From Objects to Actors:
Study of a Limited Symbiosis in Smalltalk-80

Jean-Pierre BRIOT

Equipe Mixte LITP - RankXeroxFrance,
Université Paris VI,
4 place Jussieu, 75005 Paris, France
briot@litp.ibp.fr

Research Report RXF-LITP, No 88–58, Paris, France, September 1988.

Abstract

In this paper we describe an implementation of actors in Smalltalk-80, named Actalk. Actors are active and autonomous objects, as opposed to standard passive Smalltalk-80 objects. This attempt is designed as a minimal extension preserving the Smalltalk-80 language.

An actor is basically a serializer built around some usual Smalltalk-80 object. We will study the cohabitation and synergy between the two models of computation: transfer of a single thread of activity between passive objects with a call/return discipline, and simultaneous activities of autonomous actors communicating through asynchronous communications. We propose a sketch of methodology in order to have a safe combination between these two programming clichés. As an example we propose a slight change into the MVC (Model View Controller) model for interface design in order to integrate it with the actors.

We show how to extend the Actalk kernel into various extensions to define the basic Actor model of computation, higher level programming constructs such as the 3 types of message passing (asynchronous, synchronous, and eager) proposed in the ABCL/1 programming language, and some distributed architecture. All these examples are constructed as simple extensions of our kernel model.

Keywords

object, class, Smalltalk-80, message passing, concurrency, process, serializer, actor, asynchronous, autonomy, passive/active, cooperation, future, forwarder, ConcurrentSmalltalk, Act*, ABCL/1, Distributed Smalltalk.

1 Introduction

The Smalltalk-80 language is the best example of Object-Oriented Programming (OOP) languages. One important characteristic of Smalltalk-80 we will focus on in this paper is its sequentiality. The text of a method body is a sequence of expressions, which are evaluated sequentially. The Smalltalk-80 computation model is based on a single thread of activity which is passed among objects in a call/return fashion when sending messages. The introduction of processes into the language allows multiple activities but objects remain passive by nature. The process level is also unfortunately orthogonal with the object level. As a consequence combination of these two styles is uneasy as well as unsafe. If inside two distinct processes two messages are sent to a same object, there is no guarantee in the composition of the messages. If there is some state change (assignment of variables) in the bodies of the methods activated, this could lead to chaos. Semaphores are provided in Smalltalk-80 as tools for synchronization. But semaphores are very low level tools. The nature of processes and semaphores, used to synchronize processes and their accesses to shared data, remain totally orthogonal to the nature of objects.

There have been attempts to modify Smalltalk-80 in order to unify objects and processes. The main proposals are Actra [Thomas and Lalonde 85] [Lalonde et al. 86] and ConcurrentSmalltalk [Yokote and Tokoro 86] [Yokote and Tokoro 87]. These proposals model multiple activities of objects. A receiver may resume its computation after replying to the caller, allowing the sender and the receiver to resume their respective computations simultaneously. These models remain based on synchronous communication. They will be summarized in section 10 on related work. Meanwhile we may already remark that these models did some change into the Smalltalk-80 model of computation and virtual machine. This quotation from [Yokote and Tokoro 87] explains why ConcurrentSmalltalk evolved from a first version compatible with Smalltalk-80 [Yokote and Tokoro 86] towards a second version in order to alleviate some primitive problems.

> *"Through experience gained from writing programs in ConcurrentSmalltalk, we have become aware of some disadvantages, such as the treatment of: an object shared by more than two processes and the behavior of a block context. These disadvantages originate from the concurrency control that had been designed to maintain compatibility with Smalltalk-80."*

Our approach is somewhat different because we don't intend to do any change to the Smalltalk-80 model and virtual machine but we want to propose a minimal extension needed to introduce and study concurrency (i.e. potential parallelism as opposed to physical parallelism) into Smalltalk-80. This approach of studying concurrency models and constructs as an extension of Smalltalk-80 is also advocated by [Bézivin 88]. But in contrast, our approach is specifically related to our concern with the *actor* paradigm for Object-Oriented (or Object-Based) Concurrent Programming [OOCP 87]. Smalltalk-80 appears to be a wonderful language to explain and discuss the actor languages, with some impressive economy of tools.

This choice of introducing actors into Smalltalk-80 raises the question of their cohabitation with standard Smalltalk-80 objects, and by going further, their complete combination within the current Smalltalk-80 environment and methodology. This issue of combining *passive* objects with *active* objects is one of the fundamental issue of OOCP, as pointed out by [Bézivin 87, page 404]. In order to ensure safety of combination of our actor kernel and Smalltalk-80 we will sketch some methodological rules.

It is time now to summarize our goals:

1. **preservation**, there should be no change made to Smalltalk-80. The system consists in a small extension (a set of classes and methods) to the current image, named Actalk.
2. **minimality**, this extension should be minimal, in order to be easily understood and ported. All what is non strictly necessary or uneasily implemented has been rejected.
3. **extensibility**, this minimal kernel will be easily extended in order to model actors able to replace their behaviors as stated in the Actor model of computation [Agha 86], synchronous and eager types of message passing, as proposed by the ABCL/1 model [Yonezawa et al. 87], distributed computation, and further constructs or models.
4. **synergy**, the current model of objects and the extended set of actors should be compatible and able to cooperate. Some rules are edicted in order to keep a safe mixin.
5. **pedagogy**, we are personaly involved in the actor family of OOCP languages [OOCP 87]. We intend to use this extension in graduate courses to introduce actor-based programming to usual OOP programmers.

2 Review of the Smalltalk-80 Computation Model

Before introducing our model, we will start with a review of the basic principles of the current Smalltalk-80 computation model.

2.1 Sequential and Synchronous Message Passing

2.1.1 Control of Computation

The Smalltalk-80 model of computation is based on procedural activation. There is is only one conceptual thread of activity which is passed along objects when messages are sent. The sender of the message transfers the activity to the receiver which will compute the message. (The receiver could in turn send a message to another object.) The sender is waiting for the reply from the receiver and remains blocked until this moment. When the receiver finishes its computation of the message it sends back a value (or by default itself) as the reply to the sender. Then the activity ends in the receiver and resumes back in the sender.

In summary only one object is active at a time, and when sending a message the activity is moved to another object before going back to the caller which then resumes its computation.

2.1.2 Environment of Computation

This model of activity is very close to procedural or function calls of procedural or functional languages like Lisp. One main difference is that there are no free variables to consider, except the global variables. Only three environments are considered when an object is activated by a message: the *environment of the object* (made of the bindings of its instance variables, and its class or pool variables which are inherited through its class), the *environment of the message* (made of the bindings of the parameters of the message) and the *global environment* (a dictionnary self-referenced through one of its global variable named `Smalltalk` which contains the bindings of all global variables inside the system). The environment of the sender object is irrelevant and hidden to the receiver. Information exchange between objects should be made explicit through the initial message by passing some arguments, or by explicit messages sent to obtain access to the sender. If we compare with Lisp we note that the environment of an object has to be explicitly and extensionally defined. Objects don't rely on some implicit closure mechanism. Only the global environment could be shared implicitly.

2.1.3 Blocks

There is however one exception to this model: *blocks*. Blocks in Smalltalk-80 are equivalent to Lisp lambdas, i.e. anonymous functions. They introduce delayed evaluation and abstraction into Smalltalk-80. Blocks are mainly used for control structures [Deutsch 81] (note that there is no macro mechanism in Smalltalk-80), and processes. A block (instance of the class `BlockContext`, or `BlockClosure` depending on the version) closures the context of its definition. It has already been argued that blocks are not perfectly coherent within the object-oriented philosophy proposed by Smalltalk-80, because objects and blocks have been designed separately [Borning and O'Shea 87] [Ungar and Smith 87] and that they may lead to tricky programming and errors in a concurrent environment. For instance, if an object returns a block as the value of some method invocation, the object is now shared by the block. Thus if the object or the block is shared by a process, this could lead to conflicts as pointed out in [Yokote and Tokoro 87].

2.2 Multiple Activities

2.2.1 From Blocks to Processes

In order to bring multiple activities into the language, processes have been introduced into the Smalltalk-80 system, for instance to run controllers which are waiting for interaction associated to a view on the screen. They could also be created and controlled by the user. The virtual machine schedules the running processes in a pseudo-parallel fashion bringing multiple and simultaneous threads of activities into the language. The problem for the user is the orthogonality between processes and objects. As pointed out in [Yokote and Tokoro 87], this leads to difficulties for the programmer to specify one concept into two different entities: *object* and *process*, of different levels of abstraction. The existence of a third concept: *block* leads to even more complication.

2.2.2 Concurrent Accesses to a Shared Object

Because one single object may be shared by several processes, there is a need to protect its internal state from undeterministic effects if there are state changes. This could lead to bugs, known as timing errors, very hard to detect. The relative schedulings of processes accessing to some single data may produce unpredictable results. Smalltalk-80 provides low level tools for synchronization: semaphores. Semaphores allow to control the execution of processes, specially when synchronizing accesses to some shared data.

2.2.3 Synchronization Techniques

Most usual problem is to ensure mutual exclusion of processes, i.e. only one of them may execute the critical section where there is some mutation of the object, i.e. change in its data structure. But state changes are specified in the bodies of methods at the object (or rather class) level whereas synchronization should be done at the data level. This dissociation of expressing the changes and synchronizing them makes it difficult to program. Most of the encapsulation gained from OOP is lost. It is not always obvious for the programmer to figure out which data of which object needs to be protected (and which needs not), and to introduce the semaphore `signal` and `wait` messages at the right place into the right methods.

An example of how to protect a shared object with the use of semaphore objects is presented in a progressive way in [Goldberg and Robson 83, pages 258–263]. The class `SharedQueue`, which belongs to the standard image, implements a FIFO (First In First Out) data structure whoze size is unbound (except the size of the image!). The queue is protected against timing errors in case of concurrent accesses, and synchronizes the dequeueing requests onto the enqueueing requests if the queue is empty.

3 From Passive to Active Objects

Our goal is to solve this synchronization problem while providing more autonomy to the objects. Current Smalltalk-80 objects are passive, because they undergo the activation request of the sender. An active object should gain its ability to decide *when* and *how* it will compute the messages. This section describes step by step this quest for autonomy.

3.1 Monitored Objects

To ease the programming task, we propose to enforce every object which could be shared by other objects to be safely built as the `SharedQueue` objects are. This protection must be automatic. It is difficult to automatically generate such a protection at data access level. Our solution is pragmatic and will be at the message level in order to monitor the objects, i.e. to ensure the execution of at least one message at a time.

Note that *monitoring* (ensuring mutual exclusion) and *atomicity* (ensuring the completion or abortion of the computation) are distinct notions. Atomicity is a stronger concept than monitoring, it ensures the "one-piece" nature of transformation, and is for instance used for crash recoveries. [Pascoe 86] reviews and models both of them in Smalltalk-80. In this paper we'll be only concerned with monitoring.

The intuitive technic is to use a semaphore for mutual exclusion of messages. Such a technic of encapsulating objects in Smalltalk-80 has been proposed in [Pascoe 86]. Message passing to the object is encapsulated in a critical section protected by a semaphore. An encapsulator is defined around the object. When receiving a message, the encapsulator sets the semaphore, forwards the message to the object, and when it has been completed releases the semaphore. (The value of the inner transmission is assigned to a temporary variable and returned after the release of the semaphore.)

3.2 Serialized Objects

We intend to go a step further in order to make the object become more active. A serializer [Hewitt and Atkinson 79] will encapsulate the object and serialize the incoming messages in a

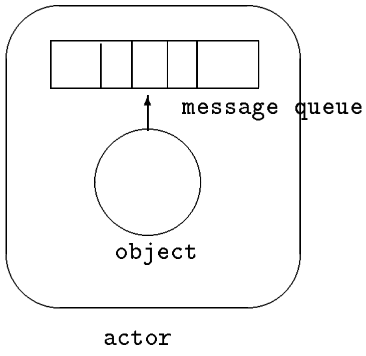

Figure 1: Modeling an actor as a serialized object.

message queue. Then the initial object may compute messages one at a time. The message queue will be an instance of the class `SharedQueue` and it represents the mailbox of the object.

3.3 Message Acceptance Autonomy

This policy may seem a bit too rigid: it forces the serialization of every message although some could perhaps be processed simultaneously safely, for instance when there is no assignment. For instance two *read* messages sent to a dictionary don't need to be serialized. But by setting the synchronization control onto the messages arrival and buffering the incoming messages in the order of their arrival, the object may decide how it will accept and perform the messages. The simplest model is to accept and perform the messages one at a time in the order of their arrival. The object may rather process messages in a pipe-line fashion, depending on their semantics. This other model, known as the Actor model of computation [Agha 86], will be proposed as an extension in section 7.

In the following we will call *actor* a serialized and active object, built as an encapsulation of a usual Smalltalk-80 passive object. The complete implementation will be described in section 4. The *actor* paradigm for programming has been invented by Carl Hewitt [Hewitt 77].

3.4 From Synchronicity to Asynchrony

The original message passing model of Smalltalk-80 is synchronous. We could keep a synchronous model for communication between actors. However one of our goals is also to introduce concurrency into Smalltalk-80. Asynchronous message passing increases concurrency between actors because the sender does not have to wait for the receiver to be ready to complete its transmission. Three steps are now fully distinct (and ordered by causality): *message sending* (from the sender), *message delivering* (from the mail system to the receiver), *message accepting* (from the receiver). In our current implementation the two first steps are merged into one, but they could be unrelated in a distributed implementation (see section 9).

We also believe that asynchronous message passing is a simpler concept, and it is easier to model and implement. Synchronous message passing will be later built from asynchronous in section 8.1. Note that in our implementation, asynchronicity is itself built from synchronism plus buffering.

3.5 Modeling Actors with Objects

The Figure 1 shows a representation of an actor in our Actalk actor kernel. This representation is close to the abstract representation of an actor proposed in [Agha 86, page 24]. Our abstract representation also gives a good insight of the implementation model, shown in Figure 2.

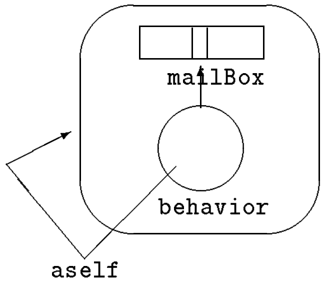

Figure 2: Implementation of an actor.

