Distributed Simulation 1988
©1988 by The Society for Computer Simulation
ISBN 0-911801-29-4

ModSim: A language for distributed simulation

Joel West
Palomar Software, Inc.
P.O. Box 2635
Vista, CA 92083
jww@sdcsvax.ucsd.edu

Alasdar Mullarney
CACI
3344 North Torrey Pines Court
La Jolla, CA 92037
ihnp4!clyde!cacilj!alasdar

Abstract

ModSim is a modular object-oriented simulation language. Although initially implemented on single-threaded processors, it is designed for use with parallel MIMD computers.

The object-oriented approach provides an important conceptual tool for designing a model and structuring the relationships between simulated objects. At the same time, the use of message-passing for interactions within the model provides a framework for realizing medium-grain parallelism.

This paper outlines the design objectives behind ModSim and the alternatives considered in its implementation, as well as the constructs deemed necessary to provide language-level support for distributed simulation.

1. Introduction

This paper is a summary of a project at CACI to develop a new simulation programming language, defined by **[Mullarney and West 1987]**. This project, sponsored by the Army Model Improvement Program Management Office, is an outgrowth of an earlier feasibility study **[West 1985]**, which concluded that a programming language designed for distributed simulation should be based on the object-method metaphor and incorporate the principles of behavior inheritance.

2. Design objectives

The ModSim (Modular Simulation) language is a hybrid object-oriented language that incorporates strong typing and data hiding for modular development.. Among its fundamental constructs are process-based discrete simulation capabilities and the incidental library support generally considered necessary for stochastic simulation.

The basic design is intended to allow execution of a ModSim simulation on MIMD (Multiple Instruction stream, Multiple Data stream) parallel-processing hardware, based on a realization of medium-grain (procedure by procedure) parallelism. Such constructs are, where possible, compatible with the existing implementation of the Time Warp operating system **[Jefferson et al 1987]** .

The explicit requirements of the sponsoring agency include:

- Object-oriented development framework
- Discrete simulation using processes
- Modular development
- Direct support for expert systems
- Implemented as a translator to a target language (initially C)
- Complete syntactic validation by translator

ModSim is most similar to Modula-2 **[Wirth 1985]** and Object Pascal **[Tesler 1985]**, but has several important concepts not present in either language.

The syntax of a ModSim program resembles that of Modula-2. The language differs significantly from the Object Modula-2 proposed analagous to Object Pascal, most notably in the explicit provision of message-passing concurrency and support for multiple inheritance. The language also includes direct support for process-based simulation.

As with most modern object-oriented languages, new object types can be defined in terms of existing types and the properties (behavior) of the prior type inherited by the new type. ModSim allows the flexibility of multiple-path inheritance, a rarity among existing object-oriented languages, in part because of implementation and efficiency difficulties for compiled languages.

Within the family of object-oriented languages, the goals for ModSim are most closely related to those for software engineering, such as those that led to the development of Modula-2 and Ada. These include support for large software development efforts (100,000+ lines of code) with medium- and large-sized teams, developed and maintained over a period of many years. The data hiding between objects also encourages the development of algorithms that can be used for parallel processing.

3. Declaring Objects

An object type is a structured type similar to a Modula-2 or Pascal record in that it includes a list of component fields. Unlike a record type, an object type may not have variant sections describing different field layouts.

In addition to component fields, an object type may also include a group of component procedures, which describe standard operations to be performed on the object.

Operations on the contents of an object are performed through these dedicated procedures, which are known as *methods*. Methods are invoked using an `ASK` or `TELL` statement, a reference to an appropriate object, the

name of the method, and the method parameters, if any, as in

```
TELL theplane TO goTo('Houston');
```

We call the externally accessible properties of an object — the mechanism by which other procedures communicate with it — the object's protocol. The fields and procedures together describe the properties of the object.

An object may be declared in terms of one or more previously-declared object types. We refer to the new type as a *derived type* of the existing type(s), while each of existing types is a *base type* of the new type. An *underlying type* of the new object is either one of the base types, or an underlying type of one of the base types.

The new type includes all the fields and methods of its underlying types and thus *inherits* the properties of those types. The new type can also define new fields and methods. If the object has no base type, these are the only fields and methods defined for that object.

