Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

THE SIMSCRIPT III PROGRAMMING LANGUAGE FOR MODULAR OBJECT-ORIENTED SIMULATION

Stephen V. Rice

Department of Computer & Information Science
The University of Mississippi
University, MS 38677, U.S.A.

Ana Marjanski

CACI Products Company
1455 Frazee Road, Suite 700
San Diego, CA 92108, U.S.A.

Harry M. Markowitz

1010 Turquoise Street, Suite 245
San Diego, CA 92109, U.S.A.

Stephen M. Bailey

P.O. Box 20812
El Cajon, CA 92021, U.S.A.

ABSTRACT

SIMSCRIPT III is a programming language for discrete-event simulation. It is a major extension of its predecessor, SIMSCRIPT II.5, providing full support for object-oriented programming and modular software development.

1 INTRODUCTION

SIMSCRIPT was among the first programming languages for computer simulation. SIMSCRIPT I was developed by the RAND Corporation for the U.S. Air Force in 1962 (Markowitz, Hausner, and Karr 1963) and was followed by SIMSCRIPT I.5 from CACI in 1965. The SIMSCRIPT II language was also developed by RAND for the U.S. Air Force (Kiviat, Villanueva, and Markowitz 1968). Its successor was SIMSCRIPT II.5, introduced by CACI in 1971.

The SIMSCRIPT II.5 language (CACI 1997) has been enhanced by CACI over the past 30 years, and implementations of the language have been developed for many computing platforms ranging from mainframes to personal computers. SIMSCRIPT II.5 facilitates the programming of simulation models by its:

- *language features*—built-in timing routine and scheduling mechanism, sets, random-number generation, and statistics gathering;
- *language syntax*—English-like self-documenting syntax to facilitate the communication and verification of simulation models;
- *support libraries*—for animation, graph generation, graphical user interfaces, and database access;
- *integrated development environment*—for program editing, creating graphical elements, and automatic project building.

The SIMSCRIPT III programming language is the successor to SIMSCRIPT II.5. It is a superset of SIMSCRIPT II.5, with significant new features for object orientation and modularity.

The object-oriented paradigm provides a powerful and elegant way to represent real-world objects in a simulation program. In fact, the need to represent real-world objects in a simulation program inspired the development of the first object-oriented programming language, SIMULA, in the 1960s (Nygaard and Dahl 1978).

Despite the popularity of the object-oriented approach and its applicability to simulation programming, there are few object-oriented simulation programming languages. SIMULA, ROSS, and MODSIM are examples. RAND developed the ROSS language, based on Lisp, for the U.S. Air Force in the early 1980s (McArthur, Klahr, and Narain 1984). CACI developed the MODSIM language, based on Modula-2, for the U.S. Army in the late 1980s (CACI 1996).

SIMSCRIPT III is a new object-oriented simulation programming language based on the venerable SIMSCRIPT II.5. Its object-oriented features were influenced by C++, Eiffel, Java, MODSIM, and SIMULA. SIMSCRIPT III is also a general-purpose object-oriented programming language.

This paper introduces the SIMSCRIPT III language to readers unfamiliar with SIMSCRIPT II.5. A previous paper by the authors focuses on how SIMSCRIPT II.5 was extended to create SIMSCRIPT III (Rice et al. 2004).

2 LANGUAGE BASICS

A SIMSCRIPT III main module consists of a block of declarations known as the "preamble," followed by one or more routines, one of which is named **main**. Declarations in the preamble are "global," i.e., they apply to every routine in the module. Declarations within a routine are "local," i.e., they apply only to the routine in which they are declared. Program execution begins with the first statement in **main** and continues until **main** returns or a **stop** statement is executed.

Programmer-defined names and language keywords are case insensitive. A programmer-defined name is a sequence of letters, digits, periods, dollar signs, and underscores. Except for **and**, there are no reserved words. A numeric constant is a sequence of digits with an optional period (i.e., decimal point) and optional scientific notation. A comment begins with consecutive apostrophes `''` and continues to the end of the line or until a second pair of apostrophes is encountered.

