Development of an Environment for Specification and Execution of Active Objects on Parallel Machines

Philippe Gautron, Jean-Pierre Briot
Hayssam Saleh, Sylvie Lemarié, Loic Lescaudron

Rank Xerox France & LITP, Institut Blaise Pascal
Université Paris VI
4 place Jussieu, 75252 PARIS CEDEX 05, France
briot@litp.fr, [gautron,lemarie,lescaudron,saleh]@rxf.ibp.fr

Abstract

This paper describes the work achieved by the RXF/LITP team during the two last years under the European Community Parallel Computing Action. The goal of the project was the design and the implementation of an environment based on object-oriented techniques to program concurrent and distributed applications on parallel machines.

First, this paper introduces an execution environment written in C++ dedicated to a parallel machine. This environment includes a concurrent extension of C++ towards active objects, a system layer for object allocation and execution, a class library to interface the system layer and a high level tool to build distributed applications. Then the paper presents Actalk, a concurrent extension of Smalltalk to prototype and visualize programming environments on a standard workstation. The integation of active objects in a sequential environment is discussed, particularly with regard to object visualization and scheduling. An assessment of the project concludes the paper.

1 Introduction

This paper describes the work achieved by the RXF/LITP team during the two last years under the European Community Parallel Computing Action. Implementation, experience gained and assessment are expressed in relation to the previous goals.

The overall goal of the project [Briot and Gautron 89] was the design and the implementation of an environment based on object-oriented techniques to program concurrent and distributed applications on parallel machines. Rather than, for example, focusing on a formal approach, we advocate the needs for specific environments (1) to help specifications, (2) to visualize experimentations, and then (3) to translate applications on a parallel execution environment. Object-oriented techniques are the common denominator to the different components of the project.

The environment we propose includes two different components: (1) an execution environment written in C++ and running on a parallel computer, and (2) a specification and experimentation environment running on top of Smalltalk-80[1] on a standard workstation. These two components can be stand-alone but were designed to be integrated in a distributed application development chain.

An important characteristic of our project was to rely on existing standard languages (Smalltalk and C++), on a commercial machine dedicated to parallel computation (T-NODE) and on an available operating system suitable to a parallel machine (Helios[2]). These choices ensure a certain durability of our work in a domain where portability is an important issue. However, the languages cited were fairly designed with sequential environment in mind. This characteristic underlies the object model they supply: activation of passive objects within the executing environment of the whole application. On the contrary, concurrent and parallel programming needs the support of active objects with a separate executing environment attached to each running object. Thus the first observation we can do about our project is that the two components needed, at first, to extend the object model of Smalltalk

[1]Smalltalk-80 is a trademark of Parc Place Systems.

[2]Helios is a trademark of Perihelion Software.

and C++ to support active objects.

The rest of the paper is organized in two parts. Each part describes goals, implementation and experience of one component of the project. An overall assessment concludes the paper.

The hardware and software platform we used during the project include SUN-4 workstations, a T-NODE machine with 16 transputers (4M), the Helios operating system version 1.[1-2], Smalltalk-80 version 2.[4-5] and C++ version 2.[0-1].

2 System Layer

2.1 Objectives

The objective of the system layer was to provide the foundations for running concurrent and distributed applications on a parallel environment [Saleh and Gautron 91]. During the specifications, the following points were of specific interest:

- design and implementation of a concurrent extension of C++ towards active objects.
- design and implementation of a minimal system layer and of a class library in order to allocate and execute active objects on different nodes.
- extension of C++ to support object migration and persistence
- high level execution layer for building distributed applications.

Almost all these objectives have been reached, sometimes in a different manner that we advocated. For example, with respect to migration, we found unnecessary to extend the language and we decided to provide this service through a class library, matching de facto the object-oriented paradigms. The sections below describe the results of our work with regard to the objectives listed above.

2.2 A Synchronization Mechanism for Concurrent Object-Oriented Programming

This section focuses on the integration of concurrency in class-based object-oriented languages. Many researchers have already pointed out the interference between object-oriented features and concurrency control mechanism [Saleh and Gautron 91]. Our goal was to implement a concurrency control mechanism for C++ objects that does not interfere with other language features. More precisely, our motivation was to address the following issues:

- it should be possible to design parallel object-oriented applications by reusing sequential components as such. For example, ACT++ [Kafura and Lee] and Rosette
  [Tomlinson and Singh 89] supply a concurrency control mechanism requiring to re-write existing library components, thus restricting code reusability.

