Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

AN ASSESSMENT OF THE MODSIM/TWOS PARALLEL SIMULATION ENVIRONMENT

David O. Rich
Randy E. Michelsen

Analysis And Assessment Division
Los Alamos National Laboratory
Los Alamos, NM 87545

ABSTRACT

The major flaw with the ModSim/TWOS system as it currently exists is that there is no compiler support for mapping a ModSim application into an *efficient* C/TWOS application. Moreover, the ModSim language as currently defined does not provide explicit hooks into the Time Warp Operating System and hence the developer is unable to tailor a ModSim application in the same fashion that a C application can be tailored. Without sufficient compiler support, there is a mismatch between ModSim's object-oriented, process-based execution model and the Time Warp execution model. In this paper we present our assessment of ModSim/TWOS and also discuss both components in isolation.

1 INTRODUCTION

There are two primary difficulties in developing large-scale, discrete-event simulations for analysis and production. First, there is the issue of software manageability over the life-cycle of a simulation. Simulations, by their very nature, are dynamic and requirements typically evolve as a system is used. Second, for large enough systems, execution speed (in terms of user response time) becomes a concern. Simulations with poor response time are not attractive tools for doing analysis. In the specific area of large, discrete-event combat simulations, both software manageability and run speed are important. Two areas of software systems research address these problems directly, but independently: object-oriented software engineering (OOSE), and parallel, discrete-event simulation (PDES). OOSE is quickly gaining widespread acceptance as the approach of choice for solving the software development and management problem (see Meyer 1988 and Booch 1991). At the same time, PDES research is directly addressing the run speed issue (Fujimoto 1990 presents an excellent survey of the current state of research in parallel, discrete-event simulation). Ideally, a system should provide a high-level simulation language that embodies well-known software engineering principles combined with a high-performance parallel execution environment. One recent attempt at building such a system is the ModSim/TWOS parallel simulation environment.

The Time Warp Operating System (Jefferson et al. 1987), TWOS, has been the focus of significant research in parallel, discrete-event simulation. A new language, ModSim, has been developed for use in conjunction with TWOS. The coupling of ModSim and TWOS is an attempt to address the problem of developing large-scale, complex simulation models for parallel execution (what we like to think of as PDES *in-the-large*). The inherent difficulty with this approach is the mapping of the simulation application to the parallel run-time environment. To use TWOS, Time Warp applications are currently developed in C and must be tailored according to a set of constraints and conventions (we will refer to these as C/TWOS applications). C/TWOS applications are carefully crafted using explicit calls to Time Warp primitives; thus, the mapping is done by the application developer. The disadvantage to this approach is the questionable scalability to larger software efforts; the obvious advantage is the degree of control over managing the efficient execution of the application. The ModSim/TWOS system provides an automatic mapping from a ModSim application to an equivalent C/TWOS application. The major flaw with the ModSim/TWOS system as it currently exists is that there is no compiler support for mapping a ModSim application into an *efficient* C/TWOS application. Moreover, the ModSim language as currently defined does not provide explicit hooks into the Time Warp Operating System and hence the developer is unable to tailor a ModSim application for execution on TWOS in the same fashion that a C application can be tailored for execution on TWOS. Without sufficient compiler support, there is a mismatch

between ModSim's object-oriented, process-based execution model and the Time Warp execution model.

As part of the Parallel Eagle Project here at Los Alamos National Laboratory (LANL), initiated in January 1989, we have completed an initial assessment of the coupling of ModSim and TWOS from an application-intensive perspective. To date, the Parallel Eagle model comprises over 15,000 lines of ModSim code.

In the following sections we will present our assessment of ModSim/TWOS and also discuss both components in isolation. The paper proceeds as follows. In section 2, we provide an overview of the Parallel Eagle model as motivation for the Parallel Eagle study. In section 3 and 4, an overview and assessment of the ModSim language (independent of TWOS) is presented. Section 5 briefly discusses parallel, discrete-event simulation in general and section 6 provides a look at the Time Warp Operating System in particular. In Section 7, we present our evaluation of the ModSim/TWOS system and conclude with related work in section 8.

2 THE PARALLEL EAGLE MODEL

The Parallel Eagle model is a deterministic, division level combat model developed using an object-oriented approach and targeted for execution under the Time Warp Operating System. It has been derived from the Eagle model originally developed by the Army (TRADOC Analysis Command, Ft. Leavenworth) and LANL. The baseline version was the original Eagle prototype (circa 1988), with selected elements of the follow-on effort (circa 1989-90) chosen for inclusion in the Parallel Eagle model. The functionality developed for the model includes elementary maneuver, command and control, direct-fire attrition, a basic intelligence process, and terrain representation (Powell 1989). The primary purpose of the parallel model development effort is to produce a realistic, large, object-oriented combat simulation model for use in evaluating the utility of TWOS paired with the ModSim programming language. A secondary goal was to study the process of transitioning from a sequential, object-oriented application written in Lisp and KEE (IntelliCorp's Knowledge Engineering Environment) to a parallelized version developed in ModSim.