2.1 Basic Data Types

There are several basic data types, called "modes" in SIMSCRIPT parlance: **integer**, **real**, **double**, **alpha**, **text**, and **pointer**. **Integer** is implemented on most platforms as a signed 32-bit value. **Real** and **double** are single- and double-precision floating-point values, respectively. **Alpha** holds one 8-bit character; an **alpha** constant is surrounded by quotation marks, e.g., `"B"`. **Text** is a dynamic string holding a sequence of zero or more characters; a **text** constant is also surrounded by quotation marks: `"Hello, world!"`. Built-in functions are available for string operations and type conversions. **Pointer** is a generic (untyped) reference value, usually implemented as a 32-bit address.

2.2 Variables and Arrays

An **integer** variable named `X` is declared by the following statement: `define X as an integer variable`. If the statement is specified in the preamble, the variable is global; if specified within a routine, the variable is local to the routine. All variables are automatically initialized to zero, except **text** variables which are initialized to the zero-length string `""`.

A one-dimensional **double** array named `Y` is declared by: `define Y as a 1-dimensional double array`. An array is dynamically allocated, and its number of elements determined at run time, by executing a **reserve** statement, e.g., `reserve Y as 100`. The number of elements in an array can be obtained by calling the built-in function `dim.f`; for example, `dim.f(Y)` returns 100. The first element of the array is stored at index 1. The elements of `Y` therefore are `Y(1)`, `Y(2)`, …, `Y(100)`. Each element is automatically initialized to zero. Multi-dimensional arrays may also be declared. The **release** statement de-allocates an array, i.e., frees its storage.

2.3 Expressions

Arithmetic expressions may use any combination of arithmetic operators: unary `+` and `-`; binary `+`, `-`, `*`, `/`, and `**` (exponentiation). Built-in functions may be called to perform other arithmetic operations, including logarithms, modulus, square root, and trigonometric functions.

Logical expressions may use relational operators, `=`, `<>`, `<`, `<=`, `>`, `>=`, and logical operators **and** and **or**. Logical negation is specified by appending **is false** to a logical expression. The expression `J >= 1 and J <= dim.f(Y)` may be abbreviated as `1 <= J <= dim.f(Y)`. Logical expressions use "short-circuit" evaluation; that is, if the first operand of **and** evaluates to false, or the first operand of **or** evaluates to true, the second operand is not evaluated.

2.4 Basic Statements

Multiple statements may appear on one line, and one statement may span multiple lines. A semicolon is not required or allowed after a statement.

The following statement assigns the value 10 to the variable named `X`: `let X = 10`. However, the **let** keyword is optional and may be omitted: `X = 10`. The statement, `add 1 to X`, is equivalent to `X = X + 1`. Likewise, `X` may be decremented by `subtract 1 from X`.

The **read** statement reads free-form and formatted input. The **write** and **print** statements produce formatted output.

The **if** statement specifies a logical expression followed by a sequence of statements to execute if the expression is true, and optionally by **else** and a sequence of statements to execute if the expression is false. It is terminated by the keyword **always**. For example:

```
define J as an integer variable

read J

if 1 <= J <= dim.f(Y)
   write Y(J) as "The value is ", d(7,2), /
else ''invalid entry
   write as "The index is out of bounds!", /
always
```

The **select** statement is a "case" statement in which one of several blocks of statements is chosen for execution based on the value of an expression.

2.5 Loops

A loop is specified by one or more control phrases followed by the body of the loop, which is either a single statement or a sequence of statements between the keywords **do** and **loop**. A **for** phrase causes the body of the loop to be executed once for each value assigned to a control variable, for example, `for J = 1 to N`. A **while** (or **until**) phrase specifies a logical expression and terminates the loop when the expression is false (or true). A **with** (or **unless**) phrase specifies a logical expression and executes the body of the loop for the current iteration when the expression is true (or false). These phrases may be combined to control loop execution. In addition, **leave** and **cycle** statements may be specified in the body of the loop: a **leave** statement terminates the loop, and a **cycle** statement terminates the current iteration of the loop.

A **find** or **compute** statement may be specified in the body of a loop. A **find** statement terminates the loop when the body is executed for the first time and is followed by an **if found** (or **if none**) phrase which evaluates to true if the body of the loop was (or was not) executed. For each execution of the body of the loop, a **compute** statement evaluates an arithmetic expression and computes statistics (e.g., sum, mean, maximum, minimum) from the values of the expression over the life of the loop.

