Two Models of Concurrent Objects[1]

Position paper

O.M. Nierstrasz[2]

We propose two models of concurrent objects that address, respectively, methodological and semantic issues of object-oriented programming languages. The first is a conceptual model to aid in the design of object-oriented languages for concurrent and distributed applications, and the second is a computational model that can be used to define the semantics of such languages. The second model has evolved, in a sense, from the first, though it is intended to be both more neutral and more general.

Traditional approaches to concurrency can be divided into two camps: those that view the world in terms of synchronized accesses to shared memory, and those that view everything in terms of message passing. A pure, shared memory view is inappropriate for object-oriented languages, since it separates data from the processes that manipulate them. Once we add data abstraction to the shared memory view, however, the differences between the two camps begin to cloud over. The remaining difference is between the kinds of "objects" that are passive or active.

We propose a unifying model consisting of *processes* and *threads*. Processes have a state, and may be either *active* (changing state) or *dormant*. Threads are virtual, and simply indicate which processes are active. There is at most one thread in a given process at any time. Threads may move from one process to another if the latter process is dormant. A thread that is not in any process is *suspended*. The total amount of concurrent activity in a system at any time is thus defined by the total number of threads actually in processes, that is, the number of active processes. Each thread originates in some process that "owns" it. We can then distinguish between passive "server" processes without their own threads, and autonomous processes that own one or more threads.

We can now use these notions of processes and threads as a reference model for describing the view of concurrency in the object model of a particular language or system. For example, we can distinguish passive and active objects by whether they have their own threads or not. We can also identify the granularity of concurrency by the correspondence between objects and processes. Typically "top-level" objects will map to processes, and sub-objects will map to part of the state of a process, but we may also consider objects with internal concurrency that correspond to systems of processes. The differences between various shared memory models and message-passing models can be understood in terms of the policies which determine when a thread may enter a process (i.e., locks, waits and signals, synchronous or asynchronous message-passing, etc.).

We have designed and implemented a concurrent object-oriented language called *Hybrid*, based on this model, in which an object is either a process, or is inside a process, as part of another object [Nierstrasz 1987; Konstantas, et al. 1988]. Objects communicate with one another by invoking operations and responding to invocations in a remote procedure call fashion. The trace of *call/return* communications corresponds to a thread. We can understand communication between objects in different processes in terms of the policy for admitting a thread. Threads are suspended on a queue (effectively a message queue) if the target is either busy, or blocked on a call of its own. This basic policy can be modified through the use of two language constructs, *delay queues* and *delegation*. A delay queue enables a process to selectively delay threads attempting to invoke certain operations. Delegation in Hybrid is a mechanism that enables a process to switch between threads by *not* blocking when calling an object in another process. Two other constructs, one for managing hierarchies of threads, and another for managing transactions, were designed, but not implemented. Although the language design adopted a message-passing communication model, the prototype implementation modeled threads as lightweight processes, and processes as shared, passive

[1]ACM SIGPLAN Notices, volume 24, number 4, Proceedings Workshop on Object-Based Concurrent Programming (San Diego, Sept 26-27, 1988), April 1989, pp. 174-176.

[2]Author's address: University of Geneva, Centre Universitaire d'Informatique, 12 Rue du Lac, CH-1207 Geneva, Switzerland. E-mail: oscar@cgeuge51.bitnet, oscar@cui.unige.ch, mcvax!cernvax!cui!oscar.

entities. The point is that the conceptual model of processes and threads made it fairly easy to propose and design communication and synchronisation primitives consistent with a concurrent object-oriented paradigm, independently of the implementation strategy.

Although this model is useful as a framework for understanding concurrent objects and for designing language constructs, it is inadequate as a computational model. In particular, it says nothing about either the "state" of a process, or the events that cause it to change state. We see the need for a computational model that will be useful:

- for defining the semantics of concurrent, object-oriented languages like Hybrid,
- for comparing mechanisms of various languages and their implementation environments,
- and for aiding language designers by providing a basis for language definition tools.

We propose a new computational model that combines ideas from CCS [Milner 1980] and Actors [Agha 1986]. Our motivation for a new approach is based on the following positions:

1. *Concurrency cannot be modeled by non-determinism.*
2. *Programs are not functions.*

The first statement means that we reject approaches that attempt to interpret concurrency by an interleaving semantics. Instead, events in a concurrent computation should be seen as being partially ordered. We believe there is an important difference between multiple observers seeing different orders of events in a truly concurrent computation, and a mono-processor non-deterministically selecting a particular total (i.e., serialized) order on the events of a pseudo-concurrent computation.

The second statement expresses the conviction that standard views of programs as functions from inputs to outputs, not only discriminate against object-oriented languages by separating program from data, but they are poor at capturing concurrent computations built up of systems of cooperating programs.

Instead, we believe that computation, especially concurrent computation, can be better modeled in terms of communicating systems of concurrent agents. Rather than distinguishing between the finite control and the "input" to a computation, we model them together as an initial system of concurrent agents (i.e., processes, or "objects"). The progress of a computation can be observed as a partial order of events, where each event represents a (synchronous) communication between a pair of agents, and yields a new, possibly concurrent, behaviour for each of the participants of the event. The "output" of a partial computation is a new system of agents, which may then continue to participate in events, if any are possible. Computations may or may not terminate.

We have designed and implemented a simple CCS-like language called *Abacus* based on a subset of these ideas [Nierstrasz 1988]. Agents are specified using behaviour expressions which encapsulate the events (communications) the agent may participate in. Behaviour expressions consist of input and output offers (i.e., guards) on event names. Operators on behaviour expressions include non-deterministic choice and concurrent composition. A behaviour expression for a system of agents is a static description of possible computations that may result. Events may take place when there are matching offers between concurrently composed agents. The resulting partial order of events is effectively a history of the computation for multiple observers. Any initial sequence of observed events yields a new behaviour expression that describes the remaining possible computations.

The current version of Abacus has only the power of finite automata (there is a finite set of reachable states for any system). We are presently searching for the right set of primitives that will extend Abacus to be computationally complete, yet permit behaviour expressions to remain directly interpretable. Our long term goal is to use Abacus as a tool for defining the semantics of languages like Hybrid, and for providing a formal and implementable basis for studying and comparing constructs of concurrent and object-oriented programming languages.

References

[Agha 1986] G.A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[Konstantas, et al. 1988] D. Konstantas, O.M. Nierstrasz and M. Papathomas, "An Implementation of Hybrid, a Concurrent Object-Oriented Language", in *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[Milner 1980] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

[Nierstrasz 1987] O.M. Nierstrasz, "Active Objects in Hybrid", ACM SIGPLAN Notices, Proceedings OOPSLA '87, vol. 22, no. 12, pp. 243-253, Dec 1987.

[Nierstrasz 1988] O. Nierstrasz, "Mapping Object Descriptions to Behaviours", in *Active Object Environments*, ed. D.C. Tsichritzis, Centre Universitaire d'Informatique, University of Geneva, June 1988.