USING OBJECTS TO DESIGN AND BUILD RADAR ESM SYSTEMS

Brian M. Barry
John R. Altoft
Defence Research Establishment Ottawa
Ottawa, Ontario K1A 0K2
CANADA

D.A. Thomas
Mike Wilson
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6
CANADA

ABSTRACT

This paper describes the application of object-oriented programming to the design of a multiprocessor ESM testbed. The ESM testbed uses an object-oriented development environment which integrates Smalltalk and C language tools with the Harmony real-time operating system in a shared memory multiprocessor. All development for an application is done using personal computers which are themselves processors in the real-time testbed. We first discuss two aspects of the ESM testbed: a framework for investigating ESM signal processing algorithms based on an object-oriented emitter database and blackboard objects which implement probabilistic reasoning; and an object-oriented ESM simulation environment which illustrates the use of object-oriented techniques for the development of complex real-time systems. In the second part of the paper we describe our software engineering approach and tools. Throughout the paper, the role played by object-oriented programming in the design of hybrid multiprocessors is highlighted.

1.0 INTRODUCTION

An Electronic Support Measures (ESM) system is used to receive and identify signals from various sources. Using this data it is possible to construct an accurate and timely picture of the current tactical situation. All of this information will be displayed for an operator and/or transmitted to other systems. In this paper we describe a testbed for prototyping new signal processors intended for naval ESM applications.

Object-oriented techniques play a central role in both the design and implementation of this system. Signal processing algorithms and control software are being implemented in a highly modular, anthropomorphic style [Gentleman 81 85, Thomas 85 87] inspired by the Actor model of programming [Hewitt 77]. An integrated development and test environment based on object-oriented languages provides software support for program design and generation, as well as performance monitoring and debugging. A Smalltalk based simulation environment has been developed to support the design process. ESM system libraries are being redesigned to use emitter objects rather than conventional data records. Signal analysis is based on interactions between data objects, blackboard objects (which hold the current status of analysis and inferencing procedures), and the ESM libraries. Objects provide the mechanism for combining these diverse elements into a single integrated package.

2.0 THE USE OF OBJECTS IN ESM TESTBED APPLICATIONS

2.1 BACKGROUND

A radar ESM system is a type of defence equipment used to intercept and identify signals from radio frequency emitters such as radars and jammers. By correlating these emitters with the ships and aircraft on which they are normally fitted, it is possible to construct an accurate and timely picture of the current tactical situation. The "front end" of an ESM system typically consists of an antenna/receiver combination, which converts intercepted radar signals into digital pulse descriptor words which describe the measured parameters of the intercepted signals. Pulse descriptor words are then fed to some combination of digital processors which sort and deinterleave trains of pulses, measure other signal parameters, and try to associate pulse trains with known emitter types stored in the ESM system libraries. Usually, the ESM system will also try to track emitters which have been identified so as to determine the relative bearings of their associated platforms, and, if possible, calculate their positions.

The front end of an ESM system using a modern wide-band receiver can produce tens of megabytes of digital data per second, far exceeding the processing capacity of a conventional processing system. The architecture for the testbed

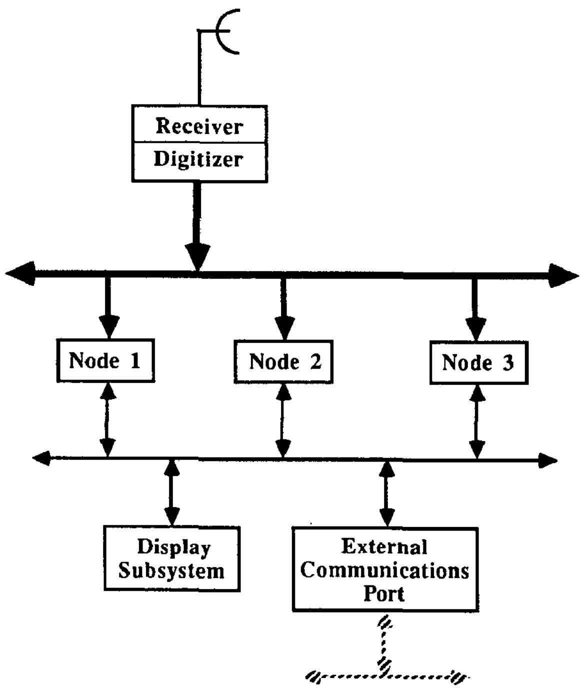

FIG. 1: NEW PROCESSOR SYSTEM CONFIGURATION

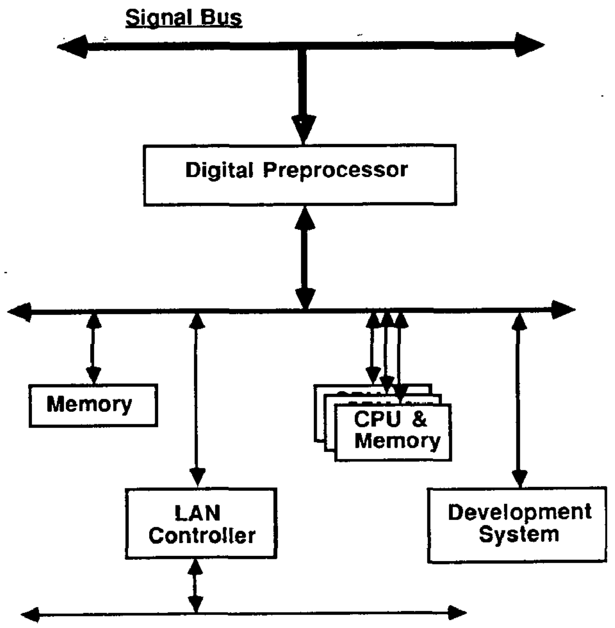

FIG. 2: MULTIPROCESSOR NODE CONFIGURATION

system employs a loosely coupled network of multiprocessor computers (Figure 1), each of which consists of a filter subsystem and a number of single board computers with associated memory (Figure 2). Physically, each multiprocessor node corresponds to a VME bus chassis; the single board computers are based on the Motorola MC68020 CPU. Incoming data is processed by a digital preprocessor, and is directly accessible by the single board computers.