2.6 Functions and Subroutines

We shall distinguish a *function*, which is a routine that returns a function result, from a *subroutine*, which does not return a function result. Functions and subroutines may have one or more *given* arguments; however, only subroutines may have *yielded* arguments. The value of a given argument is an input to the routine, whereas the value of a yielded argument is an output from the routine. In the terminology of programming languages, given arguments are passed *by value* and yielded arguments are passed *by result* (Louden 2003). **Main** is a special subroutine with no arguments.

Each function and subroutine is declared by a **define** statement in the preamble, which specifies the mode of arguments, and the mode of the function result for functions. To call a function with *n* given arguments, the function name is followed by a parenthesized list of *n* expressions. For example, `F(2, I+1, J)`, invokes the function named `F` with three given arguments. A subroutine is invoked by a **call** statement, for example, `call Analyze given A, B yielding C, D`. Recursion is allowed. A function is terminated by a **return with** statement, which specifies the function result. A subroutine terminates when a **return** statement is executed or the end of the subroutine is reached.

The following function has three given arguments: a one-dimensional array of **text** values, a **text** key to look up in the array, and a **text** value describing the order of values in the array. The function searches for the key in the array. If it is found, the index of the array element containing the key is returned; otherwise, zero is returned to indicate that the key was not found. If the third argument is `"ascending"`, the function uses binary search; otherwise, the array is searched sequentially.

```
function Search(T, Key, Order)

   define First, Last, and Index
      as integer variables

   First = 1
   Last  = dim.f(T)

   if Order = "ascending"

      ''binary search
      Index = (First + Last) / 2
      while First <= Last and Key <> T(Index)
      do
         if Key < T(Index)
            Last  = Index - 1
         else
            First = Index + 1
         always
         Index = (First + Last) / 2
      loop
      if First > Last
         Index = 0 ''not found
      always

   else ''sequential search

      for Index = First to Last
      with Key = T(Index)
         find the first case
      if none
         Index = 0 ''not found
      always

   always

   return with Index

end
```

The function must be declared in the preamble:

```
define Search as an integer function
   given a 1-dimensional text argument
     and 2 text arguments
```

The following is an example of a function call:

```
if Search(A, "Jim", "ascending") > 0
   write as "Found Jim in array A", /
always
```

3 CLASSES AND OBJECTS

A class is defined in a preamble by a **begin class** block, which specifies the name of the class and declares the attributes, methods, and sets of the class. An instance of a class, called an *object*, is identified by its *reference value* which is stored in a *reference variable*. The mode of a reference variable is denoted by the name of the class followed by the keyword **reference**. The **create** statement allocates an object, initializes its attributes to zero, and stores its reference value in the specified reference variable. The **destroy** statement de-allocates the object identified by the specified reference variable. For example:

```
begin class Vehicle
   ''class declarations go here
   ...
end
...

define Car as a Vehicle reference variable

   ''allocate a Vehicle object and store its
   ''reference value in the reference
   ''variable named Car
create Car

   ''de-allocate the object
destroy Car

   ''create an array of Vehicle objects
define Fleet
   as a 1-dimensional Vehicle reference array
reserve Fleet as N
for J = 1 to N
   create Fleet(J)
```

3.1 Attributes

The fields of an object are called *object attributes* and are declared by **every** statements in a **begin class** block. **Define** statements declare the modes of the attributes. For example:

```
begin class Vehicle

   every Vehicle
      has a Manufacturer,
          a Maximum_Speed,
      and a Current_Speed

   define Manufacturer as a text variable
   define Maximum_Speed and Current_Speed
      as double variables

end
```

An object attribute is accessed by specifying its name followed by a parenthesized reference variable:

```
Manufacturer(Car)  = "Chrysler"
Maximum_Speed(Car) = 100.0
Current_Speed(Car) = Maximum_Speed(Car) / 2
```

These statements are read as "the manufacturer of Car is Chrysler," "the maximum speed of Car is 100," and "the current speed of Car is half of its maximum speed."

Fields that are associated with the class, and not with each object of the class, are called *class attributes*. They are declared by **the class** statements in a **begin class** block. For example:

```
begin class Vehicle

   the class
      has a Count

   define Count as an integer variable

end
```

Within a method of the class, a class attribute may be accessed using its unqualified name, e.g., `Count`. Otherwise, the attribute name must be qualified by the name of the class, e.g., `Vehicle'Count`.