- concurrency control model should not have to interfere with other language behaviors. For example, Guide [Decouchant et al. 89] synchronization constraints interfere with inheritance, thus restricting language support.

- it should be possible to parameterize the concurrency control mechanism with informations included in incoming messages. For example, in the family of POOL languages [America 87, America 90], synchronization constraints can only be expressed according to the internal state of the object. Methods arguments cannot be parameters of constraints, thus restricting message acceptance policy.

We describe below how our concurrent extension of C++ copes with these issues through an example, the readers and writers problem.

2.2.1 Reuse of Sequential Components

The class definition below describes the interface of a standard C++ class with two member functions `read` and `write` that conform to the usual reader-writer scheme:

```
// Item is an arbitrary user-defined type
```

```
class ReaderWriter {
    // private stuff
  public:
    Item read (int position);
    void write (Item, int position);
};
```

Our concurrent C++ compiler automatically generates code that makes any operations to be executed in *mutual* exclusion by default. Thus, the class above implements the reader-writer problem with only one writer active at the same time.

2.2.2 Intra-Object Concurrency

Allowing different readers to be active at the same time requires a synchronization mechanism for intra-object concurrency. The class definition below introduces our model [Saleh and Gautron 92]: a derived class, `NReaders_1Writer`, specifies the synchronization constraints allowing `N` readers to be active at the same time:

```
class NReaders_1Writer : public ReaderWriter {
    boolean false () { return FALSE; }
  delay:                                    // access specifier
    false() delay read (int), read (int);   // synchronization constraint
};
```

Readers and writers own the same priority and any reader waiting on the head of the queue may be activated as soon as no writer is active.

The `delay` declaration within the `delay` specifier above defines a synchronization condition attached to the member function `read`. In this example, it states that a reader may be activated when another reader is already active only if the condition `false` is not verified. In this trivial case, this is always the case, and any reader may be activated even if another reader is running. Operations are always mutually exclusive by default: two writers, or one writer and one reader, cannot be active at the same time.

2.2.3 Message Acceptance Priority

We chose to implement message acceptance priority with a simple mechanism. An identical priority is assigned by default to each member function. This priority can be modified with a delay specification similar to the following:

Reader Priority

```
class NReaders_1Writer_ReaderPrio : public NReaders_1Writer {
  delay:
    read (int) > write (Item, int);
};
```

Writer Priority

```
class NReaders_1Writer_WriterPrio : public NReaders_1Writer {
  delay:
    read (int) < write (Item, int);
};
```

By default, all the operations applied on a same object are put on a same waiting queue and accepted for execution once their delay condition satisfied. The usual C++ comparison operators are used to arrange the priority levels. One queue per priority level is created and the object's scheduler processes a straighforward algorithm: high priority served first, FIFO mode within a same priority.

The priorities obey a sequential order. This model is simple but efficient. The number of queues is known at compile-time, simplifying implementation. Any ambiguity in the priority order entails a compile-time error.

2.3 A Basic System Layer Support for Distribution Management

Writing distributed applications amounts to define servers implementing services. Our system layer provides a framework to define servers in an object-oriented programming style. As first application to this system layer, our class library implements basic distributed services. This section briefly describes different facilities supplied by the system layer through the class library:

- object allocation, local as well as remote
- server construction. A base class `Server` is provided. A user-defined server simply inherits from this class and defines its public interface. Inheritance allows safe code reusability: for example, the base class will automatically creates the appropriate stream connections.

- remote argument passing. A distributed application has to deal with function argument encoding/decoding to send/receive requests to/from the network. A simple external representation is provided to support this task.

- remote pointers. Remote allocation means remote pointers in order to execute operations on a remote object. Universal references are supplied to support transparent synchronous and asynchronous remote/local procedure calls. This last service is interfaced by different concurrent classes.

- several other services such as early replies (also called futures) are provided for data flow and "wait-by-necessity" programming.

2.3.1 Object Allocation

Two primitives support object allocation and de-allocation:

```
void* remoteAllocation (char* sitename, size_t size);
void freeSpace (char* sitename, void* addr);
```

`remoteAllocation` allocates `size` bytes onthe remote site referred to by `sitename`, the logical configuration name of the site (`/Cluster/00` for example). The following steps are processed:

- locate the memory server on the remote site. If not present, dynamically load it and then open a connection between the local and remote sites.

- send a request for memory allocation and return the remote reference.

- close the connection and keep in a local cache the waiting port of the memory server so that future request be served faster.

Conversely, `freeSpace` frees the memory pointed to by `addr` on site `sitename`.

