Building an Unified Programming Environment for Object-Oriented Concurrent Languages

Jean-Pierre BRIOT   Loïc LESCAUDRON

Equipe Mixte LITP - Rank Xerox France,
Université Pierre et Marie Curie (Paris VI),
4 place Jussieu, 75005 Paris, France
`briot,loic@litp.ibp.fr`

Abstract

This paper describes a programming environment for modeling and visualizing object-oriented concurrent languages. This system focuses on *actor-based* languages and is named *Actalk*, which stands for *act*ors in Small*talk*-80. Actalk is based on some minimal kernel introducing actors into Smalltalk-80. The Actalk kernel extends passive and sequential objects activated by synchronous message passing into active and concurrent actors communicating by asynchronous message passing. It is composed of only two Smalltalk-80 classes which could be extended (subclassed) to simulate current object-oriented concurrent languages or design new ones. We extend the standard Smalltalk-80 programming environment in order to customize it to the concurrency specificities of actors. The current system allows visualization and interaction with concurrent objects.

In this paper, the motivations and the goals which led to design the Actalk system are first discussed. Then the structure and implementation of the kernel of Actalk is described. We then relate how we extend the standard Smalltalk-80 programming environment. We focus on two components: the graphic interface with actors, based on the standard Smalltalk-80 MVC interface methodology; and the preemptive scheduler which allows a satisfying pseudo-parallel execution of actors. Related and further work is summarized before concluding this paper.

Keywords

object, concurrent, actor, programming environment, Smalltalk-80, kernel, modular, visualization, MVC, scheduler.

1 Introduction

This paper is concerned with the design and experiment of Object-Oriented Concurrent Programming (OOCP) languages [8]. We believe that the so-called *actor* family of languages is among the most promising approaches. The original actor metaphor [1] gave rise to numerous OOCP proposals. All these *actor*, or *actor-based* languages share the foundation of active and concurrent objects communicating asynchronously.

Our experience of designing and experimenting with OOCP languages led us to the following conclusions. We remarked that, from the user point of view, it is not always easy to analyze and compare various computation models and programming languages. Comparing various semantics needs to abstract from various syntax and implementation supports. Moreover some of the prototypes are not usable outside of some labs or very specific machines.

From the implementor point of view, experience of designing actor languages shows that usually too many prototype implementations are thrown away before clarifying the design. Making a prototype modular (and improving its efficiency) is usually achieved *afterwards* and not in the early implementation effort. The reuse of previous prototypes or even other ones would greatly improve the design task.

Last but not the least, such prototype systems miss usually a programming environment which is more than a need for concurrent systems.

Notice that these goals of modularity and environments are endorsed by Object-Oriented Programming (OOP). This led us to design some modular system/environment for actor languages based on OOP, in order to satisfy these needs.

2 Overview of the Project

2.1 Goals and Design Decisions

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

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

2. *minimality and extensibility*
   We wanted some minimal kernel expressing the most general semantics of actor languages, and to further extend it in order to simulate various existing languages and to design new ones. Therefore we chose a minimal architecture in the spirit of ObjVlisp [3] which is based on a minimal kernel, and then further uniformly extended. We use inheritance to classify the various actor models.

3. *an integrated environment*
   We didn't want to restrict our system to some semantic model and raw implementation, but also to provide a full environment for pragmatic experiment with actor-oriented programming. Therefore we chose Smalltalk-80 (in the following, we will abbreviate to *ST80*) because it is the most achieved and flexible OOP system with a fully integrated programming environment. Because of the integration of our actors into the ST80 model and environment, the designer of actor languages could automatically use the standard ST80 programming environment. Furthermore, this environment is extended in order to support the specific concurrency aspects of actors.

By choosing ST80 we could also provide a minimal implementation of the system, because all entities needed to build actors: objects, classes and messages, and to express concurrency: processes and semaphores, are provided by ST80. The resulting system is named *Actalk* (which stands for *act*ors in Small*talk*-80). Actalk is an integrated environment, embedded into ST80, used as a testbed to classify, design and experiment with actor languages. Actalk does not change the underlying ST80 system, but rather merges into it.

