Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

MODSIM II - A MODULAR, OBJECT-ORIENTED LANGUAGE

Ron Belanger

CACI Products Company
3344 N. Torrey Pines Ct.
La Jolla, California 92037

ABSTRACT

MODSIM II is an object-oriented, general purpose programming language which was designed to work with both sequential and parallel processors.

It is a compiled language which is available for most systems including mainframes, work-stations and PC's. A research version has been developed for use on parallel processors.

The syntax and structure of MODSIM II is based on that of Modula-2. It has additional constructs for object types and simulation.

The built-in object-oriented constructs of MODSIM II include single and multiple inheritance, dynamic binding of objects, polymorphism, data abstraction and information hiding. These capabilities are reinforced by the modular program development environment.

MODSIM II supports automated separate compilation and importation of code from modules and libraries. This makes it ideal for large projects.

Its built-in interactive, dynamic graphics allow convenient display and manipulation of menus, charts, graphs and animated results. The graphics editor allows users to interactively design their own input and control menus, icons and charts without programming.

1. INTRODUCTION

MODSIM II was specifically designed to support large programming projects. It is a compiled, modular, object-oriented language with multiple inheritance. To protect the user's investment in applications, MODSIM can be moved to new computer systems as they become available.

Its syntax is based on that of Modula-2. Programmers trained in Pascal, Modula-2 or Ada learn the language with ease. This saves training costs and time.

Modularity in MODSIM II improves reliability and code reusability. Objects and routines performing related functions can be grouped into modules. These can be put into libraries for reuse by other programs.

The optional simulation constructs are based on CACI's widely used SIMSCRIPT II.5 programming language.

The portability of MODSIM II derives from the fact that its compiler emits "C" code which is compiled, in turn, by each computer's "C" compiler.

Finally, the integrated dynamic graphics of MODSIM II substantially reduces the time and effort needed to display results with animation and presentation graphics. It only takes a few statements to make dynamic icons, histograms, clocks and meters appear and change as the simulation runs.

MODSIM II is a complete, general purpose programming language which is ideal for large software engineering projects. Its features reduce design and coding effort, and improve reliability. The MODSIM II compiler, runtime and graphics libraries are all written in MODSIM II.

2. OBJECTS

An object is essentially an encapsulation of data and code. The data describes the object's current status. The code describes what the object does.

As an example of an object in MODSIM II, consider things that move around, such as trucks and airplanes. This is the type definition of a moving object:

```
TYPE
  MovingObj =
  OBJECT
    position : LocTyp;
    course   : [ 0 .. 359 ];
    speed    : INTEGER;
    TELL METHOD GoTo(IN dest  : LocTyp,
                     IN spd   : INTEGER);
    ASK  METHOD Stop;
  END OBJECT;
```

`MovingObj` is an object type. It has three data fields which hold information about its location, course and speed.

In addition it has two **methods**. Methods are an object's procedures or routines which define its behavior. `GoTo` makes the object go to the specified destination from its current position. `Stop` is used to set the object's speed to zero. Note that the above object type declaration simply describes the state and methods of `MovingObj` and serves as an interface to the object. The actual code for the methods is supplied separately in the object declaration block. For example:

```
OBJECT MovingObj;
  TELL METHOD GoTo(IN dest  : LocTyp,
                   IN spd   : INTEGER);
  VAR
    travelTime : REAL;
  BEGIN
    travelTime := ... { compute time }
    course := ...{ some trig calculation }
    speed  := spd;
    WAIT DURATION travelTime
      { simulation time elapses here }
    END WAIT;
    speed := 0;
    position.x := dest.x;   { update }
    position.y := dest.y;   { position }
  END METHOD;
  ASK  METHOD Stop;
  BEGIN
    speed := 0;
  END METHOD;
END OBJECT;
```

`ASK` methods are instantaneous with respect to simulation time. When an `ASK` method is invoked, the caller pauses and control passes to the invoked `ASK` method. When the invoked method completes, the caller resumes. `ASK` methods behave like a procedure call but have direct access to all fields and methods of that object. No simulation time can pass in an `ASK` method.

`TELL` methods are asynchronous. When the `TELL` method is invoked, it is simply scheduled for execution, and the caller immediately continues execution without waiting for the `TELL` method to start. Simulation time can elapse in a `TELL` method.