4 Implementing Actors in Smalltalk-80

In our implementation model we consider two classes: `Actor` and `ActorBehavior`. The class `Actor` represents the actor to which the asynchronous messages will be sent. An actor is a serializer built around some usual object. An actor has two components: the *mailbox*, or message queue, which is an instance of the class `SharedQueue`, and the object which it serializes, that we will call the *behavior* of the actor, following the terminology of the Actor computation model.

The class `ActorBehavior` represents the behavior of the actor. Any Smalltalk-80 object may be used as a behavior of a new actor if its class is a subclass of `ActorBehavior`. A background process will be created when creating the actor to implement the autonomy of the behavior of the actor. This process is infinite. The behavior will keep dequeueing the next message in the mailbox and perform it. Figure 2 gives a representation of the implementation of an actor.

4.1 The class Actor

The class `Actor` implements active objects. Its subclass is the class `EObject` which will be used for syntactic purposes and is described in section 4.4.1. It specifies two instance variables:

`mailbox` denotes the queue of messages,

`behavior` denotes the behavior (Smalltalk passive object) that the actor serializes.

```
EObject subclass: #Actor
   instanceVariableNames: 'mailBox behavior '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors'!


!Actor methodsFor: 'initialization'!

initialize
   mailBox _ SharedQueue new!

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

!Actor methodsFor: 'iv access'!

mailBox
```

```
   ^mailBox! !

!Actor methodsFor: 'message passing'!

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

Actor class
   instanceVariableNames: ''!


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

behavior: behavior
   ^self new initializeBehavior: behavior!

new
   ^super new initialize! !
```

The `behavior:` class method creates an actor which serializes a Smalltalk-80 object, its behavior. The `new` method is redefined to initialize the mailbox of the actor, by creating a new shared queue (`initialize` instance method).

The `initializeBehavior:` instance method initializes the behavior of an actor. The `asynchronousSend:` method sends an asynchronous message by enqueueing the corresponding message onto the mailbox of the actor.

Note that the decomposition into two initialization instance methods and two creation class methods is intended for modularity. This will be useful when redefining slightly some of the methods when defining extensions of this kernel. For the moment, the class method `behavior:` is the only way to create an actor with a behavior. The interface with the user in order to create actors will be discussed in section 4.5.

4.2 The class ActorBehavior

The class `ActorBehavior` implements the behavior of an actor. The user will model behaviors of actors by defining a class of behaviors as a subclass of `ActorBehavior`. (Note that there is a class named `Behavior` in the standard Smalltalk-80 class hierarchy. It models the basic characteristics of classes. Consequently we could not use this class name.) We will see in the next examples that in practice, the user will model the behavior of a class of actors as a class of behaviors, almost like defining classes of objects in standard Smalltalk-80 programming.

The subclass of `ActorBehavior` is the class `Object`. The class `ActorBehavior` defines one instance variable: `aself`, which denotes the actor built from this behavior. `aself` is used to send (asynchronous) recursive messages. The difference with usual pseudo-variable `self` will be discussed in section 4.4.1.

```
Object subclass: #ActorBehavior
   instanceVariableNames: 'aself '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors'!


!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! !
```

The **initializeAself:** method initializes the self-reference (**aself**) of the behavior and sets the background process. The **setProcess** method creates a background and infinite process for the behavior to wait for and accept one at a time the messages. The **acceptNextMessage** and **acceptMessage:** methods accept the next message in the mailbox. The process is suspended and resumed when there is a message by the semaphore synchronizing the availability of message(s) in the shared queue [Goldberg and Robson 83, page 262]. Note that our model is stated in just 2 classes and 10 small methods. The number of methods could have been further reduced but we chose this modular decomposition to easily model further extensions. For the same reason, we define the "abstract method" **performMessage:**

```
!Object methodsFor: 'message handling'!

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

4.3 The Example of the Counter

Our first example will be one of the most simple and paradigmatic example of object-oriented modelization, the counter. This will show how to define actors as usual Smalltalk-80 objects. The class **Counter** will describe the model of behavior of counter actors (which behave as counters).

There are 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: 'Actors-Examples'!


!Counter methodsFor: 'script'!

incr
   contents _ contents + 1!

reset
   contents _ 0! !
```

This definition of the class **Counter** is equivalent to standard Smalltalk-80 programming. The only difference and important point is that **Counter** must be a subclass (or a subclass of a subclass...) of **ActorBehavior**. We may create an instance of **Counter** as some usual Smalltalk-80 counter object:

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

We may use this counter object as the behavior of a new actor. In the case below, we use the `reset` method to initialize a new behavior with a null contents:

```
| aCounterActor |
aCounterActor _ Actor behavior: Counter new reset
```

The methodology for creating actors will be discussed in section 4.5.

4.4 Syntactic Issues: Compatibility with Standard Message Passing

To send an asynchronous message to an actor we need to explicitely call the `asynchronousSend:` method, and moreover create explicitely the message as in:

```
aCounterActor asynchronousSend: (Message selector: #incr)
```

This "non user-friendly" style of programming gets even worst in case of parameters, because we need to explicitely create an array to group them.

4.4.1 Transparent Asynchronous Message Passing

We would like to integrate harmoniously our new message passing model between actors with the current message passing syntax of Smalltalk-80. The technique initiated in [Ingalls and Borning 82] is based on "syntactic error trapping". A message is sent to an object which does not recognize its selector. This voluntary error is trapped by redefinition of the standard error method (`doesNotUnderstand:`) which then executes the chosen strategy.

This technique allows to model syntactic changes through redefinition of error semantics. It has been heavily used to change the message passing semantics for some classes of objects while relying on the usual syntax, thus in a modular and transparent way. Some applications are: encapsulation of objects for monitoring and atomicity [Pascoe 86], modeling various communication protocols [Bézivin 88], meta-simulation (simulation of simulation models) [Bézivin 87], serialization of objects [Yokote and Tokoro 87] and [Bézivin 88], and forwarding of messages to remote objects in a distributed implementation [McCullough 87] and [Bennett 87]. Following this technique, we redefine the `doesNotUnderstand:` method in the class `Actor`:

```
!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 a usual object will keep the standard semantics. Consequently for an actor, a message sent to `aself` will be recursively sent asynchronously to the actor itself (`a` stands for `a`ctor and `a`synchronous). The same message sent to `self` will be recursively sent in a standard synchronous way to the behavior itself.

Note that the value returned by some asynchronous transmission is not significant. Following the usual Smalltalk-80 convention, the transmission implicitly returns the receiver.

4.4.2 Implementation Issues

With this technique we assume the actor not to recognize the selector of the message because we do want to trigger an error. [Pascoe 86] and [McCullough 87] discuss the implementation of this assumption and the status of special selectors. Their scheme is based on the definition of another subclass of `nil`. In contrast [Bennett 87] defines a subclass of the class `Object`, thus all methods belonging to `Object` should be duplicated in this class and forward explicitly the messages.

Our solution is a simplification of [Pascoe 86]. In our project we are only concerned with the syntactic aspects of this technique. A new class, called `EObject` (`E` stands for `E`ncapsulator) is defined at first as a subclass of `Object`. Then it is automatically redefined as a subclass of `nil` and gets copies of the minimum basic methods of `Object` which are needed to trap errors, print, compare, and inspect such objects.

```
Object subclass: #EObject
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Encapsulator'!

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

EObject class
   instanceVariableNames: ''!


!EObject class methodsFor: 'initialization'!

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

!EObject class methodsFor: 'superclass access'!

superclass: aClass
   superclass _ aClass! !

EObject initialize!
```

The `superclass:` class method is redefined because it will be invoked when creating subclasses of `EObject`. The following method is used to copy methods from one class to another class:

```
!Class methodsFor: 'recopying method'!

copyMethodSelector: selector inClass: class
   class compile: (self sourceCodeAt: selector)
         classified: (self whichCategoryIncludesSelector: selector)! !
```

This code is designed for the implementation of Smalltalk-80 on Sun 4. There could be minor changes to port it to other implementations, just by adjusting which methods need to be copied. Thus we unfortunately cannot fully match our portability goal. The only fully portable technic we found is to define `Actor` as a subclass of `Object` and warn the user never to conflict the selectors he defines with selectors in `Object`. The most simple and general way consists in systematically starting every user's defined selector for behaviors with some special symbol, like `A` for `Actor`s for instance. But unfortunately by increasing portability, we decrease transparency! Our current choice seems to be the most appropriate.

4.4.3 External Representation

The following method defined in the class `Actor` gives a C like external representation to actors, by identifying an actor with an indirection (pointer) to its behavior:

```
!Actor methodsFor: 'printing'!

printOn: aStream
   aStream nextPut: $&.
   behavior printOn: aStream! !
```

Recall that by keeping the C conventions, `&a Counter` is equivalent to `a *Counter`. The following definition could also be used depending on personal taste:

```
printOn: aStream
   aStream nextPutAll: 'a *'.
   behavior class printOn: aStream! !
```

4.5 Methodological Issues: How to Create an Actor

4.5.1 Transparency Issues

For the moment, as we saw, there is only one explicit means to create an actor: the `behavior:` class method belonging to the class `Actor`. A more transparent usage is to create an actor from the behavior. Three major solutions are to consider:

1. creating an actor from a behavior object, instance of a subclass of class `ActorBehavior`:

   ```
   !ActorBehavior methodsFor: 'actor creation'!

   actor
      ^Actor behavior: self! !
   ```

2. creating an actor from the model of behavior, by creating a new instance of it. We could either redefine the `new` method:

   ```
   !ActorBehavior class methodsFor: 'actor creation'!

   new
      ^Actor behavior: super new!
   ```

3. or define a new selector, for instance named `newActor`:

   ```

   newActor
      ^Actor behavior: self new! !
   ```

4.5.2 Compatibility with Initialization

We found the first solution to be the most modular and compatible with initialization methods. All solutions are compatible with the following automatic initialization of a counter with a default null contents:

```
!Counter class methodsFor: 'instance creation'!

new
   ^super new reset! !
```

But if we define the usual duality class and instance methods for creating and initializing an instance:

```
!Counter class methodsFor: 'instance creation and initialization'!

contents: contents
   ^self new contents: contents! !

!Counter methodsFor: 'iv access'!

contents: anInteger
   contents _ anInteger! !
```

the `contents:` *class method* is not compatible with the `new` or `newActor` methods. However an interesting result is that the `contents:` *instance method* still combines with them, but with a distinct semantics:

The expression `(Counter contents: 100) actor` reduces to `Actor behavior: ((Counter new) contents: 100)`. The `contents: 100` message will be sent to the behavior (instance of class `Counter`) with semantics of standard message passing *before* the creation of the actor.

The expression `Counter newActor contents: 100` reduces to `(Actor behavior: (Counter new)) contents: 100`. The actor is created *and then* receives the `contents: 100` message, with the semantics of asynchronous message passing.

The semantics of both expressions is equivalent because the asynchronous message passing returns the actor as value.

In summary, equivalence is still kept but almost by accident. The semantics of behaviors and message passing will slightly evolve in sections 7 and 8. Particularly when modeling the Actor model of computation in section 7.2, we have to model explicitly creation and initialization of behaviors. Thus we have to reject solutions 2 and 3.

4.5.3 Safety Issues

If some object would be shared as a behavior by several actors, the problems discussed when an object is shared by several processes will reoccur. Section 6 discusses the issues of safety when combining objects and actors. We need to keep a one to one correspondence between an actor and its behavior.

The `actor` creation method does not guarantee that the object used as a behavior is not already shared by another actor. We will propose to redefine locally this method in a new class named `ExtendedActorBehavior`, defined a a subclass of `ActorBehavior`. This class will be extended further in next sections.

```
ActorBehavior subclass: #ExtendedActorBehavior
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions'!

!ExtendedActorBehavior methodsFor: 'actor creation'!
actor
   ^aself isNil
      ifTrue: [super actor]
      ifFalse: [self error: 'is already the behavior of an actor']! !
```

We just check if the `aself` instance variable of the behavior has already been initialized when creating an actor with this behavior.

4.5.4 Activation of the Counter Actor

We may now create and activate a counter actor, for instance in the following way:

```
(Counter contents: 100) actor incr; incr
```

4.6 Error Handling

The standard error handling system is appropriate to handle errors for behaviors of actors. Behaviors are activated by background processes. Note that because of the asynchronous semantics of communication the information on the sender of the message is lost.

4.6.1 Resetting Actors

In case of errors, it may be sometimes useful to reset an actor because its process may have been stopped. The following method may be used to restart an actor:

```
!Actor methodsFor: 'reset'!

resetProcess
   behavior setProcess! !
```

4.6.2 Safety Issues

For safety reasons, we must be careful when resetting an actor. If we start a new process whereas the current one is probably still active (because infinite process), we may get into trouble. There will be more than one process activating the same behavior on the same mailbox. Consequently the activation of methods may overlap, thus loosing the protection of the serializer. If we provide a method for process resetting, we need to slightly redefine the class `Actor` in order to keep a reference of the running process. We will define these new features into the class `ExtendedActorBehavior` (already sketched in section 4.5.3) that we will redefine with a new instance variable named `process`:

```
ActorBehavior subclass: #ExtendedActorBehavior
   instanceVariableNames: 'process '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions'!


!ExtendedActorBehavior methodsFor: 'initialization'!

setProcess
   process isNil ifFalse: [process terminate].
   process _ [[true] whileTrue: [self acceptNextMessage]] newProcess resume! !
```

We will also define its corresponding class `ExtendedActor`. Note that there could be different degrees of resetting: resetting the process, resetting the mailbox...

```
Actor subclass: #ExtendedActor
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions'!


