Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment

Jean-Pierre BRIOT

Equipe Mixte LITP - Rank Xerox France,
Université Pierre et Marie Curie,
4 place Jussieu, 75005 Paris, France
E-mail: `briot@litp.ibp.fr`

in Proceedings of European Conference on Object-Oriented Programming (ECOOP'89), British Computer Society Workshop Series, Cambridge University Press, pages 109–129, July 1989.

Abstract

This paper describes a system for designing and classifying actor languages. This system, named *Actalk*, which stands for *act*ors in Small*talk*-80, is based on some minimal kernel introducing actors into Smalltalk-80. The Actalk kernel extends passive and sequential objects activated by synchronous message passing into active and concurrent actors communicating by asynchronous message passing. This defines a sub-world of actors embedded into the Smalltalk-80 programming language and environment, which is left unchanged. The Actalk kernel is composed of only two Smalltalk-80 classes. Simulating current actor languages or designing new ones is achieved by defining subclasses of these two kernel classes. Consequently all such extensions are implicitly classified by the inheritance mechanism and unified into the Actalk environment. We are currently extending the standard Smalltalk-80 programming environment to design a specific one dedicated to Actalk concurrent actors.

In this paper, the motivations and the goals which led to design the Actalk system are first discussed. Then the structure and implementation of the kernel of Actalk is detailed. The very good integration of this kernel into the Smalltalk-80 programming language and environment is demonstrated through some examples. Interests and limits of combining objects with actors are then discussed. In the last part of the paper, we demonstrate the expressive power of the Actalk kernel by extending it to simulate the two main actor computation models and programming languages, namely, the Actor model of computation, and the Abcl/1 programming language. Related and further work is summarized before concluding this paper.

Keywords

object, Smalltalk-80, concurrency, ConcurrentSmalltalk, actor, kernel, integration, modularity, combination, Act*, Abcl/1, classification, environment

1 Introduction

This paper is concerned with the design and experiment of Object-Based Concurrent Programming (OBCP) languages. We believe that the so-called *actor* family of languages is among the most promising approaches. The theory of actors was invented by Carl Hewitt [Hewitt 77]. Plasma was the first language designed along this philosophy. It was followed by Act1 [Lieberman 81], Act2 [Theriault 83], and the latest and most achieved prototype: Pract/Acore [Manning 87], based on the latest computation model called the *Actor computation model* defined by Gul Agha [Agha 86]. Besides this stream of prototypes designed around Carl Hewitt at the AI Lab of MIT, the actor metaphor gave rise to other OBCP proposals based on the original actors, although changing some aspects of the model. All these *actor*, or *actor-based* languages keep the foundation of active and concurrent objects communicating asynchronously. One of the most representative element is the Abcl/1 language proposed by Akinori Yonezawa [Yonezawa et al. 86].

We remarked that, from the user point of view, it is not always easy to analyze and compare all these proposals. Comparing various semantics needs to abstract from various syntax and implementation supports. Moreover some of the prototypes are not usable outside of some labs or very specific machines. From the implementor point of view, experience (including our personal one) of designing actor languages shows that usually too many prototype implementations are thrown away before clarifying the design. Making a prototype modular (and improving its efficiency) is usually achieved *afterwards* and not in the early implementation effort. The reuse of previous prototypes or even other ones would greatly improve the design task. This goal of modularity is endorsed by Object-Oriented Programming (OOP). This led us recently to design some modular system/environment for actor languages based on OOP, in order to satisfy these needs. We will now discuss how we designed it.

2 Goals and Design Decisions

To design a system for integrating various actor languages in a single environment, we had the following goals in mind:

1. *uniformity and modularity*
   We wanted to unify various actor languages into some common environment and to be able to analyze and define them step by step. Therefore we chose the object-oriented paradigm as a basis for matching these two first goals. We decided to introduce actors into traditional OOP, by defining a sub-world of actors embedded into the world of objects, and without changing the underlying object system.

2. *minimality and extensibility*
   We wanted some minimal kernel expressing the most general semantics of actor languages, and to further extend it in order to simulate various existing languages and to design new ones. Therefore we chose a minimal architecture in the spirit of ObjVlisp [Briot and Cointe 87, Cointe 87] which is based on a minimal kernel, and then further uniformly extended. We use inheritance to classify the various actor models.

3. *an integrated environment*
   We didn't want to restrict our system to some semantic model and raw implementation,

but also to provide a full environment for pragmatic experiment with actor-oriented programming. Therefore we chose Smalltalk-80 because it is the most achieved and flexible OOP system with a fully integrated programming environment. Because of the integration of our actors into the Smalltalk-80 model and environment, the designer of actor languages could automatically use the standard Smalltalk-80 programming environment. Furthermore, this environment could be extended in order to support the specific concurrency aspects of actors.