3 AN OVERVIEW OF MODSIM

ModSim was designed and implemented by CACI Products Company under contract with the Army Model Improvement Program (AMIP) Management Office. In 1985, CACI performed a feasibility study for the Army (West 1985) which resulted, in 1987, in the design specification for ModSim, an object-oriented language derived from Modula-2 (Mullarney et al. 1987). In early summer of 1988, CACI completed and delivered to the Army the first ModSim compiler and run-time system for a single processor architecture. Later, in the fall of 1988, CACI delivered a modification of the compiler and run-time system for use with the Time Warp Operating System under development at the Jet Propulsion Laboratory (JPL). After completing their contract with the Army, CACI developed a commercial version, called MODSIM II (Bryan and Belanger 1989). For the remainder of this paper, references to "ModSim" refer to the Army's version of the language as developed under contract by CACI (for information regarding MODSIM II contact CACI Products, La Jolla, California).

ModSim is a strongly typed, modular, block structured language which provides support for object-oriented programming and discrete-event simulation. Its design has been heavily influenced by Modula-2, Simula, and SIMSCRIPT II.5. The language is intended to be used for building large-scale, discrete-event simulations using modular, object-oriented development techniques.

In this section, an overview of the basic ModSim language is presented. Language features particularly designed to support simulation development are also discussed. The following language discussions are based on release 1.0 of ModSim (for more information refer to the ModSim references at the end of this paper).

3.1 The Base Language

ModSim's syntax and control mechanisms resemble those of Modula-2, Pascal or Ada. The example in Figure 1, illustrates a simple procedure definition; keywords are required to be in upper-case, statements are delimited by semicolons, and comments are enclosed in curly braces as { comment }, or as in Modula-2, (* comment *).

```
PROCEDURE UpCase(INOUT str:ARRAY OF CHAR);
  VAR k : INTEGER;
BEGIN
  REPEAT
    IF (str[k] >= 'a')
     AND (str[k] <= 'z')
      DEC (str[k], ORD('a') - ORD('A'));
    END IF;
    INC (k);
  UNTIL (str[k] = 0C);
END PROCEDURE; { UpCase }
```

Figure 1: A ModSim Procedure

ModSim's approach toward supporting modularity was mainly influenced by Modula-2. For example, like Modula-2, ModSim provides the capability to decompose

an application into a *main module* and a set of *library modules*. A library module consists of two parts: a *definition module* and an *implementation module*. A definition module provides the interface information required for subsequent reference to the entities defined therein. Constants, types, variables, procedures and objects described in a definition module may be imported and used by other modules. Procedures and objects, in particular, are provided with procedural descriptions in the implementation module. This separation of definition and implementation supports functional decomposition and information hiding. A complete ModSim program may also be written as a single main module.

3.2 Object-Oriented Mechanisms

ModSim provides explicit support for object-oriented programming (Meyer 1988). In particular, ModSim supports single and multiple inheritance, dynamic binding, data abstraction, encapsulation and information hiding all within a strongly typed framework.

In ModSim, an object definition creates a new user-defined data type identified by the supplied name (we will use "object type" and "object" to mean the same thing within the context of ModSim). A ModSim object is composed of fields and methods. The fields describe an object's state while the methods define any actions the object can perform. Figure 2 shows a definition of a simple object type named `Line` in ModSim.

```
Line = OBJECT
   { Ax+By+C=0 }
   A,B,C:REAL;

   ASK METHOD InitLine(IN a,b,c:REAL);
   ASK METHOD Rotate(IN angle:REAL);
END OBJECT; { Line Object Type }
```

Figure 2: An Object Type Definition In ModSim

An important characteristic of objects is that the fields of an object can only be modified by methods defined for (or *belonging to*) that object. This formal and explicit association of data and code provides an inherent and full encapsulation of the data abstraction. Moreover, fields and methods may be defined to be `PRIVATE` and hence not part of the object's public interface; ModSim's `PRIVATE` mechanism is similar to "protected" members in C++ (Ellis and Stroustrup 1990) and is useful for implementing higher degrees of information hiding. An instance of an object type is explicitly created through the use of the procedure `NEWOBJ` to allocate the requisite storage. Similarly, object instances must be explicitly de-allocated via the procedure `DISPOSEOBJ`.