!ExtendedActor methodsFor: 'reset'!

resetAll
   self initialize.
   self resetProcess!

resetProcess
   behavior setProcess! !
```

Note that we should redefine the `actor` method in class `ExtendedActorBehavior` in order to define an instance of `ExtendedActor`, and not `Actor`. (We also could parameterize the class `ActorBehavior` with some metaclass instance variable referencing the class of actors, with a technique proposed in [Briot and Cointe 87] and [Cointe 87].) This issue will reappear along the paper when defining extensions as subclasses of the class `Actor`. We omit such redefinitions in this paper for simplicity reasons.

4.6.3 Debugging Issues

The standard Smalltalk-80 error handling system was not designed with concurrency in mind. Furthermore, because of asynchronous message passing between actors, the debugger information is no more related to the global computation history, but only to some local part. Designing debuggers for concurrent systems is still an open field of research which is out of the scope of this paper ([Manning 87] gives some good account). Meanwhile we may already point out that Smalltalk-80 gives a higher level conceptual model to manipulate information (objects and messages, rather than bits and routines). This will be obviously very useful to help debugging programs at the user level and not at the machine level as generally done.

4.7 A Generic Control of Scheduling

The scheduling policy for actors is related to the policy of the Smalltalk-80 virtual machine to schedule processes. We don't use the process priority facility when defining actors, because it is too low level control for actors.

Our solution is to introduce three "virtual methods" for scheduling, that the user may redefine in its application behavior classes. They will be called at certain points of the activity of actors and behaviors:

`scheduleAfterAcceptedMessage` denotes the acceptance of a message by a behavior,

`scheduleAfterPerformedMessage` denotes the completion of the execution of a message by a behavior,

`scheduleAfterReceivedMessage` denotes the reception by an actor of a new message.

These three methods are defined as "doing nothing" in the class `ExtendedActorBehavior`:

```
!ExtendedActorBehavior methodsFor: 'generic scheduling control'!

scheduleAfterAcceptedMessage!

scheduleAfterPerformedMessage!

scheduleAfterReceivedMessage! !
```

The two first methods are triggered while a behavior accepts a message:

```
!ExtendedActorBehavior methodsFor: 'message acceptance'!

acceptMessage: aMessage
   self scheduleAfterAcceptedMessage.
   super acceptMessage: aMessage.
   self scheduleAfterPerformedMessage! !
```

and the latter method is triggered while an actor receives some message:

```
!ExtendedActor methodsFor: 'message passing'!

asynchronousSend: aMessage
   super asynchronousSend: aMessage.
   behavior scheduleAfterReceivedMessage! !
```

The usage is mostly to redefine them locally in order to explicitly call the scheduler. The behavior may also inform its dependents. This is specially useful to trigger some redisplay in interface applications, as for instance a MVC interface for the prime number example explained in section 5.3. As an example, the following method expresses that each counter actor will be updated and scheduled after performing a message:

```
!Counter methodsFor: 'scheduling control'!

scheduleAfterPerformedMessage!
   self changed.
   Processor yield! !
```

4.8 Efficiency Issues

One could wonder about efficiency of our implementation because of all infinite processes which are started for each new actor. However such a process is kept waiting by one of the semaphores of the shared queue of messages. If the actor is no more referenced, it is garbage collected (reclaimed) by the system and so is the process.

5 Programming with Actors within Smalltalk-80

There is now a new world of actors, named Actalk, embedded into the usual Smalltalk-80 world of objects. An actor uses a passive object as its behavior. Sending a message to an actor is automatically interpreted as an asynchronous send. Sending a message to some non actor object remains current Smalltalk-80 message passing. Note that the discrimination between standard synchronous and asynchronous message passing is made upon the nature, object or actor, of the target (receiver) of the message. The sender's nature is irrelevant. Remark also that some basic selectors belonging to `Object` (for comparing, printing, inspecting...) are also recognized by actors. This is the minimum knowledge which allows the system and the user to manipulate them as other objects. We will now introduce the basic methodological concepts for programming with actors through some progressive examples.

5.1 Concept of Reply Destination: the Printer Example

We introduced in section 4.3 the first example of a counter. We can change the contents of the behavior of a counter actor, by sending `reset` and `incr` messages. Now we would like to consult its contents 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 current Smalltalk.

We will introduce synchronous messages in section 8.1, but for the moment we may use only asynchronous message passing. The naive idea is to express within the message the actor to which the reply will be returned. This has been introduced in Actor languages and named *customer* [Agha 86], or *reply destination* [Yonezawa et al. 87]. We will stick to the *reply destination* terminology. Note that we reject the *continuation* terminology because the conceptual level is different. Reply destinations will be used to implement *continuations* as described in section 5.2.1.

The following method is defined in the class `Counter` in order to consult the contents, and return it to another actor, its reply destination:

```
!Counter methodsFor: 'script'!

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

The `reply:` selector is our syntactic convention selector for replying the value to the reply destination. `replyTo:` or `AndReplyTo:` will be our conventions for the part of the keyword denoting the reply destination parameter.

In order to test this method we need an actor dedicated to handle such returned values and display them. The global variable `Transcript` refers in Smalltalk-80 to a window (view) designed for this purpose. We will define a simple model of behavior, called `Printer`, which will use this system facility:

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


!Printer methodsFor: 'script'!

reply: value
```

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

Printer class
   instanceVariableNames: ''!


!Printer class methodsFor: 'initialization'!

initialize
   Smalltalk at: #Print put: self new actor! !

Printer initialize!
```

The `reply:` method displays the value onto the `Transcript`. The global variable, `Print` defines a standard printer actor which could be used, as a first task, to display the contents of a counter actor:

```
!Counter class methodsFor: 'example'!

example
   "   Counter example   "
   (Counter contents: 100) actor incr; incr; consultAndReplyTo: Print! !
```

The result displayed will (obviously) be:

```
> 102
```

The system transcript has been serialized into the actor `Print`. Now we may send several messages concurrently to the `Print` actor without trouble. Note that of course, if we still use directly the `Transcript` object (and the system does for its own needs, but only for special tasks, like: input/output, compilation... mostly explicitly triggered by the user), the transcript window is no more protected. This cohabitation issue will be discussed in section 6.

Note that a reply destination is a *first class* actor. It may be passed along to other actors which could answer in place of the receiver [Yonezawa et al. 86, pages 265–266]. This is the basis for a mechanism called *delegation*. This mechanism is introduced and discussed in [Lieberman 86].

5.2 Continuation-Oriented Programming: the Factorial Example

5.2.1 Concept of Continuation

We will now introduce continuation style of programming into Smalltalk-80. At some point of computatio, the *continuation* represents what remains to be computed. In the Actor model of computation, a continuation is an actor which models the (local) state of computation and to which at some moment an asynchronous message will be sent in order to resume the stopped computation. Some value may be passed to the continuation to transmit the last computed value which is needed to restart the computation. We will model continuations as reply destinations, as stated in the Actor model of computation.

5.2.2 Computation of Factorial Function

The most paradigmatic example for continuation-based programming is the computation of the factorial function. The class `Factorial` models this computation:

```
ActorBehavior subclass: #Factorial
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Examples'!
```

```
!Factorial methodsFor: 'script'!

n: n replyTo: c
   n = 0
      ifTrue: [c reply: 1]
      ifFalse: [aself n: n - 1 replyTo: (FactorialContinuation n: n c: c) actor]! !
```

The first parameter of the `n:replyTo:` method is the integer whose factorial will be computed. The second parameter represents a continuation actor to which the result (`n!`) will be replied. In case of recursion, a new request for computing `n-1!` is sent to the same actor (`aself`) with a new continuation. This new continuation is created to model the remaining computation, i.e. multiplying the result (`n-1!`) with `n`, and returning this value (`n!`) to the original continuation. Here is the definition of this continuation:

```
ActorBehavior subclass: #FactorialContinuation
   instanceVariableNames: 'c n '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Examples'!


!FactorialContinuation methodsFor: 'initialization'!

n: anInteger c: aContinuation
   n _ anInteger.
   c _ aContinuation! !

!FactorialContinuation methodsFor: 'script'!

reply: v
   c reply: n * v! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

FactorialContinuation class
   instanceVariableNames: ''!


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

n: n c: c
   ^self new n: n c: c! !
```

The `reply:` method resumes the computation and expresses the equation `n! = n * n-1!`. We may activate a computation by typing:

```
Factorial new actor n: 10 replyTo: Print
```

The next line on the transcript will be:

```
> 3628800
```

Note that the initial continuation, the actor `Print`, should understand the `reply:` selector, because this is the convention we stated to receive a value. A diagram based on events showing the computation could be found in [Agha 86, page 37].

5.2.3 Concurrency Issues

From a syntactic point, this specification may seem verbose if we compare it with the equivalent definition in Scheme. One of the reasons is the explicitness of closures as already noted. This will be discussed in next section. From a semantic point, as pointed out by [Wegner 87, page 176], this way of expressing the computation of factorial seems like using a sledgehammer to crack a nut. However concurrency becomes already more visible within this example:

```
!Factorial class methodsFor: 'example'!

example
   "   Factorial example   "
   Factorial new actor n: 10 replyTo: Print;
                       n:  5 replyTo: Print! !
```

This will produce:

```
> 120
> 3628800
```

as an experimental proof that the two computations were concurrent and interleaved.

If we compare with the traditional definition in Smalltalk-80, we remark that, rather than waiting for the result of the recusive computation, the actor delegates this post-computation to a new continuation. Consequently the actor is immediately free to accept next message. There is no unnecessary waiting. Continuations increase concurrency because they allow an actor to be fluid, i.e. perform a single step of computation and "pass the bucket" to a continuation.

5.2.4 Verbosity Issues

Continuation-oriented programming is unfortunately a bit too low level to be used systematically by the user. The Actor model of computation proposes a higher level where the user could program in a functional style (answers are implicit). For instance the user will define the factorial function in the conventional recursive function (equational) style. A compiler will compile this higher level language into the basic layer, by creating the appropriate models for continuations. The layered architecture of Actor languages is described in [Agha 86] and the implementation of such a double layered architecture, named Acore/Pract, is described in [Manning 87]. A more complete model, including waiting for an answer when necessary, will be discussed in section 7.7.11.

5.2.5 Generality Issues

The uniform use of the `reply:` selector to resume and transmit information to continuations implies that exactly one argument will be systematically passed. If there is no value to transmit but just to resume the continuation, some non significant argument (`nil` or the sender) is returned. In case of multiple values transmitted by a single actor (the case of multiple values sent by more than one actor will be discussed in section 7.6.1), we have two alternatives:

1. transmit the collection of values (for instance an array),
2. use `reply:and:`... `and:` keyword selectors.

First solution preserves the `reply:` convention, but the continuation needs to explicitly extract the values from the data structure. Second solution is more transparent but looses uniformity. Uniformity is important for general purpose "collectors" like the `Print` actor which may be used in place of any continuation. We keep this issue open, while waiting for more experience with Actalk to draw a definitive methodology.

5.3 Exploitation of Concurrency: the Prime Number Example

5.3.1 Model

This example of generating prime numbers in a concurrent fashion is some classical example of concurrent programming. It has already been proposed, for instance in [Serpette 85] and [Lex 85].

A snapshot screen of this example in a Smalltalk-80 context is shown in [Yokote and Tokoro 86, page 340].

The problem is modeled through an ordered chain of filters (sieves), one filter for each prime number already found, to check if integers are prime numbers. 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 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. In addition, we need a generator which will send all integers until some maximum to the first (and initial) filter of the chain.

5.3.2 Implementation

The class `PrimeGenerator` models the generator of integers. `firstPrimeFilter` references the first filter of the chain.

```
ActorBehavior subclass: #PrimeGenerator
   instanceVariableNames: 'firstPrimeFilter '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Examples'!


!PrimeGenerator methodsFor: 'initialization'!

initialize
   firstPrimeFilter _ (PrimeFilter n: 2) actor! !

!PrimeGenerator methodsFor: 'script'!

upTo: max
   2 to: max do:
      [:i | firstPrimeFilter filter: i]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

PrimeGenerator class
   instanceVariableNames: ''!


!PrimeGenerator class methodsFor: 'creation and initialization'!

new
   ^super new initialize! !

!PrimeGenerator class methodsFor: 'example'!

example
   "  PrimeGenerator example  "
   PrimeGenerator new actor upTo: 100! !
```

The `initialize` method creates the first filter. The `upTo:` method generates all integers (until the maximum) and send them to the first filter.

The class `PrimeFilter` models each prime number filter. `n` is the prime number modeled by the filter, and `next` is the next filter in the chain.

```
ActorBehavior subclass: #PrimeFilter
   instanceVariableNames: 'n next '
   classVariableNames: ''
```

```
   poolDictionaries: ''
   category: 'Actors-Examples'!


!PrimeFilter methodsFor: 'initialization'!

n: aPrimeNumber
   n _ aPrimeNumber! !

!PrimeFilter methodsFor: 'script'!

filter: i
   i \\ n = 0
      ifFalse: [next isNil
                  ifTrue: [Print reply: i.
                           next _ (PrimeFilter n: i) actor]
                  ifFalse: [next filter: i]]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

PrimeFilter class
   instanceVariableNames: ''!


!PrimeFilter class methodsFor: 'creation and initialization'!

n: n
   ^self new n: n! !
```

The `filter:` method is the heart of the program. The parameter `i` is checked in the way explained above. When a new prime number is found, it is printed by the `Print` actor.

During the computation, the generation of integers, the activities of filters, the transmission of integers between them, and the creation of new filters are concurrent actions. Note that the generator is the most passive object, because it has a simple and independent activity (it is activated only once). Thus it could be modeled as a passive object by typing `PrimeGenerator new upTo: 100`. The only difference from user point will be waiting for control until the generator reaches the maximum. This shows a first example of how the two types of objects and types of messages may cohabit.

A video extension of this example (by using views and controllers) is easily extended from this model. But this raises the issue of how to combine actors within the current Smalltalk-80 environment and methodology. This will be discussed in next section.

6 Symbiosis between Objects and Actors

6.1 Cohabitation: Double Facet of a Single Object

Our model of introducing active objects into Smalltalk-80 consists in designing a sub-world of actors, which are basically serializers built around usual objects. The behavior of an actor is the real essence of the actor. It may consider itself as a usual Smalltalk-80 passive object, or as the behavior of an actor which encapsulates and serializes it. The two variables, `self` and `aself`, respectively reference the behavior itself and the actor. The behavior may send usual recursive synchronous message calls by using `self` (as a Smalltalk object) or asynchronous message sendings to his "address" `aself` (as an actor). We now need to study how we may combine these two styles of programming. We will focus on the safety of such combination, and sketch the methodological aspects.

Note that the "system" methods, defined in the class `Actor` and which are used for the implementation of our actor kernel, use `self` bound to an actor. But the implementation resources should not be confused with the philosophy of actor programming from the user viewpoint. This issue will be touched upon in next section.

6.2 Bivalence of Methods and Encapsulation Issues

If we focus on the instance methods which are defined in a class of behaviors. We could classify them into three categories. (We use the Smalltalk-80 debugger convention for specifying an instance method belonging to a class.)