2.1 SIGNAL PROCESSING AND ANALYSIS WITH OBJECTS

Technology and techniques are constantly evolving in response to a changing signal environment. As new radars with increasingly complex signal characteristics are developed, existing signal processing techniques must be improved or replaced to maintain the ESM system's capability to accurately identify the emitters in the electromagnetic environment. Consequently, ESM system software must be designed so as to support continual growth. Objects provide a natural solution for this requirement, offering the modularity and robust response to change that is needed.

One of the new approaches to ESM signal processing being investigated with the multiprocessor testbed is an attempt to combine symbolic reasoning with more traditional numerical signal processing algorithms. There are two key elements in this approach: an object-oriented emitter library and a data analysis system based on Shafer-Dempster belief functions. As indicated in the introduction, ESM systems include libraries of signal characteristics; traditionally, databases which are no more than a collection of data records have been used for this purpose. The database is usually accessed with a collection of customized search, retrieval and update procedures which have detailed knowledge of the internal record structure. Naturally, it is difficult to specialize the record structures for individual cases because of the wide ranging impact this could have on other application modules.

As long as the data structures are reasonably static, one can cope with this limitation by providing a field in every record for all known kinds of emitter information. A field which corresponds to information which is not applicable or available for a particular emitter can be filled by a coded entry which indicates this. Problems appear quickly, though, once the database is required to support sophisticated algorithms based on rule-based and knowledge-based processing. These procedures require a great deal of "soft" (i.e., poorly structured) data such as tactics, recent platform movements, and conjectures concerning radar design and function. Such algorithms will almost certainly be designed to make use of any specific features of particular radar systems which might make them easier to identify. Representing this kind of information within the framework of the traditional ESM database architecture is difficult.

With the emitter class libraries, data can be accessed by sending messages to the emitter objects in the library. Since each class of object has its own table of associations between message selectors and procedures, each object can have its own tailor-made procedures which are used to answer access requests. In the case of traditional emitter libraries, a common query to the library is "Does the signal just observed match any emitter type in the library?" It is the responsibility of the algorithm performing the query to understand and manipulate all data structures in the library in order to determine if a match exists. In such a class library, each entity in the library would be asked "Could you have produced the observed signal?". The library entity, in this case, is responsible for determining which aspects of itself and those of the observed signal are important and constitute the criteria for matching.

The second main element of the data-driven approach is the Shafer-Dempster data analysis system. The Shafer-Dempster Theory of Evidence is based on a generalization of probability [Shafer 76]. Based on preliminary measurements of intercepted signal characteristics, candidate emitters will be identified by the library as potential sources for the observed data. Single candidate emitters and groups of candidate emitters will be assigned numbers between 0 and 1 called masses. The support or belief for the proposition that a particular emitter is the true source is the mass assigned to it; the plausibility of the proposition is the sum of the masses assigned at all groups of emitters containing that emitter. Shafer-Dempster Theory provides mathematically consistent rules for assigning masses, and computing supports and plausibilities.

A classification system for emitters which is based solely on characteristics of the emitter which can be measured by an ESM receiver has been developed. Blackboard objects use this classification scheme to assign the candidate emitters, which were identified by the library, to groups with similar characteristics. These groups will correspond to particular types of emitters. Signal processing algorithms will be applied to the data to determine which emitter type describes the true source of the observed signal. Other procedures will be run to refine the actual values of measured parameters (for these procedures to perform optimally, it may be necessary to first establish the type of the observed emitter). The results produced by each algorithm will be used to redistribute the masses among the candidate emitters. Candidates with little or no mass may be withdrawn from consideration. This process continues until a clear winner emerges, or until only marginal gains are possible with additional processing.

Object-oriented ESM signal processing provides a number of potential benefits within an elegant framework: it provides a mathematically consistent way to quantify ambiguity; the emitter classification system provides a mechanism for classifying unknown emitters; and a capability is offered to design parameter measurement algorithms which can be directed by information specific to an emitter or an emitter type. This last option raises the possibility of actually using the special features of complex radars to identify them.

2.2 SIMULATION OF THE ESM SYSTEM

An EW simulation environment has been developed to support both the design and implementation phases of the project [Barry 87]. Complex systems such as the one we are developing quickly reach a point in the design cycle where no one individual has a sufficiently all-encompassing view of the design to reliably predict system performance. Simulations are a necessary part of the development environment, providing the feedback the design team needs to evaluate various versions of the system design, develop signal processing strategies, test algorithms, identify causes of poor response, and evaluate the cost-effectiveness of proposed modifications.

