CLAP[1]: An Object-Oriented Programming System for Distributed Memory Parallel Machines

Abstract: The Agha and Hewitt Actor model [1, 3, 4] is a natural extension of the object paradigm as we know it to the field of parallel programming. The model respects the primary principles of object-oriented programming, i.e. abstraction of data and encapsulation, making object-oriented programming a good tool for the programming of parallel computers. We *describes one Actor model implementation on a distributed memory parallel machine: the* CLAP *actor language. We explain why the Actor model can express, on distributed memory parallel machines, the natural parallelism of applications and offer other properties useful for programming this type of machine. We then discuss the implementation, currently in its final phase, of the Actor model on a distributed memory parallel machine.*

Key Words: *Parallelism, object-oriented programming, Actor language, distributed Artificial Intelligence.*

Jocelyn Desbiens, Martin Lavoie, Stéphane Pouzyreff, Pierre Raymond, Tahar Tamazouzt, Michel Toulouse
Laboratoire d'Intelligence Artificielle (Projet Volvox)
Collège militaire royal de Saint-Jean
Richelain (Québec)
J0J 1R0
E-mail: volvox@cmr.ca

1. Introduction

Object languages are built around the notions of abstract data type (ADT) and data encapsulation, the later being a kind of side-effect of the implementation of the ADT notion. ADT is implemented either statically in the form of modules or dynamically with classes, both via the definition of interface protocols. These characteristics of object languages have an effect on the way problem solving is modelled and processing is accomplished. In fact it favors a programming style in which a problem is decomposed into several independent tasks that could resulted at processing time into discontinued (loosely couple) sequences of control and data dependencies.

Abstraction and encapsulation aim to ease the development and modification of programs by enforcing the locality of operations. If the design of an application have been done correctly, chances are that objects in the program will correspond to definite structures in the application. In this case, data dependencies between objects should be weaker than intra-object data dependencies. From a parallel programming viewpoint, objects can represented implicitly the parallelism available in an application and become naturally a unit of execution in a parallel programming environment. When interface protocols are strictly enforced, control and data exchange between objects is done through message passing only, which is similar to the message passing paradigm for distributed memory computer systems. Also, strictly enforced interface protocols provide for free, synchronization mechanisms for objects running on different processors and can granted exclusive access to code section similar to monitors in parallel programming.

From a software engineering viewpoint, one of the promising factors stemming from the emergence of object languages was the notion of "code reutilization". Quickly, software designers foresaw the emergence of libraries gathering general objects from which new software will be developed by specifying those general objects. From this concern with code reutilization through object specialization stems the notion of object levels, more specifically the notion of inheritance.

As we know, classes and inheritance introduce two new categories of relationships between the objects in a system: the relationship "is a", which indicates that an object is one instance of a class; and the relationship "kind of", which

1. C++ Libraries for Actor Programming

indicates that an object's class is a subclass of another object's class. Certain languages provide primitives enabling these new relationships between objects to be expressed directly. `Smalltalk` is one example of this. These languages are referred to as class-based languages. Unfortunately, for most of these languages, communication between an object and its class, or between one class and its subclass, is not accomplished via the interface protocol. Most important, the implementation of inheritance conflicts directly with the way ADT and data encapsulation is implemented for object languages, by the very definition of an interface protocol. As a consequence, generally, the principle of encapsulation is weakened.

From a parallel programming viewpoint, the weakening of the notion of data encapsulation lessens the interest that one might have in the object paradigm. When information exchange in the inheritance chain or between an object and its class does not respect the interface protocols, it directly increases the data dependencies between objects, weakening the usefulness of designating the objects as the unit of computation in a parallel application. The uniform model of communication between tasks, the task synchronization mechanisms and the mutual exclusion of processes are also weakened all the more.

2. The Actor Computation Model

The Actor model was perfected at MIT by Carl Hewit **[3]** and his team. Within the Actor model, the analogous role played by the objects of object languages is identified by the term "actor". An actor is defined by its script, i.e. what corresponds to the definition of an object module within object languages. Essentially, an actor is an independent computation agent which carries out its actions in response to a message it has received. Actors interact on a purely local basis by sending messages to each other **[4]**. An actor may thus be modelled as follows:

- a behavior, which corresponds to the actions undertaken by the actor to process the current message;
- an address, at which a mailbox containing messages received by the actor is located.

