DEVELOPING DISTRIBUTED APPLICATIONS UNDER THE OBJECT ORIENTED MODEL

Eleri Cardozo
Tech. Institute of Aeronautics
12225 Sao Jose dos Campos - SP

Tadao Takahashi
Ethos Project
13083 Campinas - SP

*** DRAFT ***

1. INTRODUCTION

Object oriented programming (OOP) [1,2] is becoming a standard design methodology for large software system developments. OOP favors a programming style where the software components (objects) are nearly self-contained (many of them constructed by specializing existing ones), have well defined roles and clear mechanisms for interacting with other objects. Such properties decrease the efforts demanded by the implementation of complex software systems and increase the system's reliability, testability and maintainability.

Unfortunately, OOP techniques are available today at the programming language level and few extensions of such techniques to distributed environments have been reported [3]. This paper proposes an extension of DPSK (a Distributed Problem Solving Kernel) [4] in order to accommodate some OOP practices like hierarchical organization of classes, instantiation of classes, definition of methods, etc.

The main objective of this research is to derive an environment for distributed problem solving (named DPSK+P) that has the advantages of DPSK in terms of convenience and efficiency, plus some added capabilities that favor object oriented programming at the distributed application level.

2. OBJECT-ORIENTED PROGRAMMING

The object oriented programming approach is relied on two concepts: objects aggregated into classes, and methods. An object can be defined as a black-box modeling some entity of the domain. This black-box packs the data necessary to quantify the entity plus, a set of procedures that manipulate the data. Objects interact by defining procedures that can be activated by other objects. Such procedures are called methods. A method can have input parameters, and once activated produces side effects and may return some useful result. The object's internal data has a public portion and a private portion. Public data can be accessed from other objects within the same program, while private data are manipulated exclusively by the object's internal procedures. Figure 1 schematizes an object. Since objects are nearly self-contained, a better structuring of the entire code is achieved. As a result, productivity, extensibility, testability and reliability are favored by OOP when compared to other programming techniques.

A central concept in OOP is the concept of class. A class defines a behavior for a family of objects derived from it (the class' instances). A class keeps the object's data and methods[1], which are inherited by its instances. Some classes derive from one or more classes, forming an hierarchy of superclasses (ancestors) and subclasses (descendants). A class inherits the behavior (data and methods) from its direct superclasses, elaborating this behavior by

1 Some object oriented systems keep methods separated from classes (CLOS [8] is the best example of such systems).

specializing the data and by adding new features to the methods.

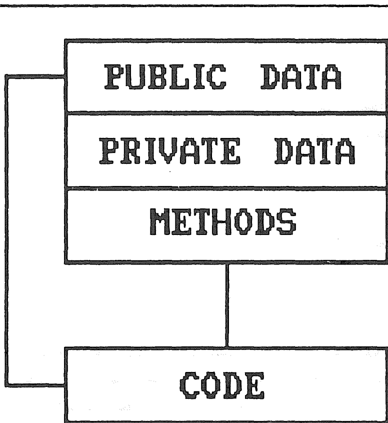

Fig 1. An object and a code (other objects) using it.

A program elaborated according to the object oriented model is composed of a hierarchy of classes and a set of instances (objects) interacting among them. For example, in the design of a drawing package, one can define a class named "shape". Geometric figures like circles and rectangles are specialization of shapes, squares are specializations of rectangles, and so on. Methods like "display" can be defined for the class "shape" and inherited by all the instances belonging to the subclasses of "shape". More specialized methods like "rotate" are defined for each geometric figure, possibly by elaborating those inherited from the respective superclasses. Calling "rotate" for an argument belonging to the class "square" will automatically invoke the method that rotates squares. This property is called polymorphism, and the binding between methods and calling parameters can be accomplished at compiling time (early binding) or at run time (late binding).

3. DISTRIBUTED PROBLEM SOLVING

In short, Distributed Problem Solving (DPS) is a methodology for software developments that favors a set of autonomous modules (the problem solving agents) running independently in a multitasking or multiprocessing machine or in a set of processors linked through a local area network [5].