A `TELL` method starts execution under control of the built-in simulation timing routine.

The data fields of an object instance are visible to all other parts of a program and may be "read" using an ASK statement. However an object's fields may be changed **only by the object itself**.

To use an object, we create an instance of that object type and send it messages using ASK or TELL when we want it to do something.

```
VAR
  boat : MovingObj;
BEGIN
  NEW(boat);
   { create an instance of a 'boat' object }
   { send a message }
  TELL boat TO GoTo(aDock, 15);
 { "go to position 'aDock' at speed 15" }
  StartSimulation;
  OUTPUT("Final location is: ",
      ASK boat position.x, " ",
      ASK boat position.y);
```

boat is a specific instance of the MovingObj type. MODSIM II uses dynamic memory allocation for objects. Thus while the object reference variable boat is declared in the VAR block, memory for the object is not allocated until a NEW statement is executed. DISPOSE(boat) releases memory during execution.

When the boat object gets the GoTo message, it does the following:

- Computes travel time from its present position to destination
- Computes and stores the course
- Stores the speed in its speed field
- Waits for the elapsed simulation time it will take to travel to the new destination
- Sets its new position in its position field

At the end of the short simulation, when the boat reaches the new location, the output statement asks the boat for its current location and prints it.

3. INFORMATION HIDING

As we've seen, the fields of an object can be changed only by the object itself. This is one level of information hiding. However it is still normally possible for any program code to "read" the value of an object's fields using an ASK statement. We can achieve a higher level of information hiding by declaring some of the fields to be private. Private fields cannot be seen except by the object itself. For example we can add a new field to the MovingObj type definition and declare it to be private :

```
TYPE
  MovingObj =
  OBJECT
    position : LocTyp;
    course   : [ 0 .. 359 ];
    speed    : INTEGER;
    TELL METHOD GoTo(IN dest : LocTyp,
                     IN spd  : INTEGER);
    ASK  METHOD Stop;
    PRIVATE
      timeStartedmove : REAL;
  END OBJECT;
```

timeStartedmove could used to calculate how far MovingObj had travelled if it were interrupted before arriving at the destination. No one outside of the MovingObj needs to know its value. It is declared as a PRIVATE field to prevent access to it.

Methods can be PRIVATE, too. Methods which are private can be invoked only by other methods of the object.

4. INHERITANCE

MODSIM II supports inheritance. With inheritance, new object types can be defined in terms of existing object definitions. While most languages only allow inheritance from one existing object type, MODSIM II supports multiple inheritance.

Here is a VehicleObj type definition created from a MovingObj:

```
VehicleObj = OBJECT(MovingObj)
  payload : REAL;
  TELL METHOD Load(IN amount   : REAL);
  TELL METHOD Unload(IN amount : REAL);
END OBJECT;
```

VehicleObj inherits all of the fields and methods of a MovingObj. In addition it adds a payload field and methods for loading and unloading the vehicle.

4.1 Overriding Methods

If an inherited method is no longer appropriate for the newly defined object, it can be overridden and replaced by a new one of the same name. The old method can be invoked by the replacement method as part of its behavior if desired.

We will define a new flying object type derived from the vehicle object type. Its GoTo method needs to be more elaborate, so we override the old one and provide a replacement:

```
TYPE
  FlyingObj =
  OBJECT(VehicleObj)
    altitude : INTEGER;
    OVERRIDE
    TELL METHOD GoTo(IN dest : LocTyp,
                     IN spd  : INTEGER);
  END OBJECT;
```

We have also added a new field, altitude, and will provide the replacement GoTo method in the object declaration block. All other aspects of FlyingObj will remain the same as VehicleObj.

```
OBJECT FlyingObj;
  TELL METHOD GoTo(IN dest: LocTyp,
                   IN spd: INTEGER);
  VAR
    departTime,
      { time spent in departure maneuvers }
    approachTime : REAL;
      { time spent in arrival maneuvers }
  BEGIN
    ... {  compute departTime }
    WAIT DURATION departTime
    END WAIT;
        { invoke the old GoTo method }
    INHERITED GoTo(airport, 350);
    ... {  compute approachTime }
    WAIT DURATION approachTime
    END WAIT;
  END METHOD;
END OBJECT;
```

The OVERRIDE indicates that the existing GoTo method will be replaced by a different one for the new FlyingObj object type. In this case the user "recycled" the existing VehicleObj GoTo method by invoking it from the new one.