- *script methods* (for instance `Counter>>incr`)
  these are the methods defined for the behavior of the actor. They are intended to be sent to the actor, and then performed by the behavior. Note that we may subdivide this first category into two types:
  - *mixed methods* (for instance `Counter>>reset`)
    these methods, although defined for initialization of the behavior at creation time, could also belong to the script.
  - *pure script methods* (for instance `Counter>>consultAndReplyTo:`)
    these methods are only intended to be used by sending messages to the actor. For instance the specification of a reply destination parameter makes no sense to directly send them to the behavior.
- *private methods* (for instance `TreeComparator>>compareElements` in section 8.6)
  these methods are analog to the *private routines* as proposed into the ABCL/1 language [Yonezawa et al. 87]. They are used by the behavior, but they are not intended to be used as messages sent to the actor.

A good methodology for combining objects with actors should exclude *mixed methods* because they may obscure the meaning of the program. This also leads to the encapsulation issue, i.e. how to restrict access to the *private methods*. But the Smalltalk-80 programming philosophy and methodology is complete visibility, in contrast to the restricted visibility of abstract data types philosophy. However Smalltalk-80 is flexible enough (this paper is also intended to be one more demonstration of this fact!) to model such an approach. A first solution would be to define explicitly the messages which are to be accepted at the actor level, and reject the others. However this will lead us to define as many subclasses of class `Actor` as subclasses of class `ActorBehavior`. A simpler solution is to restrict the acceptance at the behavior level, i.e. by explicitly checking the category of selectors before accepting and performing messages. Being more concerned with flexibility than with security in this paper, at least for the moment, we leave such extension as a trivial exercise.

6.3 Towards Rules for Safe Combination

6.3.1 Serialized and Unserialized Objects

The Actor computation model distinguishes two types of actors, *unserialized* actors and *serialized* actors. An unserialized actor never changes its state (or owns no state at all), thus it may process several messages in parallel. It could also be arbitrarily copied to other nodes in a distributed architecture. An example of unserialized actor is the factorial actor presented in section 5.2.

On the other hand, a serialized actor may change its state. Therefore it should process messages one at a time. A serialized actor may be modelled simply as an unserialized object encapsulated in a serializer. This is the approach we developped for modeling actors in Smalltalk-80. An example of serialized actor is the counter actor presented in section 4.3.

Note that standard Smalltalk-80 objects may also be considered as unserialized actors, from the actor viewpoint. But we should at the same time consider them from the Smalltalk-80 viewpoint. An object which has a constant state (immutable) may be considered as a “real (and safe) unserialized actor”. We will call it a *constant object*. Primitive objects like numbers and system or user defined objects like `true` are two examples of constant objects. An object which could change its state is an “unprotected unserialized actor”. Unfortunately most of non primitive Smalltalk-80 objects are mutable objects. We need to be careful for these objects in order to combine them with actors. Actors bring multiple activities into the Smalltalk-80 language. We need to ensure that non primitive Smalltalk-80 objects may not be activated more than once through activity of actors, or at least to assume that their state won't change. This will lead us to methodological rules rather than formal rules.

6.3.2 Sketching Rules for Safe Combination

`Rule 1.` Only actors and constant objects may be shared by more than one actor.

This first rule is very general. Rule 1 may be decomposed into the following sub-rules:

`Rule 1.1.` The behavior of an actor must not be referenced by any other object than the actor.

`Rule 1.2.` Only actors or constant objects may be shared by several behaviors.

Rule for Creation of Actors We will now discuss Rule 1.1. Our first assumption is that a behavior is not directly referenced from outside the actor. This assumption is guaranteed by the existence for the user of only one entry point: the `actor` method, which guarantees (see section 4.5) that this behavior is owned by only one actor. However the explicit creation of the behavior may lead to reference it *before* encapsulation into an actor. Therefore we have to forbid the user to reference an object which will be used as behavior for an actor. (Note that if we would have chosen the `new` or `newActor` solutions to create an actor, this would have been guaranteed by encapsulation of the instantiation of the class of behaviors.) The combination of Rule 1.1 and this assumption leads to the following methodological rule:

`Rule 1.1.1` The behavior creation and actor creation should be part of a single composite transmission.

For instance `(Counter contents: 100) actor` is a valid expression for creating an actor. But the following sequence is invalid:

```
| aCounterBehavior aCounterActor |
aCounterBehavior _ Counter contents: 100.
aCounterActor _ aCounterBehavior actor.
...
```

because the behavior is referenced by the variable `aCounterBehavior` inside the scope of the temporary binding definition.

Rule for Referencing Objects Rule 1.2., as stated, is too strong for practical use. A class is an acceptable constant object if we assume that its class and pool variables dictionaries won't be updated and that no subclasses will be defined dynamically. If we want to have full protection over every object within Smalltalk-80, we would need to redefine every non constant Smalltalk-80 object as an actor. And this hardly conflicts with our first preservation goal.

6.3.3 Safety of Message Passing Combination

Once assumed with Rule 1.1 that the behavior of an actor is single referenced by it, we may deduce that there is at most only one activation of the behavior. Consequently the behavior may safely use recursive synchronous calls to itself (`self`). We may freely combine recursive synchronous and asynchronous messages in the methods of the behavior of an actor.

6.3.4 The Dangerous Nature of Blocks

[Yokote and Tokoro 87, pages 408–409] points out the dangerous nature of block contexts, because they share the environment of the method which defines them. If a method returns a block as value, it will be shared by the object that activated the method and the object that received the value. If at least one of these objects is an actor, this could generate timing errors. This leads to the following methodological rule:

`Rule 2.` A block may never been passed as a returned value to another object.

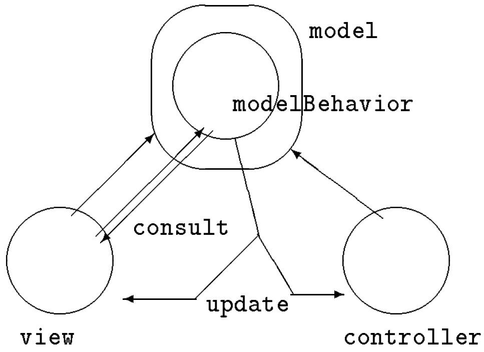

Figure 3: MVC model in the Actor world.

6.4 Combination with the MVC Paradigm

Unfortunately Rule 1.1 is a bit too strict for a good combination within the standard Smalltalk-80 programming environment and methodology. One interesting challenge is to study how the standard MVC (Model View Controller) model for interface design may combine with our actor kernel while keeping a safe concurrency. In this section we suppose the basic architecture of the MVC paradigm known, otherwise the reader may refer to [Krasner and Pope 88].

6.4.1 Model of Combination

We propose to keep almost intact the MVC methodology. The only change concerns the model. We will rely on inheritance to specialize the standard model. Only minor changes to the MVC methodology will be needed in order to keep safety.

The model (as seen from the controller and the view) will be naturally the actor whose behavior is the "real" model of the application. In this section, we will use the following terminology: *model* to denote the actor which is declared as the model of the associated view and controller, and *model behavior* to denote its behavior, which would be seen as the "real model" in a standard MVC application. Our terminology is coherent with the variable names in our design, and an encapsulation concern (view or controller's viewpoint). The model, as an actor, will transparently ensure synchronization of the activity of the model behavior and its interactions with the view and the controller.

Meanwhile the view usually needs to consult the variables of the model (behavior) in order to display information onto the screen. Message passing to an actor is asynchronous, consequently we cannot get the value of the contents. We may not use the reply destination principle, which was explained in section 5.1, because we would have to rewrite drastically the methods of the view. A synchronous message passing form will be introduced as an extension to our kernel model in section 8.1, but it does not belong to our kernel, and it could introduce some unnecessary waiting.

Our solution allows the view to reference the behavior of the model in order to consult its state. A new instance variable will be used for this purpose, and named `modelBehavior` along our terminology. Figure 3 shows the new MVC model.

6.4.2 Methodological Issues

Note that this solution violates our safety rules. We have to warn the user that interaction with the model behavior should be restricted to consultation of its state (excluding any assignment or side effect method). If the view needs to send other messages than just consultation to its model, then it has to send it to the `model` variable as usual. A programming interface such as a MVC generator could automatically ensure this assumption.

We still have to discuss the coherence issue, i.e. the assumption that the view displays current state of the behavior and that there is no delay possible. We first should explain how the dependency relationship, fundamental to the MVC, is preserved and set between the model behavior (and not the model!) and the view and controller.

When setting the dependency relationship between the view and the model, the message `addDependent:` is sent to the model, but it is performed by the model behavior. This means that the view and controller are dependents of the model behavior.

Now let's prove the coherence. The model behavior, when updating its state will send a `changed` message to itself (`self`), this will trigger the sending of an `update:` message to the view which is one of the dependents of the model behavior. This message is usually redefined in order to redisplay the view. The model behavior and the view are passive objects, thus the messages are all synchronous. This means that once the initial `changed` message passing execution is completed by the model behavior, the redisplay has been completed, as in the traditional MVC model.

6.4.3 What Needs to Be Changed?

The class `ActorView` implements this model of view dedicated to an actor model. It is a subclass of the class `View`. We will add an instance variable, named `modelBehavior`, for this purpose. When initializing a view, the `model:` message is sent to it in order to set its model. We redefine the `model:` method in order to introduce the initialization of `modelBehavior` as a complementary effect. We assume that the creation of a view *always* calls this method.

```
View subclass: #ActorView
   instanceVariableNames: 'modelBehavior '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-MVC'!


!ActorView class methodsFor: 'model access'!

model: aModel
   super model: aModel.
   modelBehavior _ aModel asObject! !
```

In the `model:` method, we use a new method, named `asObject` whose purpose is to directly access to the behavior of an object. Its definition is following:

```
!Actor methodsFor: 'behavior access'!

asObject
   ^behavior! !
```

Note that this method is not user intended because it violates Rule 1.1. It should be used only for implementation purposes and encapsulated in higher level user-oriented methods.

6.4.4 Application to the Example of the Counter

To illustrate the changes we made to the original model, we describe some MVC interface with our counter actor, defined in section 4.3. We won't describe all the interface but only the part that we changed. The remaining part is standard MVC construction, such as the one described in [Krasner and Pope 88, pages 23–27]. Note that we need to redefine the updating methods (`incr` and `reset`) of a counter behavior in order to send a `changed` message for updating dependents. The expressions which change are enclosed into `{}`.

```
{ActorView} subclass: #CounterView
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-MVC-Examples'!
```

```
!CounterView methodsFor: 'display'!

displayView
   {modelBehavior} contents printString displayAt: insetDisplayBox center! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

CounterView class
   instanceVariableNames: ''!


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

open
   self open: Counter new {actor}!

open: model
   (StandardSystemView new
      borderWidth: 3;
      label: model {asObject} class printString;
      addSubView: (self new
                     model: model;
                     insideColor: Form lightGray);
                     controller) open! !
```

Note that the `label: model {asObject} class printString` expression inside the `open:` method introduces a new call to the `asObject` method. The only purpose is to get, as usual, `Counter` as label of the view, and not `Actor`!

7 Concurrent Behaviors: the Actor Model of Computation

Now we will sketch some extensions to our actor kernel in order to model some of the most representative OCCP computation models and programming languages based on the actor concept. We won't discuss here all characteristics and advantages of the models of computation that we will talk about. For each extension, one or more references will allow the interested user to have a complete description of the respective models.

In this paper, we are mainly concerned about how we may augment our minimal actor kernel with building variations and higher level constructs while keeping our compatibility with Smalltalk-80. We don't intend here to model a complete programming language, such as the ABCL/1 language for instance [Yonezawa et al. 87]. We start this section with the Actor model of computation, as defined in [Agha 86].

7.1 The Actor Model of Computation

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, once accepting next message will specify its own replacement behavior. This leads to a causally 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. Figure 4 shows this new model of an actor.

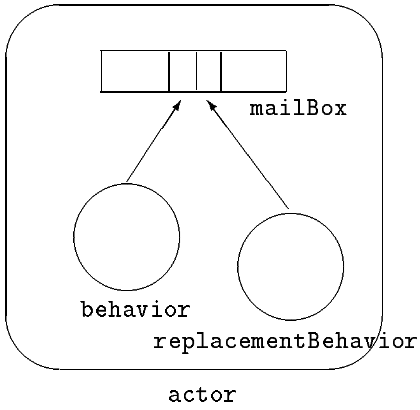

Figure 4: Modeling an actor with replacement of behavior.

7.2 Implementation

We will introduce a new class, named `AghaActorBehavior`, (because Gul Agha is the author of the latest Actor computation model!), as a subclass of `ActorBehavior`:

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


!AghaActorBehavior methodsFor: 'initialization'!

setProcess
   [self acceptNextMessage] fork! !

!AghaActorBehavior methodsFor: 'behavior replacement'!

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

Just 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). (Note that we rejected the original `become:` terminology proposed in [Agha 86], because of possible confusion with the primitive `become:` method of Smalltalk-80, which swaps pointers to objects. Nevertheless the `replace:` terminology is also used in the Pract language [Manning 87]. We will discuss the semantics of `replace:` as a method in sections 7.4 and 7.7.4.)

The semantics of the instance variable named `behavior` is slightly different than in the class `ActorBehavior`. The variable `behavior` now represents the *current behavior* of an actor. When performing a behavior replacement the actor will keep the same mailbox but change its *current behavior* reference to the *replacement behavior*. The current behavior won't be touched, thus it will complete its current computation. It could then be recovered by the system because it is no more referenced by the actor, and the process is also terminated.

7.3 The Example of the Counter Revisited

We will now redefine the example of the counter proposed in 4.3 in order to show the changes of semantics. We won't use inheritance here, because we do want the class `AghaCounter` to be a subclass of class `AghaActorBehavior`. (One solution would have been using the multiple inheritance facility, but we prefer to keep a simpler approach.)

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


!AghaCounter methodsFor: 'script'!

consultAndReplyTo: replyDestination
   self replace: (AghaCounter contents: contents).
   replyDestination reply: contents!

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

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

Each instance method must be redefined in order to specify the replacement behavior. In order to specify it, we need to express the creation and initialization of a new instance of `AghaCounter` through the usual "duality" class/instance method for the `contents:` creation and initialization method defined as in section 4.5.2.

The `reset` and `incr` methods express the state change through the change of behavior with some new contents. The `consultAndReplyTo:` method also needs to express a behavior change, although there is no state change. The new behavior is equal to the previous and current behavior, but it is distinct. If no replacement behavior is specified, the actor won't be able to accept next message, because it does not know how to perform it.