An attribute may be a scalar or array, and may be a reference variable.

3.2 Methods

The routines of an object are called *object methods* and are declared by **every** statements in a **begin class** block. A **define** statement declares the mode of a method's arguments, and the mode of the function result if the method is a function. If the **define** statement is omitted, then the method is assumed to be a subroutine with no arguments. A method specified in an **after creating** statement is called automatically after each object of the class is created. Likewise, a method specified in a **before destroying** statement is called automatically before each object of the class is destroyed. For example:

```
begin class Vehicle

   every Vehicle
      has a Construct method,
          a Destruct method,
      and a Status method

   after creating a Vehicle, call Construct
   before destroying a Vehicle, call Destruct

   define Status as a text method
      given an integer argument

end
```

In the implementation of a method, the name of the method must be qualified (e.g., `Vehicle'Status`) unless it follows a **methods** heading identifying its class. In the following example, we use the `Construct` and `Destruct` methods to update the class attribute named `Count` to hold the current number of `Vehicle` objects. The `Status`

method accepts a speed limit argument and returns a text description of the vehicle's status.

```
methods for the Vehicle class

method Construct
   add 1 to Count
end

method Destruct
   subtract 1 from Count
end

method Status(Speed_Limit)
   define S as a text variable
   if Current_Speed = 0
      S = "stopped"
   else
      if Current_Speed > Speed_Limit
         S = "speeding"
      else
         S = "traveling"
      always
   always
   return with S
end
```

An object method is called by following its name with a parenthesized reference variable and then its arguments, if any. For example:

```
if Status(Car)(45) = "speeding"
   Current_Speed(Car) = 45 ''slow down
always
```

The reference value is passed by value to the method and is accessible within the method as a local reference variable having the same name as the class—`Vehicle` in our example. (This variable is called `self`, `this`, or `current` in other object-oriented languages.) This variable is used implicitly as the reference variable when accessing object attributes and calling object methods without an explicit reference variable. For example, in the `Status` method above, `Current_Speed` is interpreted as `Current_Speed(Vehicle)`.

Routines that are associated with the class, and not with an object of the class, are called *class methods*. They are declared by **the class** statements in a **begin class** block, for example, `the class has a Speedometer method`. A class method is invoked without a reference variable.

3.3 Sets

A *set* is a doubly-linked list with a programmer-defined name. The *owner* of a set of objects named `List` has three *owner attributes*: reference variables `f.List` and `l.List`, which identify the first and last objects in the set, and `n.List`, which holds the number of objects in the set. Each *member* of this set has three *member attributes*: reference variables `p.List` and `s.List`, which identify the predecessor and successor objects in the set, and `m.List`, which indicates whether this object currently belongs to a set named `List`.

An object may own and belong to any number of sets. Each **belongs** phrase in an **every** statement names a set in which an object may be a member. Each **owns** phrase in an **every** statement names a set owned by an object. An **owns** phrase in **the class** statement names a set owned by the class. The set named in an **owns** phrase is qualified by the name of the member class. It is possible to own an array of sets.

A **file** statement inserts an object into a set. Variations of this statement permit the object to be inserted first or last in the set, or immediately before or after a specified object. If the position is unspecified, the object is placed into the set according to the "set discipline," which may be FIFO, LIFO, or "ranked," i.e., ordered according to attribute values of the members. The set discipline is declared by a **define** statement in the **begin class** block of the member class and is FIFO by default.

A **remove** statement removes an object from a set. Variations of this statement remove the first or last object, or a specific object from the set. A **for each** loop control phrase traverses a set in the forward or reverse direction, executing the body of the loop once for each member of the set. Special logical expressions test whether an object is in a set and whether a set is empty. For example:

```
begin class Vehicle

   every Vehicle
      belongs to a Service_Queue

   define Service_Queue as a FIFO set

end

begin class Repair_Shop

   every Repair_Shop
      owns a Vehicle'Service_Queue

end
...

define EZ_Auto and Ferrari_Depot
   as Repair_Shop reference variables
...

for each Car in Service_Queue(EZ_Auto)
with Manufacturer(Car) = "Ferrari"
do
   remove Car from Service_Queue(EZ_Auto)
   file Car in Service_Queue(Ferrari_Depot)
loop

if Service_Queue(EZ_Auto) is empty
   write as "Time for a coffee break", /
always
```

3.4 Inheritance

A *child* class may be derived from one or more *parent* classes, which are specified using the **is a** phrase of the **every** statement in the child class declaration. The child class *inherits* all of the attributes, methods, and sets of its parent classes. In addition, the child class may declare attributes, methods, and sets of its own. A child class may *override* any inherited object method, providing its own implementation of the method, which may invoke the overridden implementation.