5. POLYMORPHISM

Note that we now have three different object types which have a method called GoTo. The moving object type and vehicle object type share the same original method, but the flying object type has elaborated its GoTo method to fit its particular behavior. We can now TELL any of these objects to execute its GoTo method and it will execute behavior appropriate to the particular object type. As a user we don't need to know or care that some of the object types have supplied different or elaborated versions of the GoTo method. This important and versatile object-oriented capability is known as polymorphism — multiple behaviors invoked with the same method name.

6. MODULAR DEVELOPMENT

Putting all program code into a single module is feasible but not good practice. In big models it leads to unwieldy programs which are difficult to maintain.

MODSIM II programs usually consist of a main module and any number of supporting library modules. Each of these modules can be separately compiled to ease the task of development and maintenance.

Each library module typically contains declarations for a set of related procedures and objects and the executable code which constitutes the procedures and methods.

A library module actually consists of two modules, each of which is stored in its own file and is compiled separately. One is the definition module and the other is the implementation module.

The definition module is the interface between a library and the rest of the program. If something is defined in this module, the rest of the program can have access to it. There is no executable code in a definition module. Things which can be defined include object, record, array and enumerated types as well as procedures, variables and constants.

The implementation module contains the actual code which implements all procedures and methods of the library. It may also include local declarations which are needed solely within that library module.

The important point is that the definition module contains the external interface of a library. All objects, procedures, variables etc. mentioned in the definition module are available for import by other modules. The implementation module contains all of the code, but is not visible to other modules. This is another aspect of MODSIM II's data hiding and encapsulation capability.

Since any module can be separately compiled, maintenance is both simplified and accelerated. As an example, we can put our moving object into a library consisting of a definition module and an implementation module and a main module which uses the library.

```
DEFINITION MODULE MoveLib;
TYPE
  MovingObj =
  OBJECT
    position : LocTyp;
    course   : [ 0 .. 359 ];
    speed    : INTEGER;
    TELL METHOD GoTo(IN dest : LocTyp,
                     IN spd  : INTEGER);
    ASK  METHOD Stop;
  END OBJECT { MovingObj };
END MODULE { MoveLib }.

IMPLEMENTATION MODULE MoveLib;
OBJECT MovingObj;
  TELL METHOD GoTo(IN dest: LocTyp,
                   IN spd: INTEGER);
  VAR
    travelTime : REAL;
  BEGIN
    travelTime := ... { compute time }
    course := ...{ some trig calculation }
    speed  := spd;
    WAIT DURATION travelTime
  { simulation time elapses here }
    END WAIT;
    speed := 0;
    position.x := dest.x;   { update }
    position.y := dest.y;   { position }
  END METHOD { GoTo };
  ASK  METHOD Stop;
  BEGIN
    speed := 0;
  END METHOD { Stop };
END OBJECT { MovingObj };
END MODULE { MoveLib }.
```

To use the MovingObj in the MAIN MODULE, it is only necessary to IMPORT it.

```
MAIN MODULE SampleProgram;
  FROM MoveLib IMPORT MovingObj;

  VAR
    Car  : MovingObj;
    dest : LocTyp;
  BEGIN
    NEW(Car);
   { create an instance of Car object }
   { send a message }
    TELL Car TO GoTo(dest, 40.0);
   { send a message }
    StartSimulation;
    OUTPUT("Final location is: ",
       ASK Car x, " ", ASK Car y);
END MODULE { SampleProgram }.
```

7. DISCRETE EVENT SIMULATION & PROCESSES

Simulation is supported directly, as in SIMSCRIPT II.5, by built-in language constructs. The WAIT statement is used to make simulated time pass. Here is an example using the Load method of VehicleObj.

```
TELL METHOD Load (IN amount : REAL);
CONST
  rate = 0.25; { seconds per passenger }
VAR
  loadingTime: REAL;
BEGIN
  loadingTime := amount / rate;
  WAIT DURATION loadingTime
    OUTPUT("Loading completed");
  ON INTERRUPT
    OUTPUT("Loading NOT completed");
  END WAIT;
END METHOD { Load };
```

The WAIT DURATION statement causes the method to suspend execution for the indicated amount of simulation time. Once the wait is started, control returns to the scheduler which then starts execution of the next most imminent process.

