The SIMPLE 1 simulation environment

Philip Cobbin
Sierra Simulations & Software
303 Esther Avenue
Campbell, California 95008

Overview

SIMPLE 1 is an integrated modeling environment for interactive simulation using the IBM PC, XT, AT and true compatible microcomputers. SIMPLE 1 has a number of innovative features relative to simulation software and programming languages. The implementation of SIMPLE 1 combines compilation and run time systems into an integrated environment including on-line tutorials, learning modules, and a full screen editor coupled to the compiler and run time system. Errors detected by the compiler or run time system initiate a call to the editor to isolate the error and speed up the edit-compile-debug model development cycle. The language supports application programs and a "tool box" concept whereby models and programs to support simulation project activities can be written in SIMPLE 1 to post process simulation results or for data collection activities. Application programs can be run using RUNSIM which loads and executes SIMPLE 1 models compiled to disk with the commercial version of the software. A built in capability to animate simulation results using a character graphics methodology stresses a simplified approach to model animation. The language supports reading and writing of data to files and devices as standard ASCII text files in addition to animation and keyboard data input capabilities. ASCII text file I/O serves as a straightforward hook to other third party software. SIMPLE 1 is not just a pretty picture: the language support extensive statistics collection capabilities including statistics on individual elements of user defined arrays!

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 easily utilizing built in language features. SIMPLE 1 utilizes a repetitive approach to control to facilitate goal seeking modeling and run length definition based on model behavior. 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 of entities. The entity grouping feature of the language is particularly suited for modeling assembly operations in manufacturing and complex resource management situations.

SIMPLE 1 an integrated modeling environment:

Simulation projects inherently involve the integration of many activities and analysis skills to accomplish study objectives. In addition to the obvious requirement to construct, execute and analyze simulation results: Data collection and analysis, model validation, and convincing decision makers as to the merits of simulation are important issues in simulation that emerging simulation software must address.

SIMPLE 1 has been implemented an integrated modeling environment to facilitate simulation related activities by organizing the software into a set of modules accessed through function key driven menus. Editing of models is accomplished via a full screen text editor coupled to the compiler and run time system error detection routines. The compiler and run time error recovery mechanisms endeavor to return the modeler to the editor and point out the source of the problem to expedite debugging. The text editor also serves as a means for reviewing model reports and editing data files. SIMPLE 1's main environment display is illustrated in Figure 1 and Figure 2 summarizes the unique features of SIMPLE 1.

Figure 1 - SIMPLE 1 main environment display.
Figure 2 Summary of SIMPLE_1 Features:
Information on various aspects of SIMPLE_1 are available through on-line tutorial screens and learning modules. These features facilitate debugging and learning the language quickly. Information on syntax and language elements are available through extensive on-line tutorials. In addition to on-line documentation on language elements, special learning modules written in SIMPLE_1 are accessed via menus to assist beginners in learning the language. The learning modules feature animated simulations to illustrate language concepts. Figure 3 is a "road map" of the simulation environment menu structure.

The language tutorials are organized into groups and can be accessed from the main environment, or from within the editor via a few key strokes using the function keys on the keyboard.

Managing disk files is performed with a function key driven operating system to manage disk directories, active path names and drives, etc. When editing a file one can exit the editor and execute a SIMPLE_1 application program. Application programs can be compiled for simulation or collection and analysis of data via "toolbox" programs written in SIMPLE_1 featuring interactive execution of models with disk or keyboard input of program variables.

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 based animation scheme combined with the run time interrupt facility have been found particularly useful for model verification, and problem isolation.

SIMPLE_1 has no special hardware requirements for graphics adapters or special monitor requirements. The software runs on the IBM PC, XT, AT, and like compatibles such as the AT&T PC 6300 equipped with either a monochrome or color monitor. Avoidance of special hardware requirements was a conscience design constraint with the monochrome IBM PC being used as the baseline machine for implementation.

The text editor is a full screen text editor coupled to the compiler and run time systems. When the compiler or run time system detects an error the editor is called after displaying a descriptive message of the problem encountered. Figure 4 is a sample reproduction of an editor display. From the initial environment the model was loaded into memory and the editor accessed by subsequently pressing the F7 key.