2.2 Current Status of the Project

The Actalk system started on a pedagogical basis, as a way to model in a minimal way various kinds of actor-based languages. (The kernel of the system and a sketch of its architecture will be described in sections 3.1 and 3.2.) Then the programming environment has been extended in order to visualize and interactively monitor the activity of concurrent objects. (This part will be described in section 4.) Actalk has been used within the Oks system in order to compile production rules into concurrent daemons [9]. It also has been used in order to describe various Distributed AI systems, by extending the concept of actor into the concept of *agent*. A current ongoing project is now to validate the system on parallel architectures, by porting Actalk on a transputer-based multiprocessor machine, with the support of the Esprit Parallel Computing Action. (These projects won't be addressed in this paper.)

3 Introducing Actors into Smalltalk-80

We will now shortly introduce the model of actors, and how we embed them into ST80.

In ST80, as in standard OOP, objects are activated by message passing. Objects are passive because they undergo the request of activation by the sender of the message. They have no activity of their own. Sending a message represents the transfer of activity from one object to another one, before going back to the sender whose activation has been suspended.

On the other hand, a concurrent and active object owns its internal activity, and will no more depend on external activations through message passing. It can decide on its own when to perform a message and has the power to complete it. Consequently the sender does not need any more to suspend its activity to transfer it to the receiver. If no reply is needed, the sender may resume its computation immediately after sending the message. Thus basic message passing between active objects is unidirectional and asynchronous in order to avoid unnecessary waiting. The receiver may process it any time after the delivery, therefore messages should be buffered in a *mailbox* associated to the receiver, before being processed.

We will call such active and autonomous objects *actors*. An actor will encapsulate a standard ST80 object to make it being active and to change the semantics of message passing. An actor is composed of a mailbox where the incoming messages will be buffered, and the active object that will process them, which we will call its *behavior*.

3.1 Implementation of the Actalk Kernel

3.1.1 Definition of the Minimal Kernel

The *Actalk kernel* defines the most general (and minimal) semantics for actor-based languages. The class `Actor` simply defines an actor through its two components: the mailbox or queue which will contain the incoming messages, and the behavior which will process them. The class `ActorBehavior` defines the standard way the behavior (of an actor) processes messages, that is processing them one at a time[1].

3.1.2 The class Actor

The class `Actor` implements actors as encapsulators built around standard ST80 objects. Its superclass is the class `MinimalObject`, which is equivalent to class `Object`, but with a minimal set of methods. (`MinimalObject` is used for the implementation of asynchronous message passing and is described in [2].) `Actor` specifies two instance variables:

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

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

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

!Actor methodsFor: 'initialization'!

initialize
   mailBox _ SharedQueue new!

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

!Actor methodsFor: 'access to instance variables'!

mailBox
   ^mailBox! !

!Actor methodsFor: 'message passing'!

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

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

behavior: aBehavior
   ^self new initializeBehavior: aBehavior!

new
   ^super new initialize! !
```

The `behavior:` class method creates an actor and initializes its behavior (`initializeBehavior:` instance method). The `new` class method is redefined to initialize the mailbox of the actor (`initialize` instance method). The `asynchronousSend:` instance method implements asynchronous message passing to an actor by enqueueing the corresponding message onto its mailbox.

[1] Actually, this is not true in some actor-based computation models, but we made this decision for the kernel in order to reach the best integration within ST80. We may then easily model them as extensions of this kernel (see [2]).

3.1.3 The class ActorBehavior

The class `ActorBehavior` implements the general behavior of an actor. Users will describe behaviors of actors by defining classes of behaviors as subclasses of `ActorBehavior`. The class `ActorBehavior` defines one instance variable: `aself`, which denotes the actor currently using this behavior. `aself` allows an actor to send (asynchronous) recursive messages to itself. A background process will be created with the actor to implement the activity and autonomy of the behavior. This process is infinite. As stated in the definition of the kernel, the behavior will keep dequeueing the next message from the mailbox and perform it.

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

!ActorBehavior methodsFor: 'initialization'!

initializeAself: anActor
   aself _ anActor.
   self setProcess!

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

!ActorBehavior methodsFor: 'message acceptance'!

acceptNextMessage
   self acceptMessage: aself mailBox next!

acceptMessage: aMessage
   ^self perform: aMessage selector withArguments: aMessage arguments! !

!ActorBehavior methodsFor: 'actor creation'!

actor
   ^Actor behavior: self! !
```

The `initializeAself:` instance method initializes the self-reference (`aself`) of the actor and starts the activity of the behavior (`setProcess` method). The `acceptNextMessage` and `acceptMessage:` methods accept and perform the next message in the mailbox. The process is suspended until the mailbox is not empty by the semaphore which synchronizes reading data from the shared queue [4, page 262]. The `actor` method creates an actor from a passive object which is used as its behavior. (In order to create an actor, one just needs to write `actor` after the definition of its behavior, for instance: `Counter new actor` in order to create an actor counter.) Note that the Actalk kernel is stated in just 2 classes and 11 small methods.