ModSim supports single and multiple inheritance in the definition of new object types. That is, a new object type in ModSim may be defined in terms of one or more existing object types. The new type is called a derived object type. Underlying types are any types which an object inherits; the most immediate types are called an object's base types. A derived object type inherits all fields and methods of the underlying types. For example a line segment object type, `Segment`, derived from the type `Line` is presented in Figure 3.

```
Segment = OBJECT (Line)
   { a segment is simply a line with end points }
   EndPoint1:Point;
   EndPoint2:Point;

   ASK METHOD InitSegment(IN e1,e2:Point);
   OVERRIDE
      ASK METHOD Rotate(IN angle:REAL);
END OBJECT; { Segment Object Type }
```

Figure 3: Inheritance In ModSim

In addition, a derived type may override any of the methods and provide more specific (or extended) implementations (e.g., the `Rotate` method for a `Line` must be extended for a `Segment` since the end points must be rotated). Since methods may be overridden, a mechanism is necessary to decide at run-time what method should be executed. This mechanism is commonly known as dynamic binding. Dynamic binding implies that the dynamic form of the object determines which method is applied (Meyer 1988). It is this polymorphism that adds to the power and flexibility of the object-oriented approach.

Interactions between ModSim object instances are handled by sending messages. ModSim provides two language constructs, namely `ASK` and `TELL` style methods, that support synchronous and asynchronous communication between instances of objects. As discussed previously, an object's fields can only be modified by methods belonging to the object. Thus, to change the state of an object a message must be sent that will invoke the desired method. For example, assume that an instance of the object type `Segment`, `aSegment`, may be rotated by use of the method `Rotate` defined for the type. This method is invoked using the `ASK` mechanism, e.g., `ASK aSegment TO Rotate (45.0)`. Similarly, the `ASK` mechanism is used to access the current value of an object's fields, e.g., `point := ASK aSegment EndPoint1`. Conceptually, both examples result in a message being sent to the object `aSegment`. In former, the result of the message is a change in the object instance `aSegment`'s state and in the latter the result of the message is the return of the

appropriate field value.

3.3 Simulation In ModSim

ModSim provides explicit support for discrete-event simulation using a process-oriented approach similar to SIMSCRIPT II.5. The constructs which support the concept of process-oriented simulation are the TELL method and the WAIT statement. Using these constructs, objects can interact asynchronously with other objects and can advance simulation time. The passage of simulation time is accomplished through the use of TELL methods which implement certain behaviors of the object. For example, `TELL object1 TO do-something(a,b) IN 5.0` will cause the message `do-something(a,b)` to be placed on the message queue of `object1` with a time stamp of `NOW+5.0` (NOW is simply a notational convenience meaning the current simulation time). The actual invocation of the method, `do-something(a,b)`, is deferred. In the context of the sender, execution continues past the TELL statement resulting in a non-blocking send. When simulation time reaches `NOW+5.0`, the object, `object1`, will process the `do-something(a,b)` message. Within a TELL method, simulation time can be advanced by waiting for something to happen (e.g., the completion of another TELL) or by waiting for some duration of time (both accomplished via a WAIT construct).

Figure 4 illustrates how the behavior of a combat unit object with a low fuel supply might be realized.

```
{ wait until refuel process completes or is interrupted }
WAIT FOR SELF TO Refuel()
   { refuel process successfully completed }
   CurrentObjective:=
      ASK Hq NewOrders(MyStatus);
   TELL SELF TO
      ProceedTo(CurrentObjective);
ON INTERRUPT
   { refuel process interrupted, potential problem }
   TELL Hq Problem(MyStatus);
   RETURN; { exit current method }
END WAIT;
```

Figure 4: Discrete-Event Simulation In ModSim

Refueling requires some amount of simulated time, so the unit must "wait" until this process is complete. Operationally, the current method is suspended until the `Refuel` method completes or is interrupted and then the method resumes accordingly. In the case that refueling is successful, the combat unit ASKs an `Hq` (Headquarters) object for a new objective given its current status and then proceeds to the objective by scheduling its `ProceedTo` method (via a TELL message). In the event that refueling is interrupted (this is accomplished via ModSim's INTERRUPT mechanism), the unit sends a message to the `Hq` object indicating a problem.

Both TELLs send messages with time stamps of NOW (the current simulation time). The ASK call is synchronous and no simulation time may elapse within the ASK method. The `WAIT FOR` suspends the current activity and resumes when the activity completes or is interrupted; the `Refuel` method may elapse simulation time.

4 AN ASSESSMENT OF MODSIM

To date, we have written and tested over 15,000 lines of ModSim in support of the Parallel Eagle Project (our early experience is documented in Rich and Michelsen 1989). Some code, such as the terrain subsystem, have undergone several design/implementation iterations over the course of the last two years. Suffice it to say that, in the process, we have gained an appreciation of both the strengths and weaknesses of the ModSim language and its implementation.

Up until this point, we have concentrated on presenting the ModSim approach to object-oriented simulation. In practice, however, a concept is only as good as its realization. On the whole, we find the ModSim language and development environment to be generally usable. There are, however, deficiencies in both the language design/implementation and development environment that we consider important. These deficiencies amount to inconsistencies and oversights in the design and implementation of the language and development environment including the treatment of memory management, exception handling, the type system, inheritance, and separate compilation across "projects." In the following sub-sections, we will highlight our concerns regarding the current implementation of ModSim.