When an error is detected by the SIMPLE_1 compiler or run time system, a message describing the nature of the error is displayed. After displaying the error message SIMPLE_1 returns control to the editor with the cursor initially at the problem area. Once returned to the editor the usual routine is to consult with the on-line tutorials to check syntax or language concepts. Once the error is isolated and fixed in the editor the user re-compiles the revised model. SIMPLE_1's coupling of a full screen text editor with the compiler, run time, and tutorial systems provides an effective mechanism for program development and speeds up the learning process for beginners.
The SIMPLE_1 Simulation Environment

SIMPLE_1 "Toolbox" programs:

The SIMPLE_1 language and environment support development and use of a "toolbox" approach to systems analysis. Programs can be written in SIMPLE_1 to collect and analyze data: real or synthetic data. The learning modules introduced with version 3.0 of the software where written in SIMPLE_1 and illustrate language concepts via animated examples. Examples of tool box/application programs include the histogram analysis program supplied with the software: A generalized stochastic cash flow analysis program described in [7]; and an inventory ordering module for teaching purposes described in [8]. In addition SIMPLE_1 application programs have been run under an expert system shell. The ability of the user to build "toolbox" programs in SIMPLE_1 provides an open ended means of expanding the capabilities of the system. The open ended nature of SIMPLE_1 is a direct consequence of merging simulation language concepts with general purpose programming language concepts common in BASIC, Pascal, C, or FORTRAN.

SIMPLE_1: The Language

SIMPLE_1 employs a number of unique approaches to simulation from a language design point of view. The development of SIMPLE_1 evolved with the intention of providing basic building blocks, or language primitives, to model systems of both a discrete and a continuous nature. A network approach to modeling has been demonstrated to be a highly effective vehicle for describing and documenting models of systems. Accordingly, SIMPLE_1 was designed as a network oriented language with an activity on node orientation. The language integrates modeling concepts with general purpose programming concepts to unify modeling efforts. The language includes concepts common to high level programming languages including string and file variables for ASCII I/O operations to devices and files. In addition to data structure concepts the language implements "C" like side affects whereby block parameters can assign a value to a variable as a side affect of a blocks' execution. A repeatative approach to run control is used so that results from one simulation run can be used to establish parameters for subsequent simulations. For example, the decision to halt the simulation is established by the model and does not require "compiling in" the number of runs, run length etc. common to traditional discrete simulation languages. Models are structured into five segments, one of which is used to declare variables. The other segments of a model describe the discrete and continuous nature of the system and run control aspects of model execution. Table 1 summarizes the block types that make up the SIMPLE_1 language and Table 2 is a listing of the built in functions. The blocks summarized in Table 1 can be used to open and close files, buffer keyboard input and perform discrete/continuous modeling of systems. In addition general purpose concepts like an IF-THEN-ELSE and a WHILE loop construct are included in the language. SIMPLE_1 also contains a number of built in function to perform arithmetic operations and access statistics and internal SIMPLE_1 variables.

SIMPLE_1's repetitive approach to run control employs a PRERUN and POSTRUN model segment to set initial conditions and analyze run results. Figure 5 illustrates SIMPLE_1's approach to running the user's model. The PRERUN section of the model is executed first to establish model parameters and run control limits such as the stopping time for the simulation. After execution of the PRERUN code the DISCRETE and/or CONTINUOUS sections of the model are processed. Using SIMPLE_1's repetitive approach to run control one can look at the results of a simulation to base decisions for parameter values of the next run.

Discrete event aspects of the 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 fourth order fixed step procedure 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.

![Schematic of run control in SIMPLE_1](image)

SIMPLE_1 is a declarative language supporting user defined variables. Variable identifiers can have up to 20 significant characters including the underscore to facilitate self documentation of the model. The language supports the declaration of the following classes of data structures:
1) Globally scoped reals: scalars and arrays
2) Entities: each type having their own unique number of attributes.
3) Screens: Windows for animation backgrounds or menu screens.
4) Files: File variables for reading and writing to files.
5) Strings: scalar and string arrays with individual string size limits.

Statistics on globally scoped 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.