3.1.4 Transparent Asynchronous Message Passing

In order to gracefully integrate the message passing model between actors (`asynchronousSend:` method) with the standard message passing syntax of ST80, we use some technique initiated in ST80. A message is sent to an object which is assumed not to recognize the selector of the message. This error is trapped by redefinition of the standard error method (`doesNotUnderstand:`) which then executes some specific strategy. Local redefinition of the `doesNotUnderstand:` method ensures the locality of changes. Thus we redefine it in the class `Actor` in order to redefine the semantics of message passing *locally* to actors:

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

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

A message sent to an actor will get an asynchronous semantics whereas the same message sent to some standard ST80 object will keep the standard synchronous semantics. (Note that the value returned by asynchronous transmission is not significant.)

When specifying the behavior of an actor which sends recursive messages, the programmer may choose between *pseudo*-variables `aself` or `self`. Sending to `aself` uses Actalk asynchronous message passing. The message is put in the mailbox of the actor, and the behavior will process it later. Sending to `self` relies on standard ST80 synchronous message passing. This implies immediate and "internal" processing of the message by the behavior.

Figure 1 gives a representation of the implementation of an actor.

Figure 1: Implementation of an actor in Actalk.

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

3.2 Architecture of Actalk

The Actalk kernel has been easily extended to simulate some of the most representative OOCP computation models and programming languages based on the actor concept. These extensions use inheritance to refine the semantics of the Actalk kernel. For example, we simulate Agha's Actor computation model through the class `AghaActorBehavior` as a subclass of `ActorBehavior`. We also simulate Yonezawa's Abcl/1 [11] language constructs through the class `AbclActor` subclass of `Actor`. This shows the merits of modularity for our kernel. Because the kernel and its extensions are related by inheritance, one could easily compare them. Inheritance helps not only to classify various actor models, but also to clearly relate and to reuse their various implementations.

Figure 2 shows the hierarchy of the classes of the Actalk kernel, augmented by some of its current extensions, and the example of the counter. Note that the class `ActorBehavior` and all its extensions are *abstract classes*, i.e., don't have instances. These classes only give the semantics of how to compute messages, not the semantics of messages themselves. Only the application classes, like `Counter`, will generate actual instances, the behaviors. Actors will be instances of class `Actor` and its subclasses.

4 Extending the Standard Smalltalk-80 Programming Environment towards Actors

Because Actalk actors are well integrated into the ST80 system, they automatically benefit from the standard ST80 programming environment, which is a great help when designing languages and applications. A further goal is to extend this standard environment to support the specificity of actors. A first step is to extend the standard ST80 MVC[2] paradigm to take this specificity into account. The second step is to control the way ST80 schedules actors (to have a real scheduler instead of the non preemptive standard scheduler).

[2] The *M*odel *V*iew *C*ontroller (MVC in short) is the paradigm for developping interfaces in ST80. Different classes take over the operations related to the application domain (the *model*), the display of the application's state (the *view*) and the user interation with the model and the view (the *controller*) [5].

Another challenge is to extend the current ST80 debugger towards a specific debugger for actors (likewise for the compiler). A first prototype has already been implemented but, it's out of scope for this paper.

4.1 Actalk MVC

4.1.1 Lacks of the Standard MVC Model

The ST80 language is probably so popular because of its environment. Browser, inspector, debugger are indeed very powerful and easy to use. Those utilities are built along the MVC methodology. But, as for every powerful system, it has some limitations/drawbacks. The following two points are of main importance in respect to our goal of reusing MVC for actors:

- *complexity* The MVC turns out to be non trivial to apply when implementing user interfaces, especially because it implies a good knowledge of the standard MVC classes (nearly 10),
- *sequentiality* Some important assumption of the MVC is the sequential nature of ST80 computation. The MVC assumes that while the user interacts with a model through the controller, the model is inactive. Because actors are active entities, this assumption does not stand anymore. For example, the user can *move* the view, via the controller, while the model wants to *update* it, via the MVC dependency system. Running standard MVC on actors could lead to such conflicts on the views. Therefore we have to extend MVC to support actor specificities.