4.1 Memory Management

Automatic memory management is considered an important characteristic of any modern object-oriented language system (Meyer 1988). Without automatic garbage collection, the user must dispose of objects explicitly. In ModSim, there is no built-in support for managing dynamic object memory other than the allocation and de-allocation procedures, NEWOBJ and DISPOSEOBJ. Explicit de-allocation without any support for managing "reference" information may introduce dangling reference bugs; in practice, we have found explicit memory management to be the major source of run-time errors.

4.2 Exception Handling

The programmer is responsible for instrumenting code using conditional tests. There is no built-in support for encapsulating run-time exception handlers within objects or within the program itself. Thus, failures that are not caught by the conditional tests result in an ungraceful termination of the program execution (e.g., undefined reference or pointer, or a divide-by-zero). The importance of software robustness cannot be overstated and exception handling support is a critical component in providing such behavior.

4.3 Type System

ModSim is a strongly typed language. That is, type correctness is checked at compile time. Compile time type checking, however, cannot assure type safety since ModSim supports both a REFERENCE and an ADDRESS type. An assignment of an object instance to a variable of type REFERENCE, or a record instance to a variable of type ADDRESS results in the loss of type information for the respective entities. Thus, a variable of type REFERENCE (for example) containing a reference to an object instance can be legally assigned (i.e., "cast") to any object type variable. Such operations are not type-safe since they can lead to failures during program execution if an object instance referenced by a variable of type REFERENCE is incompatible with the object type of another variable to which it is later assigned (see Figure 5).

```
{ ...implementations of X & Y not shown... }
TYPE
   X = OBJECT
     ASK METHOD A();
   END OBJECT;

   Y = OBJECT
     ASK METHOD B();
   END OBJECT;

VAR
   x : X; y : Y; r : REFERENCE;
...
r := x;  { legal assignment }
y := r;  { still ok }

ASK y TO B();  { run-time error -- segmentation fault }
```

Figure 5: Type Correct Versus Type Safe In ModSim

In the absence of parameterized types, however, such a "type casting" mechanism is necessary to support type coercion; type casting mechanisms such as those found in C++ can be imitated in ModSim by assigning first to a REFERENCE variable and then to the variable of the desired object type (Booch 1991 provides examples where type coercion is useful). As shown in Figure 5, it is up to the programmer to manage type safety. Clearly, a parameterized type mechanism is a more complete and desirable solution.

4.4 Inheritance

In general, the design and implementation of ModSim's inheritance mechanism results in unnecessary constraints and inconsistencies. For example, identical field names in two or more base objects do not conflict when multiply-inherited. These field names are not directly accessible within the derived object and can only be accessed once the object instance has been assigned to a reference variable of the appropriate object type. For example, in Figure 6, Z's method D cannot access either A in X or A in Y, but method D can access both fields B and C. To get at one or the other A fields in Z, an object instance of type Z must be *demoted* (using type coercion) to either type X or Y.

```
{ ...implementations for X, Y & Z not shown... }
TYPE
   X = OBJECT
     A:REAL; B:BOOLEAN;
   END OBJECT;

   Y = OBJECT
     A:INTEGER; C:CHAR;
   END OBJECT;

   Z = OBJECT (X, Y)  { Z inherits types X and Y }
     ASK METHOD D();
   END OBJECT;

VAR
   x : X; y : Y; z : Z;
   c : CHAR; b : BOOLEAN;
   r : REAL; i : INTEGER;
...
c := ASK z C;  { ok }
b := ASK z B;  { ok }

{ "ASK z A" is ambiguous, so... }
x := z;
r := ASK x A;  { A in X — a REAL }
y := z;
i := ASK y A;  { A in Y — an INTEGER }
```

Figure 6: Inheritance Of Fields In ModSim

In contrast, identical method names in two or more base objects are flagged by the compiler when multiply-inherited — even if the parameter list and/or return types

are different (i.e., they have different "signatures"). The user must explicitly deal with this situation either by overriding the method definition and providing an implementation to disambiguate or, in the case of methods with the same name but different signatures, changing the method names in the base objects. Furthermore, since ModSim does not manage repeated inheritance (Meyer 1988) the treatment of conflicting method names becomes an especially tedious concern of the programmer.

4.5 Compile-Time And Run-Time Support

The development environment is an important component for any object-oriented language implementation. Amenities such as an interactive source-level debugger, automatic re-compilation of separate, but dependent modules (for compiled languages), class hierarchy browsers, etc., all help define the "usability" of a system. The Eiffel system, Saber C++ and IntelliCorp's ProKappa are all examples of development environments for object-oriented languages. The ultimate environment takes you from analysis to design to implementation seamlessly; this, of course, is the goal of CASE (Computer Aided Software Engineering), and object-oriented CASE in particular.

The ModSim development environment is comprised of a compiler, a compilation manager, a genealogy (or object type) browser and a run-time library with a built-in traceback utility.

