DESIGN OF AN OBJECT BASED MULTIPROCESSOR SYSTEM

3N-6

S. Watari, L. Nagamatsu, I. Morishita

Department of Engineering, University of Tokyo

1. INTRODUCTION

This paper gives an outlook of the design of a parallel object-based system, with special considerations on the communication aspects, regarding both computational model and implementation architecture.

The system primitives are designed to be sufficiently general to allow experiments with several kinds of object based languages. Concerning objects granularity, our purpose is to start experimenting with relatively small objects (medium-sized C programs), and then gradually improve the architecture design for giving support to finer grain objects.

From the architecture standpoint, our targets are tightly-coupled multiprocessors. As detailed in SECTION 4, we have given special support for synchronization between objects, introducing a special hardware for managing a global object table.

2. OBJECTS AND ACTIVITIES

The central concepts of the system are objects and activities. An object abstracts a particular aspect (either a concrete "thing" or a concept) of the real world. It is the unit of modularity in the system and encapsulates both the data representation and the behavioural representation (i.e., procedures, functions, etc. that act upon the data). An activity abstracts the actions taken by an object in the system, i.e., an activity is the system representation of the dynamic behaviour of the object.

The concept of activities has helped us to understand (and also to implement) paralellism. Each activation of an object, be it sequential or parallel, is represented by an activity. Therefore, a typical object can have many activities associated with each concurrent (potentially parallel) execution.

Problems in parallel execution of objects arise when we allow objects that change their internal state to be shared among other objects. With regard to the state of the object, the system supports two kinds of objects: *history sensitive objects*, whose internal state depends on the activation - and *non history sensitive objects* - whose internal state does not change between successive activations.

Non history sensitive objects can be shared without restrictions. It can be activated any time, by any object (once the activating object possess the necessary rights).

History sensitive objects must be synchronized their activations if their internal state is to be kept consistent. Regarding mechanisms for controlling activations, the system supports two kinds of history-sensitive objects: *atomic objects* and *quasi-atomic objects*. *Atomic objects* do not allow simultaneous activations. Their entrance are serialized, and at any time, there is at most one activity representing its execution. *Quasi-atomic objects* allow simultaneous activations, but only in particular points of their execution. The name is boorowed from the concept of quasi-concurrency, and the mechanism of control required to synchronize quasi-atomic objects closely ressembles *monitors*, a way how mutual exclusion and synchronization on quasi-concurrent processes are implemented.

3. INTER-OBJECT COMMUNICATION

Objects are activated by sending them messages. In our system implementation, rather than considering synchronization signals as single-slot messages, we have decomposed the message passing mechanism into two parts: the synchronization part and the data transfer part. The same mechanism handles both pure synchronization signals (as those utilized in a monitor for condition variables) and the synchronization for activation of/return from objects.

The synchronizing primitives provided by the system are *wait* and *signal*. They are capable of implementing both semaphore operations as well as monitor's operations on condition variables. For supporting special types of activations and returns, they allows for multiple signals (broadcasting) and multiple waits. The basic function of *wait* is to temporarily stop execution of an activity. *Signal* can either be the synchronization part of an *activate* message or utilized to restart (awake) a stopped activity.

With regard to synchronization, the system provides for two kinds of activations, namely synchronous and assynchronous. Assynchronous invocation is performed with a primitive operation with a syntax like this:

activate(target object, entry method, ...)

When a further result is needed, the invoking activity executes an *accept* primitive with an argument specifying the *event condition* associated to the result. The invoked object executes a *return* primitive for sending back the required result.

Synchronous activation always require a return message from the invoked activity. It is supported by a *call* primitive, which is quite the same as performing the two assynchronous primitives, *activate* and *accept*, in sequence.

4. PROTOTYPE ARCHITECRURE

The main distinctive characteristic of the prototype system is the adoption of a microprogrammed *Central Event Synchronizer (CES)* linked to each computer module by a common bus (*EBus*, for Event Bus). The *CES* holds a centralized global object table. Each computer module has a relatively autonomous processing capacity, with a processor element and a local memory for holding object's code and data.

Though there are arguments against centralizing a heavy-used feature such as the global object table, there are some motivations for doing so.

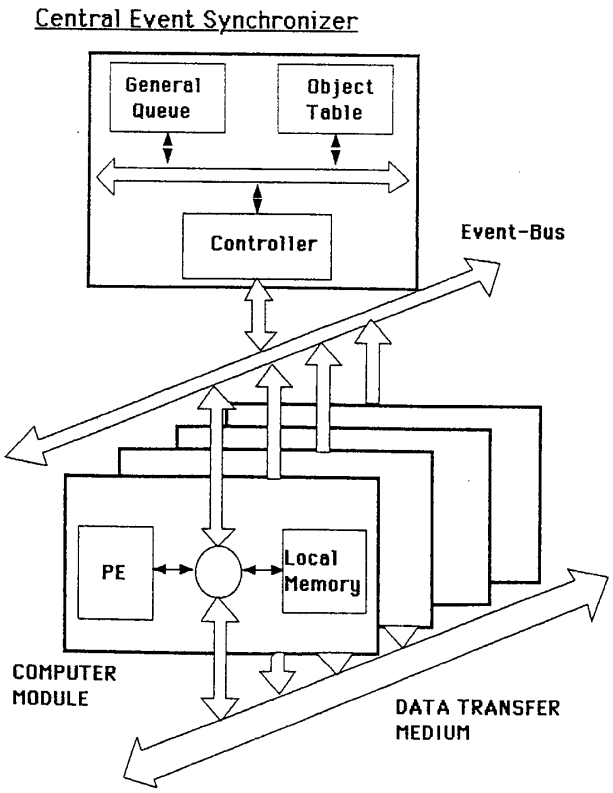
In an environment where objects are supposed to be created and deleted dynamically, at every time, there are needs for managing the name (i.e., the precise location of the object) in a centralized way. The approach we have adopted is to try to reduce the centralized part to a minimum and give architectural support for avoiding traffic congestion. The central table and the *Ebus* serve the only purpose of handling synchronization between objects. Data transfer, the other aspect of inter-object communication, is carried out through a second transportation medium, eventually slower than *EBus*.

5. CURRENT STATUS OF THE PROJECT

The prototype system is being implemented on a multiprocessor system with three MC68020s. The CES is an extension of the unit developed in an earlier project [KOTOKU87] [NAGAMATSU87] with additional capabilities for handling a global object table and parameterized inter-computer module signals.

An emulation system is being coded on a SUN workstation running UNIX. Both systems are being coded in G++, a GNU version of C++

SYSTEM CONFIGURATION



[STROUSTRUP86]. The same language, extended with concurrency features, is planned to be implemented on the system.

REFERENCES

[KOTOKU87]
Kotoku, T., The Design and Implementation of a Synchronization Management Unit for Multiprocessors (in Japanese), Department of Mathematical Engineering and Information Physics, University of Tokyo, Master Thesis, 1987.

[NAGAMATSU87]
Nagamatsu L., Kotoku, T., Watari, S., and Iwao, M., Event and Process Management Hardware on a Multiprocessor Environment (in Japanese), Proc. JIPS 35'th National Conference, 1987.

[STROUSTRUP86]
Stroustrup, B., The C++ Programming Language, Addison Wesley, 1986.