2.3.2 Server Construction

A user-defined server derives from the class `Server`. The derived class provides as constructor argument an array of pointers that describe its service. For example:

```
// a code server for class X
class X_Server : public Server {
  public:
      // constructor arguments are passed to the base class constructor
    X_Server (char* serverName, MemFctPtr* serverFct[])
    : Server (serverName, serverFct) {}
};
```

`serverFct` is an array of functions (not described here) defined to encapsulate the calls to member functions of class `X`. This mechanical task can be automatically achieved with the support of a high level tool such as a stub generator (see section 2.5).

2.3.3 Remote Procedure Calls

Client-server communications are usually implemented with the support of a remote procedure call (RPC) mechanism. Any RPC protocol requires the client and the server to agree on argument passing format. In our library, packing and unpacking of arguments are achieved through the medium of the class `Message`. The following example outlines how the arguments of a simple member function `void X::m(Arg a1, Arg a2)` are encoded:

```
void X::m(Arg a1, Arg a2) {
    Connection connection ("X_CodeServer");
    Message mess (2, 2*sizeof(Arg));

    mess.set (opcode(m));          // built-in opcode
    mess.insert (a1, sizeof(Arg);
    mess.insert (a2, sizeof(Arg);
    connection.SendRequest (mess);
    ...
}
```

Again, this mechanical task can be automatically achieved with the support of a high level tool.

2.3.4 Remote Pointers

Remote pointers, as well as usual pointers, may be created from any arbitrary type. Some kind of genericity must be thus provided for allowing the creation of arbitrary remote pointers. Our library supplies a parameterized class[3] `Pointer`, with the original type of the remote pointer as class argument:

[3]Template in C++ terminology.

```
template <class T> class Pointer<T> {
    // private stuff
  public:
    Pointer (char* sitename = 0);  // creates a remote pointer
    T* operator->();               // operator overloading
};
```

The class constructor allocates a remote pointer (local by default) to an instance of the class argument. This latter class must provide a constructor without argument. For example:

```
class X {
    ...
  public:
    X();
    void initialize();
    void m();
};

Pointer <X> rptr;    // class instantiation and
                     // remote instance creation: call to X::X()
rptr->initialize();  // remote initialization
rptr->m();           // remote call to m
```

Remote pointers allow remote procedure calls to appear as local calls. In the parameterized class declaration above, the message send operator "`->`" is redefined. A call through a `Pointer<T>` object is the same whereas the object is local or remote. The "`->`" operation returns a reference to the object if the caller and the callee are located on the same site, a reference to a stub connected to the code server of the remote object otherwise. This approach makes remote calls to appear local to the client code without language extension or compiler modification.

2.4 Object Migration and Persistence

The migration issue has been addressed, but is not yet fully implemented. The persistence issue has not been addressed. Our previous decision was to provide migration through a language extension. We found it unnecessary and, instead, we provide a library class: each potentially migrable object inherits from the generic class `Migrable`. For example:

```
template <class T> class Migrable<T> {
    // private stuff
```

Figure 1: System layer object model.

```
  public:
    Pointer<T> Migrate (char* sitename);
};

// Object is an arbitrary type
class MigrableObject : public Migrable<Object> {
   ...
};
```

The particularity of this model is to consider migration not as a language feature but as an object property. Indeed, making an object migrable only requires a user class to inherit from the class `Migrable`.

2.5 High Level System Interface

Although our system layer is interfaced with a class library that significantly reduces the complexity inherent to distributed applications, the programmer still has to write mechanical and error-prone tasks. To face this issue, we provide a high level tool, a stub generator, with the user code as input and the different pieces necessary to build a distributed application as output. This includes code servers, stub objects, encoding/decoding of arguments, translation of local address to universal adresses when needed. This tool is supplied as an independent C++ application.

2.6 Object Model

The figure 1 summarizes our object model as supplied by the system layer and used by the stub generator. It shows three nodes referred to as `/Cluster/0[0-1-2]`. Three pointers are created from node `00`, one local (`ptr0`) and two remote (`ptr[1-2]`). A procedure call through `ptr0` is just a local indirection to the real object. `ptr1` and `ptr2` are two stubs with each two fields, the object address on a given site and the local address of the code server for the class referred to by these pointers. Procedure calls through `ptr1` and `ptr2` are forwarded to the remote site by the local and remote parts of the code server.

2.7 Conclusion

Our class library supplies a clean, modular and efficient interface to the system layer. The class library illustrates the power of taxonomy claimed by the object-oriented technology and the expressiveness of the generic class model. For example, the transparent localization eases porting applications written on monoprocessor machine onto a distributed memory machine.