139
The SIMPLE_1 Simulation Environment

**FUNCTION** | **DESCRIPTION** | **FUNCTION** | **DESCRIPTION**
--- | --- | --- | ---
**ARITHMETIC:**
ABS | Absolute Value | MAX | Max of 2 arguments
ARCCOS | Arc Cosine | MIN | Min of 2 arguments
ARCOSIN | Arc Sine | MOD | Modulus operation
ARCTAN | Arc Tangent | ROUND | Round to integer
COS | Cosine | SIN | Sine
EXP | e taken to a power | SORT | Square root
LOG | Base 10 log | TAN | Tangent
LN | Natural log |

**BLOCK STATISTICS**
AVE_NUM | Average Activity level | MIN_NUM | Min activity level
COUNT | Execution count | NUM | Current block usage
MAX_NUM | Max activity level |

**ENTITY GROUPS**
NUM_ENTITY | Number of entities of a given type in current group

**INTEGRATED VARIABLES**
LAST_STATE | Last state value | LAST_DERIV | Last derivative
DERIV | Current Derivative |

**RANDOM NUMBER GENERATORS**
UNIFORM | Uniform distribution | LOGNORMAL | Log normal distr.
NORMAL | Normal distribution | POISSON | Poisson distribution
EXPON | Exponential distribution | SEED | Seed settling function
TRIAG | Triangular distribution | DISC_STEP | Discrete values

**VARIABLE STATISTICS**
OBSERVE_AVE | Average value | TIME_AVE | Time weighted
OBSERVE_MIN | Minimum value | TIME_STD | T. W. std. deviation
OBSERVE_MAX | Maximum value | TIME_MAX | Maximum value
OBSERVE_N | No. of observations. | TIME_MIN | Minimum value
OBSERVE_STD | Standard deviation |

**TIME RELATED**
TIME | Simulation time | SYS_TIME | Hardware time

**RUN CONTROL (pre-declared variables)**
KILL_COUNT | Termination count | STOP_TIM | Stop time
STEP_SIZE | Step size | KEY_PRESSED | Keybd info.

**FILE RELATED**
ECF | Returns End-Of-File status |

Table 2 - Summary of SIMPLE_1 functions & variables

Screens can be declared in SIMPLE_1 which define a character schematic to be used as a background over which animation of the model state is to be performed. One or more screens can be declared in each simulation and activated by SCREEN blocks in the model.

Entities are created by name and have their unique attributes. Entities with identifiers like: CPU_BOARD and CHIP_SET can be declared each with differing attribute requirements. CPU_BOARD can be declared to have one attribute while CHIP_SET entities can have say five attributes associated with them. Entities can be brought together into groups without loss of their individual attributes in SIMPLE_1. The maintenance of entity values is an important feature of the language allowing sophisticated statics gathering on entity movements and resource decision making in complex systems. Manipulation of entity attributes by their unique name simplifies referencing attributes and improves the self documentation of models. When multiple entities of the same type are present in a group the individuals are referenced by the ^ operator. 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.

String variables are typically used for defining input and output file names at run time using keyboard inputs and can also be used for customizing simulation reports. A prototype model generator application program uses string variables to generate SIMPLE_1 source code based on menu responses by a novice modeler. String variables are invaluable for applications work and they are used extensively in the version 3.0 system's learning modules to manipulate text messages and animation screens.

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, screens, and so forth. The PRERUN and POSTRUN sections execute in a basic subroutine like manner much like BASIC or FORTRAN. SIMPLE_1 models typically employ the language's seven (7) 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 which will be described in detail later.

Discrete system models involve construction of networks defining the flow in time of entities. Conceptually, entities are distinct individual objects that flow through blocks in the network model. Typically, entities are used in models to represent real objects: tools, parts, people, and so forth. The network model is used to define the interrelationship between entities and other elements of the system. In the most basic form, network models describe the processes to:

1) CREATE :Groups of entities
2) QUEUE :Entities until specified CONDITIONS are met.
3) ACTIVITY: Activities are undertaken by entities
4) BRANCH: To alternative parts of the model.
5) KILL: Disposal of entities when they are no longer needed.
6) SET: Variable values to update system state or entity attributes.
7) INTEGRATE: Define derivatives for integrated variables.
**CONDITIONS** block: A key language element

The **CONDITIONS** block is used to define the state conditions required for entities to leave queues. The block is a cornerstone concept of the language and provides a unified queue release mechanism. The block functions somewhat analogously to *achameleon*, in that a **CONDITIONS** block can be configured for a diversity of queue release constraints. The schematic representation of the block and example statement are illustrated in Figure 6. 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.