The modules need to cooperate in order to achieve a common goal: the solution of the problem being processed. Cooperation is composed of two basic activities: communication and control. Communication is the exchanging of information among the modules and control is the coordination necessary to guarantee that each module will perform its task at the right time and with the proper resources allocated to it. Basic control activities are inter-module sequencing (managing tasks like the access to critical resources) and execution control (managing tasks like starting of modules).

Inter-agent communication can be achieved through mechanisms based on message passing, sharing of data, remote procedure call, etc. Mechanisms for control include semaphores, events, barriers, etc (for sequencing) added to the proper operating system directives for execution control. Several tools for aiding DPS developments have being designed and implemented. Such tools provide communication and control services based on one or more mechanisms mentioned above.

The basic communication activities found in centralized object oriented programming languages and systems can be extended to distributed environments. Communication in OOP is based on the calling of methods and sharing of data (the public portion of the object's data). In distributed environments, the calling of methods may be relied on a client/server communication protocol (like rendezvous or remote procedure call); while the sharing of data needs only an accessing discipline to keep the data consistent and correct under concurrent access.

The model for distributed computing proposed in this paper is based on the concept of shared objects. A shared object is a macro structure aggregating an amount of public (shared) data and a set of active methods[2]: the smallest amount of computation managed by the operating system (processes in the case of UNIX). Active methods can be a set of ordinary objects if coded according to the OOP model, or any piece of sequential code. Without loss of generality, we assume that each problem solving agent in a distributed environment is an active method attached to some shared object.

Shared objects interact among them by sharing their public data added to a client/server communication protocol for accessing the active methods. The public data is a set of attribute-value pairs (slots), and the protocol for method activation remembers the rendezvous mechanism found in the ADA programming language. Figure 2 shows schematically a shared object.

To illustrate the usage of shared objects as an integration medium, consider an environment for system's identification, optimization and control as described in [6]. This environment has several modules (integrated around an executive) performing optimization, real time control, system's identification, etc. If these modules are to be distributed through many processors, these services could be understood as shared objects performing well defined tasks. Taking system's identification as an example, algorithms as Recursive Least Squares, Extended Least Squares, etc, are active methods of the shared object performing system's identification. Separated active methods can be employed for each algorithm, or more than one algorithms be grouped into an active method. In the last case, the appropriate action (algorithm) is selected according to the parameters passed to the active method during its invocation (see section 7).

---

2 All the code attached to a shared object will be referred as a method, even those not performing interfacing with other shared objects.

Even if the system's modules are not coded according to the object oriented model, the concept of shared object brings to the system's architecture the central ideas behind object oriented programming. As a result, it is expected an increasing in many desired attributes like reliability, testability, etc.

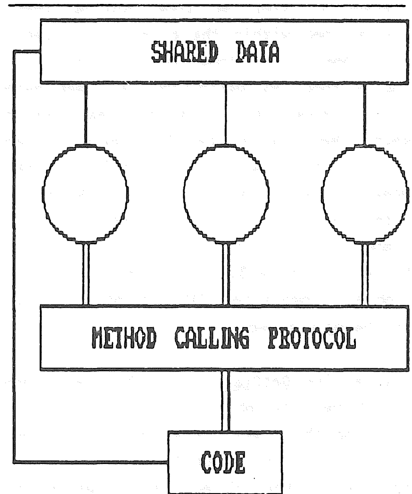

Fig 2. A shared object.

4. COMMUNICATION IN DPSK+P

As stated before, communication in DPSK+P is relied on the sharing of data and calling of active methods. The following subsections describe how shared objects can be employed for communication.

4.1 Defining Classes

DPSK+P allows the user to define a hierarchy of classes shared by all the problem solving components. A class is defined by a set of

slots (attribute-value pairs) and may be derived from another classes forming a graph of superclasses and subclasses. Classes are identified by a global name. A single primitive[3] is supplied for class definition: DefineClass. The primitive is employed for building a class giving its direct superclasses, its data attributes (slots), and its active methods (the respective addresses of compiled code holding them). Slots defined within the direct superclasses are automatically inherited by the class being defined. By definition, a class named "t" is a superclass of every class (except itself). New classes can be created at any time and redefining an already existing class causes the replacement of the old definition by the new one. In such case, all the instances of the old class are marked as "obsolete" (see 4.3). The primitive returns an object identifier (ID) pointing to the class being created.