A child class is a specialization of its parent classes. In our example, we derive a `Gas_Vehicle` class from the `Vehicle` class. Each `Gas_Vehicle` object thereby acquires the attributes and methods of a `Vehicle` (`Manufacturer`, `Maximum_Speed`, `Current_Speed`, `Construct`, `Destruct`, and `Status`) and may be a member of a `Service_Queue` set. We declare new attributes `Current_Gallons` and `Tank_Capacity`, and a new method `Gas_Gauge` that returns the current fuel level. We override the `Status` method to append the current reading of the gas gauge to the status message. The implementation of the `Status` method calls the built-in function `int.f`, which rounds its floating-point argument to the nearest integer.

```
begin class Gas_Vehicle

   every Gas_Vehicle
      is  a Vehicle,
      has a Current_Gallons,
          a Tank_Capacity,
      and a Gas_Gauge method, and
      overrides the Status

   define Current_Gallons and Tank_Capacity
      as double variables
   define Gas_Gauge as a double method

end
...

methods for the Gas_Vehicle class

method Gas_Gauge
   return with
      Current_Gallons / Tank_Capacity
end

method Status(Speed_Limit)
   define S as a text variable

   select case int.f(4 * Gas_Gauge)
      case 4  S = "full"
      case 3  S = "3/4 full"
      case 2  S = "1/2 full"
      case 1  S = "1/4 full"
      case 0  S = "empty"
   endselect

   return with
      Vehicle'Status(Speed_Limit) +
      ", gas tank is " + S
end
```

The reference value of a child object may be assigned to a reference variable of any of its parent classes. This allows specialized objects to be treated more generally. In our example, a `Vehicle` reference variable may contain a `Vehicle` reference value or a `Gas_Vehicle` reference value. If the `Status` method is called using this variable, `Vehicle'Status` is invoked for a `Vehicle` object and `Gas_Vehicle'Status` is invoked for a `Gas_Vehicle` object.

4 SIMULATION FEATURES

4.1 Random-Number Generation

SIMSCRIPT III utilizes a linear congruential generator (LCG) to produce uniform pseudo-random 31-bit values ranging from zero to 2,147,483,647. A predefined array named `seed.v` contains ten seed values equally spaced throughout the period of the LCG; however, any seed values may be assigned by the program to this array. A "stream" number between 1 and 10 selects a seed value from this array.

The values from the LCG are transformed by built-in functions into pseudo-random numbers from the following probability distributions: beta, binomial, Erlang, exponential, gamma, lognormal, normal, Poisson, triangular, uniform (continuous and discrete), and Weibull.

4.2 Process Methods

A *process method* is a subroutine that can be executed immediately by calling it (using the **call** statement) or can be executed at some future simulation time by scheduling it (using the **schedule** statement). An attribute is implicitly defined having the same name as the process method. This attribute is an object attribute if the process method is an object method and is a class attribute if the process method is a class method. A process method that is an object method is invoked on behalf of an object and can be thought of as describing an activity of the object. For example:

```
begin class Vehicle

   every Vehicle
      has a Trip process method

   define Trip as a process method
      given ''miles to travel and
            ''average speed in miles per hour
         2 double arguments
      yielding ''duration of trip in hours
         1 double argument

end
```

The **schedule a** statement creates a *process notice* and inserts it into the future-events set. The `time.a` attribute of the process notice is assigned the simulation time at which

the process method is to begin execution. The units of simulation time may be defined by the programmer; by default, one unit of simulation time is equal to one day.

The global variable named `time.v` contains the current simulation time and is initially zero. The **start simulation** statement passes control to the built-in timing routine. While the future-events set is not empty, the timing routine removes the process notice with the smallest `time.a` value from the future-events set, updates `time.v` to the value of `time.a`, and calls the corresponding process method.