The system layer is independent of a specific architecture. Although Helios was our privileged target, porting our implementation should only require the underlying operating system to provide network communication facilities, a server management support and a task system on each node.

3 Prototyping Programming Environments

3.1 Objectives

The second part of the project focuses on programming environments for concurrent (and distributed) programs. It is obvious that because of the multiple, complex, and non deterministic nature of concurrent programming, sophisticated programming environments to describe, visualize, monitor, and debug such programs are not a feature but a necessity. The nature and computation models of concurrent programming are not fully mature yet. Consequently it makes more sense for us to *prototype* programming environments for concurrent systems which should be modular and flexible enough for further evolution. We advocate designing new programming environment prototypes starting from existing sophisticated programming environments that we can find in standard (sequential) programming languages. We keep the same object-orientation for the whole project. Meanwhile, as opposed to the system-layer sub-project (see Section 2) which chose C++ as the development language, we chose Smalltalk-80 for its rich, flexible and reusable programming environment.

Our starting point is a testbed for object-oriented concurrent programming (OOCP) languages based on Smalltalk-80. This system (namely Actalk) gives a good integration of active objects (we call them *actors*) into the Smalltalk-80 environment. We will first introduce the Actalk system. Then we will focus on the way we extend standard Smalltalk-80 programming environment to provide a prototype programming environment for object-

oriented concurrent systems. This will include: extension of the Smalltalk-80 user-interface framework (MVC), and design of a generic scheduler. Other environment tools will also be discussed. This work focuses on the Smalltalk-80 system, but we firmly believe that our experience and design of prototype tools is general enough to be useful for other systems.

3.2 Overview of Actalk

Goals and Architecture Actalk is a testbed for modeling, classifying and experimenting with OOCP languages. Actalk is based on Smalltalk-80 (referred to as ST80 in the following). It introduces active objects communicating through asynchronous message passing in ST80. We call them *actors* and Actalk stands for *act*ors in Small*talk*.

The Actalk system started on a pedagogical basis, as a way to model in a minimal way various kinds of actor-based languages (for instance the Actor model of computation [Agha 86], and ABCL/1 [ABCL 90]). Such descriptions are implemented as subclasses of the two classes composing the Actalk minimal kernel, namely:

the class `Actor`, expressing the structure of actors and the semantics of asynchronous message passing,

and the class `ActorBehavior` expressing the semantics of behaviors (of actors) handling incoming messages.

Actalk has been further described in [Briot 89].

Programming Environment Because Actalk actors are well integrated into the ST80 system, they automatically benefit from the standard ST80 programming environment. We further extended this standard environment to support the specificity of actors. Tools such as a user-interface generator for actors, based on the ST80 MVC paradigm (in section 3.3), and a generic scheduler (in section 3.4) have been developed.

Applications Actalk has been used within the Oks [Voyer 89] system in order to compile production rules into concurrent demons. Actalk has been also used in order to describe various Distributed AI systems, by extending

the concept of actor into the concept of *agent*. Platforms for coarse grain cognitive agents [Bouron et al. 90], as well as for fine grain reactive agents [Drogoul et al. 91], have been successfully developed on top of Actalk.

An Example: Generating Prime Numbers In order to give a taste of the development and visualization of programs in the Actalk environment, we shortly describe an example and show a display of its execution. This example models the generation of prime numbers through an ordered chain of filters (sieves), one filter for each prime number already found. Every integer, until some limit, will be checked through this chain of filters. Each filter checks if the incoming integer is a multiple of the prime number it models. If so, the integer is rejected, otherwise it will be checked by (sent to) next prime filter. If the filter is the last of the chain, this means that a new prime number has been discovered. Consequently a new filter is created for this new prime number and linked at the end of the chain.

Definition of class `PrimeFilter` is given below. `n` is the prime number modeled by the filter, and `next` is the next filter in the chain. We omit definition of method `n:` for initializing instance variable `n`.

```
ActorBehavior subclass: #PrimeFilter
   instanceVariableNames: 'n next '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Examples'!

!PrimeFilter methodsFor: 'script'!

filter: i
   i \\ n = 0 ifFalse: [next isNil
     ifTrue:[next:= (self class new n: i) actor]
     ifFalse:[next filter: i]]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- "!

!PrimeFilter class methodsFor: 'example'!

upTo: max
   "PrimeFilter upTo: 100"
   | two |
   two := (self new n: 2) actor.
   2 to: max do:[:i | two filter: i]! !
```

A snapshot of the corresponding computation, introducing some of the tools of the Actalk programming environment, is shown in Figure 2.