Unfortunately, simulations of complex ESM systems have proven to be difficult to build and virtually impossible to thoroughly validate. As a consequence, most systems engineers tend to regard results derived from simulations with suspicion, preferring to rely instead on laboratory testing and field trials for performance evaluation. Consequently, we felt that new simulation approaches were required which would allow detailed simulations to be developed quickly by the members of the design group rather than by a team of programmers.

The ESM simulation environment is based on the Smalltalk programming language. The environment supplies a large number of re-usable objects which can be used to model the basic subcomponents of ESM systems, as well as essential features of the electromagnetic environment. Additional objects implement graphically oriented user interfaces. Simulations are constructed by assembling the appropriate building blocks taken from the library of ESM objects. Since the organization of the object library has been designed to reflect the problem domain, end-users find it much easier to understand and use. Our initial experience with the prototype environment suggests that detailed and understandable models of ESM systems can be designed and built in as little as 10% of the usual time required.

The organization of the simulation is illustrated in Figure 3. In general, the simulation objects can be divided into two groups: those used to simulate ESM systems and the electromagnetic environment, and those used to construct user interfaces. The electromagnetic environment is described by a Scenario object. Scenarios are built around Map objects which provide the physical reference points for describing the motion of objects in the Scenario. A Scenario has associated with it lists of Platforms, Emitters, and EWSystems. Platform and Emitter objects hold the state information needed to describe their activity in the Scenario (e.g., platform motion, emitter on/off times, etc.). Heuristically, one tends to think of emitters as being a collection of

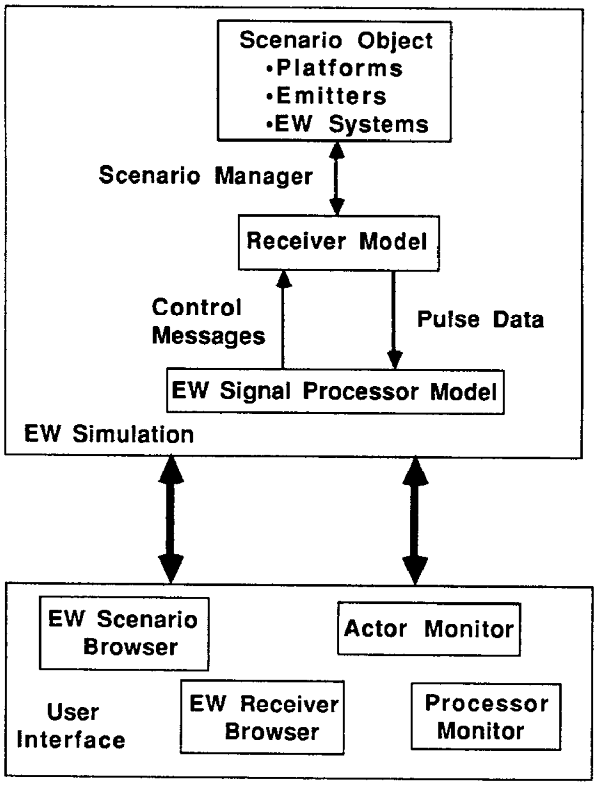

FIG. 3: ORGANIZATION OF SIMULATION OBJECTS

independent objects acting in parallel, suggesting that emitters should be modeled by a number of asynchronous processes. Unfortunately, this generates more computational overhead than can be easily managed with our current implementation. The solution we adopted was to include in the simulation an abstract actor called a ScenarioManager, whose function is to provide an interface between the emitters and any ESM receiver models in the Scenario. The other principal actors in the current implementation are Receivers, DataBuses, and Processors.

Receiver objects are associated with one of the platforms in a Scenario; they receive pulse data from a ScenarioManager. Signal processing within a Receiver is modeled by a SignalFlowNet: when the Receiver receives a Pulse object from a ScenarioManager, it sends a "process this pulse" message to the SignalFlowNet. A Pulse object is essentially a list of parameter values, specifying the RF (radio frequency), TOA (time of arrival), PW (pulse width), DOA (direction of arrival), PA (pulse amplitude), and source emitter for a pulsed signal. The SignalFlowNet is a network of abstract devices: Filters, Detectors, Digitizers, Limiters, Transformers, and BusTerminators. By interconnecting a number of these abstract devices we can quickly build functional models of most receivers.

The main objects used to simulate multiprocessor computing systems are DataBuses, Processors, Tasks, and Messages. A DataBus has a collection of BusTerminators and BusAdaptors; its function is to model the movement of data on a bus according to some bus arbitration protocol. Each Processor has a unique identifying number, a boot block which contains initialization information, priority queues for tasks and messages, and pointers to the currently executing task and the last received message. Message objects have state which records the sender and receiver of the message, the times sent and received, a message type symbol, and the body of the message (which can, in principal, be any object). Task objects model Harmony tasks (Harmony will be described in the next section); they are independent processes which communicate and synchronize with message-passing. A particular task in a multitasking system is modeled by creating a subclass of Task which includes the appropriate protocol for the task being modeled.

As shown in Figure 3, the user interface is a collection of software tools which are designed to interact with specific classes of simulation objects. Since these tools are themselves objects, they can be easily modified to support new requirements, and the tool set can be expanded, with no danger that unexpected interactions would render existing software unusable. We are using two types of interface objects at the moment, Browsers and Monitors.

Both provide access to the state of simulation objects: browsers support extensive off-line (i.e., while the simulation is not running) interaction between the browsed object and the user, while monitors are used during the running simulation to examine the state of dynamically changing simulation objects.