The compiler translates ModSim source to C code on a line-by-line basis without benefit of *any* optimization. As in Eiffel, this translation process could benefit from optimizations such as call simplification for non-overridden methods and the removal of unneeded code (see Meyer 1988 for a discussion of the Eiffel optimizer).

The compilation manager is a simplified UNIX *make* utility. As such, it manages separate compilation of ModSim programs consisting of multiple modules by determining which modules have been edited since the last compilation and re-compiling only those modules and any other modules which depend on them. The compilation manager, however, does not support "smart" compilation across projects. Thus, changes in the source of one project are not detected in the importing project. A project in ModSim is a collection of modules that implement a single object-oriented concept such as a graphics library, or a string manipulation library; i.e., a set of objects and procedures for manipulating the objects. Changes in the implementation of one project (e.g., only implementation modules are edited) simply requires a re-link on the part of the importing project. Such modifications go unnoticed in the current version of the compilation manager. The only project treated by the compilation manager is the built-in run-time library. Any changes to the run-time library will cause all ModSim projects to be completely re-compiled (this, in itself, is not done very smart since only a re-link may be required in some cases).

The genealogy browser is implemented in a textual format utilizing a screen-at-a-time view of information. Each screen provides information and a list of selections that are entered from the keyboard; the browser is not a GUI (Graphical User Interface). Moreover, the browser uses a data file generated by the compiler; thus, in order to browse updated source it must first be re-compiled.

Finally, run-time debugging can be accomplished using dbx (or another C source-level debugger) on the C source code. To accomplish this, the programmer must configure the compilation manager so that it will not delete corresponding C source files. The traceback dump is of minimal utility and is only reported when a run-time bug is caught by the run-time system (an application must be compiled with the traceback option enabled, and the application must be properly instrumented).

4.6 Graphics Support

Graphics support can come in different forms: a user interface toolkit, a user interface builder, a structured graphics toolkit, data visualization software, etc. Simulation developers, like most software developers, want a combination of all these capabilities.

The ModSim system as initially delivered to the Army provided no graphics or data visualization support. Since that time, two libraries were developed (one by the Army and the other developed here at the Laboratory) to address the graphics issue. Both packages provide minimal support for structured graphics (e.g., points, lines, circles, text) and fall far short of the needs of simulation developers. However, since ModSim interfaces to C in a fairly straightforward manner, existing packages — that are in C or that easily interface to C — might be utilized. The problem then becomes just how tightly can the graphics objects be coupled to simulation objects; in other words, an application might want to recognize a mouse click on a graphics object and map it into a reference to the corresponding simulation object (for browsing the object's state perhaps).

5 PARALLEL, DISCRETE-EVENT SIMULATION

The need for significant computer performance improvements has long been recognized in many application domains. This ever increasing demand for computational power has created wide-spread interest in exploiting parallelism in both hardware and software.

Within the modeling and simulation community, this has spurred research in the area of parallel (or, in the traditional nomenclature, distributed) simulation.

Object-oriented discrete-event simulation development naturally results in a simulation model that is comprised of a set of actors (or agents) which interact via the scheduling of events. This corresponds closely to the formulation of parallelism in discrete-event simulations as presented by Chandy and Misra 1981 (see also Fujimoto 1990). In this model of parallel simulation, the traditional global simulation clock and global event queue are eliminated in favor of analogous entities associated with each simulation actor. Thus, an actor may, as dictated by its own behavior, be at a different local, logical simulation time than any other actor in the simulation. Clearly, the association of logical simulation clocks with individual actors defines a set of partial orderings of events with respect to the set of simulation actors (Lamport 1978). This treatment of time provides additional opportunity for parallel execution amongst the simulation actors. The outstanding requirement is therefore the establishment of a total ordering over the entire set of events such that the behavior of the system as a whole is correct (i.e., preserves the correct event causality).

Much of the research in parallel simulation has concentrated on effectively establishing this ordering in the face of actor-level parallelism. Viewed in a slightly different context, the basic issue is the synchronization of actor behaviors, given that actors interact via messages and that each actor may, at an arbitrary point in the execution of the simulation, be operating at a different point along the continuum representing simulated time. This synchronization has been traditionally approached using two different strategies, generally classified as either conservative or optimistic.

The conservative approach is best characterized by the work of Chandy and Misra 1981. A basic tenet of the conservative approach is that no actor process a message until it can be guaranteed that no message with an earlier time stamp will be received. This approach toward synchronization ensures that a simulation actor never inadvertently processes events in incorrect order (i.e., out of time sequence), but does pose a number of technical difficulties that have been examined in depth by others (e.g., Wagner, Lazowska and Bershad 1988).

