An Object/Agent Oriented Programming Language and System for Heterogeneous Distributed Computing

(Extended abstract, incomplete, to be revised)

Fah-Chun Cheong
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
(313) 763-2153
fcc@eecs.umich.edu

March 27, 1992

1 Motivation

When creating a distributed application for solving a problem, the very first thing to be considered is the decomposition of the problem into subproblems. Individual pieces of code to solve these subproblems are then constructed more or less independently. And finally all of these pieces are to be integrated into a holistic whole to solve the original problem. The traditional approach for doing this is the top-down stepwise-refinement methodology, characterized by the functional or procedural nature of its decomposition tree that results from breaking a problem according to the control flow at the abstraction level of functions and procedures of its solution program. The alternative object-oriented approach favors breaking up a problem according to the structure of its data required for solving the problem.

Among other things, the object-oriented approach has the advantage of being more concrete in the sense that when looking at the big picture, a programmer sees *objects* or *classes* of objects at the top level and could relate them to things or categories of things in the real world. This is intuitively appealing because after all, a program has to model the real world quite closely in order to solve real world problems.

In trying to extend the object-oriented paradigm to a distributed setting, there are two interesting possibilities, depending on whether the programming model wants to deny or acknowledge the reality of physical distribution. In the former case, objects are spread across the network transparently, with the language runtime system hiding the physical distribution of the machines from the programmer. In the latter case, it is user-level object-owning entities called *agents* that are spread transparently across the network. The objects themselves are handled by agents under explicit program control.

To extend our the real world analogy, the former case would correspond to say, a roomful of things under the control of an invisible man. But the latter case would correspond to a roomful of people with control over things. It seems that the latter viewpoint has much more intuitive appeal. Among other things, it lends support to the thesis that a distributed application is meant to interface with people as a decentralized society of intelligent agents.

The thrust of our research work lies in the realization of this viewpoint with the ultimate goal of creating a usable research prototype for running on a heterogeneous network of popular workstations. The Oasis project, for *Object Agent Specification and Implementation System*, has recently been completed as part of the author's Ph.D. dissertation at the University of Michigan. We are pleased to announce the Alpha version of Oasis as its first research prototype that can be obtained via anonymous ftp from Michigan. The Alpha version runs on a heterogeneous network of Sparcstations, DECstations, IBM RS/6000's, NeXTstations and HP/Apollos 400's.

For the rest of this paper, we present an overview of Oasis by discussing three major areas at which our efforts have been directed, i.e. construction of the object/agent model, design of the programming language, and implementation of the system.

2 Object/Agent Model

We recognize the differences in granularity between objects and agents, and we have considered how they could be represented in terms of the underlying machine and operating system. Agents are mapped to Unix processes on multiple workstations in a network. And objects are simply records residing in the heap space of individual agents. The object/agent-oriented model immediately raises a number of questions commonly encountered in distributed programming languages and systems. How do agents communicate? How do agents synchronize? What does garbage collection mean? We shall address these questions in the following subsections.

2.1 Communication

In Oasis, agents communicate with each other using remote procedure calls. Both synchronous and asynchronous remote procedure call mechanisms are supported. In other words, after sending a message to a receiver, the sender agent can choose to either wait or not wait for the reply to come back before continuing. An important issue when considering remote procedure call mechanisms is how to deal with pointer-passing across address spaces. In relating to our earlier real world analogy of agents with people and objects with things, we have to consider the nature of *things*. If things are considered to be physical, then mapping them to bits and bytes inside the machine would have given them an added degree of freedom. After all, in a computer the bits and bytes are information that could be freely replicated, processed and distributed. But for physical entities, additional processing would have been required to simulate and preserve their physical properties. When we consider how pointers are handled in many current systems, they are in effect simulating at a remote procedure call level the physical reality of the *oneness* of the thing being pointed to. For many distributed applications, doing that at the remote procedure call level would have been overkill, and in many cases complicating the runtime implementation itself.

Therefore, our premise is that: the object represents information. An object in Oasis is simply a structured piece of information with an associated set of *methods* that could be used to operate on it. Our intuition about information is that it could become outdated and sometimes misleading, but generally useful. By drawing upon the real world as our source of inspirations and considering its information flow, it seems that a community of talking and thinking agents could serve as a basis for the design and implementation of many interesting distributed applications.