ScenarioBrowsers provide an electromagnetic scenario generation and analysis capability (Figure 4), while ReceiverBrowsers permit the user to create receiver models directly from system block diagrams. The ActorMonitor is a kind of "generic" interface designed to inspect the state of any actor; ProcessorMonitors are a subclass designed specifically for inspecting Processor objects. ActorMonitors show the time on the actor's local clock and the message state (i.e., Send, Receive, Reply). In addition, a capability to intercept and display all messages to and from the actor is provided. Outgoing messages can be edited by the user before being forwarded. ProcessorMonitors can display additional information about the internal state of the processor, such as the currently executing task and the priority queues.

In the simulation environment every effort is made to communicate with the user in the language of the application. Two objects which illustrate how this can be designed into a simulation are LimitedChoiceViews and DimensionedNumbers. The parameter values associated with most objects in a typical EW simulation are known to lie within certain specified limit. Such facts as the maximum speed of a ship, the stalling speed of an

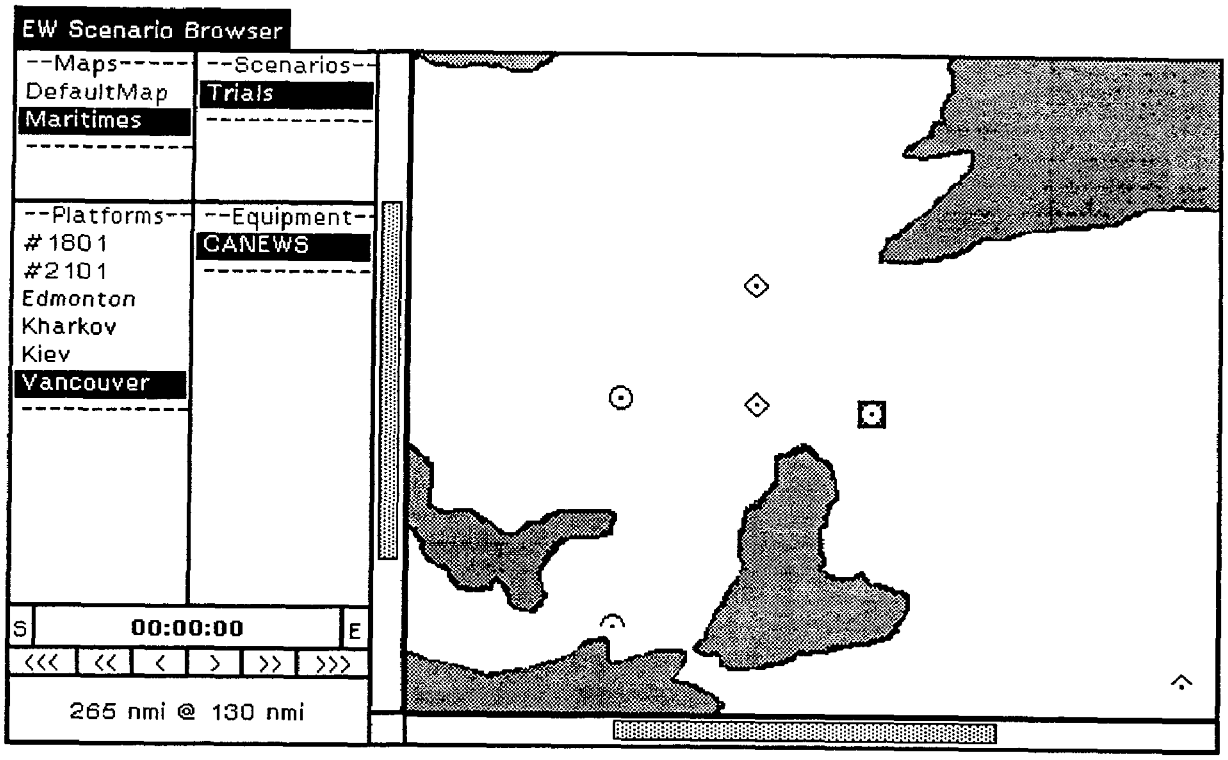

FIG. 4: SCENARIO BROWSER

aircraft, and the design of a radar all impose limits on the values of associated parameters. LimitedChoiceViews provide a "sliding scale" display which allows the user to select parameter values by moving an indicator; the endpoints of the scale are defined by the known limits on the parameter values. Moreover, parameter values are usually represented as DimensionedNumbers, which store both a magnitude and the unit of measurement. Arithmetic operations performed on DimensionedNumbers check the units of measurement and generate an error when dimensionally incorrect operations are invoked. Whenever the parameter value is displayed, its associated units are displayed as well.

The objects which comprise the simulation environment provide the systems engineer with powerful tools which focus on model-building rather than program-building. In the next section, we will see that objects also play an integral role in the testbed's software development environment.

3.0 SOFTWARE ENGINEERING METHODOLOGY AND TOOLS

3.1 IN SEARCH OF AN OBJECT-ORIENTED DESIGN METHODOLOGY

While much has been written recently about object-oriented programming [Cox 86], there is no well established software engineering methodology for using powerful tools like Smalltalk and Objective-C. Our problem was further complicated by the fact that none of the existing structured methodologies addressed the issue of multiprocessor system design. In the sections that follow we assume the reader is familiar with the object-oriented design of sequential programs. We describe our current approach for organizing collections of cooperating tasks in the multiprocessor.

