Plasma-II: an Actor Approach to Concurrent Programming

Guy Lapalme

Département d'informatique et de recherche opérationnelle
Université de Montréal
CP 6128, Succ "A"
Montréal Québec Canada H3C 3J7

Patrick Sallé

Laboratoire LSI
Université Paul-Sabatier
118 route de Narbonne
31062 Toulouse France

1. Introduction

This paper describes Plasma-II, a "minimal" extension of Plasma which was the first actor language defined by Hewitt and that has been implemented and in use in Toulouse for about 10 years: a portable version of a Plasma interpreter was developed using a virtual machine called LILA (Carré et al. 1984).

The actor paradigm describes a program as a set of autonomous actor instances that communicate via message passing. The only assumption on a message is that it is guaranted to be received in a bounded time interval and that an actor waiting to execute will enventually do so. So depending on the current context, actors can be executed either in a time-sharing, in real parallelism or even sequentially without any impact on the final results (except for the duration of the program).

2. Presentation of Plasma-II

Plasma-II[1] is an actor language designed to be executed on a set of virtual machines communicating by messages that can be distributed on diverse types of hardware architectures. Each virtual machine is running on a fixed physical processor that can execute many actors in a time sharing mode with the classical mechanisms of priority, critical section, etc. An actor can send a message to another one on any other virtual machine.

There are two kinds of actors in Plasma-II: pure actors and serialized actors.

2.1. Pure actors

A pure actor is created when it receives a message actor and it dissappears when its script has been executed or more precisely when its continuation is empty. It is the type of transmission with or without waiting for an answer and with or without continuation that determines if it is necessary to create a new process on a virtual machine to execute this instance or if the computation can be executed within the sending process. The following table shows the four kinds of transmission modes in Plasma-II where the message *M* is sent to the actor *A*:

[1] the name Plasma-II can be interpreted either as Plasma two or as Plasma parallel: an "upward" compatible parallel extension of Plasma

TRANSMISSIONS	BLOCKING	NON BLOCKING
SEQUENTIAL	*A <= M* the sender waits the answer the process sending M is idle it can thus execute the processing of M defined by the actor A anologous to a function call	*A <== M* as in this case the continuation of the sender is killed, the processing of M can be substituted in the process of the sender analogous to an Unix *exec*
PARALLEL	*A <- M* the continuation of the sender receives immediately a place holder for the value. The sender waits when it accesses the value if it has not finished computing A new process has to be allocated	*A <-- M* the sender continues its processing and so a new process has to be allocated. Analogous to an Unix *fork* and *exec*

2.2. Serialized actors

The automatic creation of a new actor at each transmission makes it hard to program a shared resource controler because in this case one would like to have a sequential processing of the message and be able to memorize the state of the resource. To ease this kind of processing, we give the possibility to *serialize* an actor which creates an unique instance of this actor with a message queue. This actor deals with the messages sequentially and can change its behavior between the messages.

2.3. Combining and synchronizing actors

Actors can be grouped together using sequences of which there are two forms:

- the sequential one which computes its elements sequentially

$$[A_1 \ldots A_n] \Rightarrow [eval(A_1) \ldots eval(A_n)]$$

- a parallel one which does not preserve the order of evaluation but keeps the order of the results

$$\{A_1 \ldots A_n\} \Rightarrow [eval(A_1) \ldots eval(A_n)]$$

the A_i are evaluated in parallel, the sequence of results being built only when all A_i have been evaluated.

2.4. Broadcasting

A message can be broadcasted with any type of transmission

([A1 ... An] <-- M) is equivalent to a "fork" by sending *M* to all A_i in parallel, and returns a sequence of *fail* or *success* to the current continuation

([A1 ... An] <== M) the same but killing the current continuation

([A1 ... An] <= M) returns the list of sequentially evaluated results

([A1 ... An] <- M) returns a sequence of *futures* that will be gradually replaced by their values.

3. Controling the distribution

Actors in Plasma-II are usually distributed automatically on the virtual machines but a user can also control it. If the receiver of a message is an actor whose value is a virtual machine,

(machine <= expression)

returns the value of *expression* but indicates where this value will be *installed* (and not computed). This can be used either to install instances of actors to deal explicitely with the distribution of the computing power or to create distributed data structures which are then sent messages. We see that the actor paradigm permits a very simple unification of data and processing elements which are designated

uniformly. So, in Plasma-II, not only are data and programs represented in the same way but this identity is extended to the computing elements.

This mechanism distributes the data and allows a data parallelism style of programming. We have shown in (Lapalme, Sallé 1988) how the *xapping* notion of Connection Machine Lisp can be represented in Plasma-II.

4. Implementation

Plasma-II is being implemented on the SMART[2] virtual machine. It is a parallel extension the LILA (Carré et al. 1984) virtual machine and consist of a set of virtual machines distributed on physical machines communicating by message passing. The first version is running on a network of Sun workstations. When the Plasma-II interpreter is started, we choose the machines to be used for installing virtual machines to execute processes in a time-sharing mode. Each virtual machine gives access to the usual mechanisms of priority, critical section and passive waiting. Processes can send messages to any other independently of their physical location.

Each virtual machine has a local memory to keep data structures, variables and informations that are global to all the processes it executes. Each process is associated with an execution context comprising a set of registers and a stack kept as a list of continuations in the local memory. Two processes on the same site communicate by sending only the address of the message in local memory. The communication between virtual machines is done using Unix "sockets" and so the message have to be translated into a "printable" form upon sending and retranslated upon reception. Garbage collection can be easily implemented for all processes on the same virtual machine but the general problem of garbage collecting distributed processes still remains.

5. References

Carré F., Durieux J.L., Julien D., Sallé P. (1984) "LILA: Langage d'Implémentation pour Logique et Acteurs", Actes des Journées AFCET sur les langages orientés objets, Bulletin Bigre, p 68-85.

Lapalme G., Marcoux A., Maurel C., Sallé P. (1988) "Plasma-II: version 88", rapport LSI, Université Paul-Sabatier, Toulouse.

Lapalme G., Sallé P. (1988) "An Actor Based Unified View of Control and Data Parallelism", rapport LSI, Université Paul-Sabatier, Toulouse.