USING IMPORT TO DEVELOP WARGAMES

Jeffrey W. Wallace, Guy E. Leonard, Larry J. Peterson, Alex Vagus, Carol L. Kropp
Space and Naval Warfare Command Systems Center San Diego
San Diego, CA
{jwallace | gleonard | ljp | kropp}@spawar.navy.mil

Jay F. Caldwell
Advanced Telecommunications, Inc.
San Diego, CA
caldwell@spawar.navy.mil

ABSTRACT

Wargaming systems developers are facing the problem of how to design and implement systems comprised of existing and new simulators which run with multiple levels of simulation resolutions, using heterogeneous computing facilities, including high performance computers. Several technical issues confront these architects, designers, and developers:

1. Complex Interactions
2. Scalability
3. Cognitive Process Modeling

This paper will address how these issues are being addressed by United States Department of Defense (DoD) scientists responsible for wargaming systems development.

INTRODUCTION

Two decades of DoD wargaming simulation development research has led to the development of the Integrated Modular Persistent Objects and Relations Technology (IMPORT) computer language [Whitehurst 95]. IMPORT has been used by the US Army and the US Navy for the rapid and successful construction of complex simulation systems. DoD's High Performance Computing (HPC) Modernization Program has tasked personnel at the Space and Naval Warfare Command to extend IMPORT to operate on parallel high performance computers, and networks of such computers. In addition, it must be delivered to the HPC community as an end user product for development of simulation and wargaming systems. IMPORT is a fifth generation language (5GL) tailored to develop large-scale, complex simulations using object-oriented technology.

The advantage to using IMPORT in wargame development stems from its ability to solve several daunting technical challenges facing wargame developers. These issues are:

Interaction Complexity - the ability to permit clear, concise implementations among a diverse set of models which represent some significant portion of the real world.
Scalability - the ability to efficiently execute over a wide range of simulation sizes, e.g. from 10 - 10^6 active simulation objects
Cognitive Process Modeling - provide support for the modeling of human decision making.

COMPLEX INTERACTIONS

The solution to the problem of interaction complexity in IMPORT is to provide a set of simulation abstractions for the developer which are keywords in the language. These keywords permit a very direct mapping between dynamic object-oriented analysis (OOA) models and the subsequent source code representation.

The keywords in IMPORT which represent simulation abstractions are:

1) WAIT - Basic simulation primitive, passes simulation time.

2) ASK - One object invokes a method on another object. If *Object1* and *Object2* are not simulation objects, then no simulation code is generated. If they are, and we have the following situation

```
Object1 = OBJECT
...
ASK Object2 TO DoSomething();
...
END OBJECT;
```

Object1 sends an event to *Object2* which will invoke the **DoSomething** method on *Object2*. *Object2* then sends an event back to *Object1* to continue execution immediately following the method call on *Object2.*

3) FUTURES/WAITFOR - A variable of type FUTURE suspends execution until the FUTURE is set, like a semaphore, except that time is managed.

4) INTERRUPT - Used to retract pending events and facilitate the implementation of state diagrams.

```
WAIT(some_amount_of_time)
   ...
   ON INTERRUPT
      WHEN interrupt_tag1:
      ...
      WHEN interrupt_tag2:
      ...
END WAIT;

WAITFOR(some_FUTURE)
   ...
   ON INTERRUPT
      WHEN interrupt_tag1:
      ...
      WHEN interrupt_tag2:
      ...
END WAIT;
```

This requires a function, `INTERRUPT`, to resume, or activate, the WAIT or WAITFOR. The time at which the interrupt occurs is now the SIMTIME for the object whose WAIT or WAITFOR has been interrupted. In the exception clauses one can now transition into other states. Typically interrupts are used to indicate that change in another part of the system under study has changed, and a reaction to this change must be determined.

5) TELL - This syntax is used to spawn an asynchronous method, or process as follows: `TELL Object1 TO Evolve();`. This spawns an ASK method asynchronously.

The OOA fragments in Figure 1 and Figure 2 depict the basic concept which is the abstraction from which ships and other waterborne vehicles and vessels derive, and this class is called the Water Movement Platform (*Water_MP*).