The derived type can also *override* the definition of any method from its underlying types by redeclaring the method with the `OVERRIDE` reserved word. In such a case, the new definition replaces the previous one for the new type and any types derived from it. If the type is derived from multiple base types and a method is multiply inherited, then that method must be overriden in the new type.

The implementation of a method is similar to that for a Modula-2 procedure. All methods for an object must be grouped within a module with the same name as the corresponding object, as in:

```
MODULE PackageObject;
    METHOD linearMeasure: CARDINAL;
    BEGIN
(* implied argument SELF: PackageObject*)
        RETURN  height+depth+length;
            (* fields of SELF *)
    END linearMeasure;
```

4. Standard Library

ModSim comes with a library of standard object definitions, including modules for grouping objects, statistics gathering, and, of course, simulation.

The user will normally declare the base type of his/her most simple object as

```
MyObject = OBJECT
```

By default, all objects are assumed to be derived from StandardObject, which provides standard component fields and methods. Among objects that include `StandardObject` as an underlying type, one standard property is the ability to belong to a group of objects, or arbitrary number of groups. Another standard property is an error protocol, a default mechanism for handling errors that is, at the same time, user-extensible. Any such object will accept an `error` message, which can be overridden by the user to provide an exception-handling capability.

Objects declared in terms of another object type use a slightly different declarative syntax, as in

```
MyProcess = OBJECT(StandardProcess)
```

5. Grouping Objects

The library includes standard object definitions for grouping objects in ordered and unordered collections.

A deliberate departure has been made from the SIMSCRIPT approach towards lists of data structures, which has several disadvantages. First, any object that may belong to a list must be declared as such, and may only belong to one list with the same identifier.

More significantly from a parallel processing standpoint, each SIMSCRIPT data structure contains within it the direct reference to the next and previous structure in the list. This means that if the objects are different processors, a series of (probably slow) cross-processor references is necessary to find each object in the group.

Groups in ModSim are implemented using link data structures. For each object that belongs to a group, there is a link containing a reference to that object in the standard system-defined form. The links would remain on the same processor (and same virtual time) as the object that owns the group, since they are but a representational convenience for the state of the group object. These links could be used to reference any object on any processor.

ModSim includes iterators to perform operations on each members of a group, similar to those of SIMSCRIPT or CLU.**[Liskov 1979]**. However, when used with unordered collections, the iteration block may be performed in parallel, much as the `PAR` keyword of Occam **[Inmos 1984]** is used to denote a parallel iteration block.

6. Simulation Concepts

Existing process-based simulation constructs, such as those provided by Simula and SIMSCRIPT, provide for a block-and-wait capability and the transparent execution of multiple threads of control. Each process is an object that has a state that can be suspended and resumed, without regard to the implementation effort required to provide that transparency.

Each process object (one with `StandardProcess` as an underlying type) can use a `WAIT` statement to await completion of some condition. The language provides direct support for two conditions, passing of a specified period of simulated time and the completion of a method execution. The `WAIT` statement can also be used to await an arbitrary condition, which is indicated by the change in the value of a special type of monitored field, known as a *trigger*.

The ModSim process design allows for a process object to be doing more than one thing at once. It will may receive multiple messages, and process those

messages simultaneously, some of which will require time-elapsing sequences of actions.

Each of the operations that a process may engage at once is termed an *activity*. The execution of a method for a process is always part of an activity. An activity continues until terminated, either by completing a method or an explicit operation.

The interactions between these activities and the process (and thus the concept of an activity itself) are transparent to the programmer in writing simple process solutions.

However, certain abnormal conditions may require that all these activities to be stopped. A process may be interruped, which by default stops any waiting and returns the process to the inactive state.

Other applications will require more elaborate interfaces, such as being able to cleanly terminate only some of the activities, one at a time. The definition of an `ActivityObject` type allows reference to these individual activities, although the detailed implementation of activities cannot be used. Each `ProcessObject` has a group of all its activities.

An activity is either current or waiting; the process method `currentActivity` returns a reference to the current activity. When an activity terminates (becomes inactive), it is deallocated by the system. Therefore, an inactive process is one that has no activities.

An activity can be interrupted using the `INTERRUPT` statement, as in:

```
INTERRUPT findActivity(moveTo);
```

Interrupting the current activity has no effect. Interrupting an activity that is waiting will cause it to execute the exception clause of the `WAIT` statement. If no such clause is included, the interrupt will terminate the activity. A process has a standard method `interruptAll`, which interrupts all the activities of that process.

