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

MODSIM II - AN OBJECT ORIENTED SIMULATION LANGUAGE FOR SEQUENTIAL AND PARALLEL PROCESSORS

Otis F. Bryan, Jr.
CACI Products Company
1600 Wilson Boulevard, Suite 1300
Arlington, VA 22209, U.S.A.
703-875-2900

ABSTRACT

MODSIM II is an object-oriented general purpose and simulation language designed to work with both sequential and parallel processors. Its objects have both single and multiple inheritance.

It has a single syntax that works across a variety of systems including main frames, work stations and PC's. A parallel version is currently under development on the BBN Butterfly parallel computer under the Time Warp operating system.

MODSIM II is based on the syntax of Modula-2. The majority of statements are identical with those in Modula-2. It has a few additional constructs that let the user write discrete process simulations.

MODSIM II has built-in object oriented constructs, including single and multiple inheritance, data abstraction and information hiding. In addition, it supports separate compilation. This makes it useful for large projects.

It contains interactive dynamic graphics to improve input and output.

INTRODUCTION

Computer science has made great strides in hardware and software since the introduction of FORTRAN in the mid 1950's. Work stations, new data structures and dynamic graphics have added important capabilities to programming.

Many of the languages have attempted to keep up with the field by adding new features. The original structure did not anticipate these changes, so it is difficult to add them elegantly. Eventually a new language needs to be written.

The introduction of the object, parallel processing and dynamic graphics have made it worth while to build a new simulation language from scratch.

FEATURES

MODSIM II was designed from the beginning to support large programming projects. It was designed to be a compiled, object-oriented language with multiple inheritance.

It's syntax is based on that of Modula-2. Programmers trained in Pascal should have no difficulty in learning the language. This saves training costs and time.

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

The simulation constructs were part of the design. They are based on CACI's 27 years of experience with discrete event languages. MODSIM II can be ported to new computer systems quite easily. The MODSIM II compiler actually produces C code. This is compiled, in turn, by the system's C compiler.

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

MODSIM II is a complete, powerful, general purpose and simulation language for large software engineering projects. Its features reduce work and improve reliability when compared with other languages.

OBJECTIVES

MODSIM was developed under contract to the US Army Model Improvement Program Management (AMIP) Office at Ft. Leavenworth. MODSIM II is the commercial version of MODSIM.

The AMIP Office laid down the following objectives for MODSIM:

- Object-oriented

- Discrete simulation using processes

- Modular development

- Direct support for expert systems

- Implemented as a translator to a target language (C)

- Complete syntactic validation by the translator

All of these objectives were achieved by August, 1988, except for two. MODSIM is a compiler, not a translator. The expert module was deferred to work on parallel processing.

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 horses and airplanes. This is the definition of a moving object:

```
MovingObj = OBJECT;

    X, Y,
    Speed,
    TimeLastChange,
    VX, VY: REAL;
    ASK  METHOD SetLoc (IN x,y: REAL);
    ASK  METHOD SetSpeed (IN speed: REAL);
    TELL METHOD GoTo (IN destination:
            MovingObj);
END OBJECT {MovingObj};
```

The brackets, {}, after END OBJECT enclose comments.

The MovingObj is an object. It has six data fields: two for its current location (X,Y), one for Speed, and two for its velocity vector (VX,VY). The sixth, TimeLastChange, is used internally for movement.

In addition it has three methods:

- SetLoc uses the two input parameters (x,y) to set the initial values of X,Y.

- SetSpeed uses the input parameter, speed, to set the Speed field.

- GoTo makes the object go to the destination from its current position. It asks the destination for its current location. Then, using the Speed, it calculates the components of the velocity vector (VX, VY). Finally makes simulated time to pass.

ASK methods are instantaneous: no simulated time passes. Control passes to the method and returns to the calling object. They are equivalent to a procedure call.

TELL methods are asynchronous: simulated time can pass. The TELL method is scheduled to execute, and the object immediately goes to the next commmand.

A scheduled TELL method starts execution only when control is passed to a master "timing routine", and it then passes control to the TELL method.