3.3 Actalk MVC

The Model View Controller (MVC) is the standard paradigm for developing interactive applications in ST80. Three abstract classes are provided to take over the operations related to a particular aspect of the user interface:

the *model* which holds the application's state and behavior,

the *view* which is concerned with displaying the application's state,

and the *controller* which coordinates user actions (or inputs) with the model and the view [Krasner and Pope 88].

3.3.1 Lacks of the Standard MVC Model

The MVC paradigm provides, at the abstract level, a convenient methodology for modular interface construction. However MVC is not directly usable to visualize concurrent objects because it makes several assumptions about the nature of objects (they are supposed to be passive objects, i.e., no activity on their own). The use of the MVC interface may become frustrating and error generating for the end-user because of the interferences that non-passive objects (actors) generate on the screen. In the snapshot shown in Figure 3, the user is currently interacting with the system browser, while some Actalk simulation of Conway's Life's game is concurrently displaying successive generations.

A careful study of the MVC abstract classes leads us to the following observations:

- Models are supposed to be passive. In the standard MVC framework, the state modification of a model is controlled by the user (via the controller part of the interface).

  As opposed to passive objects, active objects (actors) may change their state independently of the user interaction. Actors may ask their views for a redisplay in a window that could be inactive, or while the user is moving or resizing it. This can lead to interferences and confusing representations of objects on the screen.

Figure 2: Example: generating prime numbers.

Figure 3: Illustration of interferences: display of Life's game.

- The original MVC uses bitmap images (caches) in order to restore back contents of windows when they are moved, resized or closed. The problem is that a window cached image is updated in only one case: when the user disactivates it. A window image (accordingly to the ST80 MVC paradigm) cannot change while the window is inactive because models are passive objects.

  This assumption does not stand for actors because they can change their state at arbitrary time. Activating a window in such a case may have the effect of displaying the cached image which stores an obsolete representation of the model state. Other subtle interference problems may occur, and are described in [Bouabsa 90].

3.3.2 A New MVC Dedicated to Actors

Goals and Design As explained above, we have to extend the MVC in order to provide a satisfying interactive interface for actors. Our extension should ensure that:

- the user is able to take control over the representation (view) of an actor

at any time: while the actor is computing, and even if it attempts to update its view,

- the actor is not suspended while the view is accessed by the user in order to keep a high level of concurrency.

We chose the following strategy: giving higher priority to control (interaction) over update. This means that the view will not be updated while the user interacts with the controller. We also give higher priority to computation over update. This means that the actor will keep computing concurrently during interaction, and that consequently updates will be temporarily disabled. At the end of the interaction, the view will be updated only once (the updates during inhibition are not bufferized).

The MVC extension should also allow:

- the visualization of actors in partially overlapped windows,
- the co-existence of standard tools (such as browsers, inspectors, debuggers, ...) with the new Actalk tools: the end-user *must* keep benefits of using the standard tools provided by Smalltalk-80.

Managing Visualization in Overlapping Windows The solution we adopted to manage overlapping windows is to slightly extend the MVC. In this solution, views associated to an actor must display information on the cache of the window. Each time the contents of the cache is updated, the window is asked to display it in its visible parts, without affecting the contents of windows that may overlap it (its obscured parts). This type of windows is subdivided into visible parts (parts that are not overlapped by any window) and obscured parts (invisible parts).

Informations about visibility of windows is automatically updated by the window manager of ST80 each time the structure of the screen is changed (i.e., each time a window is moved, reframed, collapsed, opened or closed). During this updating, displaying of actors is disabled until new visibility information has been computed.

In order to keep a high level of concurrency, windows that may want to update themselves on the screen while visibility information is not available are not suspended: they inform the window manager that their contents have

Figure 4: Interface generated for prime numbers algorithm.

changed, and let their models (actors) continue execution. The window manager automatically redraws those windows as soon as it finishes computing visibility information.

3.3.3 Automatic MVC Generator

In order to ease the programmer with the construction of MVC-based interfaces for actors, we offer an automated interactive generation of the interface (namely the view and controller classes). Figure 4 shows an example of interface automatically generated from each of the prime number filter composing the algorithm described in section 3.2.

Design We consider two complementary aspects as basic to visualize and control actors:

state and messages : As in standard MVC, instance variables are displayed, and messages to be sent to the actor by the user may be selected through a menu.

activity : This part is specific to actors. We display the contents of the mailbox of the actor, as well as the current message being processed. A menu provides the user some monitoring (suspend, sleep, step, resume...) of the actor activity.

