the software. The user's perspective of the software environment is that of working from the editor so that while editing a file the compiler can be called to compile and run a simulation. Multiple files can be edited with file size limited to the amount of DOS RAM available on the system. When a compiler or run time error is detected the software returns the modeler to the editor and points out the source of the problem. This system is linked to a hot-key syntax look up routine to aid in debugging. The text editor is also the principle means for reviewing model reports and editing data files.

To facilitate learning the system, extensive language documentation is accessible quickly using the keyboard function keys. On-line syntax and theory of operation information for language concepts further support the debugging of models and learning the system.

Animation of simulations uses SIMPLE_1 language elements to direct updating of the monitor to reflect the changing state of the simulation model. A debugging facility is included in the run time system to interrupt the model and review or change program variables. The character and bit-mapped animation schemes combined with the run time debugger have been found particularly useful for model verification and problem isolation.

Run Time Debugger

The run time debugging facility like the editor is an integral part of the environment. The event scheduling aspect of discrete event model execution makes simulation models particularly difficult to debug using traditional debugging approaches.

The SIMPLE_1 debug facility is a menu driven system to interactively investigate the status of a model while it is running. The menu system is popped up when a key is depressed during the execution of a simulation. The debugging subsystem allows the user to:

- o Halt/Continue the simulation.
- o Lookup and change global variables in the model.
- o View and Change block contents
- o View and change the value of individual entity attributes.
- o Obtain a report on current simulation results.
- o Temporarily exit to a DOS shell.

Language Concepts

The language was developed to provide a concise set of basic building block constructions for modeling discrete and continuous systems. A network approach to modeling has been found to be a highly effective vehicle for describing and documenting models. An activity on node orientation is used in the language documentation. SIMPLE_1 provides basic modeling primitives that are employed using a block diagramming concept coupled to general purpose programming language elements for manipulation of data structures. A repetitive approach to run control, based on Continuous System Simulation Languages was merged with discrete system modeling concepts to provide a unified structure for modeling. Declaration of variables is required in SIMPLE_1 to avoid the casual declaration problems inherent in languages like FORTRAN and to provide a means of flagging statistics collection, integration, and event triggering conditions.

Code organization is designed to segregate variable declaration, model definition, and run control code elements of a simulation program. SIMPLE_1's code is organized into five sections associated with key the words: DECLARE, PRERUN, DISCRETE, CONTINUOUS, and POSTRUN. The DECLARE section is used to define data structures such as entities and global variables. SIMPLE_1 uses repetitive run control with a PRERUN and POSTRUN phase of model processing. A program initially executes with the PRERUN being called to set model variables and bind run control parameters. When the PRERUN code completes execution, the processing of entity movements through SIMPLE_1 blocks to model discrete processes is performed. Discrete event aspects of the model are defined using an activity on node network structure. The discrete segment of the model is processed by an event scheduling mechanism to sequence the flow of entities through blocks in the network model. Simulation of the CONTINUOUS elements of the system is executed by numerical integration of continuous variables using INTEGRATE blocks to define variable relationships. Using SIMPLE_1's repetitive approach to run control one can look at the results of a simulation to establish parameter values for the next run. The Continuous aspects of the system model are described using algebraic state equations which define variables via first order differential equations. The Continuous aspects of the model are simulated using either a Runge Kutta fourth order fixed step, or a Fehlberg integration procedure with the step size assignable by the modeler.

The language is similar to Pascal in that one syntax error tends to cascade into a large number of ghost errors. Accordingly, the detection of syntax errors during compilation halts after the first error and calls the editor after displaying a diagnostic message.

SIMPLE_1 variable identifiers can have up to 20 significant characters including the underscore character to facilitate self documentation of the model. The language supports the declaration of the following classes of code and data structures:

Global variables

Scalars and arrays with single or double subscripts.

Entities Discrete objects, having their own unique number of attributes.

Screens Windows with associated text for model animation and menus.

Files File variables to control reading from and writing to files and logical devices.