We may create and activate such a counter actor exactly the same way as in previous definition as a simple actor (section 5.1). In this example the reply destination of the consultation is the `Print` actor which could be some usual actor (`ActorBehavior`) or this new type of actor (`AghaActorBehavior`). They are totally indistinguable and exchangeable seen from the outside, as stated in the encapsulation principle.

7.4 Replacement Behavior Specification, Implementation, Optimization and Safety Issues

For modularity reasons we could replace the explicit invocation of the name of the class of behavior by the expression `self class`, as for instance in:

```
!AghaCounter methodsFor: 'script'!

incr
   self replace: (self class contents: contents + 1)! !
```

By doing so, we allow this method to be used by some subclass of `AghaCounter`. However if some subclass defines more instance variables, the class method for creating and initializing a behavior will change. This problem is already existent in standard Smalltalk-80 programming and due to the fixed number (and order) of keywords, as opposed to the more flexible model of keywords in CommonLisp.

Another simplification is possible when specifying an identical replacement behavior, i.e. without any state change. The expression `self replace: (AghaCounter contents: contents)` may be replaced by `self replace: self copy` which could also express the replacement of a (copy of a) constant behavior.

Note that the expression `self replace: self` is not valid in the Actor computation model, because the two behaviors are conceptually distinct although they are identical. However it is still valid in our implementation model because the two processes associated to the two behaviors are distinct. There is no assignment in this model, consequently a single behavior may be shared by distinct processes for processing distinct messages. These tricks could be viewed as optimizations of our implementation of the pure Actor computation model. As application, we may introduce a new method for replacement, named `ready`, as in the Pract language [Manning 87]. Its semantics is to specify an identical behavior as replacement, and consequently to be ready to accept next message. (Note that the expression `self replace: self` cannot be further reduced, otherwise next message could not be accepted.)

```
!AghaActorBehavior methodsFor: 'behavior replacement'!

ready
   self replace: self! !
```

Consequently the `consultAndReplyTo:` method may be redefined into:

```
!AghaCounter methodsFor: 'script'!

consultAndReplyTo: replyDestination
   self ready.
   replyDestination reply: contents! !
```

Our last remark will be about Pract implementation. Pract allows to update the state of a behavior rather than creating another one as a stronger optimization. This is valid in Pract because of the implementation of behavior activation through function invocation with local parameters bound to the state variables of the behavior. Thus the state may be updated without affecting the previous behavior still being computed. We cannot use this optimization in our implementation in Smalltalk-80 because behavior activation relies on process activation and the current and replacement behaviors processes share the same behavior as state.

7.5 Concurrency Issues

7.5.1 Autonomy Increases Concurrency and Higher Level Synchronization

Recall our definition of active objects in contrast to passive objects by bringing them autonomy (in section 3.3). Two *read* messages (such as the method `consultAndReplyTo:` for a counter actor) may now be concurrently performed by two behaviors. There is no more need to complete a message to start performing next one. The computation of incoming messages may be *pipelined*. It is now the responsability of the object to decide *when* and *how* it could perform next message. Thus *read* messages may be simultaneous, and *write* messages may be synchronized very precisely in order not to restrict concurrency. There is only one high level construct, *behavior replacement*, for expressing state change (and more general changes) and synchronization at the method definition level.

7.5.2 Concurrency of Actions of a Method

The Actor model of computation makes no assumption about the ordering of actions inside the body of a method. It may interesting for the reader to remark that the expressions inside script methods verify this assumption. There is no correlation between them, consequently they may performed simultaneously. However our implementation relies on the underlying sequential Smalltalk-80 model. To simulate concurrency of actions would greatly complexify our model and decrease the efficiency. In this respect, our implementation reveals an implicit optimization about sequencing of actions.

7.5.3 Efficiency Issues

> *"Quoique j'invente, quoique je fasse, je serai toujours au dessous de la vérité. Il viendra toujours un moment où les créations de la science dépasseront celles de l'imagination."*
> Jules Verne.

Our last remark will be about efficiency. As already pointed out when discussing some optimizations, the model if taken litteraly is not very efficient on conventional mono-processor architectures. The goal of the actor model is to anticipate new giga-processors architectures by expressing the maximum amount of concurrency, which means *potential* parallelism.

7.6 Change of Behavior: the Example of Join Continuations

We may notice that in the examples we proposed, the model (class) of behavior remains constant, although there could be some state change. In this section we will show *how* to express a complete change of behavior, and *why* it is fundamental to easily model general computations.

7.6.1 Join Continuations

In section 5.2.1 we introduced the concept of *continuation* to express some remaining computation of a single actor. The continuation is resumed by some actor which sends a `reply:` message which may include some partial result. In case of computations distributed among several actors, the partial results may come from several distinct sources. We need continuations which may get the values one at time and resume the computation once all values have been received. Such a continuation synchronizes the junction of concurrent sub-computations. It is usually called a *join* continuation, like in data flow graphs.

7.6.2 A Binary Adder

A simple example of join continuation will be to model an arithmetic operator which needs two values to compute its result. Once received a first value, the binary adder will change its behavior to become some unary adder, memorizing the value and waiting for the second (and last) value. Once received the second value, the unary adder may perform the addition and return the result to the initial continuation.

The following classes `BinaryAdder` and `UnaryAdder` model the evolutive behavior of this actor:

```
AghaActorBehavior subclass: #BinaryAdder
   instanceVariableNames: 'c '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Examples'!


!BinaryAdder methodsFor: 'initialization'!

c: aContinuation
   c _ aContinuation! !

!BinaryAdder methodsFor: 'script'!

reply: v
   self replace: (UnaryAdder v1: v c: c)! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

BinaryAdder class
   instanceVariableNames: ''!
```

```
!BinaryAdder class methodsFor: 'instance creation and initialization'!

c: c
   ^self new c: c! !
```

7.6.3 A Unary Adder

```
AghaActorBehavior subclass: #UnaryAdder
   instanceVariableNames: 'v1 c '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Examples'!


!UnaryAdder methodsFor: 'initialization'!

v1: anInteger c: aContinuation
   v1 _ anInteger.
   c _ aContinuation! !

!UnaryAdder methodsFor: 'script'!

reply: v2
   c reply: v1 + v2! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

UnaryAdder class
   instanceVariableNames: ''!


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

v1: v1 c: c
   ^self new v1: v1 c: c! !
```

7.6.4 Trapping Excessive Messages

Note that the `reply:` method of the `UnaryAdder` does not specify some replacement behavior. This is unnecessary because no other message should arrive to the binary adder, once the two values received. We may use a special behavior, named `NoMoreMessage`, to trap these messages:

```
AghaActorBehavior subclass: #NoMoreMessage
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Extensions'!


!NoMoreMessage methodsFor: 'script'!

doesNotUnderstand: aMessage
   self ready.
   self error: 'no more message'! !
```

and use it for `UnaryAdder`:

```
!UnaryAdder methodsFor: 'script'!
```

```
reply: v2
   self replace: NoMoreMessage new.
   c reply: v1 + v2! !
```

Note that trapping messages is a recurrent issue along this paper. However, we may not directly define class `NoMoreMessage` as a subclass of `EObject`, as in section 4.4.1, because it is already a subclass of `AghaActorBehavior`. We need to define some special class, named `AghaActorBehaviorEObject`, as a subclass of `EObject` to trap any kind of message. The same issue will reappear in section 7.8.4.

7.6.5 The Fibonacci Example

We will now use our model of binary adder in order to compute the function fibonacci. The form is quite similar to the factorial function we defined in section 5.2. Binary adders will be used to join and add the two recursive computations:

```
ActorBehavior subclass: #Fibonacci
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Examples'!


!Fibonacci methodsFor: 'script'!

n: n replyTo: c
   | join |
   n < 2
      ifTrue: [c reply: 1]
      ifFalse: [join _ (BinaryAdder c: c) actor.
                aself n: n - 1 replyTo: join.
                aself n: n - 2 replyTo: join]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Fibonacci class
   instanceVariableNames: ''!


!Fibonacci class methodsFor: 'example'!

example
   "  Fibonacci example  "
   Fibonacci new actor n: 10 replyTo: Print! !
```

Note that we combined two types of actors, the initial model (`ActorBehavior`) for the fibonacci actor, and the Actor model (`AghaActorBehavior`) for the binary adder. This shows that they are totally exchangeable. On the other hand the specification of a join continuation would have been much less natural in our initial model of actors.

7.6.6 Memoization

In order to avoid several recomputations of the function, we may define a new version of the fibonacci actor which will use some memory to memorize the values already computed. We may easily design such an extension by inquiring the memory for the value. If already computed, the memory will directly reply to the continuation. If not, it will request the computation to the fibonacci actor with a continuation that will get the result and return it both to the initial reply destination and to the memory in order to memorize this new value. There could be interesting variations made along optimistic or pessimistic approaches, i.e. deciding to start a computation before or after the response from the memory. Although easy to model, we won't show the definitions in this paper. This is left as an exercise.

7.6.7 Identification Issues

This example of join continuation has symetric inputs, i.e. the ordering arrival of replies is not significant. However, for instance in case of a divider operator, the join continuation may need to discriminate the values. There are several techniques available, but the basic one is to "stamp" the messages with some distinctive information, like the value of a counter which increments, or the sender. This technique will be touched upon in section 7.7.7.

7.7 Waiting for Information: the Concept of Insensitivity

Actors have the ability to take *decisions*, but tests should be *local* to preserve the *locality* philosophy of OOP. This means that decisions could be only based on the local state of the behavior of an actor. In some non trivial cases, the actor needs some extra-information which it does not own, therefore it needs to request this information to some other object. If this object of which it needs some information is some Smalltalk-80 object, then some synchronous message passing will be used, and it will get the value. But if this object is an actor (actually it *should* be an actor if its state may change because it is referenced by an actor) only asynchronous message passing may be used. The problem is *who* will and *how* to get the reply. By creating a continuation, the actor will delegate to it the remaining computation that the continuation will resume once received the reply. Consequently the actor may immediately accept next message, without waiting. However sometimes, the actor needs this extra information to specify its replacement behavior. Then some waiting is necessary, and the actor may not process incoming messages before knowing how to do it. We will discuss how to model such waiting in the Actor model of computation through a progressive example following [Agha 86, pages 52–55].

7.7.1 The Bank Account Example

We will now define a model of behavior for bank accounts as a kind of conceptual derivation from the class `Counter`. The bank account example is the standard example of modeling objects with changing state in the Actor model of computation, for instance in [OOCP 87, page 46].

A bank account replies some acknowledgement to an operation request. The `#receipt` response means a completion of the operation requested. An `#overdraft` response occurs if the withdrawal is more important than current balance.

Note that we could have modeled the acknowledgement through several distinct methods, such as: `deposit-receipt:` *deposit-amount*, `withdraw-receipt:` *withdraw-amount*, and `overdraft-warning:` *overdraft-amount*. But our first solution may use the `reply:` selector, thus the `Print` actor may be used as the actor to which the bank account will acknowledge the operation. This simplifies our example:

```
AghaActorBehavior subclass: #BankAccount
   instanceVariableNames: 'balance '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Examples'!


!BankAccount methodsFor: 'initialization'!

balance: anInteger
   balance _ anInteger! !

!BankAccount methodsFor: 'script'!

balanceAndReplyTo: replyDestination
```

```
   self ready.
   replyDestination reply: balance!

deposit: amount replyTo: replyDestination
   self replace: (BankAccount balance: balance + amount).
   replyDestination reply: #receipt!

withdraw: amount replyTo: replyDestination
   balance < amount
      ifTrue: [self ready.
               replyDestination reply: #overdraft]
      ifFalse: [self replace: (BankAccount balance: balance - amount).
                replyDestination reply: #receipt]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

BankAccount class
instanceVariableNames: ''!


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

balance: balance
   ^self new balance: balance! !

!BankAccount class methodsFor: 'example'!

example
   "   BankAccount example   "
   (BankAccount balance: 200) actor deposit: 150 replyTo: Print;
                                    withdraw: 400 replyTo: Print! !
```

Note that in case of overdraft, the balance won't change (before the justice decision!). The error handling of such situation by the reply destination is left out of the scope of this paper. However protection against overdrafts will be studied in next section.

7.7.2 Waiting for Information: The Protected Bank Account Example

Now, following [Agha 86, pages 52–55], we propose some specialization of a previous bank account example. In order to manage overdrafts, a protected bank account will own some private bank account, called its `savings`, as an overdraft protection. In case of overdraft, the protected bank account will inquire its savings for protection, by sending to it a request for withdrawal of the overdraft amount. If the overdraft could be covered by the savings (`#receipt` response), this amount is withdrawn from the savings and used to equilibrate the balance of the protected bank account (which then becomes null). If the overdraft is too important for the savings (`#overdraft` response), then previous behavior is unchanged.

```
BankAccount subclass: #ProtectedBankAccount
   instanceVariableNames: 'savings '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Examples'!


!ProtectedBankAccount methodsFor: 'initialization'!

savings: aBankAccount
   savings _ aBankAccount! !
```

```
!ProtectedBankAccount methodsFor: 'script'!

deposit: amount replyTo: replyDestination
   self replace: (ProtectedBankAccount balance: balance + amount savings: savings).
   replyDestination reply: #receipt!

withdraw: amount replyTo: replyDestination
   balance < amount
      ifTrue: [{(savings asObject withdraw: amount - balance)} = #receipt
                  ifTrue: [self replace:
                              (ProtectedBankAccount balance: 0 savings: savings)]
                  ifFalse: [self ready]]
      ifFalse: [self replace:
                  (ProtectedBankAccount
                      balance: balance - amount savings: savings)]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

ProtectedBankAccount class
   instanceVariableNames: ''!


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

balance: balance savings: savings
   ^self new balance: balance; savings: savings!

balance: balance savingsAmount: savingsAmount
   ^self balance: balance savings: (BankAccount balance: savingsAmount) actor! !
```

As you may have noticed, inside the `withdraw:replyTo:` method, the expression `{(savings asObject withdraw: amount - balance)}` is enclosed in `{}`, because it is invalid! We would like to express the request to the savings as well as the value of the response replied in order to take decision. However our message passing model between actors is asynchronous. We may not use a continuation to get the reply because the behavior of the protected bank account needs it in order to decide what will be its replacement behavior. And without replacement behavior it cannot accept the reply! This leads to a "semantic deadlock".