The primitive GetClass takes the class' name and returns its object ID. Primitives for navigating the class hierarchy are provided: GetParents and GetChildren returns respectively the direct superclasses and subclasses of a given class. Once the class structure has been defined, the agents can start to create instances of classes.

4.2 Instantiating Classes

A shared object is an instance of some class (shared object and instance will be employed as synonymous). Instances of classes are created by calling the MakeInstance primitive. This primitive takes the pointer to the class (returned by DefineClass or GetClass) and a set of slots to be added or to replace those inherited from the class. After the shared data has being built, a special method called "constructor" is invoked (if defined by the class). The constructor initializes the shared data, calling other methods if necessary. MakeInstance returns an object ID pointing to the shared object just created.

If not specified otherwise, shared objects are created in the processor where the calling agent is running (not where the class is defined as in the current version of DPSK). An optional third argument in MakeInstance directs the creation of the instance to a different processor.

4.3 Accessing and Updating Instances

Instances can be accessed by the GetInstances primitive. This primitive accepts as input a pointer to the class and a set of specialization slots that must be present in the instances accessed. A list of object IDs pointing to the instances satisfying the given specialization slots is returned.

With the object ID pointing to an instance it is possible:

a. To access any slot in the instance.
b. To delete, add or update a particular slot in the instance.

Accessing or updating an obsolete instance will cause the calling of a special active method named "repairer" (if defined within the class). This method has the mission of making the instance built under an older version of the class compatible with the new definition of it. The mark "obsolete" is removed when the repairer exits.

4.4 Deleting Instances

The primitive DeleteInstance destroys a shared object. An object ID pointing to a deleted object becomes unbounded, turning without effect all operations over it. Immediately before the instance destruction, a special active method called "destructor" is invoked (if defined in the class). This method performs the clean-up relating the instance under destruction (perhaps calling other methods informing the elimination of the shared object). The object is unbounded after the destructor exits. Active methods attached to the a deleted object are killed.

---

3 The term primitive is used interchangeably to refer a C, C++ or Lisp function; a C++ member function; or a CLOS [8] generic function.

4.5 Defining and Calling Shared Methods

Until now we noticed the existence of three special methods supplied by the user when defining a class: the constructor, destructor and repairer of instances. Any other active method can be defined by given them different names. Active methods defined for a class are instantiated (started) in behalf of a shared object when a CallMethod or StartMethod primitive is called given the shared object as target. After its starting, a method can exit after the processing of the call, or remains up waiting for new requests. Each method has a code implementing it (again, a process in UNIX). When many lightweight methods are defined for a class, all of them can be grouped into a dispatcher. The dispatcher selects the desired method by the input parameter in the call (see section 7).

A method is invoked by calling the CallMethod primitive. This primitive takes three parameters: the target--a class or instance to which the call is directed, a single object encoding the input parameter of the call (that in its turn can hold many other objects), and a timeout for the calling request. The caller remains blocked until the call is processed by an active method or a timeout occurs.

The protocol for calling methods is as follows. Calls can be directed to classes or instances. Calls directed to instances can be processed only by active methods attached to this instance. Calls directed to classes can be processed by active methods attached to any instance of the class. When a call to the active method M is directed to an instance:

a. If M is already running within the instance, then queue the call; otherwise

b. If M is defined in the instance's class, then start a clone of M within the instance and queue the call. If the class does not define such method

c. Move up in the class hierarchy until M is found. Start a clone of M within the instance and queue the call. If the method can not be found, return a null value for the call.

Notice that item c. provides inheritance of methods.

If a call is directed to a class, it is queued in the class until an instance accepts it (no method is started).

Active methods accepting calls are servers that keep performing the following cycle :

a. Get a call request.

b. Process the call.

c. Return the result to the caller.

d. Exit or go to a.