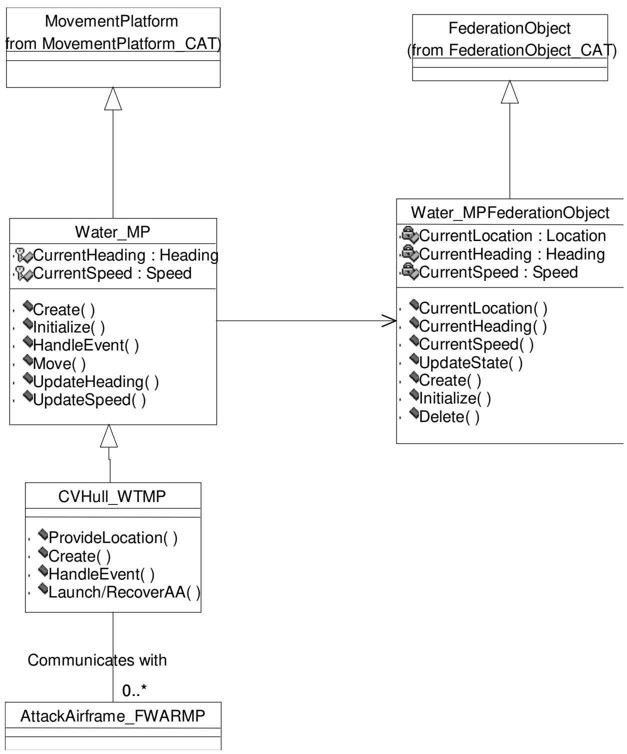

Figure 1. Simplified Water Movement Platform class hierarchy.

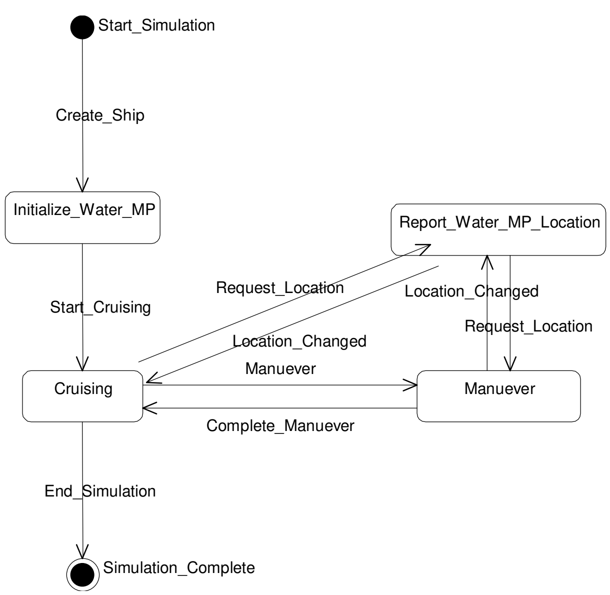

Figure 2. OOA State Diagram for the Water Movement Platform.

The use of design patterns and frameworks is crucial to elegant and easy to understand implementations. In the examples to follow we will utilize the Model-View-Controller (MVC) paradigm and design pattern [Leonard 97] to implement inter-object interactions in a way that maximizes model re-use and leads to comprehensible and maintainable implementations.

The MVC pattern implementation in IMPORT provides us with a clear implementation of cause-and-effect relationships. The *ModelObj* class, shown below, has two primary features. The first is a collection of *Dependents*, which provides the basic unit of synchronous coordination between input stimuli and output response. The second, the **Change** method, is invoked whenever the *ModelObj* changes state in some significant manner. The list of *Dependents* is looped, and their *ChangeState* synchronization semaphores are released when the **SetChangeState** methods are called.

```
ModelObj = OBJECT

      Dependents : ARRAY[MAXDEP] OF Dependent;
      NumDependents : INTEGER;

EXPORT ASK METHOD Change()
VAR
   I : INTEGER;
BEGIN
   IF NumDependents > 0 THEN
      FOR I := 0 TO NumDependents-1 DO
          ASK Dependents[I] TO SetChangeState();
      END FOR;
   END IF;
END METHOD;
END OBJECT;
```