The term "actor" itself emphasizes another aspect of programming within the Actor model, i.e. programming via concurrent processes. The object paradigm was designed for a monoprocessor environment, in which objects are scheduled each in their turn. At the opposite, an "actor" designates not only the user program code but also the execution environment of this code. In this sense, an actor is active, contrary to an object which does not possess its own run time environment.

The notion of class does not exist within the Actor model. The creation of a new object (actor) is done via cloning and not by instantiation of a type as in class-based languages. The object from which the clone copy is taken is called the prototype. The new object can, in turn, become a prototype for another object. By the omission or addition of properties, the new copy may not be identical to its prototype. In this way it is possible to model a problem incrementally in an Actor language without having to use a class hierarchy.

2.1. Delegation

The notion of inheritance is also absent from the original Actor model. The only way in which two objects may exchange information is by passing through the interface protocol. This property of Actor model guarantees that the principle of encapsulation will be respected. From a parallel programming viewpoint, this property makes the object paradigm entirely worthy of consideration once again, since the Actor languages have properties which force program execution to be isomorphic to the semantics of their script.

Hewit's **[3]** original Actor programming model did not provide for a code-sharing mechanism. It was Lieberman **[6]** who, in 1986, first suggested a mechanism for the reutilization of codes within the Actor model which does not violate the principle of encapsulation, i.e. the delegation mechanism. Delegation is a protocol for sending messages between actors that could have the same effect as inheritance in class-based languages. The communication protocol of the delegation schema is uniform: delegation is accomplished via message passing and through the interface protocol.

3. The CLAP Programming Environment

Based on the principles given above, we have created un object-oriented programming environment for a distributed memory computer architecture that implemented the following concepts of the actor model: the notion of *actor, behaviors*, the *mailbox, parallelism at the actor level* and we added *intra-actor parallelism to the original Actor model.* This environment have being implemented in `C++`. We chose `C++` as platform rather than `Smalltalk` or `ABCL/1` because it is a compiled language and therefore is *a priori* more efficient than `Smalltalk` or `ABCL/1`; secondly, `C++` cross-compilers generating code for the target platform (a 64 processors transputer-based computer) are widely available, and finally because we were able to obtain the `ACT++` system class library, a system developed by D. Kafura **[5]** of Virginia Tech. University, whose objectives were similar to ours.

3.1. Overview of the CLAP system

An application running under the CLAP system is made of at least one program. Generally however, this application will be made of many programs, each compiled separately and distributed over the processors of a parallel computer by a load balancing algorithm. When linked with the CLAP library, each of these programs is actually a task under the control of the CLAP run time environment. In CLAP, each actor is member of a given task. The CLAP run time environment is there to give services to the actor members of the task. In the actual version of CLAP, those services implement a form of uniform address space in the distributed memory system. We used for that XDR filters and types information for inter-processor messages encoding and decoding, and a scheduler that controls processes execution inside the task. In each task there is also a message server that controls the message reception for actors in the task.

It is up to the programmer to decide how many actors there will be for a given task. Generally, if this number is large it could means the lost of potential parallelism in the application. On the other side, if the computation is communication intensive between some actors (sometime resulting from a bad design), inter-processor communications may cause an overhead larger then the gain obtained from the parallel execution. In this case, actor concurrence by time sharing of a processor in the same task might be a wiser option.

3.2. The Behavior Class

The Behavior class determines the actions that an actor may take while processing a message. In the Actor model, an actor is an object with several interface protocols; this means that an actor may behave in several different ways.

The principal method of the Behavior class is become, whose role is to change the actor's interface protocol after or while a message is processed. The notion of changing the interface protocol during the existence of a particular actor within the Actor model plays two principal roles:

- it enables the actors to have a distinct behavior and gives them access to the computation history;
- it enables an actor to process several messages at once, thus adding another level of concurrence to that of the concurrent execution of several actors within the system.

The become may not be localized at the end of an actor's script, and thus can enable a level of intra-actor parallelism to exist. The become as well as the processing of the next message can be initiated, even if the processing of the current message is not finished. In this way, we can have several copies of the same actor executing concurrently. The role of the become position in the script is then similar to a semaphore, guaranteeing one process exclusive access to shared actor variables in the section of the script preceding the become. It is the programmer's responsibility to ensure that the become is placed in the correct location in order for the variables shared by possible concurrent processes to be updated while respecting the semantics of the program.