4.1.2 A New MVC Dedicated to Actors

As explained above, a view on some actor may be accessed concurrently by the model and by the controller. Views become critical shared resources. The concept of *semaphore* offers a trivial and trusted way for preventing concurrent accesses to a critical resource. But, the limitations are:

- a requestor is suspended as far as the resource is not available.
- there's no preemption. So, no kind of priority may be supported.

In case of MVC over actors, we don't want such a behavior because:

- the controller should have a higher priority since it represents the user. In other words, we don't want the user to wait for a resource (namely a view). One should be able to take control at any time, even if an actor model attempts to update its view.
- the actor shouldn't be suspended while the view is locked, in order to keep a high level of concurrency.

A further and more drastic step would be to represent views as actors (in some event-driven fashion including priority events). Actually we are currently investigating such methodology and its implementation within ST80/Actalk.

Meanwhile, in order to quickly get some useable system well integrated within standard MVC, we chose the following strategy. We give higher priority to control over update. This means that the view won't be updated while the user interacts with the controller. We also give higher priority to computation over update. This means that the actor will keep computing concurrently during interaction, and that consequently updates will be temporarily disabled. At the end of the interaction, the view will be updated only once (the updates during inhibition are not bufferized).

4.1.3 The VC Generator

Despite this extension, the resulting *Actalk-MVC* still shares with standard MVC the *complexity* weakness. The user needs to master programming of the view and controller components. Consequently, as for interface generators, we propose semi-automatic generation of the interface (namely the view and controller components, thus we call it the *VC-generator*).

One can expect many different ways of visualizing actors. We consider two complementary aspects as basic to visualize and control actors:

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

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

Figure 3: An example of generated view.

Figure 3 shows an example of interface automatically generated for the *prime number* example, which will be described in section 4.1.4.

The VC generator offers many others features and especially the following:

- *interactive description* The user can choose, at generate time, whose instance variables will be visualized. This happens to be very useful to visualize some specific instance variables at debugging time, and to abandon them at execution time.
  Next step will include goodies to create graphical views. For example the user will be asked interactively about how to visualize variables (for example, by using a gauge).
- *modularity* The generator takes the hierarchy into account. When defining a hierarchy of models, their respective views and controllers automatically match this hierarchy.
- *genericity* The generator is not limited to some specific actor model. For example, this generator supports Agha's Actor computation model [1]. If the behavior is replaced by a structurally different one (e.g., whose instance variables differ), the view (and the controller) change accordingly.
- *debugging tools* As can be seen in Figure 3, one can control the way an actor computes messages (step by step, sleep...). Those tools are well suited to spy and monitor the activity of actors.

4.1.4 The Prime Number Generator Example

In this example, we model generation of prime numbers through an ordered chain of filters (sieves), one filter for each prime number already found. Every integer until some limit will be checked through this chain of filters. Each filter checks if the incoming integer is a multiple of the prime number it models. If so, the integer is rejected, otherwise it will be checked by (sent to) next prime filter. If the filter is the last of the chain, this means that a new prime number has been discovered, consequently a new filter is created for this new prime number and linked at the end of the chain. In addition, we need a generator which will send all integers until some maximum to the first (and initial) filter of the chain. Following is the key method for filtering (`n` is the prime number modeled by the filter, and `next` is the next filter in the chain):

```
filter: i
   i \\ n = 0
      ifFalse: [next isNil
                  ifTrue: [next _ (self class new n: i) actor]
                  ifFalse: [next filter: i]]
```

A display of computation is shown in Figure 4.

Figure 4: Generation of prime numbers.

4.2 Scheduling Actors

4.2.1 The ST80 Scheduler

The standard ST80 scheduler is based on an extended[3] coroutine model. This means that the ST80 scheduler is non-preemptive. A process keeps computing as long as it does not schedule explicitly. This forces the user to include explicit scheduling (the expression `Processor yield`) inside the methods describing behaviors of actors. This is of course too much low-level constraint.

4.2.2 First Step towards a Real Scheduler

In order to give a more modular and abstract level to specify scheduling, we first proposed a set of generic event-driven methods. Each is associated to some event (e.g., receive a message, compute a message...). By default the behavior of these methods is empty. The user may redefine them in applications, by calling `Processor yield` to control scheduling.