The Actalk-MVC generator offers many others features and specially the following:

- *interactive description:* The user can choose, at generate time, what instance variables will be visualized. We intend to provide also non-textual (graphical) representations of numerical variables (for example, a gauge).

- *modularity:* The generator takes the hierarchy into account. When defining a hierarchy of models, their respective views and controllers automatically match this hierarchy.

- *genericity:* The generator is not limited to a specific actor model. For instance, this generator supports Agha's Actor computation model [Agha 86]. If the behavior is replaced by a structurally different one (e.g., whose instance variables differ), the view (and the controller) change accordingly.

- *debugging tools:* The user can dynamically control the way an actor computes messages (step by step, sleep...). Those tools are well suited to spy and monitor the activity of actors.

3.4 Scheduling Actors

Implementing concurrent activities on a monoprocessor involves a mechanism to simulate parallelism by scheduling activities. Smalltalk-80 provides a multi-process mechanism controlled by a scheduler. But this scheduler is too limitative for our goals. We describe why and how we extended the standard ST80 scheduler for a finer and more generic control over concurrent activities.

3.4.1 The Standard Smalltalk-80 Scheduler

The standard ST80 scheduler is non-preemptive. A process keeps computing as long as it does not schedule explicitly. In our case, this forces the user to include explicit scheduling (the expression `Processor yield`) inside the methods describing behaviors of actors. This is of course too much low-level constraint.

3.4.2 A Preemptive Scheduler for Actalk

Architecture The Actalk scheduler is described by a subclass of standard ST80 scheduler `ProcessorScheduler`, named `PreemptiveProcessorScheduler`. The scheduling policy of this scheduler is described by a method named `run` belonging to this class. The Actalk scheduler is implemented by a specific sleeping process which periodically awakes in order to perform scheduling, by calling method `run`. This scheduler process has a high priority in order to take immediate control when being awaken.

The Actalk scheduler is generic. It may be customized by defining subclasses and refining the `run` method. `run` methods are described as compositions of *primitive scheduling actions*. Currently two primitive scheduling actions are provided:

- *time-slicing* between ST80 processes, as provided by the ConcurrentSmalltalk scheduler [Okamura and Tokoro 90]. This is implemented by method `yield`.
- *scheduling* between actors. This is implemented by method `schedule`.

The distinction between these two concepts (two methods) provides a more precise and generic control over the scheduling of actors. For instance our generic scheduler may be customized for Agha's Actor computation model [Agha 86]. Such actors may have several activities concurrently. In that case, the CPU time allocated to such an actor is shared among its activities, being controlled by the scheduler.

The decomposition of the scheduling strategy in primitive scheduling actions is analog to the decomposition of the execution strategy of simulated objects in the simulation system described in [Goldberg and Robson 83, pages 443–445]. In this system, the user describes activities of a simulated object by defining the method `tasks` as a sequence of primitive activities such as `holdFor: n`, `produceResource`...

Time-slicing Time-slicing is achieved by performing `yield` action, defined in class
`PreemptiveProcessorScheduler`. It consists in changing the order of the activable processes in the ST80 scheduler table.

The ST80 scheduler contains a eight entries table. Each entry corresponds to a different level of process priority and contains an ordered list of processes (waiting for the processor). The ST80 policy for choosing the next running process is to search in the ST80 scheduler table (in a decremental order accordingly to the priority) for the first non empty list. Then the first process belonging to this list is chosen.

To ensure time-slicing, the following strategy is used. When the Actalk scheduler process awakes, the standard ST80 scheduler puts back the process which was currently running (and just got replaced by the awaking scheduler process) at first position in its corresponding (priority) list. The Actalk scheduler then moves it from the first to the last position in the list. Consequently when the Actalk scheduler process returns to sleep, next (different) process in the list will be scheduled.

Scheduling We want (1) to ensure concurrency between ST80 applications and concurrent applications (written in Actalk) and (2) to control the way Actalk actors are scheduled. Consequently we defined Actalk processes as specific and owning their own scheduling policy, being distinct from standard ST80 processes.

Actalk processes belong to class `PreemptibleProcess`, a subclass of `Process`. The Actalk scheduler holds its own table of Actalk processes. It ensures that at most one Actalk process at a time is included in the ST80 scheduler.

The policy for choosing the next Actalk process to be scheduled among the ST80 processes is generic, and defined in the `schedule` method belonging to class `PreemptiveProcessorScheduler`. We can customize our scheduler, for instance to take into account sub-scheduling of behavior activities specific to Agha's Actor computation model.