By choosing Smalltalk-80 we could also provide a minimal implementation of the system, because all entities needed to build actors: objects, classes and messages, and to express concurrency: processes and semaphores, are provided by Smalltalk-80. The resulting system is named *Actalk* (which stands for *act*ors in Small*talk*-80), as a spiritual offspring of *ObjVlisp* (*obj*ects in *v*irtual *Lisp*). Actalk is an integrated environment, embedded into Smalltalk-80, used as a testbed to classify and design actor languages. Actalk does not change the underlying Smalltalk-80 system, but rather merges into it. Because of the optimal integration of actors within objects, Actalk may also be used as a basis for studying relation and combination between objects and actors. We are currently using Actalk in graduate courses to introduce and teach actor-oriented programming to object-oriented programmers.

3 From Smalltalk-80 Objects to Actors

We will now shortly introduce the model of actors, and how we embed them into Smalltalk-80. We will focus on *why* and *how* to introduce them into a standard OOP model, namely Smalltalk-80, which is extended towards concurrency.

In Smalltalk-80, as in standard OOP, objects are activated by message passing. Objects are passive because they undergo the request of activation by the sender of the message. They have no activity of their own. Sending a message represents the transfer of activity from one object to another one, before going back to the sender whose activation is suspended.

To achieve concurrency, Smalltalk-80 introduces multiple activations, called processes. Then multiple messages may activate concurrently several objects. But if a single object receives concurrently two messages, the two activations may compete to control the object. If both methods assign concurrently a same variable of the object, the resulting state cannot be predicted and is inconsistent. The problem is that an object is still passive, therefore it should be protected against multiple simultaneous activations.

The simplest and most pragmatic solution to this problem is to ensure the single activation condition, also called *mutual exclusion*. Such a technique is described in Smalltalk-80 by [Pascoe 86]. A further solution ensures this condition by providing its own activity to the object. This changes the model of computation from passive into active objects. An object will now possess its own internal activity, and will no more depend on external activations through message passing. The object becomes *active* and *autonomous*. It gained the ability to decide on its own when to perform a message and has the power to complete it.

Because the receiver has its own activity to perform the message, the sender does not need any more to suspend its activity to transfer it to the receiver. If no reply is needed, the sender may resume its computation immediately after sending the message. Message passing becomes unidirectional and asynchronous. Thus, sending a message reduces to its delivery to the receiver. The receiver may process it any time after the delivery, therefore messages should be buffered in a *mailbox* associated to the receiver, before being processed.

The initial model of passive objects synchronously activated by messages has been extended towards a model of concurrent active objects communicating asynchronously by passive messages. We will call such entities *actors*. An actor will encapsulate a standard Smalltalk-80 object to make it active and to change the semantics of message passing. An actor is composed of a mailbox where the incoming messages will be buffered, and the active object that will process them, which we will call its *behavior*.

4 Implementation of the Actalk Kernel

4.1 Definition of the Minimal Kernel

We will now define the most minimal and general kernel for our Actalk system. We will call it the *Actalk kernel*. Two classes define its semantics. The class `Actor` defines the semantics of actors. The class `ActorBehavior` defines the semantics of behaviors (of actors).

The class `Actor` simply defines an actor through its two components, the mailbox or queue which will contain the incoming messages, and the behavior which will process them. The implementation of asynchronous message passing led us to define a third class, presented in section 4.4, in order to clearly separate the implementation technique required from the semantics of the kernel.

Designing the class `ActorBehavior` needs careful attention. We want to define the most general semantics of how a behavior processes messages. The Actor computation model of Gul Agha states that a behavior processes only one message and has to specify which behavior will process next message. This generalizes the usual assignment of instance variables in OOP languages. The Actor computation model is minimal and general enough to express any other kind of computation model, as discussed in [Agha 86]. However we do not choose it as the candidate for the kernel, because we want an extension of the Smalltalk-80 language with minimal implementation and maximal integration within Smalltalk-80. We would like the programmer to specify behaviors of actors as he would specify behaviors of standard Smalltalk-80 objects. Therefore we must keep usual variable assignment to specify state changes. Because there could be state changes during the processing of a message, messages have to be *serialized* [Hewitt and Atkinson 79], i.e., processed one at a time.

We believe that the basic semantics of behaviors we choose is minimal *and* general enough. It expresses the default semantics of the activity of behaviors, and it can be extended or redefined by the extensions of the kernel. For instance the Actor computation model of Gul Agha is expressed as an immediate extension of the kernel (one method is redefined and another one is added), as shown in section 7.1.

Figure 1 gives a representation of the implementation of an actor. All Smalltalk-80 source for Actalk will be printed in the standard *fileOut* format, where the underline character _ means assignment.

4.2 The class Actor

The class `Actor` implements actors as encapsulators built around standard Smalltalk-80 objects. Its superclass is the class `MinimalObject` which will be used for the implementation of asynchronous message passing and is described in section 4.4. It specifies two instance variables:

Figure 1: Implementation of an actor in Actalk.