But such scheduling may occur only if events occur. If some actor keeps computing without any sending/receiving of message, there won't be any scheduling.

4.2.3 The Actalk Scheduler

Using event-driven generic methods to simulate scheduling appears not to be satisfying enough. So we designed a new scheduler in ST80 in order to offer a satisfying pseudo-parallel execution of actors.

Our requirement was to leave the standard ST80 scheduler unchanged. The overall idea is to set a specific sleeping process which periodically awakes in order to schedule the currently active actor process (see Figure 5). The Actalk scheduler has two components:

- *Actalk scheduler list*: the list of active Actalk processes (i.e., associated to actors),
- *Actalk scheduler process*: a standard ST80 process whose task is to periodically take next process in the Actalk scheduler list[4], and to set it into the standard ST80 scheduler (by sending message `resume` to the process).

The Actalk scheduler process has a high priority in order to take immediate control when being awaken. Its sleeping time (by using delays) may be parametrized by Actalk processes. Thus one can give higher delay to specific actors in order to assign them a higher "priority" (more CPU time).

Like the Actalk-MVC, the Actalk scheduler is generic. Actors which are defined along Agha's Actor computation model may have several activities concurrently. In that case, the CPU time allocated to such an actor is shared among its activities. Also note that Actalk-MVC and Actalk scheduler are modular, thus they may be used independently or together.

[3] As opposed to standard coroutine model, the scheduler implicitly chooses the next process to schedule.

[4] Before that, the current Actalk process is removed from the standard ST80 scheduler (by sending message `suspend`).

Figure 5: The Actalk scheduler behavior.

4.2.4 Visualizing the Scheduler

In order to show how scheduling is done between actors, we designed a graphic tool. It is based on *Pie menus* described in [6], and Figure 6 gives a sample of its use.

5 Related and Further Work

Friend projects to ours are ConcurrentSmalltalk [10] and Actra [7]. They went further than Actalk in terms of integrating actors into ST80. But to achieve it, they had to slightly change both the ST80 virtual machine and the semantics of original actors, whereas our objective was to extend the underlying environment and to simulate various actor systems into it.

Besides the results and projects described in section 2.2, other prospective experiments with the Actalk platform include: modeling processor communication protocols, and modeling strategies for allocation of actors and tasks. One of our current activities is porting the system onto parallel hardware with the support of the Esprit Parallel Computing Action.

Figure 6: Visualizing the Actalk scheduler.

6 Conclusion

In this paper we described a system, named Actalk, based on ST80 which provides an environment to model, classify and experiment with various actor-based languages. The kernel of Actalk introduces actors into the current ST80 system. Its implementation was completely described. The extension of the standard ST80 programming environment towards actors was also discussed, by focusing on how we extended standard ST80 interface and scheduler.

We thank Chork-Huat Ky, Marc Le Bris and Gilles Moussion for taking part in the design of the first prototypes of Actalk-MVC and scheduler.

References

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

[2] Briot J.-P., Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment, *European Conference on Object-Oriented Programming (ECOOP'89)*, *British Computer Society Workshop Series*, Cambridge University Press, pages 109-129, 1989.

[3] Cointe P., Metaclasses are First Class: the ObjVlisp Model, Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Special Issue of SIGPLAN Notices, ACM, Vol. 22, No 12, pages 156-167, December 1987.

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

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

[6] Lalonde W. and Pugh J., Pie Menus, Journal of Object-Oriented Programming, Vol. 2, No 1, May-June 1989.

[7] Lalonde W.R., Thomas D.A. and Pugh J.R., 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.

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

[9] Voyer R., Reflex Compilation: a New Efficient Implementation Model for Systems that Use variables and Work on a Forward Chaining Basis, Proceedings of the Fourth International Symposium on Computer and Information Sciences, Vol. 2, pages 1135-1145, 1989.

[10] Yokote Y. and Tokoro M., Experience and Evolution of ConcurrentSmalltalk, Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Special Issue of SIGPLAN Notices, ACM, Vol. 22, No 12, pages 406-415, December 1987.

[11] Yonezawa A., ABCL: An Object-Oriented Concurrent System, *Series in Computer Science*, MIT Press, Cambridge MA, USA, 1990.