Combining Time-Slicing and Scheduling The default scheduling policy defines a 20% ratio of scheduling/time-slicing (i.e., one actor scheduling every four ST80 processes time-slicing). The `run` method is defined accordingly:

```
!PreemptiveProcessorScheduler methodsFor: 'scheduling'!

run
   (nbTimeSlice >= 5)
      ifTrue:[nbTimeSlice := 0.
```

Figure 5: Chronogram: overview visualization of scheduling.

```
        self schedule]
ifFalse:[nbTimeSlice:=nbTimeSlice+1.
         self yield]! !
```

Visualization In order to visualize scheduling between actors, we designed a graphic tool. It is based on *Pie menus* described in [Lalonde and Pugh 89]. Figure 2 gives a sample of its use. Each actor is represented by a slice of the pie. The slice is highlighted when the actor is scheduled and removed when the actor dies. This tool allows dynamic visualization of scheduling events.

A complementary tool, called chronogram, records scheduling events (creation, destruction, activation ... of actors), and is used to display a graphical overview (see Figure 5).

The user can easily find informations about:

- *time life:* creation and end of actors,
- *time slicing:* scheduling of actors,
- *statistics:* waiting duration between two activations of an actor, ...

As opposed to the Pie menu which provides visualization of the scheduler instantaneous state, the chronogram provides a global overview of the scheduler behavior during some time interval.

3.5 Porting Actalk on the T-NODE

We started porting the Actalk system on the T-NODE multiprocessor. Therefore we first ported the Actalk kernel on another dialect of Smalltalk, named GnuSmalltalk, whose sources are in C and in public domain. We are porting GnuSmalltalk onto the Transputers under Helios. This porting turned out to be much more difficult than we expected because of some GnuSmalltalk implementation strategies. This porting has not been completed yet. We extended the GnuSmalltalk virtual machine to support built-in actors and asynchronous message passing, as well as remote objects. A multi-process simulation of a network of connected GnuSmalltalk interpreters has also been implemented [Lemarié 91].

4 Conclusion

In this paper, we described a concurrent and parallel object-oriented programming environment. This includes two components: a C++based concurrent extension and distribution system layer, and a Smalltalk-80-based prototype programming environment for object-oriented concurrent programs. During the progress of the project, some related activities were achieved:

- a graphical tool for the network configuration of parallel applications [Briot et al. 91].
- the multi-process simulation of CDL [Perihelion 89], a language for component distribution on parallel machines [Perret and Gautron 91].

All the tools mentioned in the paper have been distributed to other research centers and are available.

The overall results of the project have to be slightly relativized. We achieved numerous software components on programming language, distribution system layer, and programming environment perspectives. However

we could not fully achieve the most ambitious part of our project, that is to join these components into an integrated chain of development. The main reason of this failure was the real difficulty encountered in porting existing sequential tools on a parallel machine. Nevertheless we hope that the numerous software components that we designed and implemented, as well as our experience discussed in several research papers, will be useful for next developments of software for parallel computers.

References

[ABCL 90] "ABCL: An Object-Oriented Concurrent System". *Computer Systems Series*. Edited by A. Yonezawa. MIT Press, 1990.

[Agha 86] G. Agha, "Actors: a Model of Concurrent Computation in Distributed Systems". *Series in Artificial Intelligence*, MIT Press, 1986.

[America 87] Pierre America, "Inheritance and Subtyping in a Parallel Object-Oriented Language". In *ECOOP'87*, Paris (France), June 1987.

[America 90] Pierre America, "A Parallel Object-Oriented Language with Inheritance and Subtyping". In *OOPSLA'90*, Ottawa (Canada), October 1990.

[Bouabsa 90] M. Bouabsa, "Une Extension du MVC pour Actalk". Dea Report, University Paris VI, France, September 1990.

[Bouron et al. 90] T. Bouron, J. Ferber and F. Samuel, "A Multiagent Testbed for Heterogeneous Agents". In *2nd European Workshop on Modelizing Autonomous Agents and Multi-Agent Worlds (MAAMAW'90)*, Paris (France), July 1990.

[Briot 89] J.-P. Briot, "Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment". In *ECOOP'89*, Nottingham (GB), July 1989.

[Briot 90] J.-P. Briot, "OOCP = OOP + C". In *TOOLS PACIFIC'90*, Sydney (Australia), November 1990.

[Briot and Gautron 89] J.P. Briot and P. Gautron, "Development of an Environment for Specification and Execution of Active Objects on Parallel Machines". *European Community Parallel Computing Action: Application N0 4232*, Research Proposal. April 1989.