Optimistic schemes incorporate the notion of virtual time and are exemplified by systems such as Time Warp (Jefferson 1985) and Moving Time Window (Sokal, Briscoe and Wieland 1989). The basic notion employed in these approaches is that all actors progress as dictated by the events queued for them at any point in the simulation execution. Only when an message with a time stamp earlier than the current logical simulation time is processed by an actor is any synchronization required. This synchronization causes the actor to rollback its simulated time to receive the message. This rollback initiates a restoration of local state equivalent to the actor's state at the time stamp of the message. Further, this causes the retraction of all events generated by the actor subsequent to the time stamp of the message initiating the rollback. Since the recall of these events may also force other actors to rollback, rollbacks potentially cascade through the actors comprising a system as the simulation is re-synchronized. Military combat models differ from many traditional simulation application areas (e.g., circuit simulation) in that any of the entities may potentially interact and that these interactions are very difficult to predict. The highly dynamic nature of these problems thus precludes the static mapping or determination of simulation actor interaction. Thus, in the general case, the conservative approach can not be effectively utilized in this class of problem.

6 THE TIME WARP OPERATING SYSTEM

The Time Warp Operating System version 2.4.1 developed by JPL (Hontalas et al. 1990) is a complete implementation of the Time Warp mechanism. As discussed earlier, Time Warp is a distributed protocol for virtual-time synchronization based on process rollback and message annihilation. TWOS is a special-purpose operating system designed to support parallel, discrete-event simulation and other computations that can be expressed as a system of logical processes that interact through time stamped messages. In recent literature, several studies have reported performance improvements for applications utilizing the Time Warp mechanism in general (see Fujimoto 1990) and the TWOS system in particular (Ebling et al. 1989, Hontalas et al. 1989, Presley et al. 1989, and Wieland et al. 1989). JPL's Time Warp system runs a single simulation at a time concurrently on all allocated processors in a distributed computer system. According to Hontalas 1990, computer systems currently supported include Sun-3 and Sun-4 workstations, the BBN Butterfly GP-1000, the CalTech/JPL Mark III Hypercube, and the Transputer-based Definicon DSI-T4. In a combined effort with JPL, we also ported TWOS (version 2.4.1) to our BBN Butterfly TC-2000. A detailed discussion of TWOS primitives (e.g., message send and receive) is beyond the scope of this paper. In the remainder of this section we will discuss Time Warp application development and performance issues; in particular, there is an obvious though not well understood trade-off between software "manageability" and performance.

In Time Warp, an application must be decomposed

into a system of logical processes that interact through time stamped messages (for a complete discussion of Time Warp theory see Jefferson 1985). To use TWOS, in particular, an application must be written using the C programming language. In constructing a C/TWOS application, general parallel programming guidelines must be followed such as selecting the appropriate granularity and maximizing parallelism. In addition, the programmer must be intimately familiar with the synchronization mechanism in order to maximize performance. As one might expect, code designed to run well on TWOS is often fragile and difficult to modify and maintain (Fujimoto 1990). This last point must be taken in the proper context. For instance, it may not be important for an application to be easily modified and maintained; once the application is fine tuned for performance it may not be necessary to touch it again. In practice, however, software systems (especially simulations) tend to be dynamic and evolve over time (perhaps to satisfy new or updated requirements, for example). In this case, the modifiability and maintainability of the software system becomes important. What is not well understood is just how to balance performance and software quality in a Time Warp application.

Time Warp (and related) research is ongoing. In particular, TWOS research continues at JPL with memory and performance related issues dominating. Open issues in Time Warp and related research include hardware approaches to the memory intensive problem of state-saving and rollback, virtual memory support for very large applications, exception and interrupt handling, and support for real-time applications as well as the application development and performance issues we mentioned earlier.

7 MODSIM/TWOS

At the beginning of this paper we discussed a couple of basic requirements for an "ideal" simulation environment that would support large-scale simulation development for analysis and production use, namely the ability to manage the simulation software and provide acceptable response time during analysis. A solution to this problem is an environment that combines a high-level simulation language that embodies well-known software engineering principles with a high-performance parallel run-time system. The inherent difficulty with providing such an environment is the problem of mapping a simulation application to the parallel run-time system. To use a system like TWOS effectively this mapping must be carefully completed by hand; to reiterate, this is accomplished in JPL's Time Warp system by writing the application in C and utilizing TWOS primitives. For large applications that must be flexible to changes in requirements, this approach is neither desirable nor practically feasible. The coupling of ModSim and TWOS is an attempt to address these very issues. The ModSim/TWOS system does indeed provide an automatic mapping from a ModSim application to an equivalent Time Warp application in C. However, the major flaw with the ModSim/TWOS system as it currently exists is that there is no compiler support for mapping a ModSim application into an *efficient* C/TWOS application.