To use an object, we have to give it a specific name and send it messages when we want it to do something.

```
MAIN MODULE;

    VAR

        Barn, Horse: MovingObj;

    BEGIN

        NEWOBJ (Barn);
        NEWOBJ (Horse);
        ASK Barn  TO SetLoc (100.0, 100.0);
        ASK Horse TO SetSpeed (1.0);
        ASK Horse TO SetLoc (35.0, 27.5);
        TELL Horse TO GoTo (Barn);
        StartSimulation
        OUTPUT (ASK Horse X, " ", ASK Horse Y);

END MODULE {Main}.
```

The Barn and Horse are specific instances of the MovingObj, even though barns, in reality, don't go any where.

MODSIM II uses dynamic memory allocation for objects. While the Barn and Horse are declared in the VAR block, memory for them is not allocated until a NEWOBJ statement is executed. DISPOSEOBJ (Barn) releases memory during execution.

Since the current location of the Barn cannot be changed directly, the MAIN MODULE sends a message to the Barn telling it to set its location. This is done with the ASK statement.

When the Barn gets the message, it changes its current location (X,Y) to 100.0, 100.0, and returns. MODSIM II sets the initial values of all variables to 0.0. Because the barn doesn't move, its Speed is left at 0.0. It is a static object.

The Horse sets its Speed and location as directed by the ASK statements.

The Horse is then told to go to the Barn. The GoTo method asks the Barn for its location, calculates the velocity vector and lets time pass until the Horse gets to the Barn.

When the Horse reaches the Barn, the MAIN MODULE asks the Horse for its current location and prints it.

INFORMATION HIDING

Access to an object's data fields and methods can take several forms depending on the implementation in a

language.

In MODSIM II, data in an object can be read or changed only by the object itelf -- no outside statement can get direct access to the data: it has to send a message with an ASK or TELL statement.

Even the indirect access can be prevented by using the PRIVATE option. For example,

```
MovingObj = OBJECT;
    X,Y,
    Speed,
    VX, VY: REAL;
    ASK METHOD SetLoc (IN x,y: REAL);
    ASK METHOD SetSpeed (IN speed: REAL);
    TELL METHOD GoTo (IN destination:
            MovingObj);
    PRIVATE
        TimeLastChange
END OBJECT {MovingObj};
```

TimeLastChange is used to calculate when the MovingObj will arrive at its destination. No one outside of the MovingObj needs to know its value. It becomes a PRIVATE field to prevent access to it.

Methods can be PRIVATE, too. PRIVATE methods are invoked only by other methods in the object.

INHERITANCE

Inheritance lets simple objects be expanded without having to be rewritten completely. MODSIM II provides for single and multiple inheritance.

Here is a VehicleObj created from a MovingObj:

```
VehicleObj = OBJECT (MovingObj);

    Payload: REAL;
    TELL METHOD Load (IN amount: REAL);
    TELL METHOD Unload (IN amount: REAL);

END OBJECT {VehicleObj);
```

The VehicleObj has all of the fields and methods of a MovingObj. In addition it has a payload plus two methods for loading and unloading the vehicle.

Here is an example of the use of a VehicleObj:

```
MAIN MODULE;

VAR
    DullesAirport:  MovingObj;
    Flight217:      VehicleObj;

BEGIN

    ASK  DullesAirport TO SetLoc (39.0, -78.0);
    ASK  Flight217 TO SetLoc (50.0, 0.0)
    TELL Flight217 TO Load (324.0); {Passengers}
    ASK  Flight217 TO SetSpeed (560.0);
    TELL Flight217 TO GoTo (DullesAirport);
    TELL Flight217 TO Unload (324.0);
    StartSimulation

END MODULE {MAIN}.
```

Flight217 is going to fly from London's Heathrow Airport to Washington's Dulles Airport. First, it will take some time to load 324 passengers.

Once loaded it will use the methods of the MovingObj to set speed and fly to Dulles. Upon arrival it will use the Unload method of the VehicleObj to unload the passengers.

Multiple inheritance works the same way, but the declaraion is a little different:

```
AirplaneObj = OBJECT(VehicleObj,
        NavigationObj);
```

The AirplaneObj inherits the data and methods of both the VehicleObj and NavigationObj.

MODIFYING METHODS

An object's methods cannot be modified by another object. But they can be inherited and inherited methods can be modified by the inheriting object.

If the MovingObj uses Cartesian coordinates, then the calculations for Flight 217 will be wrong. A different algorithm is needed for locations given in latitude and longitude.