The basic *View* class has a method **ActivelyView**, which get spawned as a thread which waits on a semaphore *ChangeState* to be set. The key elements of the base class View are:

```
View = OBJECT(Dependent)

      Model : ModelObj;
      superView : View;
      subViews : ARRAY[MaxView] OF View;
      numSubViews : INTEGER;
      viewID : INTEGER;

EXPORT ASK METHOD ActivelyView()
BEGIN
WHILE( NOT ASK SELF TO GetDone() ) DO
      stdout()<<"waiting in view at time =
"<<simtime()<<"\n";
      {EXTRACT STATE FROM MODELS HERE}
   ASK SELF TO WaitForChange();
   ASK SELF TO ResetChangeState();
END WHILE;
END METHOD;

EXPORT ASK METHOD WaitForChange()
BEGIN
   waitFor(ChangeState);
END METHOD;
END OBJECT;
```

The process **ActivelyView** suspends execution at the statement

```
ASK SELF TO WaitForChange();
```

since it is invoked synchronously. Examination of the method **WaitForChange** shows the semaphore suspend operation

```
waitFor(ChangeState);
```

as we described above.

The implementation of the water movement platform class is specified in the following code listing. The **Move** method implemented below and shown in Figure 1 above implements the intent of the state diagram illustrated in Figure 2.

The **Move** method models motion using initial conditions set at the time of the last state change and a specified equation of motion. In the example above, the initial conditions are heading and speed, a point in Euclidean space, and the equation of motion is a line. The **Move** method responds to methods such as **SetNewHeading**. Typically, **SetNewHeading** is called from a Controller class. A new heading is loaded into the *CommandHeading* variable, and the new command to change heading is set. The **Move** method is then activated by the `set(NewCommand)` invocation. The *NewCommand* instance variable, of type FUTURE, also known as a simple FUTURE, is a time managed semaphore. This means execution is suspended until the semaphore is set, and simulation time is recorded automatically when the FUTURE is set, and execution resumes immediately following the FUTURE. The full features of the FUTURE concept are beyond the scope of the discussion here.

```
CommandType = (ChangeHeading, ChangeSpeed, Other);

Water_MP = OBJECT(MovementPlatform,ModelObj)
   CurrentHeading, CurrentSpeed, X,Y,Z : REAL;
   LastChangeTime : INTEGER;
   CommandHeading: REAL;
   CommandSpeed:   REAL;
   NewCommand : FUTURE;
   Command : CommandType;

EXPORT ASK METHOD Move()
BEGIN
   waitFor(NewCommand);
   reset(NewCommand);

   CASE Command
      WHEN ChangeSpeed:
         ASK SELF TO Update();
         ASK SELF TO UpdateSpeed();
         ASK SELF TO Change();
      WHEN ChangeHeading:
         ASK SELF TO Update();
         ASK SELF TO UpdateHeading();
         ASK SELF TO Change();
      WHEN Other:
         ASK SELF TO Update();
         ASK SELF TO Change();
   END CASE;
   TELL SELF TO Move();
END METHOD;

EXPORT ASK METHOD ResetNewCommand()
BEGIN
   reset(NewCommand);
END METHOD;

EXPORT ASK METHOD Update()
VAR
   NewX, NewY : REAL;
BEGIN
   NewX := ASK SELF TO GetX() +
   CurrentSpeed*cos(CurrentHeading)*(simtime() -
   LastChangeTime);
   NewY := ASK SELF TO GetY() +
   CurrentSpeed*sin(CurrentHeading)*(simtime() -
   LastChangeTime);
   LastChangeTime := simtime();
   ASK SELF TO UpdateLocation(NewX, NewY, 0.0);
END METHOD;
EXPORT ASK METHOD SetNewHeading(IN heading : REAL)
BEGIN
   CommandHeading := heading;
   Command := ChangeHeading;
   set(NewCommand);
END METHOD;
EXPORT ASK METHOD UpdateHeading()
BEGIN
   CurrentHeading := CommandHeading;
END METHOD;
END OBJECT;
```