The current approach taken in ModSim/TWOS is to map all ModSim objects to TWOS logical processes (also called TWOS objects). At first blush, this might seem like a reasonable solution. The problem, however, is that a well designed object-oriented application in ModSim is not necessarily a well designed TWOS application. An object-oriented design typically results in relatively fine-grained object definitions from which the application is composed. A TWOS-oriented design typically results in large-grained object definitions. There are some elements of object-orientedness apparent in TWOS applications, but to a large extent, these applications must be designed with a different set of decomposition and encapsulation rules than a typical object-oriented program (Booch 1991). For example, in designing the terrain component of the Parallel Eagle model we initially used a straight-forward object-oriented decomposition. At the time of the design and implementation of terrain the ModSim system supported a granularity control mechanism. This mechanism allowed objects to be identified as either *process* or *simple* objects; process objects would be mapped to TWOS objects and simple objects would not. The actual implementation of such a mechanism is difficult and this approach was abandoned by CACI in favor of the current one-to-one mapping approach. Early on in the terrain design (Rich and Michelsen 1989), it was our intention that only a few object types would be identified as process objects and hence TWOS object instances would number in the hundreds. Without the granularity control mechanism initially in ModSim, a small terrain data set results in instances of TWOS objects numbering in the tens of thousands; quite a task for any Time Warp system. The simple fact is that we were never able to successfully run any part of the Parallel Eagle model in ModSim/TWOS.

A report prepared by Katherine Morse while at CACI demonstrated that speedup could be achieved by using ModSim/TWOS (Morse 1989b). The application benchmarked was under one thousand lines of ModSim source code and its design was TWOS-oriented; that is, the object definitions were large-grained and instances manipulated record structures internally and

communicated with other objects infrequently. Furthermore, the number of TWOS object instances was in the hundreds.

8 DISCUSSION

8.1 Summary

Both ModSim and TWOS have their strengths and weaknesses when viewed independently. The ModSim language system is a generally usable object-oriented simulation environment that benefits from the integration of an object-oriented approach within a modular, process-based simulation framework. The major weakness of the ModSim language system is that it suffers from inconsistencies and oversights in its current realization. The Time Warp Operating System has been demonstrated to be a viable vehicle for doing parallel, discrete-event simulation; performance studies are encouraging. Many issues, however, are still open in the area of Time Warp research and TWOS, in particular, is still experimental. The combination of ModSim and TWOS is notionally a good idea. Yet the failure to generate a workable, comprehensive mapping of the ModSim execution model to the Time Warp execution model is a debilitating deficiency. Without sufficient compiler support there is a mismatch between these two execution models. The current ModSim/TWOS system does not provide this kind of compiler support.

8.2 Continuing And Related Work

There is an ongoing research effort to redesign the ModSim/TWOS system. This effort is sponsored by the Army and is currently underway at Jade Simulations in Calgary, Canada. The initial phase of Jade's effort has been completed (Baezner 1991) and the redesign phase is underway. ModSim language issues are being addressed concurrently with the ModSim/TWOS redesign.

Jade Simulations also developed and marketed Sim++ (Jade 1990), a process-oriented, discrete-event simulation embedded in the object-oriented programming language C++. Sim++ is specially designed so that programs are able to execute sequentially in a single processor environment or in parallel in a multiple processor environment using Jade's proprietary implementation of the Time Warp mechanism. Like ModSim/TWOS, Sim++ does not provide an automatic mapping from a simulation application to an efficient Time Warp application; Sim++ programs must be carefully designed for maximum performance (not unlike C/TWOS applications).

Finally, CACI is continuing its development of MODSIM II. Proposed enhancements to the MODSIM II system include an implementation of parameterized types and support for visual programming (Duncan 1990). We are unaware of any development work on a parallel processing version of MODSIM II.

ACKNOWLEDGMENTS

Thanks to Dennis Powell and Fred Weber for their valuable comments on earlier versions of this paper. This work was sponsored by the U.S. Army Model Improvement and Study Management Agency (MISMA) under contract DPG-880108 and performed under the auspices of the United States Department of Energy (DOE) as part of the DOE contract W-7405-ENG-36.

For information about Eagle contact Dr. Robert LaRocque, Director, Operations Analysis Center, TRADOC Analysis Command, Ft. Leavenworth, Kansas.

REFERENCES

Baezner, D. 1991. Preliminary Assessment of ModSim. Draft Report. Jade Simulations, Calgary, Canada.

Belanger, R., B. Donovan and K. Morse. 1989a. ModSim User's Manual (Release 1.0). CACI Products, La Jolla, California.

Belanger, R., S. Rice, B. Donovan and K. Morse. 1989b. ModSim Reference Manual (Release 1.0). CACI Products, La Jolla, California.

Booch, G. 1991. *Object-Oriented Design with Applications.* California: Benjamin/Cummings Publishing Company, Inc.

Bryan, O. and R. Belanger. 1989. MODSIM II — An Object-Oriented Language for Sequential and Parallel Processors. Draft Report. CACI Products, La Jolla, California.

Chandy, K. and J. Misra. 1981. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM* 24, 11 (November): 198-205.

Duncan, H. 1990. Personal Communication. CACI Products, La Jolla, California.