The VehicleObj can still use the MovingObj, but it has to provide its own method for great circle navigation.

```
VehicleObj = OBJECT (MovingObj);

    Payload: REAL;
    TELL METHOD Load (IN amount: REAL);
    TELL METHOD Unload (IN amount: REAL);
    OVERRIDE
        TELL METHOD GoTo;

END OBJECT {VehicleObj};
```

The OVERRIDE indicates that the existing GoTo method will be replaced by a different one for VehicleObj's. In this case the user will have to write it himself.

MODULAR DEVELOPMENT

Putting all objects into a single MAIN MODULE is feasible but not very good practice. In big models it leads to unwieldy programs.

One approach is to collect objects with related functions into a module and put modules in libraries. This improves cohesion and reduces coupling. Rather than copy the object into the program one just imports it by

referring to the object and module.

Moreover, it is desirable to separate the definition of an object from the actual coding of its methods. Since the other parts of the program can only send messages to invoke methods, they do not have to know how the methods work.

This leads to the concept of DEFINITION MODULEs and IMPLEMENTATION MODULEs.

```
DEFINTION MODULE MoveLib;

    MovingObj = OBJECT;
        X, Y,
        Speed,
        VX, VY: REAL;

        ASK  METHOD SetLoc (IN x,y: REAL);
        ASK  METHOD SetSpeed (IN speed: REAL);
        TELL METHOD GoTo (IN destination:
                MovingObj);
    END OBJECT {MovingObj};

END MODULE {MoveLib}.

IMPLEMENTATION MODULE MoveLib;

    OBJECT MovingObj;

        ASK METHOD SetLoc (IN x,y: REAL);
            BEGIN
                X := x;
                Y := y;

        END METHOD {SetLoc};

    END OBJECT {MovingObj};

END MODULE {MoveLib}.
```

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

```
MAIN MODULE;

    FROM MoveLib IMPORT MovingObj;

    VAR

        Barn, Horse: MovingObj;

    BEGIN

        ASK Barn TO SetLoc (100.0, 100.0);
        ASK Horse TO SetSpeed (1.0);
        ASK Horse TO SetLoc (35.0, 27.5);
        TELL Horse TO GoTo (Barn);
        StartSimulation (ASK Horse X, " ", ASK Horse
                Y);

END MODULE {Main}.
```

The DEFINITION MODULE is the interface between the object and the rest of the program. If something is defined in this module, the rest of the program can have access to it.

The source code for an IMPLEMENTATION MODULE does not have to be present, as long as an object file is available when the program is linked. This is information hiding with a vengence.

In fact, the IMPLEMENTATION MODULE doesn't have to be present at all. MODSIM II will compile a program consisting only of a MAIN MODULE and DEFINITION MODULEs. Implementation can be deferred until the architecture is complete.

This makes MODSIM II an excellent design tool.

DISCRETE SIMULATION USING PROCESSES

Adding simulation to Modula-2 involved adding a few more statements and developing the supporting libraries to execute them -- no mean task.

The WAIT statement is used to make simulated time pass. Here is an example using the Load method of the VehicleObj.

```
    TELL METHOD Load (IN amount: REAL);
        VAR
            rate,
            loadingTime: REAL;
        BEGIN
            rate := .25; {seconds per passenger}
            loadingTime := amount / rate;
            WAIT DURATION loadingTime;
                OUTPUT ("Loading completed");
            ON INTERRUPT
                OUPUT ("Loading stopped");
            END WAIT {DURATION loadingTime};

    END METHOD {Load};
```

The WAIT DURATION statement causes the object to suspend execution for the indicated amount of time. Control returns to the scheduler which starts execution of the most imminent process.

When the completion of loadingTime is the next event, control returns to this method at the statement after the WAIT statement.

Objects can interrupt any of their methods waiting for completion. If the method receives an interrupt command, it executes the part of the WAIT statement after ON INTERRUPT. This is essentially a two-part branching statement.

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

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

This statement invokes the Load method of Flight217 and waits for it to complete. Note that this is different from the usual method. The usual method schedules Load and proceeds without waiting.

The other form of the WAIT statement uses semaphores or TriggerObj's to synchronize methods.

WAIT FOR ControlTowerLight TO Fire;

This statement makes the Flight217 wait for permission from the ControlTowerLight before it moves. The ControlTowerLight is an object. It has a TELL method that raises a semaphore when it fires.

IMPLEMENTATION AS A COMPILER

Transporting programs from one computer system to another has always been a problem. Frequently they have to be extensively rewritten to modify machine dependencies.