In order to provide a continuous view of this discrete event simulation class, and ultimately interact with the Department of Defense (DoD) High Level Architecture (HLA) Run Time Infrastructure (RTI) and other simulation frameworks, we introduce the concept of the *Federation Object* (FO) class. This is sometimes referred to as an object proxy [Steinman 98]. For purposes of illustration, our FOs will contain the essential state variables and the equation of motion of the associated class, and demonstrate their utility in providing a continuous view of a discrete event simulation. Thus, the key code fragments of a *Water_MPFO* would be:

```
Water_MPFO = OBJECT
   X,Y,Z : REAL;
   CurrentHeading, CurrentSpeed : REAL;
   LastChangeTime : INTEGER;
EXPORT ASK METHOD Update(IN x,y,z,heading,speed :
REAL)
BEGIN
   X := x;
   Y := y;
   Z := z;
   CurrentHeading := heading;
   CurrentSpeed := speed;
   LastChangeTime := SIMTIME();
END METHOD;
EXPORT ASK METHOD GetLocation(OUT x,y,z : REAL)
BEGIN
   x := X +
CurrentSpeed*cos(CurrentHeading)*(simtime() -
LastChangeTime);
   y := Y +
CurrentSpeed*sin(CurrentHeading)*(simtime() -
LastChangeTime);
   z := 0.0;
END METHOD;
END OBJECT;
```

A view class can now be created, call it the *FOMView* class, which is a dependent of the *Water_MP*. When the *Water_MP* changes, and invokes its **Change** method, the FUTURE *ChangeState* in the *FOMView* class is set. The *FOMView* process **ActivelyView**, illustrated in the code below, resumes execution, extracts the relevant state data from the *Water_MP*, by calling the methods **GetLocation**, **GetHeading**, and **GetSpeed**. We then invoke the *Water_MPFO* method **Update** to load the updated state information into the FO. A code fragment from the *FOMView* class which implements this is:

```
FOMView = OBJECT(View)

    MP : MovementPlatform;
    MPFO : MovementPlatformFO;

EXPORT ASK METHOD ActivelyView()
VAR
   x,y,z : REAL;
BEGIN
   WHILE( NOT ASK SELF TO GetDone() ) DO
      ASK MP TO GetLocation(x,y,z);
      ASK MPFO TO Update(x,y,z,
                         ASK MP TO GetHeading(),
                         ASK MP TO GetSpeed());
```

```
      ASK SELF TO WaitForChange();
      ASK SELF TO ResetChangeState();
   END WHILE;
END METHOD;
END OBJECT;
```

Reactive Simulation using MVC Networks

One of the difficult problems in complex system simulation is a realistic portrayal of cause-and-effect relationships, which is manifested in the ability of simulation objects to react to system state changes. To demonstrate the effectiveness of IMPORT in implementing reactive simulations, we solve one of the classical problems in discrete simulation easily in the following example. Consider the case of an airplane tracking and intercepting a *Water_MP*. In event-based systems, the intersection is calculated and scheduled. In stepped simulations, at some point in time intersections are tested nearly every step.

The classic problem occurs when the *Water_MP* changes its motion while the airplane is tracking it. In conventional event-based simulation, quite a bit of condition checking and design effort must be employed to ensure correct functionality of the implementation, let alone the clarity of the solution. Process-based simulation primitives, such as those found in IMPORT and MODSIM by CACI [CACI 89], however, lead to logical, correct, easy to understand, and cost-effective to maintain implementations of such common, yet nontrivial, behaviors.