7.7.3 The Concept of Insensitivity

The basic idea to solve the problem, is to sleep for a while, while waiting for the reply. Messages arriving to the actor during this period of "lethargy" will be buffered in order to be accepted after the actor regained a sensitive behavior. This concept is called *insensitivity* in the Actor model. This insensitivity should not be total because the actor should be able to detect the right reply and discriminate it with other messages which could arrive during that time.

7.7.4 Implementation Models

A first conceptual model of implementation is proposed in [Agha 85]. When becoming insensitive, the actor will get a special replacement behavior, called *insensitive behavior*. The semantics of this behavior is to check if the message it accepts is the one it is waiting for. Such a pattern of message is: `becomeSensitiveWithBehavior:` *replacement behavior*. We call this message the *replacement message*, *replacement behavior* is the behavior which will be used as replacement behavior when the actor will regain sensitivity. Otherwise the message is recursively sent to the actor. Any other message will keep being accepted and recursively resent before being eventually accepted after the actor gets a sensitive behavior. Of course this first strategy, although easy to model and implement, is unrealistic in a real implementation, because of the computation overhead due to all the messages "turning" around the actor before it gets sensitive.

The second model, as proposed in [Agha 86], introduces a buffer actor (or object) in order to keep all the messages which are waiting to be performed. When the actor will become insensitive (get an insensitive behavior), it will create a new buffer. Every message but the replacement message will be kept into this buffer. When the actor will get a sensitive behavior again, it sends to the buffer a request to redirect all these buffered messages to itself. This second model appears as an optimization of the first model. Meanwhile we are now using two data structures to buffer messages, one message queue for the actor, and a buffer for the insensitive behavior. This is a kind of waste.

The third model, as proposed in [Manning 87], keeps storing the messages into the message queue. The replacement message is checked at the actor level, and no more at the behavior level. This makes impossible to express this model of insensitivity in the Actor model of computation itself, as the first two models could. But of course we gain much efficiency. In fact we will lose some ability to check the identity of the messages that is important. This last issue is not preeminent in our paper, it will be discussed after completing our protected bank account example, in section 7.7.7.

7.7.5 Strategy for Implementation

Our implementation of the third solution is immediate in our model. The replacement into an insensitive behavior becomes trivial: it is equivalent to avoid any behavior replacement. But we still need a means to wake up the "sleeping beauty" from the outside, because the current behavior did not specify a replacement. This should be done at the actor level, as stated in previous section. We simply redefine the replacement message (`replace:`), initially for behaviors, now for actors. For modularity reasons, we define it in the class `AghaActor`, subclass of `Actor`:

```
Actor subclass: #AghaActor
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Extensions'!


!AghaActor methodsFor: 'behavior replacement'!

replace: replacementBehavior
   self initializeBehavior: replacementBehavior! !
```

7.7.6 The Return of Safety Issues

This solution again leads to some safety issues. Suppose that in case of programming bug, some `replace:` message is delivered to some actor which is sensitive, i.e. whose replacement behavior was already or will be computed. The result is disastrous because we will get two unrelated behaviors which are active simultaneously. They are not causally related like a behavior and its replacement behavior should be. Note that this problem was already existant before, because there is no checking of how many `replace:` actions may be specified in a method of a behavior.

Our solution to ensure the unicity of the chain of behaviors uses a flag to avoid double replacement. This flag is named `isReady` and indicates whether an actor owns a behavior which is ready to accept a message. The initial value of the flag is `false` (before a behavior is assigned to a new actor). The flag is set at `true` each time there is a behavior assignment and initialization (the `initializeBehavior:` method, which is called both when creating an actor with some initial behavior, and when there is a behavior replacement). The flag is set at `false` each time the current behavior accepts a message (the `acceptMessage:` method). It is legal to compute some behavior replacement from the acceptance of a message until some behavior replacement is computed. Otherwise an error is raised.

For protection purposes, we need an additional semaphore for mutual exclusion in order to monitor the flag. The new class for actors is called `ExtendedAghaActor` and a subclass of `AghaActor`. The `initializeBehavior:` method is redefined in order to check and set the flag:

```
AghaActor subclass: #ExtendedAghaActor
   instanceVariableNames: 'isReady isReadySemaphore '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Extensions'!


!ExtendedAghaActor methodsFor: 'initialization'!

initialize
   super initialize.
   isReady _ false.
   isReadySemaphore _ Semaphore forMutualExclusion!

initializeBehavior: aBehavior
   isReadySemaphore critical:
      [isReady
          ifTrue: [self error: 'replacement behavior could be specified only once']
          ifFalse: [isReady _ true.
                   super initializeBehavior: aBehavior]]! !

!ExtendedAghaActor methodsFor: 'iv access'!

unsetIsReady
   isReadySemaphore critical: [isReady _ false]! !
```

The `acceptMessage:` is also redefined to unset the flag. For coherence and modularity, we also redefine the `replace:` method for the behavior as calling the `replace:` method on the actor:

```
AghaActorBehavior subclass: #ExtendedAghaActorBehavior
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Extensions'!


!ExtendedAghaActorBehavior methodsFor: 'message acceptance'!

acceptMessage: aMessage
   aself unsetIsReady.
   super acceptMessage: aMessage! !

!ExtendedAghaActorBehavior methodsFor: 'behavior replacement'!

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

This solution detects errors in programming but it does not avoid them. In the Acore/Pract system, this is ensured by the compiler. The user never directly specifies internal or external behavior replacement, but specifies its programs in a higher level functional like form. The compiler itself generates the appropriate continuations and behavior replacement basic Pract commands. The control of the interface is some important point within the Actor model.

7.7.7 Safety of Identification

One important problem, that we skipped in section 7.7.4, is the identification problem. We based the identification of the replacement message on the sole nature of its selector, `replace:`. Note that specifying the replacement behavior may be *internal* or *external*. Our first implementation of `replace:`, in section 7.2, was internal, because it was encapsulated in a "command" used within the behavior of the actor itself. In this first case there was no need for identification. However the needs for external `replace:` in order to model insensitivity, in section 7.7.4, lead to this identification issue.

To ensure the identification, the usual technique is to "stamp" the message. This stamp must be unique, therefore we will create an actor specially for this purpose as a *key* for accessing to and waking up the actor. This identification actor will be created and referenced by the actor, then communicated as a parameter when sending the request and waiting for its reply. The replacement message now becomes: `replace:` *replacement behavior* `key:` *key-actor*. The actor will check the identity of the key it knows about and the key it receives. If the keys differ, an error is raised. Note that in our implementation, the key is memorized by the actor. It may also be kept by the current behavior.

```
ExtendedAghaActor subclass: #KeyedExtendedAghaActor
   instanceVariableNames: 'key '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Extensions'!


!KeyedExtendedAghaActor methodsFor: 'iv access'!

key: anActor
   key _ anActor! !

!KeyedExtendedAghaActor methodsFor: 'replacement behavior'!

replace: replacementBehavior key: aKeyActor
   aKeyActor = key
      ifFalse: [self error: 'wrong behavior replacement key']
      ifTrue: [self replace: replacementBehavior]! !
```

The `becomeInsensitiveWithKey:` method is just an assignment method to set the value of the key.

```
ExtendedAghaActorBehavior subclass: #KeyedExtendedAghaActorBehavior
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Extensions'!


!KeyedExtendedAghaActorBehavior methodsFor: 'insensitiveness'!

becomeInsensitiveWithKey: anActor
   aself key: anActor! !
```

7.7.8 Coherence Issues

We decided to raise an error in case of key mismatch. However imagine some complicate (pathological?!) case. Suppose that the insensitive actor is intended to become a forwarder (see section 7.8.4) to some other insensitive actor. If the first insensitive actor receives the replacement message intended for the second one, it should not reject it, but buffer it in order to forward it later. Consequently we need to put the wrong replacement message in the message queue. This unfortunately leads to some dirty Smalltalk programming. In any other case, the sensitive behavior of the actor, once got the right replacement message, will reject the previous wrong replacement message and automatically raise an error (because the `replace:key:` selector does not belong to behaviors). This leads to the following new definition:

```
!KeyedExtendedAghaActor methodsFor: 'replacement behavior'!

replace: replacementBehavior key: aKeyActor
   aKeyActor = key
      ifFalse: [mailBox nextPut:
                  (Message selector: #replace:key:
                           arguments:
                              (Array with: replacementBehavior with: aKeyActor))]
      ifTrue: [self replace: replacementBehavior]! !
```

7.7.9 Application to the Bank Account Example

We may now solve our protected bank account problem. We will create an actor to manage the overdraft, as explained in [Agha 86, pages 53–55]. It will be sent as reply destination to the savings bank account. It will get the reply of the savings and send the `replace:` message to the protected bank account with a replacement behavior depending on the response. Note that the protected bank account becomes implicitly insensitive because there is no specification of replacement behavior.

```
!ProtectedBankAccount methodsFor: 'script'!

withdraw: amount replyTo: replyDestination
   balance < amount
      ifTrue: [savings withdraw: (amount - balance) replyTo:
                 (OverdraftManager account: aself balance: balance savings: savings
                                   replyDestination: replyDestination) actor]
      ifFalse: [self replace:
                  (ProtectedBankAccount
                     balance: balance - amount savings: savings)]! !
```

The overdraft manager is modeled below:

```
AghaActorBehavior subclass: #OverdraftManager
   instanceVariableNames: 'account balance savings replyDestination '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Examples'!


!OverdraftManager methodsFor: 'initialization'!

account: anInsensitiveActor balance: anInteger savings: aBankAccount
                           replyDestination: anActor
   account _ anInsensitiveActor.
   balance _ anInteger.
   savings _ aBankAccount.
   replyDestination _ anActor! !

!OverdraftManager methodsFor: 'script'!

reply: response
   replyDestination reply: response.
```

```
   response = #receipt
      ifTrue: [account replace: (ProtectedBankAccount balance: 0 savings: savings)]
      ifFalse: [account replace:
                  (ProtectedBankAccount balance: balance savings: savings)]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

OverdraftManager class
   instanceVariableNames: ''!


!OverdraftManager class methodsFor: 'creation'!

account: account balance: balance savings: savings replyDestination: replyDestination
   ^self new account: account balance: balance savings: savings
            replyDestination: replyDestination! !
```

When the overdraft manager receives the response from the savings bank account (`reply:` method), it redirects (delegates) the response from the savings to the initial reply destination. Then the appropriate behavior is sent for replacement to the protected bank account.

We may now test our protected bank account:

```
!ProtectedBankAccount class methodsFor: 'example'!

example
   "   ProtectedBankAccount example   "
   (ProtectedBankAccount balance: 200 savingsAmount: 100) actor
      deposit: 150 replyTo: Print;
      withdraw: 400 replyTo: Print;
      balanceAndReplyTo: Print! !
```

This will produce:

```
> receipt
> receipt
> 0
```

Note that, like in the example of factorial in section 5.2, the overdraft manager, as a continuation, closures the bindings it needs for resuming the computation. These bindings memorize: the protected account itself (`aself` closured in the variable `account`), instance variables (`balance` and `savings`), and the message parameter (`replyDestination`). We could also closure the behavior itself rather than its instance variables. In that case, we need to define methods in the class `ProtectedBankAccount` to access these instance variables.

7.7.10 Application of the Identification Technique

If we use the technique of identification based on keys, the methods will slightly change. Note that the overdraft manager itself is used as the key.

```
!ProtectedBankAccount methodsFor: 'script'!

withdraw: amount replyTo: replyDestination
| overdraftManager |
   balance < amount
      ifTrue: [overdraftManager _
                 (OverdraftManager account: aself balance: balance savings: savings
                                   replyDestination: replyDestination) actor.
               self becomeInsensitiveWithKey: overdraftManager.
               savings withdraw: (amount - balance) replyTo: overdraftManager]
      ifFalse: [self replace:
```

```
                    (ProtectedBankAccount
                        balance: balance - amount savings: savings)]! !

!OverdraftManager methodsFor: 'script'!

reply: response
   replyDestination reply: response.
   response = #receipt
      ifTrue: [account replace: (ProtectedBankAccount balance: 0 savings: savings)
                        key: aself]
      ifFalse: [account replace:
                    (ProtectedBankAccount balance: balance savings: savings)
                         key: aself]! !
```

7.7.11 Complementarity of Continuation and Insensivity Concepts

As already pointed out in 5.2.4, continuation and insensitivity constructs are not intended to be explicitly used by the programmer. A compiler, such as the Acore/Pract compiler [Manning 87] compiles functional style programs into continuations. If the replacement of the behavior occurs inside the scope of a binding to some function call, this means that the behavior needs to wait for information from another actor to be able to specify its replacement behavior. In such a case an insensitivity management is generated by the compiler.

7.8 From Insensitivity to Eager Computation

This concept of insensitivity, defined to model waiting for information, will be now paradoxally used to increase concurrency between actors.

7.8.1 The Concept of Future

The concept of future was initially invented by Carl Hewitt and introduced into the Act 1 programming language [Lieberman 87]. It then has been applied to other languages, for instance Multilisp [Halstead 84], as a programming construct to introduce concurrency. The intuitive idea is to reference an object before it gets known. This is specially useful for replying immediately an object to a customer before it gets computed. Consequently the two actors do not need to wait for completion of the computation of the object to exchange a reference of this object. Of course if the customer gets to need the real nature of the object, it will then have to wait for completion of its existence, but there is no unnecessary waiting. Some metaphor of the concept of future is the following story:

> *"Some customer wants to buy a chicken from a farmer. The farmer gets the money, gives him an egg while telling: "Sorry, I've no more chicken ready to sell. I'll give you an egg right now. After a while this will become a chicken. But you may carry it around, including selling it to someone else, starting from now. It is already yours.""*

As suggested by the business transaction between the farmer and the customer, there is a contract between them. The farmer guarantees that this egg will *eventually* become a chicken, although he does not tell exactly when. The egg is a *future* chicken, a *promise* (contract) to get a chicken. This future object may be used in place of the real object, e.g. as an argument of a message to transmit it to another object, as long as there is no need for its real value, in that latter case such requests need to wait.

7.8.2 Modelization with Insensitivity

Now let's consider this concept of future in the context of our actor kernel. The future is an actor which will buffer all messages until its real nature gets computed. This means that a future owns an insensitive behavior. A replacement behavior will be sent to the future when its behavior will get computed.

A future actor is simply an actor without behavior. Consequently the implementation is immediate. The class `AghaFutureActor` models future actors. It is defined as a subclass of `ExtendedAghaActor`, without any new definition:

```
ExtendedAghaActor subclass: #FutureActor
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Extensions'!
```

The following example creates an actor still unknown, and sends some messages to it. These messages (and the following ones) will get accepted as soon as the replacement behavior will get known and communicated to the future.