From a technical point of view, treating Oasis objects as information means that objects could be automatically marshaled and unmarshaled by the Oasis runtime system during remote procedure calls and replies. Each object is described by some *class* in the program. It directs the Oasis compiler to generate the appropriate code for use in packing objects into a contiguous buffer for shipment across the network and in unpacking the buffer at the receiving end. In addition, the Oasis runtime system takes special care to faithfully replicate objects with shared structures and not expand them into trees prior to shipment. We believe that this will greatly simplify agent communications in Oasis.

2.2 Synchronization

An Oasis agent supports multiple threads of control, each running on its own stack but all sharing the same heap space within the Unix process. The execution of these threads are non-preemptive. In other words, when a thread is scheduled to run, no other threads within the same agent could be run unless the current thread voluntarily relinquishes its control. Non-preemption turns out to be the right policy because the basic unit of parallelism in Oasis is the agent. In addition, the agent is also the basic unit of protection, which means that outside agents could not directly address the threads or objects internal to some other agent. But an outside agent could make a remote procedure call on an agent and cause it to dynamically create a new thread which it then schedules on the ready queue. This new thread is responsible for servicing the remote procedure call upon whose completion an answer will be returned to the caller.

Through remote procedure calls, the problem of synchronization between communicating agents has been reduced to the problem of internal synchronization of threads within an agent. There are plenty of choices here. In the context of Oasis, counting semaphores and conditions look rather promising. However, counting semaphores has the potential for unstructured use, although they are generally useful for accounting resources. On the other hand, when agents are viewed as monitors, conditions become very attractive but are less convenient for counting purposes without additional state variables. In Oasis, we have devised a more general notion of conditions by combining the best of these two mechanisms.

The act of posting a notice to an Oasis condition is regarded as giving a *hint*, as in Mesa, and not about establishing some *truth* as in Hoare's or Brinch Hansen's. In Oasis, the *post* operation can be parameterized by a non-negative integer which indicates how many threads waiting on the condition are to be notified. As a special case, parameterizing the post operation with the number zero means broadcasting to all threads waiting on the condition. Similarly, the corresponding *wait* operation takes a non-negative integer which indicates how many notices must be received by the thread before it becomes ready again. As a special case, the number zero means to put the currently active thread at the end of the ready queue and run the next ready thread, i.e. do one step of round-robin scheduling. This extended notion of conditions had been very simple to implement and we have found them to be generally useful for counting resources. For example, they are used for counting slave agents when running a distributed branch-and-bound algorithm coded in Oasis for solving the Traveling Salesman Problem.

2.3 Garbage Collection

In Oasis, objects are *passive*, but agents could be either *reactive* or *active*. Passive objects are collectible in the usual manner, and for performance reasons we have chosen to use an almost tag-free garbage collector for Oasis. Reactive agents never die, i.e. their lifetimes extend as far as that of a run of the distributed program, and they are never collected. Active agents have limited lifetimes, and they are responsible for making the appropriate arrangements before terminating.

It is quite meaningless to apply the usual reacheability criterion to the Oasis model when trying to determine whether an agent is in use or not. An active agent that is unreacheable could still initiate remote procedure calls on other agents and affect the final outcome of the computation. So we have decided to eliminate the need for distributed garbage collection at the global runtime system level but instead have relied upon the agents themselves to individually deal with it locally at the user program level.

3 Programming Language Design

Now that the stage has been set for object/agent-oriented programming, the question becomes: what would be an effective programming language that would support the notions of objects and agents? It is conceivable that existing languages can be extended to support the object/agent paradigm, but we are interested in exploring alternatives. Our efforts result in a new programming language that is both simple and elegant. And more importantly, programs written in this language can be easily compiled into efficient machine code.

The Oasis programming language draws upon ideas from logic and object-oriented programming. A number of existing languages and systems, including Prolog, SML, C++, Argus, Actor, POOL, CLOS, Smalltalk, Mesa, Linda and Ada, heavily influence its design in various ways.