To solve this problem, consider the *Air_MP* class, whose partial implementation is listed below. The *Air_MP* class is patterned after the *Water_MP* presented earlier. The *Air_MP* class is equipped with a mechanism to intercept, or track, another movement platform. Note that we have added a prototype synthetic natural environment (SNE) representation, and added a command set and instance variables which allow a movement platform to be tracked and intercepted.

```
Air_MPCommandType =
(TrackMP,TrackedMPChanged,OtherAir_MP);

Air_MP = OBJECT(MovementPlatform)
   Air_MPCommand : Air_MPCommandType;
   CurrentMPTarget : MovementPlatformFO;
   Ocean : OceanFO;
   Ground : GroundFO;
   Atmosphere : AtmosphereFO;
   TimeToIntercept : REAL;
   Tracking : BOOLEAN;
ASK METHOD Move()
BEGIN
   ASK SELF TO WaitForNewCommand();
   ASK SELF TO ResetNewCommand();
   CASE ASK SELF TO GetCommand()
     WHEN ChangeSpeed:
        ASK SELF TO Update();
        ASK SELF TO UpdateSpeed();
        ASK SELF TO Change();
     WHEN ChangeHeading:
        ASK SELF TO Update();
        ASK SELF TO UpdateHeading();
        ASK SELF TO Change();
     WHEN Other:
       CASE Air_MPCommand
          WHEN TrackMP:
               TELL SELF TO InterceptMP();
          WHEN TrackedMPChanged:
               TELL SELF TO InterceptMP();
          WHEN OtherAir_MP:
               ASK SELF TO Update();
               ASK SELF TO Change();
       END CASE;
   END CASE;
   TELL SELF TO Move();
END METHOD;
```

The **Move** process of the *Air_MP* class, is very similar to the *Water_MP* class presented earlier. In this case, however, in addition to the inherited command set *ChangeHeading*, *ChangeSpeed*, and *Other*, we add a set of *Air_MP* commands*, TrackMP*, *TrackedMPChanged*, and *OtherAir_MP*.

The **InterceptMP** process is designed to calculate an interception course for *the MovementPlatform (MP)* being tracked, given that the *MP* equation of motion is available via the *MP* FO. One the interception path is registered, in this case the heading is required, the interception event is scheduled in the `WAIT(TimeToIntercept);` statement in the code listing below. If the MP being tracked changes course before the interception event occurs, a calling mechansim must be established and the following command issued:

```
INTERRUPT(AMPref,"InterceptMP",TrackedMPChanged);
```

This causes the **InterceptMP** process to resume execution in the `ON INTERRUPT` clause immediately following the `WHEN TrackedMPChanged:` statement in the code below. This causes the **InterceptMP** process to be restarted to recalculate the interception path and time and reschedule the interception event. The statement `TELL SELF TO InterceptMP();` accomplishes this task.

```
ASK METHOD InterceptMP()
VAR
   Heading : REAL;
BEGIN
   Tracking := TRUE;
   ASK SELF TO GetInterceptData(Heading,
                                TimeToIntercept);
   ASK SELF TO Update();
   ASK SELF TO SetHeading(Heading);
   ASK SELF TO UpdateHeading();
   ASK SELF TO Change();
   WAIT(TimeToIntercept)
      STDOUT()<<"Intercepted Target";
      ON INTERRUPT
         WHEN TrackedMPChanged:
            TELL SELF TO InterceptMP();
         WHEN DifferentTarget:
            TELL SELF TO InterceptMP();
         OTHERWISE:
            ASK SELF TO Update();
            ASK SELF TO Change();
            TELL SELF TO Move();
   END WAIT;
END METHOD;
```

The *AMPController* class, which controls *Air_MP* objects, is designed to have the access to an *Air_MP* so that it can dynamically interrupt and redirect, or have execution resume in, the *Air_MP*. This is illustrated in the following code inside the *AMPController*'s **ActivelyControl** process:

```
AMPController = OBJECT(Controller)

   TModel : Air_MP;

EXPORT ASK METHOD ActivelyControl()
VAR
   heading, speed, sign : REAL;
BEGIN
      WAIT(9);
      stdout()<<"Input a new state: \n";
      ASK TModel TO SetTrackMP();
END METHOD

EXPORT ASK METHOD TrackedMPChange()
BEGIN
 INTERRUPT(TModel,"InterceptMP",TrackedMPChanged);
END METHOD;
```

When the **ActivelyControl** process is spawned, nine seconds elapse, due to the `WAIT(9);` statement. The **Move** process is then interrupted, or resumed, by invoking the:

```
ASK TModel TO SetTrackMP();
```

statement. This invocation sets the command to be executed , the *Air_MPCommand*, to *TrackMP* and sets the *Command* variable inherited from *MovementPlatform* to *Other*, and sets the *NewCommand* FUTURE, which will activate the **Move** process. The **Move** process then resumes execution immediately following the **WaitForNewCommand** method. The process execution then traverses the CASE structure through the *Other* branch, and unifies the *Air_MPCommand* with *TrackMP*. This causes the **InterceptMP** process to be spawned.