3.2 THE HARMONY MESSAGE PASSING REALTIME KERNEL

Harmony is a portable real-time multiprocessing kernel designed for applications such as robotics and command and control. It implements the send, reply, and receive constructs developed in the Thoth [Cheriton 82, Gentleman 81] family of real-time operating systems. The rationale for blocking send, blocking receive and asynchronous reply are well described by Cheriton [Cheriton 82]. Harmony supports both multiprocessing and multitasking with priorities, using light weight tasks. Any task can send a message to any other task in the same or a different processor. Since both local and remote tasks are referenced in the same way, tasks can be moved to different or other processors without modifying the application software. The tasks which are executed in a given processor are determined at configuration time. Any number of instances of a task can be created and destroyed at runtime.

A task can communicate with any other task on any processor in the system. The "send" message causes the executing task to block or suspend until it receives a "reply"; "receive" causes the executing task to become blocked until a corresponding request has been received. This simple and straightforward protocol is augmented by two special forms which implement a non-blocking receive and interrupts. Additional primitives for creating and terminating tasks and supporting stream-oriented input/output are also provided.

3.3 USING ACTORS TO ORGANIZE MULTIPROCESSOR APPLICATIONS

We use an object-oriented design philosophy which has its roots in the Actor model of programming, often referred to as Anthropomorphic Programming. In our system, an actor may be thought of as a Harmony task which has a list of subtasks to perform (its "script"). Actors synchronize their activities and communicate by sending one another messages. Designers tend to assign actors personified roles such as servers, clients, administrators, workers, couriers, etc. Actors have local state variables, but there is no global state in the usual sense. Until recently, the intuitive appeal of the actor model has been offset by concerns that it would lead to designs which would entail prohibitive overhead, especially in real-time systems. Harmony provides a robust, industrial strength vehicle for realizing actor-based designs.

The usual steps when designing such as system (as outlined, for example, by Hewitt [Hewitt 77]) are:

1. Decide on the natural kinds of actors to have in the system.
2. Decide what messages an actor receives.
3. Decide what actions each kind of actor should perform when it receives each kind of message.

Actors are consequently defined by their behaviour or function. One of the actions permitted an actor is to simply forward the message to another actor; this process, called delegation [Lieberman 86], allows an actor to share the responsibility for performing an action with one or more other actors. In addition, in our system, actors can be decomposed into a number of "lower level" objects. These lower level objects may communicate directly with each other; however, to send messages to objects outside the actor's domain of definition, they must communicate via the actor. This discipline leads to "modularity in the large" which complements the "modularity in the small" provided by object-oriented programming languages. In this way, actors encourage encapsulation, permitting the system designer to create abstractions which hide detail and separate specification from implementation.

Given the above facilities to create and communicate with active objects, it is possible to develop some generic actors which will prove useful for a wide number of applications. Typical examples include clients, servers, workers, couriers and notifiers [Gentleman 85]. Clients are actors which make requests on servers using a normal message send. Servers are objects which encapsulate a shared resource; this resource might be a hardware device, data, or perhaps knowledge about how to perform some action. The default behaviour of a server actor is to create its worker and courier objects (which report for work) and then to loop awaiting client requests. Servers can be designed as straightforward state machines which manage computations carried out on their behalf by worker actors. Sometimes it is necessary for a server to itself request something from another server actor. A courier is simply a vehicle for sending a request to another server; it avoids blocking the sending server while work is being performed by the second server (Figure 5).

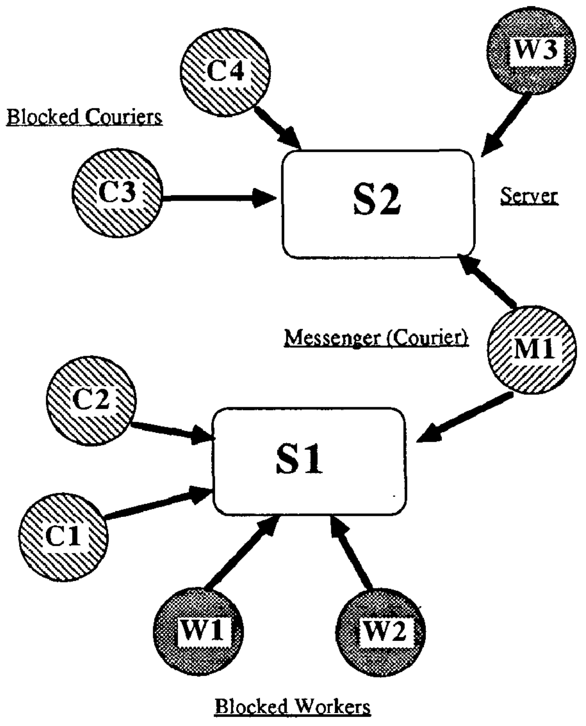

FIG. 5: SERVER-SERVER COMMUNICATIONS USING MESSENGERS (COURIERS)

When created, a worker/courier begins execution by sending a message to the server indicating that it is waiting for a reply containing the work to be performed. The server upon receiving this message keeps track of the available workers/couriers and dispatches work to them using the reply primitive. When a server receives a request from a client, it finds an available worker and replies to it with some operation required for that client. In this way, servers do not stay blocked for any request (Figure 6).