`mailbox` denotes the queue of messages, an instance of the standard class `SharedQueue`,

`behavior` denotes the behavior which processes the messages, an instance of a subclass of the class `ActorBehavior` described in next section.

```
MinimalObject subclass: #Actor
   instanceVariableNames: 'mailBox behavior '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Kernel'!

!Actor methodsFor: 'initialization'!

initialize
   mailBox _ SharedQueue new!

initializeBehavior: aBehavior
   behavior _ aBehavior.
   behavior initializeAself: self! !

!Actor methodsFor: 'access to instance variables'!

mailBox
   ^mailBox! !

!Actor methodsFor: 'message passing'!

asynchronousSend: aMessage
   mailBox nextPut: aMessage! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

!Actor class methodsFor: 'instance creation and initialization'!

behavior: aBehavior
   ^self new initializeBehavior: aBehavior!

new
   ^super new initialize! !
```

The `behavior:` class method creates an actor and initializes its behavior (`initializeBehavior:` instance method). The `new` class method is redefined to initialize the mailbox of the actor (`initialize` instance method). We designed this decomposition in two distinct initialization protocols in order to improve modularity and reuse of the kernel when extending this basic implementation (but this won't be shown in this paper). The `asynchronousSend:` instance method implements asynchronous message passing to an actor by enqueueing the corresponding message onto its mailbox.

4.3 The Class ActorBehavior

The class `ActorBehavior` implements the general behavior of an actor. Users will describe behaviors of actors by defining classes of behaviors as subclasses of `ActorBehavior`. The class `ActorBehavior` defines one instance variable: `aself`, which denotes the actor currently using this behavior. `aself` allows an actor to send (asynchronous) recursive messages to itself. The difference with standard pseudo variable `self` will be explained in section 4.4.

A background process will be created with the actor to implement the activity and autonomy of the behavior. This process is infinite. As stated in the definition of the kernel, the behavior will keep dequeueing the next message from the mailbox and perform it.

```
Object subclass: #ActorBehavior
   instanceVariableNames: 'aself '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Kernel'!

!ActorBehavior methodsFor: 'initialization'!

initializeAself: anActor
   aself _ anActor.
   self setProcess!

setProcess
   [[true] whileTrue: [self acceptNextMessage]] fork! !

!ActorBehavior methodsFor: 'message acceptance'!

acceptNextMessage
   self acceptMessage: aself mailBox next!

acceptMessage: aMessage
   self performMessage: aMessage! !

!ActorBehavior methodsFor: 'actor creation'!

actor
   ^Actor behavior: self! !
```

The `initializeAself:` instance method initializes the self-reference (`aself`) of the actor and starts the activity of the behavior (`setProcess` method). The `acceptNextMessage` and `acceptMessage:` methods accept and perform the next message in the mailbox. The process is suspended until the mailbox is not empty by the semaphore which synchronizes reading data from the shared queue [Goldberg and Robson 83, page 262]. The `actor` method creates an actor from a passive object which is used as its behavior. It is the only method that the end user has to know about the kernel, i.e., the public interface, as shown in section 5.1.

Note that the Actalk kernel is stated in just 2 classes and 11 small methods. The number of methods could have been further reduced, but as already stated in previous section, we chose this modular decomposition to easily describe further extensions. For the same kind of reason, the implementation primitive `performMessage:` method is defined in class `Object`:

```
!Object methodsFor: 'message passing'!

performMessage: aMessage
   ^self perform: aMessage selector withArguments: aMessage arguments! !
```

4.4 Transparent Asynchronous Message Passing

4.4.1 Local Redefinition of Message Passing Semantics

Now we would like to integrate harmoniously our new message passing model (`asynchronousSend:` method) between actors with the current message passing syntax of Smalltalk-80. We use a technique initiated in Smalltalk-80 in order to change the syntax (to introduce compound selectors for multiple inheritance [Ingalls and Borning 82]) or the semantics (to encapsulate objects [Pascoe 86]) of message passing through redefinition of error semantics. A message is sent to an object which, on purpose, does not recognize the selector. This error is trapped by redefinition of the standard error method (`doesNotUnderstand:`) which then executes some specific strategy. Local redefinition of the `doesNotUnderstand:` method ensures the locality of changes. Thus we redefine it in the class `Actor` in order to redefine the semantics of message passing *locally* to actors:

```
!Actor methodsFor: 'message passing'!

doesNotUnderstand: aMessage
   self asynchronousSend: aMessage! !
```

A message sent to an actor will get an asynchronous semantics whereas the same message sent to some standard Smalltalk-80 object will keep the standard synchronous semantics. Note that the value returned by asynchronous transmission is not significant. (Actually the value of the expression is the receiver of the message, due to the Smalltalk-80 convention.)

When specifying the behavior of an actor which sends recursive messages, the programmer may choose between *pseudo*-variables `aself` or `self`. Sending to `aself` uses Actalk asynchronous message passing. The message is put in the mailbox of the actor, and the behavior will process it later. Sending to `self` relies on standard Smalltalk-80 synchronous message passing. This implies immediate and "internal" processing of the message by the behavior.