To complete the example, we need to demonstrate how to establish the *View-Controller* linkage required to complete the feedback loop. The *AMPController* class has the **TrackedMPChange** method, which contains the appropriate interrupt command. We need only define a *View* class which attaches to the *MP* being tracked and has a reference to the *AMPController*, so that it may invoke the *AMPController*'s **TrackedMPChange** method, which then interrupts the *Air_MP*'s **InterceptMP** process, and cancels the interception event which is no longer valid. This cancellation causes the **InterceptMP** process to be restarted. The restarting of the **InterceptMP** process causes a new interception path to be calculated and a new interception event to be scheduled, as desired.

The following code describes the *MPTrackView* class, which has an **ActivelyView** process which monitors *MP* change and propagates that change via a powerful, yet easy to understand event retraction capability using interrupts.

```
MPTrackView = OBJECT(View)
    ampController : AMPController;
EXPORT ASK METHOD ActivelyView()
BEGIN
   WHILE( NOT ASK SELF TO GetDone() ) DO
      ASK SELF TO WaitForChange();
      ASK SELF TO ResetChangeState();
      ASK ampController TO TrackedMPChange();
   END WHILE;
END METHOD;
END OBJECT;
```

SCALABILITY

The IMPORT CASE tool is used to generate object-oriented simulation frameworks from the specification language demonstrated above. The resulting scalability of an IMPORT simulation is tied directly to the scalability of the underlying simulation framework. The original simulation framework generated by IMPORT [Whitehurst 95], the QuickThreads Simulation Runtime (QTSR), is based on the QuickThreads thread package [Keppel 93], and optimized for execution on single CPU, low-cost PCs and workstations. This version of IMPORT is currently available as a GOTS product.

The Director of Defense Research and Engineering's (DDR&E) High Performance Computing Modernization Program (HPCMP) is sponsoring the adaptations required to make IMPORT generate the SPEEDES parallel and distributed simulation framework [Steinman 98], which is also a GOTS simulation software product originally developed by NASA's Jet Propulsion Laboratory (JPL). The resulting version of IMPORT [Whitehurst 98] will soon be available as a GOTS simulation software engineering product as well.

In addition, the Joint Simulation System (JSIMS) Maritime Program will be using IMPORT to execute their implementation for objects in the Maritime domain. Complying with the JSIMS architecture will require adapting the IMPORT translator to generate the TEMPO/THEMA parallel simulation framework [West 98].

An unintended consequence of these requirements are that in addition to the original ISLE project goals of write once, run anywhere, we will soon be able to specify which simulation framework we want to link against based on the hardware architecture at hand. The time required for each framework to process an event, as communicated to us by the authors, is listed in Table 1 below. The time is in units of microseconds(10^{-6}).

Simulation Framework	Event Processing Time
QTSR	6.5
TEMPO	10
SPEEDES	10 - 25

Table 1. Simulation event processing times in microseconds.

These results are consistent with the fact that the target hardware architectures are increasingly more general. The QTSR framework is designed for single CPU execution, the TEMPO framework is designed for shared memory multiprocessors or shared memory clusters of computers, and the SPEEDES framework is designed for both parallel computers (shared memory and distributed memory computers) and networks of computers.

We have built several scalability-benchmarking simulations SPEEDES, and executed them on a 32 processor Convex Exemplar enterprise server, and we present some of these results here. The Joint National Test Facility has been rigorously benchmarking SPEEDES on networks of workstations, but these results are not available at this writing. The initial results are very promising, yielding linear scalability.