![CONDITIONS block diagram](image)

**EXAMPLE STATEMENT:**

\[
\text{CONDITIONS, NUM(HEAD I) \leq NO(HEAD I) \text{ TOOLS(SHIFT INDEX)),} \\
\text{HEAD I, NUM(HEAD I) = NUM(FINAL I), HEAD I,} \\
\text{SERVO Q, HEAD I,} \\
\text{Secondary blocking constraint on a subset of active tools and status of downstream queue}
\]

Figure 6 **CONDITIONS** block: a unified queue release mechanism

Notably absent in **SIMPLE_1** is the concept of a resource for modeling complicated queue-server relationships. **SIMPLE_1** does not employ resources because its nature, 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 "customer" 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 entity attributes and system state variables.

The example statement in Figure 6 is taken from a model of a typified winchester disk drive assembly process. The three queues: **HEAD I**, **SERVO Q**, and **SERVO Q** are used to store work in process inventory (WIP) and parts inventory. Sub assemblies are contained in the **HEAD I** (each composed of multiple part entities). The sub assembly entity group is added to and the assembly activity is initiated when the **CONDITIONS** block detects all of the required release criteria have been met. The **CONDITIONS** block in this case executes the entity movements only when the upstream queues each have a minimum of one element and both of the boolean expression are true.

The **CONDITIONS** block is an important means for organizing entities into groups. Returning to the passenger-bus analogy; a model of such a system using **SIMPLE_1** organized passenger entities with a vehicle entity into groups like that schematically shown in Figure 7. The decision to release a passenger entity from the group involved checking the list of individual passengers to see if any were to leave at the current stop. The value of a bus attribute in this case contained the groups current location. When passengers were at their final destination, the model **SPLIT** them off and routed them to the exit segment of the model.

![Entities organized into a group for modeling a busline operation](image)

Figure 7 Entities organized into a group for modeling a busline operation

The **CONDITIONS** block 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. For example when modeling product assembly, the 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, shift staffing, and other constraints on system operation can typically be added in stages without a major restructuring of the model.

Run Control Concepts

Four specialized blocks are used to control simulations. A CLEAR block is used to control clearing statistical accumulators and a RESET can be used in the POSTRUN to eliminate all entities in existence in the discrete portion of the model. A standard report on system performance can be obtained using the REPORT block in the POSTRUN. In addition, I/O concepts in the language fully support customized report writing. The key run control block is the STOP block. The STOP block is used in the POSTRUN to halt execution of the model.

A discrete system example

An original GPSS example of a basic TV inspection and adjustment situation will be used to introduce SIMPLE_1. This example, based on Scriber’s SIMULATION using GPSS; 1974) models TV’s arriving by inspecting by one of two available inspectors. After inspection good sets are routed to shipping and defective sets are routed to an adjusting station. The SIMPLE_1 version of the TV inspection and adjustment system is:

```
DECLARE;
  GLOBALS: TIME_IN_SYSTEM OBSERVE_STATS;
  INVENTORY TIME_STATS;
  ENTITIES: TV(1);

  { $ SCREEN:DEF { "--- Include file used to define animation screen " END;}
    PRERUN;

  { $ SCREEN:PRN { "--- Include file used to activate animation screen " END;)
    END;

  DISCRETE;

  { $ ANIMATE:TV { "--- Include file used to animate simulation " CREATE TV, UNIFORM( 3, 5, 7, 1); SET TV(1) := STIME: INVENTORY := INVENTORY +1;

  WAIT: INSPECT QUEUE FIFO;

  CONDITIONS: NUM(NINSPECT) < 2, WAIT: INSPECT, INSPECT;

  INSPECT ACTIVITY UNIFORM(6, 12, 1);

  BRANCH 0.65, PACK: 0.15, WAIT: INSPECT;

  WAIT: ADJ QUEUE FIFO;

  CONDITIONS: NUM(ADJUST) < 1, WAIT: ADJUST, ADJUST;

  ADJUST ACTIVITY UNIFORM(20, 40, 1);

  BRANCH, WAIT: INSPECT;

  PACK SET TIME: IN SYSTEM := STIME - TV(1);

  INVENTORY := INVENTORY +1;

  KILL;

  END;

  CONTINUOUS: END;

  POSTRUN;

  REPORT;

  STOP;

  END;
```