Requests are accepted by active methods through the primitive AcceptCall. This primitive has a timeout which controls the amount of time the active method wishes to wait for a call. The method becomes blocked until the arriving of a call for it or a timeout occurs. The primitive returns the parameter passed by the other end (the caller) and an object identifying the Call-Accept pair. When the method finishes the processing of a call, the primitive ReturnCall completes the communication by returning the result of the computation to the caller. The accepting protocol is as follows:

a. If there are pending calls for this method directed to this shared object, process the oldest call, otherwise

b. If there are pending calls for this method directed to the class, process the oldest call.

If no call is pending to the accepting target, steps a. and b. are performed until the timeout specified in AcceptCall occurs.

Active methods are selected by name and by the object holding the method (as in the Flavor System [7]). No method combination

strategy is provided by DPSK+P. Binding occur at run time.

Active methods can discover the object they belong by calling the primitive GetWorld. The primitive returns a pointer (object ID) to the shared object the method belongs. With this pointer the method can operate in the object's shared data.

4.6 Remarks on the DPSK+P Communication Structure.

The communication structure adopted in DPSK+P attends both data-driven and goal- driven problem solving strategies. Data-driven strategies will use the manipulation of shared data to progress the problem solving activity while goal-driven strategies will use the method calling protocol in order to invoke methods for solving subproblems generated during the planning/decomposition phase.

With these two complementary communication structures, we believe that DPSK+P fulfills the communication requirements for a wide range of practical applications.

5. CONTROL IN DPSK+P

5.1 Locking

Under concurrent access, the shared objects may become inconsistent if proper control is not provided. The accessing of the shared data kept by the shared object needs a locking mechanism that assures consistency and correctness of the shared objects. DPSK+P provides a locking mechanism based on transactions [8].

Transactions can be performed over shared objects, classes or hierarchy of classes. The scope of the locking is the following:

a. A transaction over a shared object will lock only the shared data kept by the object.

b. A transaction over a class is equivalent to a transaction over all the shared objects belonging to this class.

c. A transaction over a hierarchy of classes is equivalent to a transaction over the starting class and all of its subclasses.

The scope of the locking ranges from the shared data kept by a single shared object to the entire shared data (transaction of a hierarchy of classes starting at the class "t").

A shared object is not available for locking if:

a. Its constructor is still running, or

b. Its destructor is already running, or

c. The object is already locked for read/write, or

d. The object is already locked for read-only and a read/write lock is being requested.

A lock is obtained by calling the BeginTransaction primitive. This primitive accepts an indicator of locking type (read/write or read-only) and a list of objects to be locked. The primitive returns a locking object if the transaction was accepted, or a null object otherwise. The primitive EndTransaction and AbortTransaction finishes and aborts a transaction in progress. Both primitives accept the locking object returned by BeginTransaction as argument.

If serializability only is demanded (without atomicity and persistence), a simple and faster mechanism based on the method calling protocol can be implemented (as illustrated in the section 7).

5.2 Execution Control

In the object oriented model data and code are aggregated into objects. Following this line, DPSK+P allows the control of the code running within the active objects, that is, the shared methods. Two primitives are

provided: StartMethod and InterruptMethod. StartMethod starts an active method attaching it to some given shared object (duplication of an active method in the same object is allowed). The method must be defined in the object's class or in any of its superclasses. InterruptMethod sends a signal to an active method of a given shared object. Signals provided are those found on UNIX like:

a. KILL: kills the method.

b. SUSPEND: freezes the execution of a method.

c. RESUME: resumes the execution of a suspended method.

d. HANGUP: hangs-up (branches the execution to a given exception handler) a method.

e. Any other signal.

The handling of the signal is user's responsibility (and not possible in some programming languages).

5.3 Synchronization

The basic synchronization mechanism provided by DPSK+P is the method calling protocol. Using this protocol as a basis, it is simple to implement other synchronization mechanisms like events, semaphores, barriers, etc. Section 7 illustrates this feature.

6. INTERFACES

DPSK+P will provide interfaces for C, C++ and CLOS (Common Lisp Object System) [9] programming languages. The C interface is based on a library of functions, while the C++ and CLOS interfaces are based on a set of classes and some few functions. The C++ interface to DPSK+P is listed in the appendix A.