Our SPEEDES-based benchmarking results for constructive wargaming simulation for real-time C2 training are shown in Figure 3. The models were based on physics-oriented motion and sensor models from existing, operational wargames. In order to determine the range of simulation speeds available to an exercise controller, it is necessary to measure the achievable ratio of simulation time to wall clock time for a given exercise size. The ratio achieved for various entity counts and processors are plotted on the vertical axis. The horizontal axis plots the number of processors employed by the Convex Exemplar to execute the configuration. Note that the scalability is linear for light, moderate, and heavy loading.

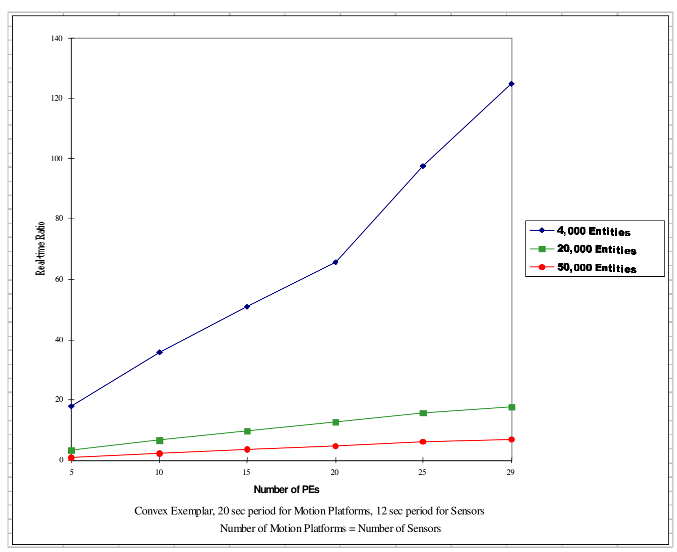

Figure 3. SPEEDES constructive wargaming simulation benchmarking results on a 32 processor Convex Exemplar.

COGNITIVE MODELING AND SIMULATION

As was noted in the development of the FSTAM wargame prototype [Wallace 91], which was implemented in IMPORT's predecessor MODSIM [CACI 89] and the logic programming extension MODLOG, separating behavior into physical and cognitive operations makes good conceptual modeling practice. In addition, this approach also scales well in terms of ease of implementation and software engineering flexibility. In wargaming simulation we are most interested in military group cognitive behaviors, but we are also concerned with non-military behavior too, i.e., the thinking behavior not only of a commander and their staff, but possibly in the trenches war-fighters and civilians as well.

The military model in FSTAM was based on the abstractions *Commander*, *Unit*, and *Asset*, and the basic design concepts are shown below modified slightly from the original MODSIM to IMPORT.

```
Commander = OBJECT (ModelObj)
   supereriors  : ARRAY[MAXSUP] OF Commander;
   subordinates : ARRAY[MAXSUB] OF Commander;
   Unit         : UnitAssociation;
END OBJECT; <-- Commander
Unit = OBJECT (ModelObj)
   Commander : CommanderAssociation;
   superUnit : Unit;
   subUnits  : ARRAY[MAXUNIT] OF Unit;
   Assets    : ARRAY[MAXASSET] OF Asset;
END OBJECT; <-- Unit
Asset = OBJECT (ModelObj)
   AssetID : INTEGER;
   AssetName : STRING;
END OBJECT; <-- Asset
```

The *Commander* class hierarchy represented the military chain of command for the scenario. The *Commander* classes established communications within the chain of command via a C2 network framework which modeled message transmission, and permitted communications bandwidth modeling. This feature allowed bandwidth to be a scarce commodity. Several economic models governing bandwidth allocation were used. *Commander* classes are associated with the *Unit* class they command. The *Unit* is a both a modeling abstraction and an implementation class to organize the *Asset* classes, the physical things both human and equipment, which comprise a military unit. The *Water_MP* and *Air_MP* classes used in our illustrations would be examples of *Assets*.

In FSTAM, the basic concepts of perception, or data collection, data fusion, assessment, planning, and plan execution were represented as both asynchronous and synchronous processes, depending on the command circumstance. The time duration for these processes was modeled so that the further one went up the chain of command, the longer the observed cycle times. As we have seen, process-based simulation primitives make the implementation of such functionality straightforward. The embedded artificial intelligence capabilities in IMPORT via the DOME subsystem [Wallace 97] are the modern counterparts of the MODLOG inference engine used in FSTAM to implement basic plan formation and plan execution, and to direct goal seeking behavior.