When the WAIT is complete, control returns to this method at the statement after the WAIT.

Any of the methods of an object which are waiting for completion can be interrupted. If the method receives an interrupt command, it executes the part of the WAIT statement after ON INTERRUPT.

Two other forms of the WAIT statement let methods synchronize themselves.

```
  WAIT FOR Flight217 TO Load(324.0);
    ...
  END WAIT;
```

This statement schedules the Load method of Flight217 but does not allow the invoking code to proceed with execution until the Load method has completed. Note that this is different from a normal TELL invocation which proceeds without waiting.

The other form of the WAIT statement uses the built-in trigger object to synchronize methods.

```
  WAIT FOR ControlTowerLight TO Fire;
```

```
   ...
END WAIT;
```

This statement makes `Flight217` wait for a signal from the `ControlTowerLight` before it moves. `ControlTowerLight` is a trigger object which has a `TELL` method called `Trigger`.

```
TELL ControlTowerLight TO Trigger;
```

The `Trigger` method releases all waiting methods when it is executed.

8. THE DEVELOPMENT ENVIRONMENT

Transporting programs from one computer system to another has often been a problem. Frequently programs have to be extensively rewritten to eliminate machine dependencies. MODSIM II avoids this problem. It was designed for portability.

MODSIM II compiles its source code to C. The MODSIM II compilation manager then compiles and links the C code to a standalone executable.

MODSIM II's compilation manager was designed to facilitate project management of large computer programs consisting of many separate modules and libraries. It manages separate compilation of MODSIM II programs consisting of multiple modules by determining which modules have been edited since the last compilation and then recompiling only those edited modules and any modules which depend on them. This process is accomplished automatically without need for "make" or project files to describe the process.

The compilation manager also automatically handles the integration of modules written directly in C into MODSIM II programs. This permits direct interface to the large base of existing software libraries.

9. DYNAMIC GRAPHICS

Graphically displaying results has typically been a tedious programming task. To make matters worse, programs which take advantage of graphics are usually not portable. MODSIM II solves both problems. The programming task is made simpler through MODSIM's interface to the SIMGRAPHICS II graphics editor and environment. Since SIMGRAPHICS II is supported in a variety of environments, the MODSIM II programs which use SIMGRAPHICS II are portable.

SIMGRAPHICS II has three major capabilities:

- Animated graphics tied to objects in a program
- Dynamic or static graphs tied to variables and statistics in a program
- Interactive input menus in a contemporary windowed style

Animated icons, graphs and input menus are all interactively edited using the SIMGRAPHICS II editor. These are then tied to existing objects and variables in the user's program. This greatly simplifies the task of creating a graphical user interface. The amount of coding for graphics is drastically reduced.

Figure 1 shows some of the types of presentation graphics which can be created in the editor and tied to variables and structures in a MODSIM II program. The graphs can be updated dynamically as the value of variables in the program change or they can be displayed statically under programmer control.

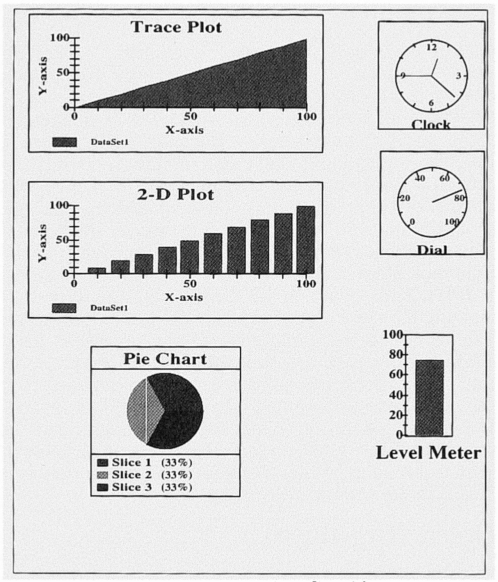

Figure 1. Presentation Graphics

The trace plot will show the value of a variable over time as simulation time elapses in a model. It will either rescale to show all information since program startup or keep a sliding window featuring only recent information. The 2-D plot is particularly useful in displaying histograms which are collected by MODSIM II's automated statistics gathering facilities.