In addition to modeling the basic flow of events for this simple example this model illustrates SIMPLE_1’s unique approach to collection of statistics on user declared variables. The global variable TIME_IN_SYSTEM is declared with the key word OBSERVE_STATS appended to signal collection of statistics. When the SET block near the bottom of the code assigns the value of TIME_IN_SYSTEM with the expression:

```
TIME_IN_SYSTEM := STIME - TV(1)
```

The creation time for the TV and the current simulation time (STIME) are used to calculate the time in the system for the exiting TV. As a side affect of the variable assignment, SIMPLE_1 updates observational statistics for TIME_IN_SYSTEM. In a similar fashion time weighted statistics are maintained for the variable INVENTORY used to track work in process inventory.

The CONDITIONS blocks in this model employ the built in function NUM which returns the current number of entity groups at a block in the model. Built in functions of the language provide access to arithmetic functions, random number generators etc. The CONDITIONS blocks in this case is used to model straight forward queue-server relationships but is readily extendable to handle such complications as:

- Blocking by a downstream QUEUE
- A variable number of servers changing over time
- Mobile resource entities that decide which queues to tend
- Model component assembly to see impact of logistics policies

Animation, Input, & Output

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. SIMPLE_1 supports I/O operations using specialized block constructs. The input and output operations supported in the language are for two types of operations. Block constructs in the language control I/O to the screen or 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 the language combined with screen generation features form a character based animation capability. In summary, SIMPLE_1 supports file and screen I/O. Operations associated with:

1) SCREEN activation to display a text background.
2) SHOW block to display numeric values on a screen.
3) CHART block to display characters on a screen.
4) ACCEPT block for reading variable values from the keyboard.
5) READ and WRITE blocks for file input/output.
6) OPEN and CLOSE files during model execution.

Include files were used in the model to include animation code into the model for clarity. Include files also allow management of various versions of a model to turn on or off compiling of animation code and for management simulation experiments. A screen is used to form a schematic of the TV inspection system over which SHOW and CHART blocks animate the state of the system by writing characters and numbers to the screen.
schematic background is defined in the include file: "SCREEN.DEF" illustrated in Figure 8.

Figure 8 - screen definition for animation of TV repair model.

During the PRERUN phase a SCREEN block contained in the include file: "SCREEN.PRN" (not illustrated) is executed to initialize the character image. The image is updated to show the changing state of the simulation using a series of MONITOR blocks that function as a kind of interrupt in the DISCRETE section of the model. Whenever a variable or labelled block listed in a monitor block changes state the MONITOR is executed. This side-effect/Interrupt mechanism provides an efficient hook for animation and generation of custom trace files to track model execution. The include file: ANIMATE,TV is used to drive the animation of the TV repair model. To illustrate the function of the MONITOR, SHOW, and CHART blocks in animation of models consider what happens when an INSPECT activity is started, or completes. Whenever an entity enters or leaves the INSPECT activity the monitor block:

MONITOR INSPECT;
   CHART .38,13,4,178,NUM (INSPECT),3,12,0;
END_MONITOR;

is executed as a side effect of the event. The MONITOR block in turn executes a CHART block to write ASCII characters onto the monitor to indicate what just happened. The number of ASCII characters written by the CHART block graphically represents the number of busy inspectors in the system.

Conversely, if only numeric information is to be updated a SHOW block would be used to update the screen and a WRITE block for outputting the information to a disk file.