We are expanding on these efforts in several directions. First, we are reconstructing the cognitive architecture described above, since this was never properly exploited by the Army. To augment the native cognitive modeling capabilities in IMPORT [Wallace 97], we are developing a taxonomy of military decision-making behaviors and classifying which artificial intelligence techniques solve which behavior modeling problem the best. Along these lines, under funding from the Office of Naval Research (ONR), neural networks and fuzzy logic frameworks in IMPORT are being readied for production use in complex system simulation [Wallace 98].

FUTURE DIRECTIONS

Our primary focus in the near future is the adaptation of the IMPORT translator to generate TEMPO/THEMA frameworks. Upon completion of this task, we should be in a position to assess the design of the translator with respect to the generation of three different simulation runtime frameworks. Devising a final simulation runtime (SR) abstraction layer in the IMPORT translator to perfect the technique of generating and linking in different simulation runtimes based on the hardware we have at hand should be straightforward at that time.

The authors would like to thank the ONR 6.1 Basic Research Program, the DDR&E's High Performance Computing Modernization Program, and the JSIMS Maritime Program for their support in conducting this work

REFERENCES

[CACI 89] CACI Products Company. 1989. *MODSIM: A Language for Object-Oriented Simulation, Reference Manual, Release 1.0*, U.S. Army Contract DABT60-86-C-1382, Deliverable A-009, CACI Products Company, 1989.

[Keppel 93] Keppel D. 1993. *Tools and Techniques for Building Fast Portable Threads Packages*, University of Washington Technical Report Number UW-CSE-93-05-06.

[Leonard 97] Leonard G., L. Peterson, and J. Caldwell. 1997. "Using the Model-View-Controller Framework as a Simulation Development Methodology," In *Object-Oriented Simulation 1997*, Wallace J. *et al* Eds., Simulation Councils, Inc., January 1997.

[Selig 98] Selig M. *et al*. 1998. " An Unmanned Aerial Vehicle Simulation Framework for Training Simulation and Simulation-based Design," In *Object-Oriented Simulation 1998*, Wallace J. *et al* Eds., Simulation Councils, Inc., January 1998.

[Steinman 98] Steinman J. 1998."Time Managed Object Proxies in SPEEDES," In *Object-Oriented Simulation 1998*, Wallace J. *et al* Eds., Simulation Councils, Inc., January 1998.

[Wallace 91] Wallace J., C. Herring, R.A. Whitehurst. 1991. "A Flexible Architecture for Combat Simulations," In *Object-Oriented Simulation 1991*, Ege R. Ed., Simulation Councils, Inc., January 1991.

[Wallace 97] Wallace J. and R. A. Whitehurst. 1997. "Using IMPORT to Implement Complex Behaviors in Simulations," In *Object-Oriented Simulation 1997*, Wallace J. *et al* Eds., Simulation Councils, Inc., January 1997.

[Wallace 98] Wallace J. 1998. " Implementing Neural Networks and Fuzzy Logic Using the IMPORT Simulation Language," In *Object-Oriented Simulation 1998*, Wallace J. *et al* Eds., Simulation Councils, Inc., January 1998.

[West 98] West D. *et al*. 1998. " Event Distribution and State Sharing in the THEMA Parallel Discrete Event Simulation Modeling Framework," In *Object-Oriented Simulation 1998*, Wallace J. *et al* Eds., Simulation Councils, Inc., January 1998.

[Whitehurst 95] Whitehurst R. A. and V. P. Morrison. 1995. "IMPORT: A Language for Simulation of Complex Systems," In *Object-Oriented Simulation 1995*, Roberts C. *et al* Eds., Simulation Councils, Inc., January 1995.

[Whitehurst 98] Whitehurst R.A. *et al*. 1998. "Parallel Execution of Process-based Simulation Models," In *Object-Oriented Simulation 1998*, Wallace J. *et al* Eds., Simulation Councils, Inc., January 1998.