4.4.2 Implementation

By using this error redefinition technique we assume that an actor won't recognize the selector of the message it receives because we *do* want to trigger an error, in order to put the message in the mailbox. (Note that, in contrast, the "Actalk implementation methods" defined in class `Actor` will be directly processed.) But methods defined in class `Object` are inherited by every class, consequently such messages sent to an actor won't fail as expected.

[Pascoe 86] and [McCullough 87] discuss various implementation strategies to ensure the assumption of unrecognized messages. Our simplified solution is to define another root of the inheritance tree, besides the class `Object`, in order to bypass it. Actually the method dictionary of this new class should not be fully empty, because a minimal set of system methods is needed to trap errors, print, compare, and inspect their instances. Otherwise the Smalltalk-80 interpreter and environment would not be able to manage properly such objects. Unfortunately the standard Smalltalk-80 environment disallows the user to define a class without superclass. Therefore the implementation trick is to define the new class, named `MinimalObject`, at first as a subclass of Object. Then its initial inheritance link is automatically removed (superclass _ nil), and a minimal set of system methods belonging to Object is copied (recompile:from:) onto it.

```
Object subclass: #MinimalObject
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Kernel-Encapsulator'!

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

!MinimalObject class methodsFor: 'initialization'!

initialize
   superclass _ nil.
   #(doesNotUnderstand: error:   ~~ isNil   = ==   printString printOn: class
     inspect basicInspect basicAt: basicSize instVarAt: instVarAt:put:)
         do: [:selector | self recompile: selector from: Object]! !

MinimalObject initialize!
```

5 A First Example: the Counter

Our first example will be one of the most simple and paradigmatic examples of object-oriented programming: the counter. This will show how well Actalk actors are integrated into the Smalltalk-80 language. The class Counter will describe the behavior of counter actors, i.e., which behave as counters.

5.1 Definition of the Counter

The class Counter defines two instance methods to reset (reset), and increment (incr) the instance variable contents of a counter. In the actor terminology, these two methods constitute the *script* of the actor.

```
ActorBehavior subclass: #Counter
   instanceVariableNames: 'contents '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Examples'!

!Counter methodsFor: 'script'!

incr
   contents _ contents + 1!

reset
   contents _ 0! !
```

We defined this class of behaviors of actors-counters, exactly as we usually define the class of objects-counters. Counter must be defined as a subclass of class ActorBehavior. We may create an instance of Counter as some usual Smalltalk-80 counter object postfixed with the selector actor, and send (implicitly asynchronous) messages to this newly created actor:

```
| aCounter |
aCounter _ Counter new actor.
aCounter reset; incr; incr
```

Notice that the selector `actor` is the only special keyword to create actors. Definition and message passing are transparent within the Smalltalk-80 language into which actors are embedded. Some difference in programming style will be when returning values, as we will see in next section.

5.2 Concept of Reply Destination: the Printer Example

Now suppose that we want to consult the contents of the counter and display it for instance. But, due to the asynchronous nature of message passing to actors, we cannot rely any more on the returned value of a message as in standard Smalltalk.

The intuitive idea is to simulate bidirectional transmission by a second unidirectional message as reply, i.e., to specify within the message the actor to which the reply will be returned. Such an actor is called a *customer* [Agha 86], or *reply destination* [Yonezawa et al. 86]. Reply destinations are also used to implement *continuations* which is one of the main concept of programming with actors [Hewitt 77], but won't be addressed in this paper. We will specify a reply destination when consulting the contents of a counter with the following method:

```
!Counter methodsFor: 'script'!

consultAndReplyTo: replyDestination
   replyDestination reply: contents! !
```

We assume that every actor (or even object, see section 7.2.2) used as a reply destination handles the selector `reply:`, convention for replying the value. For instance, some standard Actalk actor which is bound to the global variable `Print`, displays values in the Smalltalk-80 Transcript window. Its behavior is defined by the class `Printer`:

```
ActorBehavior subclass: #Printer
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Examples'!

!Printer methodsFor: 'script'!

reply: value
   Transcript show: '> ' , value printString; cr! !
```

Following is an example of use:

```
Counter new actor
   reset; incr; incr; consultAndReplyTo: Print
```

which displays:

```
> 2
```

6 Symbiosis Between Objects and Actors

Historically, classes and objects were proposed by Simula and Smalltalk to describe abstract and concrete concepts. Actors were proposed by Plasma to describe control structures and concurrency. Actalk appears as a proposal to combine both, i.e., extend classes and objects towards concurrency, or/and give a structure and an environment to describe actors. Because the Actalk sub-world of actors is fully integrated into the Smalltalk-80 language, actors may send messages to objects and vice versa. Thus the two programming styles may be combined.