[Briot et al. 91] J.P. Briot, P. Gautron, S. Lemarié, L. Lescaudron and H. Saleh, "Development of an Environment for Specification and Execution of Active Objects on Parallel Machines". *European Community Parallel Computing Action: Application N0 4232*, Project Report, March 1991.

[Decouchant et al. 89] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, X. Rousset de Pina, "A Synchronization Mechanism for Typed Objects in a Distributed System". In *ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, Volume 24, number 4, San Diego (USA), April 1989.

[Drogoul et al. 91] A. Drogoul, J. Ferber and E. Jacopin, "Viewing Cognitive Modeling as Eco-Problem Solving: the Pengi Experience". Laforia Research Report No 2/91, Institut Blaise Pascal, U. of Paris VI, Paris, France, January 1991.

[Goldberg and Robson 83] A. Goldberg and D. Robson, "Smalltalk-80: the Language and its Implementation," *Series in Computer Science*, Addison Wesley, 1983.

[Hopkins et al. 89] T. Hopkins, I. Williams and M. Wolczko, "Mushroom - A Distributed Multi-User Object-Oriented Programming Environment," Technical Report, Dept. of Computer Science, U. of Manchester, U.K., 1989.

[Kafura and Lee] Dennis G.Kafura and Keung Hae Lee. "Inheritance in Actor Based Concurrent Object Oriented Languages". In *ECOOP'89*, Nottingham (GB), July 1989.

[Kahn and Saraswat 90] K.M. Kahn and V.A. Saraswat, "Complete Visualizations of Concurrent Programs and their Executions". *IEEE Workshop on Visual Languages*, IEEE Computer Society Press, October 1990.

[Krasner and Pope 88] G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80". Journal of Object-Oriented Programming (JOOP), Vol. 1, No 3, August-September 1988.

[Lalonde and Pugh 89] W.A. Lalonde and J.R. Pugh, "Pie Menus". Journal of Object-Oriented Programming (JOOP), Vol. 2, No 1, May-June 1989.

[Lalonde et al. 86] W.R. Lalonde, D.A. Thomas and J.R. Pugh, "Actors in a Smalltalk Multiprocessor: a Case for Limited Parallelism". *Technical Re-*

port SCS-TR-91, School of Computer Science, Carleton University, Ottawa (Canada), May 1986.

[Lemarié 91] Sylvie Lemarié, "Simulation d'une architecture multiprocesseur à mémoire répartie. Application à un système acteur réparti". In *AFUU'91*, Paris, March 91.

[Manning 87] C.R. Manning, "Traveler: the Apiary Observatory". In *ECOOP'87* Paris (France), June 1987.

[OOCP 87] "Object-Oriented Concurrent Programming". *Computer Systems Series*. Edited by A. Yonezawa and M. Tokoro, MIT Press, 1987.

[Okamura and Tokoro 90] "ConcurrentSmalltalk-90". In *TOOLS PACIFIC'90*, Sydney (Australia), November 1990.

[Perihelion 89] Perihelion Software. "The Helios Operating System". *Prentice Hall*, 1989.

[Perret and Gautron 91] Jérôme Perret and Philippe Gautron, "SIMCDL: Simulating CDL Parallel Programming on a Sequential Environment". Working paper.

[Saleh and Gautron 91] Hayssam Saleh and Philippe Gautron, "A System Library for C++ Distributed Applications on Transputer". In *Applications of Transputers 91*, IOS Press, Glasgow (Scotland), August 1991.

[Saleh and Gautron 91] Hayssam Saleh and Philippe Gautron, "A Concurrent C++ Extension for Writing Distributed Applications". In *ECOOP'91 Workshop on Concurrent Object-Oriented Programming*. Genève (Suisse), July 1991.

[Saleh and Gautron 92] Hayssam Saleh and Philippe Gautron, "A Concurrency Control Mechanism for C++ Objects". *To appear in Lecture Notes in Computer Sciences*. Edited by Tokoro and al, Springer-Verlag, 1992.

[Tomlinson and Singh 89] Chris Tomlinson and Vineet Singh, "Inheritance and Synchronization with enabled sets". In *OOPSLA'89*, New Orleans (US), October 1989.

[Voyer 89] R. Voyer, "A Tool for Generating Knowledge Representation in Smalltalk-80". In *TOOLS PACIFIC'90*, Sydney (Australia), November 1990.

[Yokote and Tokoro 87] T. Yokote and M. Tokoro, "Experience and Evolution of ConcurrentSmalltalk". In *OOPSLA'87*. Orlando (US), October 1987.