Oasis is a class-based, object/agent-oriented language. The semantic runtime entities of objects and agents are categorized as compile-time *classes* in the program. Classes are organized into a collection of user-defined single inheritance hierarchies. To be meaningful, the inheritance hierarchies never intermix classes of objects with classes of agents. Different levels of protection, i.e. public, protected and private, are used for determining visibility and access control, and they could be independently assigned to individual *attributes* and *methods* of each class.

A class definition has separate specification and implementation sections for static type-checking purposes. The type system treats subtyping as inheritance. For reusability consideration, classes of objects within some hierarchy can be uniformly made generic by including one or more generic type variables in their specifications. The Oasis type system can handle genericity in a sound fashion. Classes of agents, on the other hand, can be endowed with generalized condition variables with which to do threads synchronizations.

A method in the implementation section is written as a list of clauses resembling those of Prolog in syntax but not in semantics. For compiler efficiency and ease of implementation, each argument of a method must be declared in the specification section as operating in one of input or output modes, but never both. The general unification mechanism in Prolog has thus been reduced to the special case of term-matching in Oasis, where data can flow in only one specific direction and furthermore there can be no partially instantiated data structures. In contrast to Prolog, the control strategy of a single Oasis thread is single-threaded because a method invocation can produce at most one answer. In other words, backtracking is uniformly shallow and side-effects are predictable and are never undone. Furthermore, both static and dynamic method binding mechanisms are supported.

Lists and arrays are supported as built-in data structures, in addition to the usual integers, reals and characters. User-specified values are assigned by default to attributes of a class instance if they have not been overridden. Objects and agents can refer to themselves through the reserved word `self`. There is also a notation to designate specific machines for starting up particular agents. Furthermore, existing agents can be explicitly named in a handle by specifying its internet address and port number.

4 Implementation

It is anticipated that the overhead associated with conventional tagged methods of garbage collection will impact quite severely on the performance of the language. Oasis uses an almost tag-free garbage collection mechanism derived from those of Appel and Goldberg, and which dispenses with runtime tags altogether.

We have considered a compiler-based implemetation right at the outset concurrent with the design of the language, and have developed a Transcode abstract machine upon which we could base the construction of the Oasis compiler. The Transcode abstract machine is stack-oriented and has a very clean stack convention for handling method invocation and parameter passing, and which by design can be mapped very nicely to real microprocessor architectures. Transcode translators have been implemented for several architectures, including Sparc, Mips, RS/6000, and M68k. Translators generates native code through a simple but effective stack-based register allocation algorithm. Transcode instructions that reference the top portion of the abstract machine stack are translated to assembly code that references machine registers directly.

In addition, our implementation of Oasis uses the idea of a network-transparent compiler system, under which an Oasis user would never have to worry about maintaining different binaries for different ⟨microprocessor, object file format⟩ combinations on a heterogeneous network. Our inspirations come from Postscript, which is the ubiquitous page description language that every printer knows about, even though some may use a different internal page description language. Transcode is to Oasis what Postscript is to LaTeX, or any other high-level document formating languages. Basically, the Oasis compiler system achieves heterogeneous network transparency through the use of a scheme that delays the generation of native machine code until just before program start-up time, after identical copies of Transcode program have been transported over the network from a client machine to individual target server machines. Machine specific system calls are taken care of at link time when local libraries at individual target machines become bound into their respective object code prior to start of execution.

Oasis provides an interactive programming language shell in the style of Lisp or Prolog for enhanced productivity. In trying to avoid the extra work of creating an independent Oasis interpreter, we have decided to make the Oasis compiler reusable and in addition incorporated a dynamic loading facility into the shell. The Oasis shell compiles each user command into Transcode which it translates into native assembly, which it then assembles and links to form an object code that is dynamically loaded into the shell itself for execution. The shell can thus be viewed as a special interactive agent that executes the user's command, perhaps in cooperation with other agents in the network, and subsequently returns an answer upon completion and patiently awaits the next command.

5 Acknowledgements

This is a working draft and the author will be very pleased to hear comments and suggestions from its readers as it is being finished. The author will be especially grateful for pointers into existing systems or ongoing research on systems with similar features.