Our design approach is derived from the Smalltalk model, with delegation replacing inheritance; servers play the role of objects, and workers are identified with methods. As in Smalltalk, servers are organized as a hierarchy

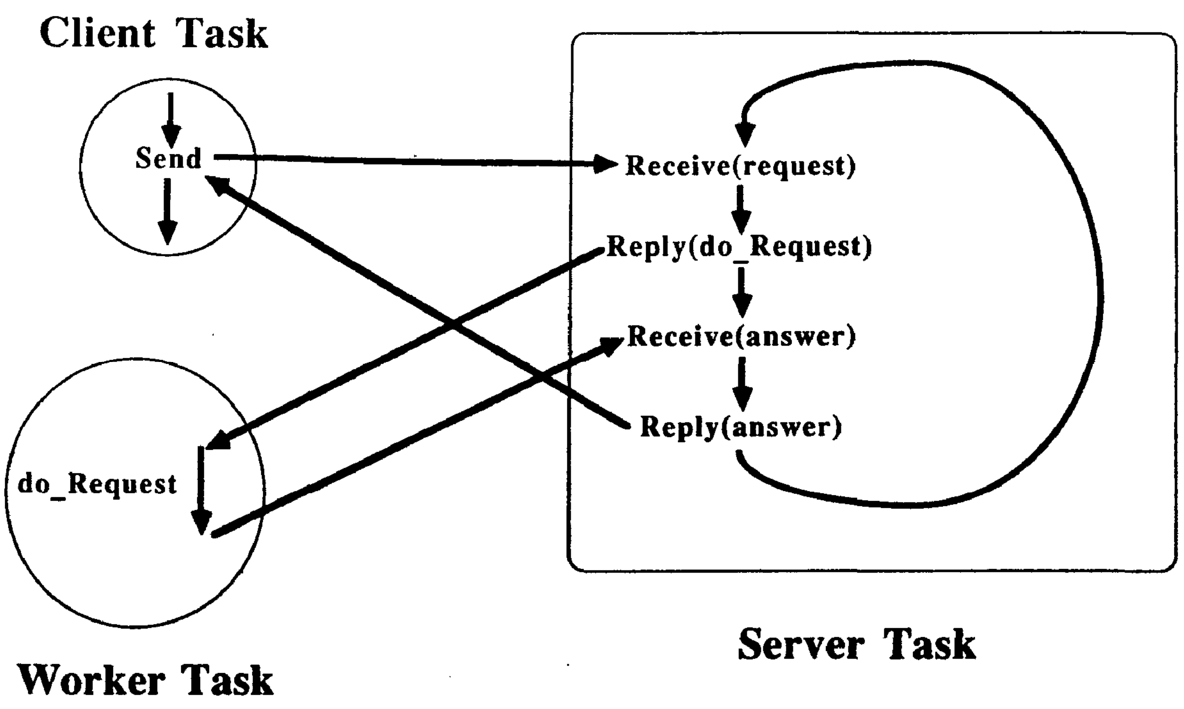

FIG. 6: CLIENT-SERVER-WORKER TASK STRUCTURE

of software objects, as opposed to the "flat" system space generated by current Harmony applications. A typical server will have a supervisor (server), a number of workers, and a number of subordinates (servers). Workers will only execute "sends" to servers; servers will only execute "replies" and "receives" to and from workers. Servers which communicate directly with one another but do not have a subordinate/supervisor relationship are referred to as colleagues; colleagues may be thought of as servers in the same design module which share a common purpose. Each server has a service dictionary which describes all of the services it provides. Each item in a service dictionary is an association between the message(s) which invokes a service and the actor responsible for providing it.

Requests for service can either be satisfied by one of the server's worker actors, or they can be delegated. Three kinds of delegation are possible: service requests which appear in the service dictionary can be assigned to a subordinate or referred to a colleague, while requests which the server cannot find are referred to its supervisor. On first glance, it might seem that the additional message sends required to implement delegation in a real-time system would be prohibitive. Fortunately, there is a simple mechanism available to cope with this. When a worker which requests a service is redirected, it can record the "route" it took to reach its final destination. Moreover, the worker can "remember" this routing so that future requests for the same service can be made directly to the provider. Thus far, it is our experience that most actors will be relatively long-lived. In such cases, delegation can provide encapsulation without significant overhead.

The hierarchical server organization, when combined with delegation, provides a mechanism for controlling inter-task communication without imposing undue restrictions on the system designer. The package defined by a server, its workers and its subordinates can be treated by the rest of the system as a single object. Extensive changes can be made to a server's functionality without impacting on the rest of the system. The design becomes much easier to understand and document, since at each level in the hierarchy there should only be a few colleague servers. The approach encourages top-down design: one first identifies the general functions and protocols of a few colleague servers, after which, as more detail is added, the servers can be decomposed into workers and subordinate servers. In this way, system designers can defer decisions as to how work will actually be performed until the later stages of the design cycle.

3.4 OBJECT-ORIENTED DEVELOPMENT AND TEST SYSTEM

A software development environment designed to realize hybrid object-oriented software has been developed for the ESM testbed [Thomas and Wilson 87]. The software is "hybrid" in the sense that two object-oriented languages are used concurrently: a version of Smalltalk modified to support the Actor concept which we call Actra [Thomas et. al.], and Objective-C, an object-oriented dialect of the C language [Cox 86].

Smalltalk is used for simulating, prototyping, monitoring, debugging, and testing system software. In addition, software tools to support object-oriented design and configuration management are under development. Objective-C is used to code tasks which must meet real-time performance criteria after they have been successfully prototyped in Actra. Software development is done using personal computers. The personal computers are directly connected to the VME bus of one of the multiprocessor nodes in the testbed (Figure 7).

take place until a new message is received. Actra modifies this behaviour to allow an Actor to reply a value to the Actor which invoked it, without terminating its own execution. At some later point, it can then make itself available to receive new messages by executing an end-method bytecode. This terminates the current computation, and puts the Actor in a receive-blocked state. This is the user's view of the Actra model.