Strings String variables for handling text information.

Procedures User written subroutine procedures.

Process Packaged sub-models called like a procedure, but involves the simulation of time in the body of the routine.

Statistics on variables of an Observation or Time Persistent nature are collected automatically by appending key words to the variable declaration. When statistics are declared for arrays; the statistics are collected for each element in the array; accordingly SIMPLE_1 models can collect extensive statistics on model variables.

Block Constructs

SIMPLE_1 employs eight (8) basic block types to define discrete and continuous models. The brevity of language concepts for discrete system modeling is due to the flexibility of the CONDITIONS block. The CONDITIONS block is a fundamental concept of the language. The eight fundamental blocks types are:

ACTIVITY Place entities in an activity for a specified amount of time.

BRANCH Select between alternative pathways in the model.

CONDITIONS Define rules for releasing entities from queues.

CREATE Create entities in the model.

INTEGRATE Define state equations for continuous variables.

KILL Disposal of entities when they are no longer needed.

QUEUE Queue entities until specified CONDITIONS are met.

SET Assign variable values to describe changes in system state.

There are a total of 37 block constructs in the language including the specialized PROCESS and PROCEDURE block types. The block types are summarized in Table One below. One third of the block types are Input-Output type blocks for operations on files and interaction with the video display.

BASIC	**EVENT MONITORING**
ACTIVITY	MONITOR
BRANCH	
CONDITIONS	**FILE I/O**
CREATE	CLOSE
INTEGRATE	OPEN
KILL	READ
QUEUE	WRITE
SET	
FLOW CONTROL	**VIDEO I/O: CHARACTER**
IF-THEN-ELSE	ACCEPT
WHILE-END_WHILE	CHART
PROCEDURE	SCREEN
PROCESS	SHOW
RETURN	
REPORT WRITING	**VIDEO I/O: BIT-MAP**
REPORT	ACCEPT
	LINE
	PLOT
	SHOW_OBJECT
RUN CONTROL	**ENTITY GROUP**
CLEAR	CLONE
RESET	ENTITY_MSG
SHELL	PREEMPT
STOP	REGROUP
	RERANK
	SPLIT
INITIALIZATION	
INIT_GROUP	

Table 1: Summary of SIMPLE_1 Block Types

Entities

Entities are conceptual objects that travel alone or in groups through SIMPLE_1 blocks. Entities are created by name and have their own unique attributes. If one were modeling the assembly of computer mother boards for example, entities with identifiers like: CPU_BOARD and CHIP_SET

might be declared, each with differing attribute requirements. CPU_BOARD can be declared to have one attribute and CHIP_SET entities can have, say, five attributes associated with them. Entities are created by name in SIMPLE_1 models and can be brought together into groups. Entities formed into groups do not lose any of their attributes in SIMPLE_1. Manipulation of entity attributes by name simplifies the reference of entity attributes for entities traveling in groups and tends to improve the documentation aspects of SIMPLE_1 models.

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.

Key Language Elements

The two key block types of SIMPLE_1 are the CONDITIONS block and the INTEGRATE block. The INTEGRATE block is used to define differential equation relationships and the CONDITIONS block controls releasing entities from QUEUE blocks in the DISCRETE section of the model. The CONDITIONS block was developed to provide a highly generalized rule based queue release mechanism. The syntax of the CONDITIONS block specifies a logical condition, or conditions, required to leave a list of queues bound to the CONDITIONS block. When all the queues listed for release by the CONDITIONS block have at least one entity, and all logical relationships are detected to be true, the CONDITIONS block executes a series of releases from the queue(s) to the specified blocks in the model. The CONDITIONS block forms a concise definition of constraints affecting entity flow and assembly of entity groups. In a basic queue/server relationship, a CONDITIONS block is used to associate a specific QUEUE with an ACTIVITY block.