In the following examples, a 200-mile car trip is scheduled with an average speed of 50 miles per hour. We can start the trip now,

```
schedule a Trip(Car) given 200, 50 now
```

or we might start the trip two days from now:

```
schedule a Trip(Car) given 200, 50 in 2 days
```

The **schedule a** statement assigns the reference value of a newly-created process notice to the object attribute `Trip(Car)`. This process notice may later be referred to in a **cancel** statement, which removes the process notice from the future-events set to cancel the pending execution of the process method. It may be rescheduled using the **schedule the** statement, which puts the process notice back into the future-events set. For example, to reschedule the trip for next week:

```
cancel the Trip(Car)
schedule the Trip(Car) in 7 days
```

The routine that executes a **schedule** statement continues on without waiting for the process method to begin executing. Eventually control passes to the timing routine, which executes the process method when it becomes the most imminent event. However, by using a **call** statement, a routine can execute a process method immediately and wait for it to complete before continuing on. For example:

```
call Trip(Car) given 200, 50
   yielding Trip_Duration
```

A routine that calls a process method receives the values yielded by the process method, if any. These values are discarded when the process method is invoked through the scheduling mechanism; however, the method may save these values in attributes for other routines to access.

A process method, or a routine called by a process method, may suspend its execution using a **wait** or **suspend** statement, pass control back to the timing routine, and later resume its execution, not at the beginning of the routine, but immediately following the **wait** or **suspend** statement. The **wait** statement inserts the process notice of the suspended routine into the future-events set to schedule its resumption. The **suspend** statement does not schedule resumption; another routine must execute a **schedule the** statement referring to the process notice of the suspended routine to schedule its resumption.

In the following example implementation, the process method `Trip` randomly generates an average speed that is the given average speed plus or minus five miles per hour, and computes the duration of the trip. It then executes a **wait** statement, which suspends execution of the process method, schedules its resumption after the duration has elapsed, and passes control back to the timing routine. Upon resumption, the actual duration of the trip is computed and returned to the caller in the yielded argument.

```
methods for the Vehicle class

process method Trip
   given Distance, Avg_Speed
   yielding Actual_Duration

   define Duration, Start_Time
      as double variables

   Duration = Distance /
      uniform.f(Avg_Speed-5, Avg_Speed+5, 1)

   Start_Time = time.v

   wait Duration hours

   Actual_Duration =
      (time.v - Start_Time) * hours.v

end
```

The **wait** statement places the process notice in the future-events set. Another routine may refer to this process notice in an **interrupt** statement to remove it from the future-events set; however, the remaining waiting time is saved in a process notice attribute. A routine may later refer to this process notice in a **resume** statement to insert it back into the future-events set, scheduling the resumption of execution to occur after the remaining waiting time has elapsed.

In our example, `Actual_Duration` will be greater than `Duration` if the trip is interrupted. Perhaps we are modeling mechanical breakdowns and repairs, and for a `Gas_Vehicle`, stops at gas stations.

```
interrupt the Trip(Car)

... ''simulation time elapses

resume the Trip(Car)
```

4.3 Statistics

An **accumulate** or **tally** statement specifies one or more statistics to compute automatically from the values assigned to an object attribute (or class attribute). A name is given to each statistic, and an object method (or class method) by that name is generated that returns the value of the statistic. Any of the following statistics may be computed: the maximum, minimum, number, sum, mean, mean square, sum of squares, variance, and standard deviation of the values assigned to the attribute. A histogram of the values may also be computed.

The statistics are weighted by simulation time if specified by an **accumulate** statement and are unweighted if the **tally** statement is used. The statistics can be computed for the entire simulation, or for particular time intervals, for example, every day or every week of simulation time. The **reset** statement is used to initialize the statistics at the beginning of a time interval.