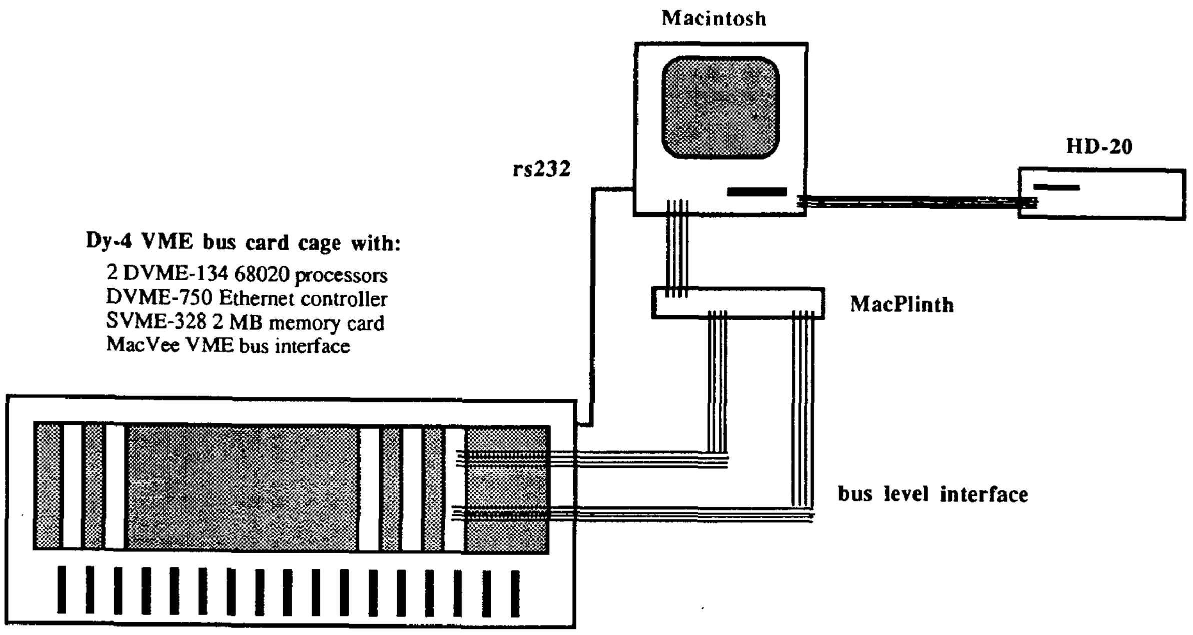
Macintosh
HD-20
rs232
Dy-4 VME bus card cage with:
2 DVME-134 68020 processors
DVME-750 Ethernet controller
SVME-328 2 MB memory card
MacVee VME bus interface
MacPlinth
bus level interface

FIG. 7: OBJECT-ORIENTED DEVELOPMENT AND TEST SYSTEM CONFIGURATION

Smalltalk processes provide traditional co-routine based uniprocessor multitasking with semaphore synchronization. Unfortunately in Smalltalk, processes are not first class objects. They can't be sent messages or specialized like other objects. They are created from blocks. While this approach is adequate for Smalltalk's requirements (simulation and window management), it is insufficient for structuring large numbers of independent processes or handling true multiprocessing.

Actra's high-level multiprocessing model is based on the concept of Actors. In the Smalltalk context, Actors are objects (i.e., instances of class Actor or its subclasses) which have the capability to execute concurrently. To implement this, the Smalltalk model of message passing has been extended in the following way. In Smalltalk, when an object sends a message to some other object it is suspended, until the receiving object returns some value. In Actra, we maintain this behaviour. However, in Smalltalk, from the point of view of the receiving object, once a value has been returned, no further actions can

From the implementation standpoint, the Actor mechanism is supported by the Harmony kernel as follows: associated with each Actor is a Harmony task. This task is essentially a copy of the Actra interpreter, which executes the bytecodes - that is, the (Actra) Smalltalk code - for the Actor. In fact, all Actors share the same code, but each executes it as a separate task. Under Harmony, the message passing mechanism allows synchronous send, synchronous receive, and asynchronous reply operations. Calling a Harmony receive causes the associated task to suspend, putting the Actor which is executing in a receive-blocked state. Calling a Harmony send causes the task to suspend until it receives a reply; consequently, when the executing Actor sends a message it will block until the receiving Actor replies a value. Obviously, this is exactly what we want to support Actors.

When an Actor sends a message, its interpreter decides whether or not the message is intended for another Actor. If not, as is the case for most messages, it is simply sent to the receiver object using the standard Smalltalk message passing mechanism. However, when a message is to another Actor, the message and any other required information is packaged in a Harmony message and sent to it using the RTOS. This allows Actors to reside on multiple processors. An important goal when we were designing the software development environment was to provide a consistant interface between Harmony tasks written in C or Objective-C, and Actors in the Smalltalk world. Given this facility, it is possible to write time critical, real-time applications while still maintaining access to the user-interface facilities of Smalltalk. In the ESM testbed, C language support has been provided both by a package of C-to-Smalltalk communication primitives, and by high-level C code debugging facilities accessible from the Smalltalk environment.