In most situations you start modeling the main processes and add embellishments to capture additional constraints on system operation. In a model of a CPU assembly process for example, one would typically start by modeling the basic production process and add additional detail in stages as required for such complexities as subassembly part logistics and so forth. The assembly aspects of system operation can have a dramatic bearing on the performance of the system and SIMPLE_1 has features especially useful for modeling assembly constraints in models of manufacturing processes. To add in an assembly constraint to a model of the CPU's basic process flow , one would add queue(s) to store the required key part/subassembly entities and augment the conditions block.

Discrete Modeling

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;
```

Figure 1: SIMPLE_1 model of an MM1 Queueing system

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.

Continuous Modeling

The INTEGRATE block type is used to define differential relationships among model variables. The values of integrated variables are obtained numerically using either a fixed or variable step Runge Kutta integration procedure.

A model of a rocket fired vertically illustrates the basics of CONTINUOUS SIMPLE_1 modeling. In this example a rocket is fired vertically with a thrust equivalent of 3500 Kg. The rocket burns propellant at the rate of 20 Kg/second. While burning propellant the rocket is thus losing mass at the rate of 20 Kg/second. The rocket weighs 300 Kg empty. Factors that influence the height attained by the rocket are the drag produced in flight, the amount of fuel loaded into the rocket, etc. The source code for this example model is listed below.

The height versus time attained by the rocket is calculated by integrator number one (Fehlberg procedure). The Runge Kutta Fehlberg method uses the CONTINUOUS segment of the model to sample derivative values for the WT_FUEL, VELOCITY, and HEIGHT state variables. There are four blocks which define the CONTINUOUS segment. The SET block is used to update a variable for tracking the maximum height attained by the rocket. The SET block also defines the drag force for the rocket which is a constant (K) function of the VELOCITY squared. Notice that the square of the velocity is calculated with a call to the SIMPLE_1 absolute value function to sign the drag force by direction of action. The Thrust is set to 0 or 3500 depending upon whether there is still fuel on board the rocket. The function of the maximum function for setting the WT_FUEL variable is to avoid negative intermediate integration step values for fuel quantity.