6.1 Safety of Combination

One of the motivation for introducing actors in the object world of Smalltalk-80 was to automatically solve the inconsistencies between objects and processes (discussed in section 3). However some unrestricted combination between objects and actors may see these problems reoccur. If several actors happen to share a single passive object, the situation will be equivalent to processes sharing an object.

One drastic and ultimate way to solve the problems is to remove passive objects and to make every Smalltalk-80 object become an actor. ConcurrentSmalltalk-II [Yokote and Tokoro 87] walked a step in this direction and reduced some of the problems by changing parts of the Smalltalk-80 system. We did not choose this way because we did not want to change Smalltalk-80 in any way.

Another way that we chose is to provide safety rules. Some of the rules are ensured by the Actalk implementation and user interface (for instance, the only way to create an actor). But some methodological rules are also necessary as compromises between too strict rules forbidding any reuse of standard objects and too weak rules leading to havoc. For instance, Smalltalk-80 classes may be safely approximated to constant objects. Consequently they may stand concurrent activations. (In the actor terminology, a Smalltalk-80 constant object is similar to an *unserialized* actor.)

6.2 Extending the Smalltalk-80 Environment towards Actors

Because Actalk actors are well integrated into the Smalltalk-80 system, they automatically benefit from the standard Smalltalk-80 programming environment, which is a great help when designing languages and applications. A further goal is to extend this standard environment to support the specificity of actors. A first step is to extend the Smalltalk-80 MVC model for interface design towards actors. The prototype basic extension of MVC that we designed allows to control representations (views) of an actor during its activation. Another challenge is to extend the current Smalltalk-80 debugger towards a specific debugger for actors. A first prototype has already been implemented. It relies on extended messages which contain the context of the sender to reconstruct the appropriate chain of contexts.

7 Extending the Kernel to Simulate Various Actor Languages

Now we will sketch some extensions to our actor kernel in order to simulate some of the most representative OBCP computation models and programming languages based on the actor

Figure 2: Architecture of the Actalk system (kernel and few extensions).

concept. Such simulations are not concerned about a complete reimplementation of some programming language environment, but to express its most essential and specific characteristics. These extensions will use inheritance to refine the semantics of the Actalk kernel. The first example will express Agha's Actor computation model as a subclass of `ActorBehavior`, whereas the second example will express Yonezawa's Abcl/1 model as a subclass of `Actor`. This shows the merits of modularity for our kernel. Because the kernel and its extensions are related by inheritance, one could easily compare them. Inheritance helps not only to classify various actor models, but also to clearly relate and to reuse their various implementations.

Figure 2 shows the hierarchy of the classes of the Actalk kernel, augmented by some of its current extensions, and the example of the counter. Note that the class `ActorBehavior` and all its extensions are *abstract classes*, i.e., don't have instances. These classes only give the semantics of how to compute messages, not the semantics of messages themselves. Only the application classes, like `Counter`, will generate actual instances, the behaviors. Actors will be instances of class `Actor` and its subclasses. The classes `ExtendedActor` and `ExtendedActorBehavior` introduce a generic control of actor events (i.e., receiving a message, computing it…) into Actalk. They are currently used to modularly change the semantics of the underlying Smalltalk-80 scheduler of processes, and design actor event driven representations (views) of actors (through a combination with our Actalk/MVC interface). Their complete description is found in [Briot 88].

7.1 The Actor Model of Computation

7.1.1 The Concept of Behavior Replacement

The Actor model of computation, as exposed in [Agha 86], replaces state change (assignment) with a much higher level concept: *behavior change*. When performing a message, the current behavior of an actor will specify its *replacement behavior*, i.e., how it will perform next incoming message. A behavior will now accept only one message. The replacement behavior in turn, on accepting the next message, will specify its own replacement behavior. This leads to a causally

Figure 3: The concept of behavior replacement.

connected ordered chain of behaviors isomorphic to the queue of messages. The two important points to highlight are the *absence of assignment* and the *separation between the successive behaviors*. As a consequence they may execute concurrently.

7.1.2 Implementation

To change accordingly the semantics of behaviors, we introduce a new class, named `AghaActorBehavior`, as a subclass of `ActorBehavior`:

```
ActorBehavior subclass: #AghaActorBehavior
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Extensions-Agha'!

!AghaActorBehavior methodsFor: 'initialization'!

setProcess
   [self acceptNextMessage] fork! !

!AghaActorBehavior methodsFor: 'behavior replacement'!

replace: replacementBehavior
   aself initializeBehavior: replacementBehavior! !
```

Only two instance methods define our extension. The `setProcess` method is redefined, and now accepts only one message in the message queue. We introduce a new method, named `replace:`, whose semantic is to specify the replacement behavior (and to initialize it).