Suppose in our example we wish to measure how well a repair shop is doing its job, and assume that after each vehicle is serviced, the time required to service the vehicle is assigned to an object attribute named `Service_Time`. A **tally** statement specifies that the average and maximum service time is to be computed from the values assigned to this attribute. An **accumulate** statement indicates that the time-weighted average of the length of the service queue is to be computed. The number of vehicles in the queue is maintained in the implicitly-defined object attribute named `n.Service_Queue`, which is automatically updated whenever a vehicle is inserted into the queue by a **file** statement or removed from the queue by a **remove** statement. `A Print_Statistics` method displays the results.

```
begin class Repair_Shop

   every Repair_Shop
      has  a Service_Time and
           a Print_Statistics method, and
      owns a Vehicle'Service_Queue

   define Service_Time as a double variable

   tally Avg_Service_Time as the mean and
         Max_Service_Time as the maximum
      of Service_Time

   accumulate Avg_Queue_Length as the mean
      of n.Service_Queue

end
```

```
methods for the Repair_Shop class

method Print_Statistics
   print 3 lines with
      Avg_Service_Time, Max_Service_Time, and
      Avg_Queue_Length as follows
   Average service time is **.**
   Maximum service time is **.**
   Average queue length is **.**
end
```

5 SUBSYSTEMS

A SIMSCRIPT III main module may utilize one or more subordinate modules called *subsystems*. Each subsystem is compiled separately and may be used by one or many SIMSCRIPT III programs. It is easier to develop and maintain a large program that has been divided into meaningful units. Subsystems promote better source code organization and facilitate the re-use of code.

A subsystem consists of a public preamble, an optional private preamble, and zero or more routines. The public preamble contains declarations that apply to the rest of the subsystem. More importantly, main modules and other subsystems may access these public declarations by *importing* the subsystem. Main modules and subsystems may import any number of subsystems.

The public preamble is typically used to declare the interface to public classes, which includes the public attributes, methods, and sets defined and inherited by the class. The private preamble declares private classes, and the private attributes, methods, and sets of public classes. These private declarations are visible only to the routines of the subsystem. The source code of the routines is also private.

A subsystem may be distributed as a source file containing only the public preamble, and one or more binary object files obtained by compiling the subsystem. The source file documents the subsystem interface and is read by the compiler when compiling a main module or subsystem that imports this subsystem. An executable program is built by linking the binary object files that were produced by compiling the main module and each of its subsystems.

Each subsystem has a name which is used to qualify the name of each class declared by the subsystem. For example, if a main module imports subsystems named `Army` and `Navy` and both subsystems declare a `Vehicle` class, then the main module can distinguish them by their qualified names, `Army:Vehicle` and `Navy:Vehicle`. The name `Army:Vehicle'Position` refers to an attribute, method, or set named `Position` of the `Army:Vehicle` class.

A subsystem may contain a special **initialize** routine which is called once before the **main** routine is executed, and is used to initialize class attributes and global variables declared by the subsystem.

The following is an outline of a main module that imports a subsystem named `Transportation`:

```
preamble
   importing the Transportation subsystem

   ''preamble declarations go here
   ...
end

''routines of the main module go here
...

main
   ''logic of the main routine goes here
   ...
end
```

The following is an outline of the subsystem named `Transportation`:

```
public preamble
   for the Transportation subsystem

   begin class Vehicle
      ''public attributes, methods, and sets
      ''are declared here
      ...
   end

   ''other public declarations go here
   ...
end

private preamble
   for the Transportation subsystem

   begin class Vehicle
      ''private attributes, methods, and sets
      ''are declared here
      ...
   end

   ''other private declarations go here
   ...
end

''routines of the subsystem go here
...

initialize
   ''logic of the initialize routine
   ''goes here
   ...
end
```

6 CONCLUSION

For more than forty years, the SIMSCRIPT language has been a valuable tool for programming simulation models. SIMSCRIPT III is the latest version of the language. It is a superset of the previous version, SIMSCRIPT II.5. Every valid SIMSCRIPT II.5 program is a valid SIMSCRIPT III main module. SIMSCRIPT III inherits from SIMSCRIPT II.5 an expressive syntax and a rich collection of data types, built-in functions, loop constructs, and executable statements. The new features in SIMSCRIPT III interoperate with the inherited features.

SIMSCRIPT III introduces classes and objects and their attributes and methods, using syntax that is similar to that used by SIMSCRIPT II.5 for entities, attributes, and routines. A SIMSCRIPT III process method acts as both a method and a SIMSCRIPT II.5 process. Sets of objects in SIMSCRIPT III are similar to sets of entities in SIMSCRIPT II.5. The SIMSCRIPT II.5 statistics-gathering feature is applied to attributes of objects and classes in SIMSCRIPT III.