7. Parallel Simulation

There are several important distinctions that must be made when examining the object-oriented interactions between processes in a multi-processor environment.

Interactions in an object-oriented language are, of course, through method invocation. One important aspect of the conceptual model is whether each method is implemented synchronously or asynchronously, that is, whether or not the object sending the message waits for receiving object to complete the requested behavior.

The distinction can often be ignored for inter-process interactions in a sequential environment. In SIMSCRIPT II.5, for example, activating or reactivating a process (similar to a method call in many ways) is always asynchronous.

However, in a parallel-processing environment, the issue is more important, since the asynchronous execution of methods offers an opportunity for realizing medium-grain parallelism, while the synchronous execution of methods provides a formal mechanism for structuring inter-process time dependencies.

One proposed approach **[West 1985]** is to have two types of message-sending operators, one used for synchronous interactions and another used for asynchronous interactions.

This is, in fact, the approach adopted by ModSim. For example,the statement

```
ASK theplane TO setCourse('Houston');
```

is assumed to execute instantaneously in simulated time, and the object sending the message will not continue execution until the corresponding method is complete. In contrast, the method call

```
TELL theplane TO goTo('Houston');
```

initiates the requested method (`goTo`) for the specified object (`thePlane`), but the sending object continues execution immediately.

If one thinks of a ModSim process as a program, and an activity as a UNIX process, then the `TELL` keyword is analogous to the `fork` function of the standard UNIX library **[AT&T 1986]**.

8. Time Warp Message-passing

The Time Warp operating system also has two ways to send a message, but uses a different criterion for the distinction between the two. The difference between the two is centered around Time Warp's approach to maintain the correct time ordering of interactions within a simulation.

Time Warp structures distributed simulations around time-stamped messages sent between individual objects. The framework assumes the availibility of each of both synchronous messages and asynchronous messages.

However, the existing implementation imposes a further restriction on each type of message. It attempts to isolate and identify any message that can cause a side-effect or change in state. Side-effects exclude returning a value to the sending object, but include any other change in the state of the receiving object or another object. Although Time Warp is event-oriented, advancing simulated time in a process-oriented language would also require such a state change and thus qualify as a side-effect.

The Time Warp o.s. performs optimistic race-ahead and time (causality) fault correction through rollbacks, and identifying potential side-effects allows it to perform an important suboptimization. By examining each message in the message queues, the Time Warp o.s. can identify which messages cause side effects and which do not. A time fault involving a no-side-effects message can be handled without rolling back the entire simulation to the saved state. For example, asking a plane its position at time T=20 may only require searching a list of previous values and changes for the location variable.

In the current implementation of the Time Warp o.s., not all possible synchronization and side-effect combinations are allowed. The matrix of available combinations in shown in Table 1.

	Synchronous	Asynchronous
side-effects	not allowed	*Event message*
no side-effects	*Query message*	not allowed

Table 1: Synchronization vs. side-effects in Time Warp

Although not required by the Time Warp paradigm, the current implementation directly supports only one type of synchronous message, the type that does not involve side-effects. Such side-effects include changing any of the object's variables or sending a message to another object that causes side-effects.

Such messages are referred to as *query messages,* since they are primarily useful for querying the state of the object — either directly stored values, such as fields, or computed values, such as estimated time of arrival. When the sending object transmits a query message to the recipient, it is always followed by a *query response message,* in which the recipient object returns 0 or more values to the sender.

This corresponds to the purest form of side-effect free functional programming. Unlike procedural languages which encourage (but do not require) such use of functions, it also applies to procedures that return multiple values through variable parameters (such as Modula-2 `VAR`), since the restriction is imposed by operating system primitives rather than the programming language.

In contrast, any message that causes side effects must be asynchronous under the current implementation. Such a message is called an *event message.* Any event method can send query messages, but no query method can send event messages.

9. Message Side-effects In ModSim

If accepted on its face, the prohibition against combining a state change with synchronous execution of a method severely limits the encapsulation of several categories of operations typical of both simulations and object-oriented programs.

A number of coding schemes are possible to work around these restrictions, but all require significantly more user code and many have other significant disadvantages. In general, these work-arounds will tend to diminish the clarity of sequential program expression (as advocated by Jackson System Design) that is provided by the process framework. They also require the client of a method to know more about its implementation and reduce the encapsulation of the method.