This redefinition slightly changes the role of the instance variable named `behavior` and defined in class `Actor`. It now represents the *current behavior* of an actor. When performing a behavior replacement this variable will be reassigned (by the `initializeBehavior:` method of `Actor`) to the *replacement behavior*. The current behavior won't be touched, thus it will complete its current computation. It will then be garbage-collected by the system because it is no longer referenced by the actor and the process is also terminated. Figure 3 shows this new model of actors.

7.1.3 Example

We easily redefine the class of counters (defined in section 5.1) as a subclass of `AghaActorBehavior` and name it `AghaCounter`. Previous assignment will be replaced by the specification of a replacement behavior. (We suppose the existence of the `contents:`

class method to create and initialize a new counter.) Remark that the `consultAndReplyTo:` method, although not changing state, needs to specify a replacement behavior (equal to the current one) in order to process next incoming message.

```
AghaActorBehavior subclass: #AghaCounter
   instanceVariableNames: 'contents '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Extensions-Agha-Examples'!

!AghaCounter methodsFor: 'script'!

consultAndReplyTo: replyDestination
   self replace: self.
   replyDestination reply: contents!

incr
   self replace: (AghaCounter contents: contents + 1)!

reset
   self replace: (AghaCounter contents: 0)! !
```

Further extensions of this initial implementation of the Actor model have been easily obtained with Actalk, e.g., various strategies for optimization, as for instance proposed in the Pract/Acore system [Manning 87], and introduction of the concept of *future*. This concept will also be dealt with, by other means, in the next section.

7.2 The Abcl/1 Model of Computation

7.2.1 Principles

The Abcl/1 language (which stands for *A*ctor-*b*ased *c*oncurrent *l*anguage) [Yonezawa et al. 86], although based on the actor philosophy, chose a more pragmatic approach. The language is not intended to be self-contained, but supports hybrid computation. The actor-oriented model of computation may combine with more traditional programming languages which could be used for expressing parts of the behaviors of actors. The main characteristic of Abcl/1 is to propose three distinct types of communication protocols between actors, called *types of message passing*, at the user level:

past is the asynchronous type of message passing, equivalent to the one we designed in the Actalk kernel. The action of sending the message is already completed (in the *past*) as soon as specified, and the sender may immediately resume its computation.

now is a synchronous type of message passing. The sender wants the reply *now* and will wait for it.

future is an eager type of message passing. The place where the reply (or possibly several successive replies) will be eventually delivered in the *future* is specified at the time of sending. Consequently the sender may start manipulating the (*future*) reply before getting its actual value.

Moreover, these three types of message passing are consistent. The same message is sent but the reply destination depends on the type. The reply destination is implicit for the *now* and *future* types. (This will be illustrated in section 7.2.5.) Consequently the receiver handles uniformly the three types of messages, and only the semantics of replying will change according to the various reply destinations.

Actually, there are some more (four) major properties in order to fully define Abcl/1. Due to space limitation, they won't be discussed here, but they have also been simulated by extending further the following extension.

7.2.2 Principles of Implementation

To simulate Abcl/1 into Actalk, we will follow the pragmatic philosophy of Abcl/1. We will define the *now* and *future* types of message passing as returning immediately some standard Smalltalk-80 object on which the sender will synchronize to get the value(s) once computed. We will call this object a *future* object (because it acts in place of the future actual value). It should behave as a queue buffering the successive replies. To implement it, we will use the standard class `SharedQueue`, already used to implement mailboxes. We just need to rename the assignment message and define a message to consult the first value by suspending until the queue is not empty. The class `MAFuture` (which stands for *m*ultiple *a*ssignment *future*) implements such *future* objects:

```
SharedQueue subclass: #MAFuture
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Extensions-Abcl'!

!MAFuture methodsFor: 'assignment'!

reply: aValue
   self nextPut: aValue! !

!MAFuture methodsFor: 'consultation'!

value
   | firstValue |
   readSynch wait; signal.
   accessProtect critical: [firstValue _ contentsArray at: readPosition].
   ^firstValue! !
```

The `reply:` method ensures our reply selector convention (defined in section 5.2) by renaming the `nextPut:` assignment method. The `value` method consults the first element of the queue (without removing it). The semaphore controlling the non emptiness of the queue is checked then reset (in order to be able to read the value once again). (This definition is minimal but relies on the implementation of standard class `SharedQueue`.)

Two more useful methods, `next` and `isEmpty`, are inherited from class `SharedQueue`. The `next` method returns (and removes) the first element in the queue. But the `value` and `next` methods need to wait if the queue is empty. Therefore the `isEmpty` method provides checking the emptiness of the queue. This is useful for an actor not to be "glued" onto a future object not yet ready, but on the contrary to do some other computation for a while. These two inherited methods respectively simulate the `next-value` and `ready?` standard Abcl/1 constructs to access future objects.

7.2.3 The Future Type of Message Passing

We may now easily define the Abcl/1 *future* type of message passing. It creates a new future object, sends the original message in the *past* type with the future object as the reply destination, and immediately returns this future object as the value of the transmission (through the underlying standard synchronous message passing level). When the receiver will finish computing the reply to this message, it will reply the returned value to the future object. Then the queue will get assigned with a first element, and will be available for consultation.