4. Details of the Current CLAP Implementation

The ACT++ classes which implement the Actor model abstractions were written presupposing a shared-memory machine. For this type of architecture, the address of an object is determined by the compiler and the operating system which associate with each variable's name the virtual or real address of that variable. Thus, all the ACT++ classes implementing the message passing between objects were based on this use of pointers. In the context of a distributed memory architecture system, reference between objects on different processors based only on pointers is not possible. Generally, operating systems for distributed memory architecture support direct references between the name of an object and its memory localization only for the processor's local memory. This situation is related to the fact that most operating systems for this kind of machine have still to implement the concept of uniform address space for local memories. For systems that need this uniform address space, it is possible to create tables that keep records of the correspondence between the name of objects and their processor localization. The implementation could be through only one global table or many local tables or a mix of both strategies. If only one global table is used, the implementation will be easy but accessing this table will create a bottleneck. The use of many correspondence tables distributed in the system will solve the bottleneck problem but coherence problems will render this strategy difficult to implement. In the CLAP system, we have chosen the global table strategy and reduce the impact of the bottleneck access on performances by creating cache memories on each task in the system. Cache memories prevent global table access for objects that are locals and also prevent repetitive access for the same object (since the first access causes the copy of the correspondence in the local cache memory and there is no object migration in the actual version of CLAP).

It is a global data base in the CLAP system that records correspondence between actors and their processor localization. This global data base is on one processor and its localization is known to every task in the system. This global data base knows all the actors existing in the system at any given moment. It is updated by a data base server as soon as

an actor is either created or destroyed. The function of this server is also to respond to data base consultation requests and send back to the creator of an actor a number identifying uniquely the newly created actor. Each entry in this data base corresponds to an actor name, but may also contain the actor's number[2]. Local cache memories consists of a network of "local actor data bases", for tasks created by an application. These local data bases are updated as soon as an actor is created locally or when the global data base is consulted.

Once the address of an actor is known, in a distributed memory system, one still has to send the message. For every argument of a method, the programmer has to know the type of the argument and its size in order to code this information in his application. In CLAP, we have chosen to ease this work and to leave the user the possibility of executing an application on different types of processors. In order to realize this objective, every message in CLAP is encoded and decoded using the XDR protocol. However, for this encoding to be possible, an object sending a message must know the type of every argument of the object method to which the message is sent. Once again, this problem is solved in CLAP by creating tables for each task involved in the execution of an application. Contrary to the actors which are created dynamically, the types of the method parameters are known at compile time. This makes possible an approach based on the code source analysis in order to build tables for message passing.

Using `lex`, `yacc` and the C++ grammar, we have developed a preprocessor that extends the syntax and grammar of C++ to the CLAP language. This preprocessor scans the user's source code and generates C code that is used to build appropriate "local message data bases". Each entry in the message data base contains all the information needed to encode a message to an actor that resides on another processor.

4.1. The `Message` Class

The `Message` class coordinates the sending of messages between actors. In cases where both actors are located in the same processor, the situation is the same as if we were dealing with a shared-memory environment and thus we have implemented `send` methods that uses pointers.

Suppose two actors **A** and **B**, based on different processors, wish to communicate. The initiator of the communication (for instance, **A**) will create a structure frame. To fill this structure with the message, the constructor of the class `Message` consults the local message data base in order to get the type of each field of the message to be built and the address of the XDR filters. To find the address of the actor **B**, a request is made to the local actor data base server. If an actor with the name **B** is not found locally, a request to the global actor data base is done which is followed by an updates of the local data base with the reply obtained. Finally, a call to network primitives is done for the sending of the message.

At the other end, Actor **B** does not receive the message directly from **A**. It is the message server of the task **T** which contains **B** that receives all messages from other processors. Once the message server of task **T** receives a message, it extracts from it the name of the actor to which the message is intended. With this name, the message server sends a request to the local actor data base which returns the initial behavior of actor **B**. Then the local message data base is searched in order to find the address of the XDR filters for the decoding of the message received for actor **B**. Once the decoding is done, the message server puts in the mailbox of the actor **B** the received message.