DPSK+P itself is implemented in plain C to favor transportability to more specialized hardware systems. For inter-machine data transferring, the UNIX interface to the TCP/IP protocol is employed. Data coding/decoding follows three different protocols. When the network is homogeneous, the binary protocol is the fastest available since data structures are shipped in their in-memory format (without coding). For heterogeneous networks two protocols are available: the eXternal Data Representation (XDR) [10] protocol (when available in all machines), or a protocol based on the ASCII code (the most general and slower available).

The kernel itself can be replicated in order to avoid interference between different applications when running at the same time.

7. AN EXAMPLE

This section shows how a synchronization mechanism based on semaphores [11] can be implemented using the method calling protocol.

The semaphore mechanism relies on two functions: P and V. When the process desires to enter the critical region it calls P. Upon returning of P it executes the critical section, calling V at the end. P has the following structure:

a. Subtract 1 from the value of the semaphore.

b. If the value is non negative return, otherwise

    i. Add the caller process to the list of blocked processes.

    ii. Block the process.

V returns if the semaphore has its value grater than the unit, blocking the caller if the value is zero or less.

The V function has the following structure:

a. Add 1 to the value of the semaphore.

b. if the value is less than or equal to zero:

    i. Remove one entry from the blocked processes.

ii. Wake up this process.

V makes a blocked process on a P call to return, granting it access to the critical region.

An interface to the semaphore mechanism can be implemented as a C++ class that defines, operates and destroys semaphores. The member functions of this class access DPSK+P in order to achieve the proper synchronization among the distributed system components.

To implement the semaphore mechanism, one can define a DPSK+P class named "semaphore" with a single active method called "PV" that implements both functions P and V. The class has a single slot ("name"). The value of the semaphore will be kept by the method PV. The class semaphore will have an instance for each semaphore defined.

The interfaces for P and V will call the method "PV" with an object as input parameter stating if the call is to be processed as "P" or "V". The method "PV" will keep a list of blocked process (step b.i of V). Callers will be blocked by "PV" simply by not returning the call. In step b.ii of V, the caller will be waked-up by returning the respective call. The list of blocked processes is, in fact, a list containing Call-Accept identifiers (see 4.5).

The code of this example is presented in the Appendix B.

8. CONCLUDING REMARKS

DPSK+P has some attractive features not found in the current version of DPSK. The communication structure found in DPSK was improved by adding a hierarchical organization of classes with inheritance of slots and methods. The method calling protocol allows a client/server communication structure (suitable for goal-driven problem solving strategies) plus a wide range of synchronization mechanisms to fit the needs of practical applications. Libraries of classes (like the class "semaphore" defined above) can be incrementally build and added to the kernel. A monitoring facility is also under development. Crash recovery and migration of shared objects started to be investigated.

9. ACKNOWLEDGEMENTS

This research is being supported by the National Science Foundation, Grant Number CDR-8742796 at the Engineering Design Research Center, Carnegie Mellon University; and Ethos Project at the Tech. Institute of Aeronautics, Brazil.

10. REFERENCES

1. Cox, B., Object Oriented Programming: an Evolutionary Approach, Addison-Wesley, 1987.

2. Meyer, B., Object-Oriented Software Construction, Prentice-Hall International, 1988.

3. Larner, D.L., Factories, Objects & Blackboards, AI Expert, April 1990.

4. Talukdar, S. and Cardozo, E., Building Large Scale Software Organizations, in Expert Systems for Engineering Design, M.D. Rychener (editor), Academic Press, 1988.

5. Demazeau, Y and Muller, J.P. (editors), Descentralized A.I., Elsevier Science Publishers B.V., 1990.

6. Hemerly. H.M., PC-Based Packeges for Identification, Optimization and Adaptive Control, IEEE Control Systems Magazine, Vol. 11, No. 2, Feb 1991.

7. Sun Microsystems Inc, Sun Common Lisp 3.0 - The Flavor System, First Edition, August 1988.

8. Date, C., An Introduction to Database Systems, Vol II, Addison-Wesley, 1983.

9. Keene, S.E., Object-Oriented Programming in Common Lisp, Addison-Wesley, 1989.

10. Sun Microsystems Inc, Network Programming, Part Number 800-1779-10, May 1988.

11. Peterson, J.L. and Silberschatz, A., Operating Systems Concepts, Second Edition, Addison-Wesley 1985.