The *future* type of message passing will reduce to the *past* type with a future object as reply destination. In order to always and easily know which argument of the message specifies the reply destination, we assume that a reply destination is *always* specified as the *last argument* of a message. Consequently the newly created future object will always replace the initial last argument of the message before sending the message in the *past* type. The initial argument is not significant, but will be used for discriminating the 3 types of message passing as shown in section 7.2.5. We now define the class `AbclActor` as a subclass of class `Actor` to add this new *future* type of message passing:

```
Actor subclass: #AbclActor
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actalk-Extensions-Abcl'!

!AbclActor methodsFor: 'message passing'!

futureCall: aMessage
   | aFuture |
   aFuture _ MAFuture new.
   aMessage arguments at: aMessage arguments size put: aFuture.
   self asynchronousSend: aMessage.
   ^aFuture! !
```

7.2.4 From Eager to Synchronous Communication

The *now* (synchronous) type reduces immediately to the *future* type of message passing plus the explicit consultation of (waiting for) the first value of the future object:

```
!AbclActor methodsFor: 'message passing'!

nowCall: aMessage
   ^(self futureCall: aMessage) value! !
```

This reduction of the *now* type to the *future* type, itself reduced to the *past* type, is similar to the reduction semantics proposed in [Yonezawa et al. 86, pages 263–264]. Like in Abcl/1, the implicit reply destination is a future object, first class object, which may be passed along or delegated to other actors. Note however that in Actalk the future object is a standard Smalltalk-80 object and not an actor like in Abcl/1. This does not limit our simulation in practice however, because of the symbiosis between objects and actors in Actalk.

7.2.5 Combination of Message Types into the Abcl/1 Model

The *now* and *future* types of message passing will now be combined with the *past* type already integrated (in section 4.4) into standard Smalltalk-80 syntax. We extend the error redefinition technique by using a symbol to specify the two new types of message passing. The last (`replyTo:`) parameter of the message is used for this purpose. Moreover the value of this parameter is not significant when sending a message in the *now* or *future* types, because their reply destination is implicit. This parameter will be replaced during the reduction process, as seen in section 7.2.3, by the real reply destination, a newly created future object.

Consequently we will redefine further the `doesNotUnderstand:` method in order to discriminate the type of message passing. We first check if there is at least one argument to the message, and in such a case then check the last one:

```
!AbclActor methodsFor: 'message passing'!

doesNotUnderstand: aMessage
   ^aMessage arguments isEmpty
      ifTrue: [self asynchronousSend: aMessage]
      ifFalse: [aMessage arguments last == #future
                  ifTrue: [self futureCall: aMessage]
                  ifFalse: [aMessage arguments last == #now
                              ifTrue: [self nowCall: aMessage]
                              ifFalse: [self asynchronousSend: aMessage]]]! !
```

The three types of messages passing are summarized in this example of consulting a counter:

`aCounter consultAndReplyTo: Print`
: will be sent in the *past* type and include an explicit reply destination (`Print`). The contents of the counter will be displayed on the Transcript, as already explained in section 5.2.

`aCounter consultAndReplyTo: #now`
: will be sent in the *now* type. The value of the expression is the contents of the counter.

`aCounter consultAndReplyTo: #future`
: will be sent in the *future* type. The value of the expression is a future object (instance of class `MAFuture`), referencing the contents of the counter.

One needs to send the message `value` to this future object in order to get the real value. Note that we also developed other implementations of future objects where the consultation of the real value is implicit.

8 Related and Further Work

Our work can be compared with similar activities which have not been already referred in the paper. [Bézivin 88] shares a similar goal on studying concurrency within Smalltalk-80, however his study is much more general, whereas ours is only devoted to study the actor paradigm for computation by providing a testbed dedicated to it. The ConcurrentSmalltalk [Yokote and Tokoro 87] and Actra [Lalonde et al. 86] projects went further than Actalk in terms of combination of objects and actors. But to achieve it, they had to slightly change both the Smalltalk-80 virtual machine and the semantics of original actors, whereas our objective was to preserve the underlying environment and to simulate various actor systems into it. We are not concerned either in designing an efficient extension of Smalltalk-80 to concurrency, but we provide a platform for specification and experiment with actor languages which takes benefit and reuse of the standard Smalltalk-80 programming environment.

We now expect to explore many fields with this unified tool. We will attempt simulating more actor languages as extensions of the Actalk kernel. For instance, Actalk is currently being used as a designing tool for the Mering-IV project [Ferber and Briot 88]. We also plan the design of a higher level language, analog to Acore [Manning 87], with a compiler generating Actalk kernel code. A group of students is currently working on a general framework based on Smalltalk-80 MVC to visualize and control Actalk actors. Other prospective experiments with the Actalk platform include: modeling communication protocols, modeling strategies for allocation of actors and tasks, and compiling production rules into concurrent daemons implemented by Actalk actors [Voyer 89].