In the Actor model, communication between actors is asynchronous. This means that an Actor **A** which sends a message to an Actor **B** may continue to do its own processing between the time after it sends the message to **B** and the time it receives the reply. Replies must be processed differently than the simple receiving of a message from another actor. They cannot be placed in the actor's mailbox for obvious deadlock reasons. The original Actor model had adopted the *insensitive actor* approach **[1]** to process these messages. A second approach, Coromel's *waiting by necessity* approach **[2]**, consists of creating a special mailbox for each actor - a `Cbox` - to receive this type of message. Actors check their `Cboxes` periodically to see whether a message has been received. This is the approach used by ACT++ and also in `ConcurrentSmalltalk` **[7]**. In ACT++, the `Cbox` class is passed through as an argument in the message sent to an actor. In a distributed environment, the sending of a class as part of a message is much harder to implement. Furthermore, the message server on each CLAP task enables us to implement the `Cboxes`' operational capacity without having to use a `Cbox` class, as is the case with ACT++. The CLAP implementation therefore consisted of creating a method in the class `Message` which unscheduled an actor when it has to await a reply. When the reply arrives, the actor is "woken up" by the message server. If the reply arrives before the actor is ready to process it, the reply is placed in a field reserved for the reply to the message in the data structure created when the original message was sent. This field is itself a complex data structure able to receive several replies via message passing. The actor uses the `received` method in the class `Message` to consult the message structure when it is ready to process a possible reply

2. The number of an actor is not contained in the data base; it is calculated each time the server accesses the data base.

to the original message. When the decoding is done, the message server puts in the mailbox of actor **B** the received message.

5. Conclusion

There are three well-known approaches to the development of programming languages for parallel machines:

- automatic parallelization of the sequential code based on the analysis of data dependencies (PTRAN, Paraphrase II and PTOOL);
- the addition of primitives to languages that are basically sequential (ABCL/1, Ada, ParallelC and ConcurrentProlog);
- the design of new languages for parallel architectures (SISAL, VAL, Linda and Occam).

The CLAP system belongs to the third category. In this system we have attempted to merge the characteristics of object-oriented programming and the resolution of parallel programming problems. Therefore, in a preliminary version of the system, a first stage involved implementing most of the Actor model concepts. This relation to the Actor model enabled us to develop an object-oriented programming system which preserved the interesting characteristics of the object paradigm in a physical parallel machine environment. We used C++ to create the Actor model abstractions.

A second phase of the research involves implementing code reutilization by delegation and the creation of actors via cloning. As soon as the CLAP system operates very well on the current T800 network, the software tools we are using will enable us to bring CLAP into a heterogeneous (sequential and parallel) computer network. Looking further ahead, our laboratory will examine the integration of the two paradigms, i.e. object-oriented programming and parallel programming, more closely. To accomplish this, we will study reflexive programming and the performance of parallel programming via complex interaction between concurrent processes. The results of these studies will be incorporated into CLAP and this system will serve as a testing platform for new programming concepts.

6. Acknowledgments

We are thankful to Professor Dennis Kafura of Virginia Tech. University for having given us the source code of the ACT++ system and for his numerous advices.

7. References

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

[2] D. Coromel, *A generalized model for concurrent and distributed object-oriented programming*, Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, 1988.

[3] C.E. Hewit, P. Bishop and R. Steiger, *A Universal Modular ACTOR Formalism for Artificial Intelligence*, in "Proceedings of the 3rd IJCAI", Stanford, California, 1973, pages 235-245.

[4] C. Hewit, *Control Structure as patterns of passing Messages*, in "Artificial Intelligence: an MIT Perspective", vol. 2, The MIT Press, Cambridge, Massachusetts, 1979.

[5] D. Kafura and K. Lee, *ACT++: Building a Concurrent C++ with Actors*, Journal of Oriented Object Programming, May/June, 1990, pages 25-37.

[6] H. Lieberman, *Using prototypical objects to implement shared behavior in object oriented systems*, in "Proceedings of the First ACM Conference on Object-Oriented Programming Systems, language, and Applications", Portland, Oregon, 1986, pages 214-223.

[7] Y. Yokote and M. Tokoro, *The Design and Implementation of ConcurrentSmalltalk*, in "OOPSLA'86 Proceedings", ACM, New York, 1986, pages 331-340.