```
DECLARE;
  GLOBALS:
    VELOCITY INTEGRATED:
    HEIGHT INTEGRATED:
    WT_FUEL INTEGRATED:
    BURNING:  K:  G:  MAX_HEIGHT:  THRUST:  WT_ROCKET:
DRAG;
  STRINGS: temp;
END;
PRERUN;
 SET
  STOP_TIME       := 150: STEP_SIZE       :=  0.25:
  WT_ROCKET       := 300: max_step        :=0.5:
  integrator      :=   1: K := 0.05:   G := 9.81:
  absolute_error :=0.01: relative_error := 0.00001:
  BURNING          := 20:  HEIGHT := 0:  VELOCITY :=
0:
  MAX_HEIGHT      :=  0:  THRUST :=3500;
  { Initial setting of derivatives}
 INTEGRATE WT_FUEL:-BURNING;
 INTEGRATE
    VELOCITY:G*(THRUST-DRAG)/(WT_ROCKET+WT_FUEL)-G;
 INTEGRATE HEIGHT : VELOCITY;
  { Show message and get initial WT_FUEL value}
 SHOW,
   5,4,'INPUT FUEL IN RANGE 500..1600:',0,0,15,0;
 ACCEPT,38,4,WT_FUEL,500,1600;
END;
DISCRETE;
END;
CONTINUOUS;
 SET DRAG:=K*VELOCITY*ABS(VELOCITY);
 INTEGRATE
   WT_FUEL :-BURNING*(THRUST>0);
 INTEGRATE
   VELOCITY:G*(THRUST-DRAG)/(WT_ROCKET+WT_FUEL)-G;
 INTEGRATE
   HEIGHT : VELOCITY;
 SET
   MAX_HEIGHT:=MAX(MAX_HEIGHT,HEIGHT):
   WT_FUEL:=WT_FUEL*(WT_Fuel>0):
   THRUST:=3500*(WT_FUEL>0);
END;
POSTRUN;
  {Show results....}
  SHOW,5,5,'Maximum height :',0,0,13,0;
  SHOW,22,5,MAX_HEIGHT,8,2,12,0;
  SHOW,5,6,'Press enter key when finished',0,0;
  ACCEPT,5,7,temp;
  STOP;
END;
```

Figure 2: SIMPLE_1 Continuous model of a simple rocket

After the set block executes, three INTEGRATE blocks are used to define rate of change for the VELOCITY of the rocket, its HEIGHT and the amount of fuel (WT_FUEL) on board the rocket. The rate of change in VELOCITY for the rocket is a function of the THRUST, DRAG, and mass of the rocket. The mass of the rocket being defined as the sum of the WT_ROCKET and the WT_FUEL. The rate of change for the fuel amount is a function of the burn rate of the rocket motor. When the fuel is exhausted the burn rate, by definition is zero. The last INTEGRATE block establishes the rate of change in height for the rocket which is defined by the VELOCITY of travel.

I/O and Animation:

The language has input and output concepts for both file I/O and screen animation with the screen being updated while the model is running. Block constructs in the language control I/O to the screen, keyboard and to DOS. Screen I/O constructs include mechanisms for writing ASCII characters and numbers coupled with template images. The character and number based display formats of SIMPLE_1 combined with screen generation features of the language form a character based animation capability.

Bit map graphics are supported for CGA, EGA, and VGA equipped systems. A TSR frame grabber is supplied to grab bitmap images created with one of the popular PC graphics programs (Dr. Halo (tm), or DeluxePaint II (tm) etc.). The frame grabber is used to capture the image into the proper format. Graphic icons are "lifted" off of the grabbed artwork image using a menu driven icon grabbing tool. Language constructs allow the modeler to load bit mapped image backgrounds, and load/display graphic icons for animation. Plotting is supported with a LINE drawing and a point DRAW block type in addition to the animation of icons.

In summary, SIMPLE_1 supports file and screen I/O Operations associated with:

1) Character display

- SCREEN activation to display a text background.
- SHOW block to display numeric values on a screen.
- CHART block to display characters on a screen.
- ACCEPT block for reading variable values from the keyboard.

2) Text File

- READ and WRITE blocks for file input/output supporting reading form text files and both writing to or appending information to text files.
- OPEN and CLOSE blocks for managing files during model execution.

3) Bit Mapped

- LOAD_IMAGE to load and display a bit mapped screen image file.
- SHOW_OBJECT to display a graphic ICON.
- PLOT and LINE blocks for bit-mapped plotting of variables.

SIMPLE_1 Object Oriented Programming concepts

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. 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;
             RETURN;
 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. When the RETURN statement is executed the entity group is returned to the calling statement in the model.

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. 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 in a simulation 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 [Dooley et al 1988]. The language has not been limited solely to manufacturing with health care applications reported by Smith. The military logistics features of the language has led to it's usage in graduate studies at the Air Force Institute of Technology School of Logistics as reported by Cooper and Westfall. 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 [1986] 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. 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.

SIMPLE_1 References

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.

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.

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.

Cobbin, Philip, "A Tutorial on the SIMPLE_1 simulation environment", Winter Simulation Conference proceedings, December 1986, Washington D.C. pp 168-177.

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

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.

Cooper, Martha C. and Westfall, Frederick W., "The impact of personal computing technology on the education of logistics managers: A comparison of a military and a civilian institutional approach" Proceedings of The Society of Logistics Engineers twenty third annual symposium-1988, Orlando Florida, pp 408.

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.

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.

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

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.

Sierra Simulations & Software: SIMPLE_1 User's guide and reference manual, 1989.

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.

Starr, Patrick J., Development of a Manufacturing Simulation Model, Final report CDC project 85M106, Mechanical Engineering Department, University of Minnesota, 1988.

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.

Thinnes, Karen, M., "Simulation of Printed Circuit Board Manufacturing" unpublished Masters Thesis, The University of Iowa, 1987.

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

Van Houten, Karen, "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.