Augmented Example:

A modified version of the above TV repair model illustrates powerful modelling features of the SIMPLER simulation language. For example, assume TVs come in two types, color sets and black and white units. By assigning a TV attribute to signify which type of set is being inspected, the INSPECT activity:

INSPECT ACTIVITY UNIFORM(6,12,1);

Can be reconfigured as an IF block with embedded activities for inspection of color and black and white sets by substitution of the code fragment:

INSPECT IF TV(Type) = Color THEN;
   COLOR ACTIVITY UNIFORM(6,12,1);
   ELSE;
   BW ACTIVITY UNIFORM(3,6,1);
END_IF;

The TVs attribute for set type is reference using the global variable Type (assigned the value of 2 and used to enhance reading of the code). Based on the TVs type the IF block executes either the COLOR or BW ACTIVITY blocks. For this example three utilization statistics are automatically tracked for the time TVs spend in the IF block and for the utilization of the COLOR and BW labelled activity blocks.

Augmenting the TV model to collect observational statistics on composite inspection time for color and black and white sets is straightforward. In the DECLARE section the global variable declaration:

  InspectTime OBSERVE_STATS

would be inserted to declare the variable "InspectTime" and signal the collection of observational statistics on the values assigned to this variable. The collection of the statistic is accomplished by side effect assignment in the COLOR and BW labelled activity blocks: For example the COLOR labelled activity block would be modified to assign the sampled activity duration to the variable InspectTime via the modified statement:

  COLOR ACTIVITY InspectTime := UNIFORM(6,12,1);

The collection of the desired statistic would be accomplished by the assignment of the InspectTime variable as a side effect of sampling an activity duration for the block. In turn the assignment of InspectTime causes updating of statistical accumulators for automatic tracking of observational statistics.
Enhancement of the model to account for differing product inspection times alters the time spent in the system as a function of product type. Accordingly, one would like to differentiate the `TIME_IN_SYSTEM` observational statistics by product type. This final modification to the model is readily implemented by changing the global variable declaration in the `DECLARE` section from:

```
TIME_IN_SYSTEM OBSERVE_STATS;
```

to a subscripted variable (vector) with dimension 3:

```
TIME_IN_SYSTEM(3) OBSERVE_STATS;
```

Then in the `DISCRETE` section the PACK labelled `SET` block:

```
PACK SET TIME_IN_SYSTEM := STIME-TV(1):
INVENTORY:=INVENTORY-1;
```

would be revised to collect the time in system for both TV types as well as a composite statistics for both types with the revised statement:

```
PACK SET TIME_IN_SYSTEM(TV(Type)) := STIME-TV(1):
TIME_IN_SYSTEM(3) := TIME_IN_SYSTEM(TV(Type));
INVENTORY:=INVENTORY-1;
```

This revised `SET` block assigns element (1) or (2) of the variable `TIME_IN_SYSTEM` based on the TVs type and assigns element (3) to the time in the system calculated for both types of TVs. The ability to collect statistics on user defined subscripted variables is a unique and powerful feature of `SIMPLE_1`.

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, and as a tool for scheduling current systems. A number of students throughout the United States have employed the language in support of their graduate work.

`SIMPLE_1` is currently being used to teach Principles of Management at Cal State Long Beach via teams of students trying their skills out with a detailed model of a production system. The language has been employed by Starr et al [11] to investigate schedule recovery strategies and currently the software is being used in applied research for scheduling an electronic assembly system in a mid-western aerospace site. Inspection issues relative to FMS systems was investigated using `SIMPLE_1` according to Hauck [9]. The language has not been restricted to manufacturing usage solely, as evidenced by Bottomly's investigation of logistics support for avionics equipment [1].

**References**


**Authors' Biography**

Philip Cobbin is the owner of Sierra Simulations & Software and is the developer of `SIMPLE_1`. Phil has developed and taught simulation to undergraduates as an adjunct professor of Industrial Engineering at San Jose State University. He holds a Master of Science in Industrial Engineering from Purdue University, a Bachelor of Science in Industrial Engineering and Operations Research from the University of Massachusetts at Amherst, and an Associate in Science degree in Manufacturing Engineering Technology from Waterbury Connecticut State College. Phil is a native of Los Angeles and has been previously employed by the General Products Division of the International Business Machines corporation performing simulation modeling and material handling engineering activities.