Ebling, M., M. DiLorento, M. Presley, F. Wieland and D. Jefferson. 1989. An Ant Foraging Model Implemented on the Time Warp Operating System. In *Proceedings of the SCS Multiconference on Distributed Simulation* 21, 2 (March): 21-26.

Ellis, M. and B. Stroustrup. 1990. *The Annotated C++ Reference Manual.* Massachusetts: Addison Wesley

Publishing Company.

Fujimoto, R. 1990. Parallel Discrete Event Simulation. *Communications of the ACM* 33, 10 (October): 30-53.

Hontalas, P., B. Beckman, M. DiLorento, L. Blume, P. Reiher, K. Sturdevant, L. Van Warren, J. Wedel, F. Wieland and D. Jefferson. 1989. Performance of the Colliding Pucks Simulation on the Time Warp Operating System: Part 1 (Asynchronous Behavior and Sectoring). In *Proceedings of the SCS Multiconference on Distributed Simulation* 21, 2 (March): 3-7.

Hontalas, P., D. Jefferson and M. Presley. 1990. Time Warp Operating System Version 2.4.1. User's Manual JPL D-6493 (Revision A). Jet Propulsion Laboratory, Pasadena, California.

Jade Simulations. 1990. Sim++: A Discrete-Event Simulation Language. Programmer Reference Manual (Release 3.2). Jade Simulations, Calgary, Canada.

Jefferson, D. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July): 404-425.

Jefferson, D., B. Beckman, F. Wieland, L. Blume, M. DiLorento, P. Hontalas, P. Reiher, K. Sturdevant, J. Tupman, J. Wedel and H. Younger. 1987. The Time Warp Operating System. In *Proceedings of the 11th Symposium on Operating Systems Principles* 21, 5 (November): 77-93.

Lamport, L. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21, 7 (July): 558-565.

Lomow, G., J. Cleary, B. Unger and D. West. 1988. A Performance Study of Time Warp. In *Proceedings of the SCS Multiconference on Distributed Simulation* 19, 3 (July): 50-55.

Maisel, H. and G. Gnugnoli. 1972. *Simulation of Discrete Stochastic Systems*. SRA.

Meyer, B. 1988. *Object-Oriented Software Construction*. New York: Prentice Hall International.

Morse, K. 1989a. Time Warp Addendum to ModSim User's Manual (Release 1.0). CACI Products, La Jolla, California.

Morse, K. 1989b. Parallel Distributed Simulation in ModSim. Draft Report. CACI Products, La Jolla, California.

Mullarney, A. and J. West. 1987. ModSim: A Language for Object-Oriented Simulation. Design Specification. CACI Products, La Jolla, California.

Mullarney, A., J. West, R. Belanger and S. Rice. 1989. ModSim Tutorial (Release 1.0). CACI Products, La Jolla, California.

Powell, D. 1989. Object-Oriented Terrain Analysis. Technical Report, LA-UR-89-3665. Los Alamos National Laboratory, Los Alamos, New Mexico.

Presley, M., M. Ebling, F. Wieland and D. Jefferson. 1989. Benchmarking the Time Warp Operating System with a Computer Network Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation* 21, 2 (March): 8-13.

Rich, D. and R. Michelsen. 1989. Writing Parallel, Discrete-Event Simulations in ModSim: Insight and Experience. Technical Report, LA-UR-89-3104. Los Alamos National Laboratory, Los Alamos, New Mexico.

Sokal, L., D. Briscoe and A. Wieland. 1988. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution. In *Proceedings of the SCS Multiconference on Distributed Simulation* 19, 3 (July): 34-42.

Wagner, D., D. Lazowska and B. Bershad. 1988. Techniques for Efficient Shared-Memory Parallel Simulation. Technical Report 88-04-05. University of Washington, Seattle, Washington.

West, J. 1985. Object-Oriented Distributed Simulation. Technical Report. CACI Products, La Jolla, California.

Wieland, F., L. Hawley, A. Feinberg, M. DiLorento, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot and D. Jefferson. 1989. Distributed Combat Simulation and Time Warp: The Model and its Performance. In *Proceedings of the SCS Multiconference on Distributed Simulation* 21, 2 (March): 14-20.

AUTHOR BIOGRAPHIES

DAVID O. RICH is a staff member in the Analysis and Assessment Division at Los Alamos National Laboratory. His interests include, object-oriented software engineering, distributed computing and simulation. He is a member of the ACM and IEEE Computer Society. Author's present address: Military Systems Analysis, MS F602, Los Alamos National Laboratory, Los Alamos, NM 87545 (dor@lanl.gov).

RANDY E. MICHELSEN is a staff member in the Analysis and Assessment Division at Los Alamos National Laboratory. His interests include software engineering, object-oriented software development and parallel computing. He is a member of the ACM and IEEE Computer Society. Author's present address: Simulation Applications, MS F606, Los Alamos National Laboratory, Los Alamos, NM 87545 (rem@lanl.gov).