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

Jean-Pierre BRIOT

Equipe Mixte LITP - RankXeroxFrance,
Université Paris VI,
4 place Jussieu, 75005 Paris, France
`litp!briot@inria.inria.fr.uucp`

Abstract

In this paper we describe an implementation of actors in Smalltalk-80, named Actalk. This attempt is designed as a minimal extension preserving the Smalltalk-80 language.

Actors are active and autonomous objects, as opposed to standard passive Smalltalk-80 objects. An actor is built from a standard Smalltalk-80 object by associating a process with it and by serializing the messages it could receive into a queue.

We will study the cohabitation and synergy between the two models of computation: transfer of active messages (message and thread of activity) between passive objects, and exchange of passive messages between active objects. We propose a sketch of methodology in order to have a safe combination between these two programming paradigms.

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 distributed architectures. All these examples are constructed as simple extensions (by using inheritance) of our kernel model.

Keywords

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

1 Introduction

The Smalltalk-80 language is the best example of Object-Oriented Programming (OOP) languages. 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. Semaphores are provided in Smalltalk-80 as tools for synchronization. But the nature of processes and semaphores, used to synchronize processes and their accesses to shared data, remain totally orthogonal to the nature of objects and messages. As a consequence combination of these two styles is uneasy as well as unsafe.

There have been attempts to modify Smalltalk-80 in order to unify objects and processes. The main proposals are Actra [Lalonde et al. 86] and ConcurrentSmalltalk [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, but changed the Smalltalk-80 model of computation and virtual machine.

Our approach is somewhat different because we don't intend to make 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 87]. 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.

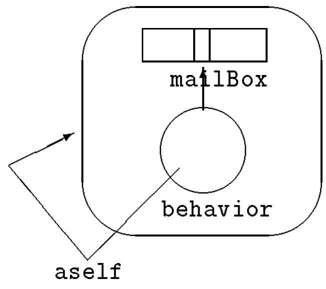

Figure 1: Implementation of an actor.

We expect our actor kernel, named Actalk, to be a basic platform to express, classify and test various computation and execution models of actor languages. We will also use it and its extensions in graduate courses to teach actors to usual OOP programmers.

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 leads to study some fundamental issue of combining passive and active objects. In Actalk, we sketch some methodological rules for a safe combination.

2 From Passive Objects to Active and Autonomous Actors

Current Smalltalk-80 objects are passive, because they undergo the activation request of the sender. A first step is to monitor the objects, i.e. to ensure the execution of at most one message at a time. The intuitive technique 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].

In order to make the object more active, a serializer [Hewitt and Atkinson 79] will encapsulate the object and serialize the incoming messages into a message queue. The activity of the object is now internal and no more external (depending on message invocation). By setting the synchronization control onto the message arrival and by buffering the incoming messages in the order of their arrival, the object may decide *when* and *how* it will accept and perform the messages. The object gained *activity* and *autonomy*, and we call it an *actor*. The initial model of active messages between passive objects has been extended towards a new model of passive messages between active objects.

Message passing in Smalltalk-80 is synchronous. We introduce asynchronous message passing between actors in order to increase concurrency, because the sender does not have to wait for the receiver to be ready to complete its transmission.

3 Implementing Actors in Smalltalk-80

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

The class `ActorBehavior` models the behavior of the actor. Some background and infinite process will be started when creating the actor to implement the autonomy of its behavior. The behavior will keep dequeueing the next message in the mailbox and perform it. Figure 1 gives a representation of the implementation of an actor in Actalk.

In order to integrate harmoniously our new message passing model between actors with the current message passing syntax of Smalltalk-80, we use some technique, initiated in [Pascoe 86]. This technique allows to model syntactic changes through redefinition of error semantics. 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. 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.

The instance variable `aself`, defined in class `Actorbehavior`, denotes the (reflexive) knowledge about the actor itself. 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.

Any Smalltalk-80 object may be used as a behavior of a new actor if its class is a subclass of `ActorBehavior`. The user may define classes of behaviors almost as he would define classes of objects. The main difference is that methods won't return implicitly a result to the caller, because of asynchronous communication.

4 Symbiosis between Objects and Actors

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 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. On the other hand, a serialized actor may change its state. Therefore it should process messages one at a time.

Standard Smalltalk-80 objects may also be considered as unserialized actors, from the actor viewpoint. An object which has a constant state (immutable) may be considered as a "real (and safe) unserialized actor". An object which could change its state is an "unprotected unserialized actor". Unfortunately all non primitive Smalltalk-80 objects are mutable objects. We need to be careful with these objects in order to combine them with actors. Such objects should not be shared by several actors. However some of them, for instance classes, may be considered as constant objects, with some assumptions about staticity of their hierarchy and environment. This will lead us to methodological rules rather than formal rules which would be too restrictive.

An interesting challenge is to study how the standard MVC (Model View Controller) paradigm for interface design may be combined safely with our actor kernel. The basic extension of MVC we designed (the class `ActorView`) allows to control representations (views) of actors asynchronously with their activities in a safe manner.

5 Extensions of the Model

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 will augment our minimal actor kernel by building variations and higher level constructs while keeping our full compatibility with Smalltalk-80. The inheritance concept of Smalltalk-80 allows us to classify and specialize such extensions and variations in a modular way.

5.1 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 the next incoming message. A behavior will now accept only one message, a new behavior being specified for next message.

The class `AghaActorBehavior` is defined as a subclass of class `ActorBehavior` with redefinition of only two methods. The process associated to each behavior now accepts only one message in the message queue. We introduce a new method to specify the replacement behavior.

We immediately model successively the concepts of *insensitivity*, *future* (eager computation), and *forwarder*, as further extensions of this first extension.

5.2 the ABCL/1 Types of Message Passing

The ABCL/1 programming language [Yonezawa et al. 86], although based on the actor philosophy, chose a more pragmatic approach based on hybrid computation and higher granularity of actors. ABCL/1 proposes three distinct types of communication protocols: *past*, *now* and *future* respectively for asynchronous, synchronous and eager types of communication.

The two latter types of communication are based on a concept of future object modeled as a passive Smalltalk-80 object. The future object is synchronized with the completion (and value) of the computation it represents. This model is analog to the `CBox` object in ConcurrentSmalltalk.

The *future* type of message passing is easily modeled with asynchronous message passing and a future object. A new future object will be used as reply destination of the asynchronous message. Consultation of the expression will automatically synchronize on the assignment of the future. A full model of current ABCL/1 future type of transmission is even more easily constructed from a multiple assignment object, modeled by the standard class `SharedQueue`.

The *now* type of message passing is simply reduced to the equivalent *future* type of transmission and the explicit consultation (and synchronization) of the future object.

5.3 Distributed Architecture

Our kernel model is compatible with remote objects, based on proxies as defined in [Decouchant 86]. Thus we expect to easily combine both models. On the other hand all these distributed models 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. Moreover it simplifies the modeling of remote communications.

6 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 sketched, with application to the MVC model. This study needs now to be pushed further in order to test Actalk in various contexts, for instance modeling new actor models or constructs, designing some higher level language compiled into Actalk (in a way analog to the Acore/Pract compiler [Manning 87]), specializing the MVC-based Smalltalk-80 programming environment for Actalk, and completing a distributed implementation designed along our model. We also need to state more formal rules for safety of combination.

The complete version of this paper [Briot 88] includes all code for Actalk kernel, extensions and examples, plus an extensive discussion (about 50 pages).

References

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

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

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

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

[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.

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

[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.

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

[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.