```
!FutureActor class methodsFor: 'example'!

example
   "   FutureActor example   "
   FutureActor new
      deposit: 200 replyTo: Print;
      withdraw: 50 replyTo: Print;
      replace: (BankAccount balance: 200);
      withdraw: 300 replyTo: Print;
      balanceAndReplyTo: Print! !
```

The display will be:

```
> receipt
> receipt
> receipt
> 50
```

7.8.3 Uniformity Issues

The limitation of this model of future is that it accepts only a behavior when getting computed. In the *universe of actors* model, as defined in [Agha 86, page 57], every conceptual entity is modeled as an actor. Thus when the value of the future is computed, the behavior of this value will be communicated to the future as its replacement behavior. However our kernel actor model is not fully uniform. Some basic entities, called *rock-bottom* objects in [Lieberman 87], are not real actors, and consequently do not have behaviors. In that case, the future cannot "become" such a primitive object. We need one more level of indirection. This will be provided by the concept of *forwarder*.

7.8.4 Forwarder Concept

We will model the behavior of a *forwarder*, i.e. an actor which forwards all messages it receives to another actor (called its *proxy*). But suppose that we don't know yet its proxy. Before knowing it, the forwarder will buffer all incoming messages. When the proxy gets known it could send all messages which have been buffered to the proxy and change its behavior into forwarding all messages to come. The forwarder before knowing its proxy behaves as a future actor.

The class `FutureForwarderActor` models a future forwarder. `FutureForwarderActor` is simply a subclass of class `FutureActor`: The `forwardTo:` method specifies the proxy of a future forwarder.

```
FutureActor subclass: #FutureForwarderActor
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha-Extensions'!
```

```
!FutureForwarderActor methodsFor: 'forwarder replacement'!

forwardTo: anActor
   self replace: (Forwarder to: anActor)! !
```

The class `Forwarder` models the behavior of a forwarder. The `doesNotUnderstand:` method is redefined in order to forward messages to the proxy (`forwardTo` instance variable).

```
ExtendedAghaActorBehavior subclass: #Forwarder
   instanceVariableNames: 'forwardTo '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-Agha'!


!Forwarder methodsFor: 'initialization'!

forwardTo: anActor
   forwardTo _ anActor! !

!Forwarder methodsFor: 'message handling'!

doesNotUnderstand: aMessage
   self ready.
   forwardTo performMessage: aMessage! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Forwarder class
   instanceVariableNames: ''!


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

to: anActor
   ^self new forwardTo: anActor! !
```

We may test our new construct on the previous example pattern (section 7.8.2). Now the value computed may be of any kind:

```
!Forwarder class methodsFor: 'example'!

example
   "  FutureForwarder example  "
   FutureForwarder new actor
      incr;
      incr;
      consultAndReplyTo: Print;
      forwardTo: (Counter contents: 100) actor;
      incr;
      consultAndReplyTo: Print! !
```

The display will be:

```
> 102
> 103
```

7.8.5 Methodological Issues

Note that our implementation may be optimized if the forwarding is performed at the actor level, and not at the behavior level. On the other hand, like in section 7.7.4, this construct is no more expressed at the behavior (conceptual) level.

Futures and future forwarders are first class actors which may be passed around to other actors. The identification issue, already discussed in section 7.7.7 is still valid here. The compiler may ensure and/or restrict the use of replacement messages.

8 Eager and Synchronous Communication: the ABCL Model of Computation

We will now define the basic computation model of another actor-oriented programming language as an extension of our kernel. The ABCL/1 programming language [Yonezawa et al. 87], although based on the actor philosophy, chose a more pragmatic approach. The language is not intended to be self-contained, but reveals hybrid computation. The actor-oriented model of computation may combine with more traditional programming languages which could be used for expressing the bodies of the methods of objects. Another main characteristic of ABCL/1 is to propose three distinct types of communication protocols between objects at the user level.

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

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

future is an eager type of communication. The place where the reply will be delivered in the *future* is specified at the time of sending. Semantics is slightly different with our model of future discussed in section 7.8 as we will see below.

There are some other characteristics of the model that we won't model here. Messages may be sent in *express mode*, i.e. with some higher priority. This is the means to control the activity of an object. The interruption issue will be slightly touched upon in section 8.6.3. The definition of *private routines* in a traditional programming language (usually Lisp) bears a strong ressemblance with our standard Smalltalk-80 methods, as already discussed in section 6.2. The *transmission ordering assumption*, restricting the *non determinism principle* of the Actor model of computation, guarantees that the ordering of two messages sent from and to same objects will not change. Due to our current implementation, this assumption is preserved in Actalk.

8.1 A Pragmatic Approach to the Concept of Future

In the Actor model of computation, waiting is simulated through insensitive behavior, as we saw in section 7.7.3. In ABCL/1, waiting (for a pattern of message) mode is primitive. This is motivated by efficiency and expressiveness concern. The synchronous *now* type of message passing may be reduced to *past* type plus *waiting mode* [Yonezawa et al. 87, pages 87–88].

We will keep this pragmatic approach in our implementation in Actalk. We would like to directly express the value replied in the *now* type of message passing. Moreover we would like this reply to be any kind of object, and not necessarily an actor. *Now* type of message passing will rely on the underlying standard synchronous message passing level. The value returned will be some standard Smalltalk-80 passive object on which the receiver will synchronize to get the real value once computed. The receiver will get the reply immediately, and then it may, when necessary, wait for the real value. This is in fact another approach to design eager type of message passing, already discussed in section 7.8. This model is analog to the `CBox` object in ConcurrentSmalltalk. But in ConcurrentSmalltalk, simple asynchronous message passing does not exist, it *always* returns a `CBox` object wherever the receiver will use it or not.

In contrast to the previous approach of futures in the Actor model of computation, the returned value may be used as any kind of Smalltalk-80 object. It does not have to be an actor (although it may be of course). For instance if the returned value is a number, the user may manipulate it as usual without the needs for modeling numbers as actors and dealing with continuations to use it.

8.1.1 Implementation

We will define a new type of future object according to this strategy. It will be a passive Smalltalk-80 object owning some location for the real value it will reference. A semaphore is used to synchronize access to the value, i.e. to force any actor which wants to use the value to wait for computation of the value.

We would like to have an implicit access to the value of the future, i.e. to use it as a standard object and to get automatic synchronization if the value is not computed yet. We will reuse the technique of trapping error messages to implicitly call an explicit consultation message onto the future. Therefore the class `Future` is defined as a subclass of class `EObject`.

```
EObject subclass: #Future
   instanceVariableNames: 'value semaphore '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-ABCL'!


!Future methodsFor: 'initialization'!

initialize
   semaphore _ Semaphore new! !

!Future methodsFor: 'consultation'!

value
   semaphore wait; signal.
   ^value! !

!Future methodsFor: 'assignment'!

reply: aValue
   value _ aValue.
   semaphore signal! !

!Future methodsFor: 'implicit access'!

doesNotUnderstand: aMessage
   ^self value performMessage: aMessage! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

Future class
   instanceVariableNames: ''!


!Future class methodsFor: 'creation'!

new
   ^super new initialize! !
```

The `value` method is used to consult the value of the future. The semaphore synchronizes such access (`wait` message). It is immediately reset (`signal` message) in order to give access to another `value` message. The `reply:` method is used to set the value of the future once computed, and to release the semaphore. The `doesNotUnderstand:` method is redefined in order to implicitly call the consultation (`value` message) of the future.

8.1.2 Safety Issues

Unfortunately this model of future violates our rules for safe collaboration between passive and active objects, because this passive object is shared by at least two actors. These are the actor which creates it and will assign its value, and the actor which gets it as a value and will consult it. It could even be shared by more actors, because as a first class object it may be communicated to other actors.

The problem is not consulting the future but assigning it. The value of the future once computed could not change. The future should be single assignment. Then our safety problem is solved because there could be only one assignment *and* the consultations may not occur *before* the assignment. The future is a constant.

In order to ensure this single assignment assumption, we need to redefine futures. The class `SAFuture` (as Single Assignment Future) is defined as a subclass of class `Future`. It adds a new instance variable, named `isReady`, which denotes a boolean to check the status of the future. A semaphore, named `isReadySemaphore`, is defined to ensure mutual exclusion when accessing and checking the flag. The flag and its semaphore are initialized by redefinition of the `initialize` method. The `reply:` method is redefined to check the flag before assignment.

```
Future subclass: #SAFuture
   instanceVariableNames: 'isReady isReadySemaphore '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-ABCL'!


!SAFuture methodsFor: 'initialization'!

initialize
   super initialize.
   isReady _ false.
   isReadySemaphore _ Semaphore forMutualExclusion! !

!SAFuture methodsFor: 'assignment'!

reply: aValue
   isReadySemaphore critical:
      [isReady
          ifTrue: [self error: 'cannot be assigned twice']
          ifFalse: [isReady _ true.
                   super reply: aValue]]! !
```

We may now redefine the external representation of a future in order to be printed as its value, once computed, otherwise it will be printed in the standard way, i.e.: `a SAFuture`.

```
!SAFuture methodsFor: 'printing'!

printOn: aStream
   isReady
      ifTrue: [value printOn: aStream]
      ifFalse: [super printOn: aStream]! !
```

8.1.3 Future Type of Message Passing

The *future* type of message passing expresses a bidirectional communication. The future object will be used as a reply destination to model it. A new future object is created for the communication. It is transmitted to the receiver as the reply destination and immediately returned as the value of the transmission. When the receiver will finish computing the reply to this message, it will return it to the future. Then the future will get assigned, and may be consulted.

In our implementation, we assume that the reply destination is *always* specified as the *last argument* of the message. We define the class ABCLActor as a subclass of class Actor to model these new types of message passing. (We omit the definition of class ABCLActorBehavior and its actor method which are obvious.)

```
Actor subclass: #ABCLActor
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-ABCL'!


!ABCLActor methodsFor: 'message passing'!

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

8.1.4 Equality versus Forwarding Issues

If the expression of a *future* type of message, i.e. a future object, receives some comparing message, we would like the future to forward it to its value. The = method is one of the primitive methods which has been copied into the class EObject in order for the system to manipulate these objects. The future will always reply (false) in place of the value it references. We need to explicitly redirect the equality message to the value of the future:

```
!Future methodsFor: 'comparing'!

= anObject
   ^self value = anObject! !
```

However the problem remains if the future object is the argument and not the receiver of the = message. The = method may be interpreted by the receiver object with various semantics depending if the receiver is primitive. Explicitely checking if the receiver is a future, in order to redirect the message to its value, would force us to drastic changes in some standard methods for primitive classes. Moreover, defining == primitive method causes some problems because it is directly encoded into byte codes and does not follow usual inheritance lookup, as pointed out in [Pascoe 86, page 345]. These forwarding issues are discussed in [Manning 87, pages 41–43].

Because of our preservation goal, our position is to propose a methodological rule in order to avoid most of problems:

Rule 3. The expression of a *future* type message transmission should appear as the receiver of binary operators.

8.2 From Eager to Synchronous Communication

The *now* synchronous type reduces immediately into this eager type plus explicit consultation of (waiting for) the value of the future:

```
!ABCLActor methodsFor: 'message passing'!

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

This reduction of types is somewhat analog to the reduction semantics proposed in [Yonezawa et al. 87, pages 87–88], although the transformation is different. Note that in case of recursive *now* type message passing, there is an immediate deadlock.

8.3 Combination of Message Types into the ABCL/1 Model

It is now time to combine these three types of message passing (*past*, *now* and *future*) into the standard Smalltalk-80 syntax for message passing. We already introduced the asynchronous type of message passing in section 4.4.1. We will extend this technique by using a symbol to specify the two new types of message passing. We will use the `replyTo:` parameter of the message to specify the type of message passing. It will then be replaced by the real reply destination, a new future object.

8.3.1 Implementation of the 3 Types of Message Passing

We will redefine further the `doesNotUnderstand:` method in order to discriminate the types of message passing. At first we have to check the size of the message, and if there are arguments, 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]]]! !
```

8.3.2 Combination of the 3 message Passing Types with the Reply Destination Concept

The *now* and *future* types of message passing are modeled with the creation of some specific reply destination: a future object. This is a first class Smalltalk-80 object, thus the future object may be passed along or delegated to other actors. Note however that the future object is a standard Smalltalk-80 object and not an actor. This does not limit our model in practice, because a future object may reference any kind of object, including an actor.

8.4 Application to the Bank Account Example

We may now propose another way to solve our example of the protected bank account (section 7.7.2).

In this example we suppose that the savings actor belongs to the `ABCLActor` category, in order to recognize *now* type message, because of our modularity choice.

```
!ProtectedBankAccount methodsFor: 'script'!

withdraw: amount replyTo: replyDestination
   | response |
   balance < amount
      ifTrue: [response _ savings withdraw: amount - balance replyTo: #now.
               replyDestination reply: response.
               response = #receipt
                  ifTrue: [self replace:
                              (ProtectedBankAccount balance: 0 savings: savings)]
                  ifFalse: [self ready]]
      ifFalse: [self replace:
                   (ProtectedBankAccount
                       balance: balance - amount savings: savings)]! !
```

Note that the state change of the protected bank account is still expressed through replacement of behavior. This example would be also valid for the initial type of actors with assignment. This shows the flexibility and modularity of our model.

8.5 The ABCL/1 Future Type of Message Passing

8.5.1 Model

We should now precise that the current ABCL/1 *future* type of message passing is not exactly the one we discussed in section 8.1.3. Its semantics is not a single assignment object, but rather a multiple assignment object which behaves as a queue. Such a future object acts as a buffer on which several successive values may be replied. This is analog to Un*x pipes, used as a communication facility between processes. Its combination with message passing was already proposed in [Serpette 85]. The difference is that in ABCL/1 this buffer object is modeled as an object and accessed through message passing rather than procedures. Consequently communication is transparent and futures could be located on remote hosts in some distributed implementation.