It would be more useful to implement the side-effect + return value capability directly for the user. Event response messages could also be emulated by the programming language library using ordinary event messages. However, this may obscure useful dependency clues for the Time Warp operating system.

The ModSim language does provide support for query messages. All function methods in ModSim are assumed to be synchronous and side-effect free, and thus can be implemented in Time Warp as query messages.

The possible combinations of synchronization and side-effects in ModSim are shown in Table 2.

	Synchronous	Asynchronous
side-effects	`ASK`	`TELL`
no side-effects	function method	meaningless

Table 2: Synchronization vs. side-effects in ModSim

For synchronization purposes, it is important to know which methods can advance simulation time. Of course, methods for non-process objects will always belong to one of the first two categories.

Therefore, methods would be separated into three categories of side-effects:

- No side-effects;
- Side-effects that do not advance simulated time;
- Time-elapsing.

The use of time-elapsing constructs are directly deduced by the compiler from ModSim source statements. However, side-effect causing methods must be explicitly declared, and then the compiler verifies the absence of side-effects in other methods.

10. Comparison with Other Languages

ModSim adopts the Object Pascal syntax for declaring objects, instance variables (fields), and methods, as well as its syntax for referencing an inherited behavior. From Clascal, ModSim adopts the grouping of related methods, adapted to use a Modula-like syntax.

ModSim's syntax for referencing variables and methods of an object is different from the other languages listed here. Although somewhat similar to Flavors, it most closely resembles an earlier proposed language for concurrent simulation based on SIMSCRIPT **[West 1985]**.

Like Object Pascal, the terminology of ModSim is built around the "object type" rather than the "class" terminology used by most other languages. However, C++ includes a simple and unambiguous terminology ("base class" and "derived class") for relating class inheritance, which is used here.

ModSim is a member of the family of typed object-oriented languages that begins with Simula and includes both Object Pascal and C++. Unlike Objective-C or Smalltalk compilers, ModSim attempts to disambiguate

object references at compile time whenever possible.

Instead of the C++ `friend` concept, private fields and procedures are obtained by clustering related object types within a module. In this case, "friends" are object types, not individual methods, and friendship is always symmetric. This is a less rigorous form of data hiding, but easier to use and does not require the introduction of additional concepts.

Data hiding of the properties of a base type from its derived type is assumed to be less important than between two object instances. Among the languages mentioned, only C++ provides data hiding to prevent access to the implementation of a base type from its derived types.

The multiple inheritance approach is similar to that used by Smalltalk-80 **[Borning and Ingalls 1982].** In particular, ModSim allows combinations of full types (instead of the "mix-in" of the Lisp Flavor System **[Weinreb and Moon 1980]**) and requires the use of named base types to resolve method inheritance ambiguities.

With the exception of Simula, none of the languages mentioned directly address the topic of simulation. Smalltalk and C++ acknowledge simulation as goals, but do not provide simulation as a fundamental language concept.

11. References

AT&T. *System V Interface Definition, Issue 2*, Volume I, AT&T, Indianapolis, Ind.: 1986.

Borning, Alan and Dan Ingalls. "Multiple Inheritance in Smalltalk-80," *Proceedings of the AAAI*, 1982, 234-237

Inmos Limited, *Occam Programing Manual*, Prentice-Hall International, London: 1984.

Jefferson, David et al. "The Time Warp Operating System," (unpublished paper). Los Angeles: UCLA Department of Computer Science, February 1987.

Liskov, Barbara et al. *CLU Reference Manual*, MIT/LCS/TR-255, Cambridge, Mass.: MIT Laboratory for Computer Science, 1979.

Mullarney, Alasdar and Joel West, *ModSim: a language for Object-Oriented Simulation; Design Specification*. CACI technical report, La Jolla, Calif.: CACI, August 1987.

Tesler, Larry. "Object Pascal Report", *Structured Language World*. Volume 9, Number 3; 1985.

Weinreb, Daniel and David Moon. "Flavors: Message Passing in the Lisp Machine," MIT-AIM-602, Cambridge, MA: MIT Artificial Intelligence Laboratory, November 1980.

West, Joel. *Object-Oriented Distributed Simulation*, CACI technical report, La Jolla, Calif.: CACI, 1985.

Wirth, Niklaus. *Programming in Modula-2*, 3rd ed. New York: Springer-Verlag, 1985.