The software development environment provides support for high-level debugging, and direct manipulation of C language code, using a family of classes for accessing C data structures. Class Struct contains several methods for directly accessing memory locations anywhere on the development system's own microprocessor or on the VME bus. It also contains methods for accessing the standard C language data types (chars, short ints, ints, unsigneds, etc.), which are built on top of the memory accessing routines. Finally, the class implements a general framework for describing arbitrary C structs. Subclasses of this class can use these routines to describe specific C structs. By creating instances of these subclasses, it is possible to shadow arbitrary C structs anywhere in memory on the VME bus.

Using this facility we are able to use Smalltalk as our symbolic debugger! Class StructInspector and class StructInspectorView, together, constitute a generic C struct inspector facility which allows C structures to be inspected exactly like Smalltalk objects. Accessing fields in the inspector actually access the real memory of the struct, and "accept"ing new values causes them to be written into memory. For example, Class TaskDescriptor is a subclass of Struct which describes the struct used to describe a Harmony task descriptor (TD). Instances behave exactly as if they were the TD struct. The interpretation of the various fields of the struct is managed internally, so that, for example accessing a field which should contain a pointer to another TD actually returns another instance of TaskDescriptor which describes it. Similarly, accessing the STATE field will return a symbolic representation of the current Harmony state of the associated task, while modification of this field can be performed by using direct entry of the state name, or by menu selection.

Given the above facilities, any C struct can be described from within the development environment, and then accessed using the full power of the Smalltalk programming environment. ESM testbed nodes are interconnected to other nodes and various hosts and workstations using ethernet. It is well known that TCP/IP based communication can be very expensive and is therefore frequently avoided for embedded systems. However, the use of a custom protocol has the disadvantage of isolating the testbed from post-processing hosts and development facilities. To solve this problem we have implemented an intersystem protocol based on UDP envelopes. The testbed supports the address resolution protocol (ARP) for TCP/IP, thereby allowing host programs to easily communicate with the testbed using standard UDP sockets. The protocol and drivers allow both remote Smalltalk and C programmers to send and receive messages from the ESM nodes.

We are in the process of developing methodology and tools to support configuration management of large object-oriented systems during the entire development cycle. Programming guidelines have been developed aimed at standardizing both Smalltalk and Objective-C code. We use macros to improve the readability of the C code, and to supply a common organization for procedural code in server, courier and worker actors. This facilitates porting software between the two languages. We are currently experimenting with automated design aids which diagram the functional relationships between servers and workers. Unfortunately, the OOPS community lacks standard diagramming conventions. This is especially frustrating for those who see how easy it is to support SADT type diagramming in Smalltalk. An active data dictionary facility is being implemented to provide consistent naming in multiuser environments.

Object-oriented projects require change management systems competitive with those offered in the traditional Unix environment [SCS, RCS]. In an environment with so much freedom, we need tools to be able to manage groups of programmers. We are investigating a system which associates a class manager with every class. It requires us to define a set of base classes which can be relied upon by all programmers. In order to share code between workstations, we assume the class manager to have ownership for the class, including the source code, object code and changes.

4.0 CONCLUSION

We have presented an overview of an integrated testbed for prototyping ESM signal processing systems. Object-oriented programming and object-oriented design have played a key role in the success of this undertaking, by providing the conceptual building blocks which allow us to manage the complexity of the application and its multiprocessor environment. The result is a vertically integrated system which, within a single uniform framework, can encompass the entire development cycle. Beginning with tools and methodologies which support the design process, continuing with extensive simulation and prototyping facilities, and following through with support for programming and testing, the testbed provides a more productive and reliable development environment for ESM systems engineers.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the contributions of the Advanced ESM Multiprocessor Design Group at DREO, the Actra Project Team at Carleton, The Dy-4 Systems Harmony Team, and the Harmony Group at NRC.

REFERENCES

1. Barry, Brian M., 1987. "Object-Oriented Simulation of Electronic Warfare Systems". To appear in Proceedings of the Summer Computer Simulation Conference (Montreal, July).

2. Thomas, D.A., 1987. "Object-Oriented Design of Multiprocessor Software Using the Harmony Operating System", Dy-4 Systems Design Note.

3. Hewitt, Carl, 1977. "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence 8: 323-364.

4. Gentleman, W. Morven, 1981. "Message Passing between Sequential Processes: the Reply Primitive and the Administrator Concept", Software Practice and Experience, 11: 435-466.

5. Gentleman, W. Morven, 1985. "Using the Harmony Operating System", National Research Council of Canada Report No. 24685. National Research Council of Canada, Ottawa, Canada (May).

6. Cheriton D.R., 1982. The Thoth System: Multi-process Structuring and Portability, American Elsevier.

7. Lieberman, Henry, 1986. "Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems", ACM SIGPLAN Notices, Vol 21, No. 11.

8. Cox, Brad J., 1986. Object-Oriented Programming: An Evolutionary Approach. Addison-Wesley Publishing Company, Don Mills, Ontario.

9. Thomas, D.A., 1985. "Supporting Interpretive Programming Languages on a Harmony Based Multiprocessor", DY-4 Systems Design Note.

10. Thomas, D.A. and M. Wilson, 1987. "An Object-Oriented Development and Test System", DREO Contract Report.

11. Thomas, D.A., W. Lalonde and John Pugh, 1986. "A Multitasking/Multiprocessing Smalltalk", SCS-TR-92, School of Computer Science, Carleton University, Ottawa, Canada (May).

12. Shafer, G., 1976. A Mathematical Theory of Evidence, Princeton University Press.