MODSIM II avoids this. The goal is to take source code and recompile it without changes.

MODSIM II is a two part compiler. The first part coverts MODSIM II code to C. The second part uses the C compiler on the machine to convert the C code to an object file. The linker completes the process of creating an executable file.

Writing a MODSIM II compiler for a new system becomes straight forward. It is a matter of adapting a small portion of the runtime library to the machine and recompiling the compiler.

With this approach, procedures written directly in C can be added to the MODSIM II code and compiled as part of the program.

DYNAMIC GRAPHICS

Plotting results from a simulation has always been a tedious task. MODSIM II automates this process.

A graph is a plot of various values of a variable. Since these values are always available to the computer, the computer should plot them as they change.

The general method, known as SIMGRAPHICS II, works this way:

- The user lays out the graph as he wants to see it on the screen using an interactive graphics editor. He selects the type of graphic (eg. pie chart, graph) from a standard library and changes the attributes, such as color, location and X-axis values.

- When satisfied with the results, he saves it as a file in his directory.

- He then attaches the graph to the object and tells the object to display a data field using the graph. Every time the data field changes a value, MODSIM II automatically plots the new point.

This process works both for presenting results and animating the simulation.

In the case of animation, the graph is replaced by an image, say an airplane and is attached to a MovingObj. When the MovingObj changes location, the image moves to the new location.

Interactive graphics are available as a forms editor. The user can create a data input form in the same way he creates a graph. As the program runs, the form will appear and the user will enter or modify data. This provides dynamic control of the simulation at run time.

PARALLEL PROCESSING

Parallel computers are currently available on the market. They promise substantial improvements in speed, especially for asynchronous programs, such as discrete process simulations.

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

The operating system, Time Warp in MODSIM II's case, has to synchronize the execution of objects on different processors. The problem occurs when one object finishes execution prior to the another one.

For example, consider two planes that fly from Los Angeles to Washington. Flight 38 leaves Los Angeles, goes through Chicago and arrives at Washington at 9 pm. Flight 164 flies directly from Los Angeles and arrives at 9:30 pm.

The object representing Flight 164 finishes executing on its processor before Flight 38 and gets the ground crew to unload the plane. When Flight 38 arrives, it has to wait for the ground crew even though it "arrived" 30 minutes earlier.

Time Warp handles this by keeping a log of messages sent and received by each object as well as copies of previous states. When it appears that something is out of order, it cancels the message and rolls time back.

In this case upon arrival of Flight 38 Time Warp will cancel the message from Flight 164 asking for the ground crew. Flight 38 will get the ground crew, and Flight 164 will have to wait.

MODSIM II works under Time Warp, but certain modifications are needed to speed execution. In general, there can be no global data. That is, every data element must be part of an object and not a module. In short, the object is the largest element of a parallel program.

TELL methods should be used exclusively. Otherwise

processors wait until control returns during an ASK method. Parallel processing is faster if asynchronous methods are used exclusively.

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

MODSIM II is currently in development for parallel processors, but considerable work still needs to be done. A commercial version is not available at this time.

BENEFITS

Any higher order language is designed to cut the work in programming a set of problems. Simulation languages make it easier to write simulations than general purpose languages do.

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

- Objects improve reliability because they isolate data fields from the rest of the program.

- They reduce development time because they can be put in libraries and reused.

- Modules permit step-wise development, particularly by separating the definition module from the implementation module.

Early results in using MODSIM II with parallel processors indicates substantial improvements in run time are possible. But, again, much work remains.

The integrated dynamic graphics will substantially reduce the time to display results.

CONCLUSIONS

MODSIM II is a robust general purpose and simulation language incorporating the latest advances in computer science.

These features will substantially reduce the time and effort in writing compute programs.

ACKNOWLEGEMENTS

My thanks to Ron Belanger, Barbara Donovan and Katherine L. Morse for their comments on an earlier version of this paper.

REFERENCES

Belanger, R., Rice, S., Donovan, B., Morse, K. (1989a) MODSIM Users Manual La Jolla, California

Mullarney, A., West, J., Belanger, R., Rice, S. (1989b) MODSIM II Tutorial. La Jolla, California

SKIP BRYAN manages the Simulation and Modeling Department for CACI Products Company. He received a B.S. in Mechanical Engineering from MIT, an M.S. in R & D Management from the University of Southern California and an MBA from the University of Chicago. His current interests include specification and design of large scale simulation models.