8.5.2 Implementation

The class `SharedQueue` is already what we need. We just have to rename messages for assigning and consulting the queue. The class `MAFuture` defines such multiple assignment future objects:

```
SharedQueue subclass: #MAFuture
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-ABCL'!


!MAFuture methodsFor: 'assignment'!

reply: aValue
   self nextPut: aValue! !

!MAFuture methodsFor: 'consultation'!

value
   ^self peek! !
```

The `value` message consults the first element of the queue (without removing it). The `next` method returns (and removes) the first element in the queue. These two methods need to wait if the queue is empty. The `isEmpty` checks if there is an element without waiting for it. This could be useful for an actor not to be "glued" to a future not yet ready and to do some other computation for a while. These two latter methods are equivalent to `next-value` and `ready?` constructs to access a future in ABCL/1.

8.5.3 Message Passing

The ABCL/1 future type of message passing is immediately deduced from section 8.1.3:

```
!ABCLActor methodsFor: 'message passing'!

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

We suppose that the `doesNotUnderstand:` method is extended to recognize some new symbol, e.g. `#ABCLfuture`, to select this new type of message passing. This last type of message passing will be used in the example of section 8.6.

8.5.4 The Identification Problem Revisited

In the semantics of *future* type of message passing described in [Yonezawa et al. 87, pages 88–89], the future object belongs to the sender of the message (the sender is also the creator of the future object). The consultation messages check the origin of the message. We could easily model such a future with restricted access by using an identification technique, like in section 7.7.7. This is left as an exercise for the reader. Note that this makes the consultation messages more verbose, and we think that it is unreasonnable unless there is a higher user level.

8.6 Concurrency and Communication: the Same-Fringe Example

To show the use of this ABCL/1 future type of message, we now define the same-fringe example into our Actalk model. This example was already stated in ABCL/1 [Yonezawa et al. 86, pages 266–267] along a model defined in [Serpette 85]. The goal is to compare two fringes of two trees concurrently.

8.6.1 Model

Three actors are used: two extractors which will extract the fringe of each one of the trees, and one comparator which will compare their elements, one with one. Two classes, respectively `TreeExtractor` and `TreeComparator`, model the behaviors of these actors.

8.6.2 Implementation

The class `TreeComparator` owns two instance variables `extractor1/2` to reference the two extractors, instances of the class `TreeExtractor`. `input1/2` instance variables reference the two future objects to which the elements of the fringe will be assigned by the extractors.

```
ABCLActorBehavior subclass: #TreeComparator
   instanceVariableNames: 'extractor1 extractor2 input1 input2 '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Actors-Extensions-ABCL-Examples'!


!TreeComparator methodsFor: 'initialization'!

initialize
   extractor1 _ TreeExtractor new actor.
   extractor2 _ TreeExtractor new actor! !

!TreeComparator methodsFor: 'script'!

sameFringe: tree1 with: tree2
   input1 _ extractor1 fringe: tree1 replyTo: #ABCLfuture.
   input2 _ extractor2 fringe: tree2 replyTo: #ABCLfuture.
   self compareElements! !

!TreeComparator methodsFor: 'private routine'!

compareElements
   | element1 |
   element1 _ input1 next.
```

```
      element1 = (input2 next)
         ifTrue: [element1 = #EOT
                     ifTrue: [self success]
                     ifFalse: [self compareElements]]
         ifFalse: [self failure]!

   failure
      Print reply: false.
      self initialize!

   success
      Print reply: true! !
   "-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

   TreeComparator class
      instanceVariableNames: ''!


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

   new
      ^super new initialize! !

   !TreeComparator class methodsFor: 'example'!

   example
      "   TreeComparator example   "
      TreeComparator new actor sameFringe: #(1 (2 ((3) 4) 5) (6))
                               with:       #(1 2 3 4 5 6)! !
```

The class `TreeExtractor` models extractors:

```
   ABCLActorBehavior subclass: #TreeExtractor
      instanceVariableNames: 'output '
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Actors-Extensions-ABCL-Examples'!


   !TreeExtractor methodsFor: 'script'!

   fringe: tree replyTo: aReplyDestination
      output _ aReplyDestination.
      self extract: tree.
      output reply: #EOT! !

   !TreeExtractor methodsFor: 'private routine'!

   extract: tree
      Processor yield.
      (tree isKindOf: Array)
         ifTrue: [tree isEmpty
                     ifFalse: [self extract: tree left; extract: tree right]]
         ifFalse: [output reply: tree]! !
```

Note that the `extract:` method specifies an explicit scheduling (`Processor yield`). We cannot use our generic scheduling methods here, because `extract:` method is private and is not received by the actor. This explicit call to the scheduler ensures a pseudo-parallel extraction of the two trees.

Trees are simulated with Arrays. We need to teach them how to behave as trees:

```
!Array methodsFof: 'tree simulation'!

left
   ^self first!

right
   ^self copyFrom: 2 to: self size! !
```

8.6.3 Interruption Issues

This example works fine, but the issue of interrupting the extractors in case of failure is still opened. Currently we cannot stop them, but we may start immediately a new computation.

A first solution would have been to use explicit coroutines, i.e. each extractor will communicate next element to the comparator and some continuation actor which may be resumed in order to get next element, and so on. In this case it is easy to stop the extractors because we explicitly resume them after each element. On the other hand we cannot take advantage of eager computation of the extractors.

A second solution, as proposed in ABCL/1 in [Yonezawa et al. 86, pages 266–267], is to use express mode message passing (some higher priority message passing mode analog to interruptions) to stop them.

A third solution would be to use a *sponsoring* mechanism, allowing to stop sponsoring the computation of the extractors. It is described in [Manning 87].

We intend to study the design of these two latter mechanisms in Actalk, before drawing firmer conclusions about the optimal construct.

9 Distributed Architecture

We will now discuss the extension of our kernel towards some distributed architecture. Our kernel model is compatible with remote objects, based on proxies as defined in [Decouchant 86], [Bennett 87], and [McCullough 87]. Thus we may expect to easily combine both models.

On the other hand all distributed models referenced above are based on *remote procedure call* (RPC) in order to keep compatibility with standard synchronous message passing in Smalltalk-80. We discussed introduction of asynchronous message passing. It is natural to keep this philosophy for a distributed extension in order not to loose the concurrency that we gained.

In case of remote reference to an actor, some special actor, called a *forwarder* and analog to the one studied in section 7.8.4, will represent it. Its behavior is to asynchronously convey the message to the remote actor on the remote machine. The model is in fact a simplification of traditional models for distributed Smalltalk. We are currently designing, implementing and testing such an extension. We intend to keep in mind our goals expressed at the beginning of the paper. We are using an interprocess communication interface which is now built inside the standard Smalltalk-80 image on Sun 4 workstations.

10 Related Work

Related work has already been referenced and partially discussed along the sections. We will now summarize and compare our work to some of the most representative activity in this field.

The merging of a sequential OOP language (called ObjVlisp [Briot and Cointe 87]) and a process layer (called Pive) in [Serpette 85], both based on Lisp, gave us a first account on combination of passive and active objects.

[Bézivin 87] investigates concurrent models and constructs for simulation in Smalltalk-80. This paper also coins the passive/active object duality that we investigated in this 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 the actor paradigm of computation.

The ConcurrentSmalltalk project, [Yokote and Tokoro 86] and [Yokote and Tokoro 87], is very close to ours, notably with the resemblance between `CBox` and `Future` functionalities. It goes further than our project in terms of design. On the other hand they had to change the Smalltalk-80 virtual machine. Our goal is not that extreme, but to study the combination of the two styles of programming without changing the underlying one. Another important difference is about the asynchronous message passing we consider as the fundamental mode, whereas they keep synchronous message passing as the fundation. Asynchronous message passing is a specialization of eager message passing in their model.

The Actra project, [Thomas and Lalonde 85] and [Lalonde et al. 86], is another project close to ours, but whose goals are different. They model actors as a group of standard Smalltalk-80 objects. Standard message passing occurs between objects belonging to a same group, whereas synchronous message passing occurs between several actors which have autonomous activities.

Proposals for distributed Smalltalk architectures have been proposed in [Decouchant 86], [Bennett 87], and [McCullough 87]. Our design of a distributed architecture is closely related to these models, the main difference is about the asynchronous of message passing which simplifies management of remote communication between actors.

[Pascoe 86] proposed a technique of encapsulating objects and its aplications to monitoring and atomicity. We modeled a simplified version of his general model, to simply trap messages sent to an actor.

The Actor model of computation [Hewitt 77] and [Agha 86], and its implementation [Manning 87] have been extensively discussed in this paper. The ABCL/1 model [Yonezawa et al. 86] gave us a more pragmatic model of actors, and models of transactions between them. [OOCP 87] gives a good account of current activity in the field of OOP and concurrency.

11 Further Work

This project is obviously not completed yet and we intend to push further this study of actors through the Smalltalk-80 pedagogical filter. We intend to study the design of a higher level language, analog to Acore, with a compiler generating Actalk kernel code. How to transpose the MVC model in a context of multiple activities needs also deeper study. A prototype implementation of distributed actors in Smalltalk will give us ground to evaluate models and techniques for allocation, migration, load-balancing, garbage collection... These experiments should also be enhanced with a more formal approach to design rules for a safe combination.

12 Conclusion

In this paper we discussed the introduction of autonomous active objects, called actors, into the current Smalltalk-80 model. We proposed a minimal kernel of actors, called Actalk. This kernel was extended in several directions to simulate the Actor model of computation, ABCL/1 types of message passing, and distributed architectures. A methodology for combining traditional Smalltalk-80 programming and actor programming was discussed all along the paper. This study needs now to be pushed further in order to test Actalk in various contexts and state formal rules for safety of combination.

All code that was shown in this paper has been tested in Smalltalk-80 on various machines and images, including Apple's image for MacIntosh, and Parcplace Systems' image for Sun 4 workstation.

Acknowledgements

We would like to thank the past, now and future Smalltalk and Actor groups for making this wonderful reactive dream a reality!

References

[Agha 85] G. Agha, Semantic Considerations in the Actor Paradigm of Concurrent Computation, Seminar on Concurrency, *Lecture Notes in Computer Sciences*, No 197, pages 151-179, Springer-Verlag, 1985.

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

[Bennett 87] J.K. Bennett, The Design and Implementation of Distributed Smalltalk, in [OOPSLA 87], pages 318-330.

[Bézivin 87] J. Bézivin, Some Experiments in Object-Oriented Simulation, in [OOPSLA 87], pages 394-405.

[Bézivin 88] J. Bézivin, Langages à Objets et Programmation Concurrente: quelques Expérimentations avec Smalltalk-80, Actes des Journées Afcet-Groplan Langages et Algorithmes, *Rouen, France, 18-19 November 1987*, Bigre+Globule, No 59, pages 176-187, April 1988.

[Borning and O'Shea 87] A. Borning and T. O'Shea, Deltatalk: an Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language, in [ECOOP 87], pages 1-10.

[Briot and Cointe 87] J.-P. Briot and P. Cointe, A Uniform Model for Object-Oriented Languages Using the Class Abstraction, Tenth International Joint Conference on Artificial Intelligence (IJCAI'87), edited by J. McDermott, Vol. 1, pages 40-43, Milano, Italy, 23-28 August 1987.

[Briot and Yonezawa 87] J.-P. Briot and A. Yonezawa, Inheritance and Synchronization in Concurrent OOP, in [ECOOP 87], pages 32-40.

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

[Decouchant 86] D. Decouchant, Design of a Distributed Object Manager for the Smalltalk-80 System, in [OOPSLA 86], pages 444-452.

[Deutsch 81] L.P. Deutsch, Control Structures in Smalltalk-80, *Special Issue on Smalltalk-80*, Byte, Vol. 6 No 8, 1981.

[ECOOP 87] ECOOP'87, European Conference on Object-Oriented Programming, *Paris, France, 15-17 June 1987*, edited by J. Bézivin, J.-M. Hullot, P. Cointe and H. Lieberman, *Lecture Notes in Computer Sciences*, No 276, Springer-Verlag, 1987.

[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.

[Halstead 84] R.H. Halstead, Implementation of Multilisp: Lisp on a Multiprocessor, Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, pages 9-17, Austin TX, USA, 6-8 August 1984.

[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, Proceedings of the National Conference on Artificial Intelligence, AAAI, pages 234-237, August 1982.

[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, *essay*, ParcPlace Systems, Palo Alto CA, USA, 1988.

[Lalonde et al. 86] W.R. Lalonde, D.A. Thomas and J.R. Pugh, Actors in a Smalltalk Multiprocessor: a Case for Limited Parallelism, Techical Report SCS-TR-91, School of Computer Science, Carleton University, Ottawa, Canada, May 1986.

[Lex 85] A. Lex, POOL-T User Manual, Doc. No 0104, ESPRIT Project 415, Philips Research Lab., Eindhoven, Netherlands, 10 September 1985.

[Lieberman 86] H. Lieberman, Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, in [OOPSLA 86], pages 214-223.

[Lieberman 87] H. Lieberman, Concurrent Object-Oriented programming in Act 1, in [OOCP 87], pages 9-36.

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

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

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

[OOPSLA 86] OOPSLA'86, Object-Oriented Programming Systems, Languages and Applications, *Portland OR, USA, 29 September - 2 October 1986*, edited by N. Meyrowitz, Special Issue of SIGPLAN Notices, Vol. 21, No 11, November 1986.

[OOPSLA 87] OOPSLA'87, Object-Oriented Programming Systems, Languages and Applications, *Orlando FL, USA, 4-8 October 87*, edited by N. Meyrowitz, Special Issue of SIGPLAN Notices, Vol. 22, No 12, December 1987.

[Pascoe 86] G.A. Pascoe, in [OOPSLA'86], pages 341-346.

[Serpette 85] B.P. Serpette, Contextes, Processus, Objets, Séquenceurs: Formes, *Thèse de 3ème cycle*, Technical Report 85-5, IRCAM - LITP, Université Paris VI, Paris, France, March 1985.

[Thomas and Lalonde 85] D.A. Thomas and W.R. Lalonde, Actra: The Design of an Industrial Fifth Generation Smalltalk System, IEEE COMPINT'85, 1985.

[Ungar and Smith 87] D. Ungar and R.B. Smith, Self: the Power of Simplicity, in [OOPSLA 87], pages 227-242.

[Wegner 87] P. Wegner, Dimensions of Object-Based Language Design, in [OOPSLA'87], pages 168-182.

[Yokote and Tokoro 86] Y. Yokote and M. Tokoro, The Design and Implementation of ConcurrentSmalltalk, in [OOPSLA 86], pages 258-268.

[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.

[Yonezawa et al. 87] A. Yonezawa, E. Shibayama, T. Takada and Y. Honda, Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, in [OOCP 87], pages 55-89.

Contents