Usually these graphs are used in a layout which features a background with icons moving to present objects in the model. The graphs present updated information about some aspect of the model's behavior.

Figure 2 shows a screen from a communications satellite model. The satellites are icons which move around the earth. The line between two satellites indicates that a message is being passed. At the top left is a trace plot of message rate versus time. At top right is a level meter showing the current mean message rate. Finally, the clock at the bottom shows that we are 31 seconds into the simulation.

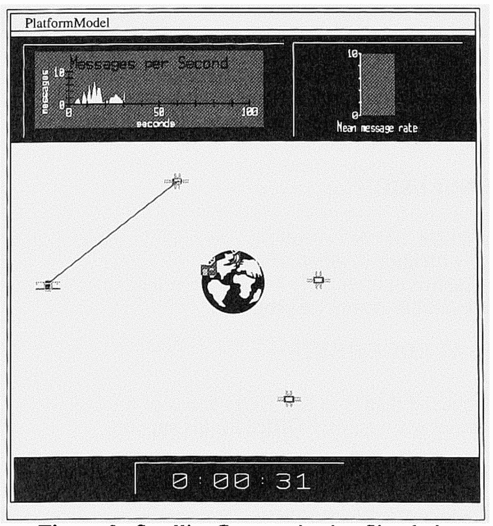

Figure 2. Satellite Communication Simulation

An important side benefit of the editor is that the appearance of these objects can be edited without recompiling or changing program code. This facilitates both design and subsequent maintenance as well.

10. PARALLEL PROCESSING

Parallel computers are now available. They promise substantial improvements in speed, especially for asynchronous programs, such as many discrete process simulations.

The object-oriented design of MODSIM II inherently works with these parallel processors. Because each object is self-contained, the operating system can assign it to any available processor. It will then execute independently of other objects.

The difficulty in exploiting these machines has been that objects executing on different processors must sometimes synchronize their activities. This is a problem when each has its own simulation clock running on its own schedule. The operating system, JPL's Time Warp in MODSIM II's case, has the task of synchronizing the execution of objects on different processors.

Preliminary results are encouraging. With appropriate programming techniques, it appears that the speed up can approach 1/2 the number of processors. That is, using 20 processors can lead to a reduction of execution time in the vicinity of 8 or 9 times.

Although initial results have been encouraging, the parallel version of MODSIM is still a research project. Considerable work still needs to be done before it can join the production version of MODSIM II in the commercial market.

11. BENEFITS OF MODSIM II

Any high order language is designed to reduce the effort needed to program a set of problems.

The object-oriented and modular features of MODSIM II substantially reduce the time and effort needed to write programs.

- Objects improve reliability because they encapsulate data fields and provide a disciplined interface to these fields.
- Development time is reduced because code can be put in libraries and reused.
- Modules permit step-wise development, particularly by separating the definition module from the implementation module.
- Inheritance allows programmers to build on top oi previous effort instead of starting from scratch each time. It is no longer necessary to take it "as is" or leave it in the software library. The programmer can take it then modify it to better suit the problem.

The integrated dynamic graphics substantially reduce the time and effort needed to build menus and display results. The graphics are portable.

The investment in programming is preserved because MODSIM II runs on most computer types.

12. CONCLUSIONS

MODSIM II is a robust, general purpose programming language with built-in graphics.

Its features substantially reduce the time and effort required to write and validate computer programs.

ACKNOWLEDGEMENTS

Our special thanks to Mr. Harry Jones of the U.S. Army Model Improvement Management Office for his support.

REFERENCES

Belanger, R., B. Donovan, K. Morse and D. Rockower (1990), *MODSIM II Reference Manual* - Revision 6, CACI Products Company, La Jolla, CA.

Belanger, R., and A. Mullarney (1990), *MODSIM II Tutorial* - Revision 8, CACI Products Company, La Jolla, CA.

Russell, E. C. (1983), *Building Simulation Models with SIMSCRIPT II.5*, CACI Products Company, Los Angeles, CA.