All global declarations in a SIMSCRIPT II.5 program reside in a single preamble that is shared by every routine of the program. Through the introduction of subsystems, SIMSCRIPT III enables large programs to be divided into modules. The public preamble specifies the module interface and the implementation details are hidden. A program may use many modules and a module may be used by many programs.

The SIMSCRIPT III programming language is supported by libraries for graphics, animation, and database access, and by the "Simstudio" integrated development environment.

REFERENCES

CACI Products Company. 1996. *MODSIM III: The language for object-oriented programming*. La Jolla, California: CACI Products Company.

CACI Products Company. 1997. *SIMSCRIPT II.5 reference handbook*. La Jolla, California: CACI Products Company.

Kiviat, P. J., R. Villanueva, and H. M. Markowitz. 1968. *The SIMSCRIPT II programming language*. Englewood Cliffs, New Jersey: Prentice Hall.

Louden, K. C. 2003. *Programming languages: Principles and practice*. Pacific Grove, California: Brooks/Cole.

Markowitz, H. M., B. Hausner, and H. W. Karr. 1963. *SIMSCRIPT: A simulation programming language*. Englewood Cliffs, New Jersey: Prentice Hall.

McArthur, D., P. Klahr, and S. Narain. 1984. ROSS: An object-oriented language for constructing simulations. Technical Report No. R-3160-AF, RAND Corporation, Santa Monica, California.

Nygaard, K., and O.-J. Dahl. 1978. The development of the SIMULA languages. *ACM SIGPLAN Notices* 13 (8): 245–272.

Rice, S. V., A. Marjanski, H. M. Markowitz, and S. M. Bailey. 2004. Object-oriented SIMSCRIPT. In *Proceedings of the 37th Annual Simulation Symposium*, 178–186. Los Alamitos, California: IEEE Computer Society.

AUTHOR BIOGRAPHIES

STEPHEN V. RICE is an assistant professor in the Department of Computer and Information Science at the University of Mississippi. He is the lead designer of the SIMSCRIPT III extensions to SIMSCRIPT II.5. Previously he designed and implemented SIMSCRIPT II.5 Database Connectivity, which gives SIMSCRIPT programs the ability to access relational databases. In the late 1980s, he co-invented the MODSIM object-oriented simulation programming language and wrote the first MODSIM compiler. His research interests also include pattern recognition and audio retrieval. His e-mail address is rice@cs.olemiss.edu and his Web address is www.cs.olemiss.edu/~rice.

ANA MARJANSKI is Head of the Technical Team at CACI Products Company. She is leading the development of SIMSCRIPT III and the MODSIM III-to-SIMSCRIPT III language converter. For the past 15 years, she has been Technical Manager for SIMSCRIPT II.5. Prior to joining CACI, she was at research Institute Mihailo Pupin, Belgrade, where she developed Operating Systems for specialized computer systems, including a Kernel for Object-Based Real-Time Simulation, and led several international software projects at Philips Sweden and Ferranti England. Her e-mail address is amarjanski@caci.com and her Web address is www.caciasl.com.

HARRY M. MARKOWITZ has applied computer and mathematical techniques to various practical decision making areas. In an article in 1952 and a book in 1959, he presented what is now referred to as MPT, "modern portfolio theory." This has become a standard topic in college courses and texts on investments, and is widely used by institutional investors for asset allocation, risk control, and attribution analysis. He developed "sparse matrix" techniques for solving very large mathematical optimization problems. These techniques are now standard in production software for optimization programs. He also designed and supervised the development of the SIMSCRIPT programming language. In 1989 he received The John von Neumann Award from the Operations Research Society of America for his work on portfolio theory, sparse matrix techniques and SIMSCRIPT. In 1990 he shared The Nobel Prize in Economics for his work on portfolio theory.

STEPHEN M. BAILEY is the lead implementor of SIMSCRIPT III. He has worked on SIMSCRIPT II.5, MODSIM II, and MODSIM III, and was a lead implementor of the SIMGRAPHICS II object-oriented graphics package used by both SIMSCRIPT and MODSIM. He developed audio-related technology for the *FindSounds Palette* software and the *FindSounds.com* Web search engine.