9 Conclusion

In this paper we discussed the design of a system, named Actalk, based on Smalltalk-80 and providing an environment to compare and design various actor languages and implementation strategies. The kernel of Actalk introduces actors into the current Smalltalk-80 system. Its implementation was completely described. A methodology for combining traditional Smalltalk-80 programming and actor-oriented programming was discussed. The extension of the current Smalltalk-80 programming environment towards actors was also sketched. The minimal Actalk kernel has been successfully extended in several directions to simulate various actor languages. We described the complete implementation in Actalk of the Actor model of computation and the communication protocols of the Abcl/1 programming language.

Because of space limitation, many topics were just sketched in this paper. They are extensively discussed in the current preliminary report on Actalk [Briot 88] which includes all code for Actalk kernel, extensions and examples.

We would like to express our thanks to Jean-Francois Perrot and the reviewers for suggesting improvements of the paper.

References

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

[Bézivin 88] J. Bézivin, Langages à Objets et Programmation Concurrente: quelques Expérimentations avec Smalltalk-80, *Actes des Journées Afcet-Groplan Langages et Algorithmes*, Bigre+Globule, No 59, pages 176-187, Irisa, Rennes, France, April 1988.

[Briot 88] J.-P. Briot, From Objects to Actors: Study of a Limited Symbiosis in Smalltalk-80, *Research Report LITP 88-58 RXF*, Université Pierre et Marie Curie, Paris, France, September 1988.

[Briot and Cointe 87] J.-P. Briot and P. Cointe, Definition of a Uniform, Reflexive and Extensible Object-Oriented Language, *European Conference on Artificial Intelligence (ECAI'86)*, Advances in Artificial Intelligence-II, North-Holland, Amsterdam, Netherlands, pages 225-232, 1987.

[Cointe 87] P. Cointe, Metaclasses are First Class: the ObjVlisp Model, in [OOPSLA 87], pages 156-167.

[Ferber and Briot 88] J. Ferber and J.-P. Briot, Design of a Concurrent Language for Distributed Artificial Intelligence, *International Conference on Fifth Generation Computer Systems (FGCS'88)*, Vol. 2, pages 755-762, Icot, Tokyo, Japan, November-December 1988.

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

[Hewitt 77] C.E. Hewitt, Viewing Control Structures as Patterns of Passing Messages, *Journal of Artificial Intelligence*, Vol. 8 No 3, pages 323-364, 1977.

[Hewitt and Atkinson 79] C.E. Hewitt and R. Atkinson, Specification and Proof Techniques for Serializers, *IEEE Transactions on Software Engineering*, Vol. SE-5, No 1, January 1979.

[Ingalls and Borning 82] D.H.H. Ingalls and A.H. Borning, Multiple Inheritance in Smalltalk-80, *National Conference on Artificial Intelligence (AAAI'82)*, AAAI, pages 234-237, August 1982.

[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 Report SCS-TR-91*, School of Computer Science, Carleton University, Ottawa, Canada, May 1986.

[Lieberman 81] H. Lieberman, Concurrent Object-Oriented Programming in Act1 in [OOCP 87], pages 9-36.

[Manning 87] C.R. Manning, Acore: The Design of a Core Actor Language and its Compiler, *Revised Master Thesis*, EE and CS dept., MIT, Cambridge MA, USA, 15 May 1987.

[McCullough 87] P.L. McCullough, Transparent Forwarding: First Steps, in [OOPSLA 87], pages 331-341.

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

[OOPSLA 86] Conference on Object-Oriented Programming Systems, Languages and Applications, Special Issue of SIGPLAN Notices, ACM, Vol. 21, No 11, November 1986.

[OOPSLA 87] Conference on Object-Oriented Programming Systems, Languages and Applications, Special Issue of SIGPLAN Notices, ACM, Vol. 22, No 12, December 1987.

[Pascoe 86] G.A. Pascoe, Encapsulators: A New Software Paradigm in Smalltalk-80, in [OOPSLA 86], pages 341-346.

[Theriault 83] D. Theriault, Issues in the Design and Implementation of Act2, *Technical Report No 728*, AI Lab, MIT, Cambridge MA, USA, June 1983.

[Voyer 89] R. Voyer, Implémentation d'Architectures Efficaces pour la Représentation des Connaissances. Application aux Langages Loopsiris et Oks, *Thèse d'Université*, Université Pierre et Marie Curie, Paris, France, February 1989.

[Yokote and Tokoro 87] Y. Yokote and M. Tokoro, Experience and Evolution of ConcurrentSmalltalk, in [OOPSLA 87], pages 406-415.

[Yonezawa et al. 86] A. Yonezawa, J.-P. Briot and E. Shibayama, Object-Oriented Concurrent Programming in ABCL/1, in [OOPSLA 86], pages 258-268.