Prototyping Programming Environments for Object-Oriented Concurrent Languages: a Smalltalk-Based Experience

Loïc LESCAUDRON* Jean-Pierre BRIOT Malik BOUABSA*

LITP,

(*Equipe Mixte RXF-LITP),

Institut Blaise Pascal,

4 place Jussieu, 75252 Paris Cedex 05, France

`loic@rxf.ibp.fr briot@litp.ibp.fr bouabsa@rxf.ibp.fr`

in Proceedings of the 5th Conference on the Technology of Object-Oriented Languages and Systems (Tools Usa'91), pages 449–462, Prentice-Hall, 1991.

Abstract

This paper discusses the prototyping of programming environments for object-oriented concurrent programming languages. We advocate designing new prototypes starting from existing sophisticated programming environments that we can find in standard (sequential) object-oriented programming languages (namely Smalltalk-80[1]).

Our starting point is a testbed for object-oriented concurrent languages based on Smalltalk-80. This system (namely Actalk) focuses on *actor*-based languages and gives a good integration of *actors* into the Smalltalk-80 system. We naturally came to reuse the standard Smalltalk-80 programming environment tools and to customize them to the specificities of actors.

In this paper, we describe why and how we customize the Model View Controller architecture to provide user-interfaces on actors, and the scheduler to simulate concurrent execution of actors. Other tools such as stepper and chronogram are also discussed. We then summarize our experience in building and using this prototype programming environment to prospect future programming environments for object-based concurrent systems.

Keywords

object, concurrency, actor, programming environment, reusability, Smalltalk-80, Actalk, visualization, control, MVC, scheduler, stepper.

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

1 Introduction

This paper is concerned with the construction of programming environments for Object-Oriented Concurrent Programming (OOCP) languages [OOCP 87]. It is obvious that because of the multiple, complex, and non deterministic nature of concurrent programming, sophisticated programming environments to describe, visualize, monitor, and debug such programs are not a feature but a necessity.

Sophisticated programming environments for standard Object-Oriented Programming (OOP) are already available (for instance Smalltalk-80). OOP sequential computation models are rather well understood and stable. This is not yet the case for OOCP, concurrent programming computation models being still under evolution. Consequently it makes more sense for us to *prototype* programming environments for OOCP which should be modular and flexible enough for further evolution.

Therefore we propose (following our argumentation in the proceedings of a previous edition of the Conference [Briot 90]) to prototype OOCP environments by extending already existing sophisticated programming environments for sequential OOP (namely Smalltalk-80). This allows fast prototyping, good reusability, and modular and flexible environments, as opposed to building a new programming environment from scratch.

In this paper we review our experience of customizing the Smalltalk-80 programming environment for concurrent objects (actors). This work focuses on the Smalltalk-80 system, but we firmly believe that our experience and design of prototype tools is general enough to be useful for other systems.

2 Actalk: an Overview

Our project of prototyping programming environments for OOCP started naturally triggered by the Actalk project. Actalk is a testbed for modeling, classifying and experimenting with OOCP languages.

2.1 Goals and Architecture

Actalk is based on Smalltalk-80 (referred as ST80 in the following). Actalk introduces active objects communicating through asynchronous message passing in ST80. We call them *actors* and Actalk stands for *act*ors in Small*talk*-80.

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

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

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

Actalk has been further described in [Briot 89].

2.2 Programming Environment

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. We further extended this standard environment to support the specificity of actors. Tools such as a user-interface generator for actors, based on the ST80 MVC paradigm (in section 3), a generic scheduler (in section 4), a chronogram... (in section 4.3.5) have been developed.

2.2.1 Applications

Actalk has been used within the Oks system in order to compile production rules into concurrent demons [Voyer 89]. Actalk has been also used in order to describe various Distributed AI systems, by extending the concept of actor into the concept of *agent*. Platforms for coarse grain cognitive agents [Bouron et al. 90], as well as for fine grain reactive agents [Drogoul et al. 91], have been successfully developed on top of Actalk by Jacques Ferber's team at Laforia, Institut Blaise Pascal.

2.2.2 An Example: Generating Prime Numbers

In order to give a taste of the development and visualization of programs in the Actalk environment, we shortly describe some algorithm and show a display of its execution. 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.

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

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

!PrimeFilter methodsFor: 'script'!

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

Figure 1: Example: generating prime numbers.

```
!PrimeFilter class methodsFor: 'example'!

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

A snapshot of the corresponding computation, introducing some of the tools of the Actalk programming environment, is shown in Figure 1.

Figure 2: Illustration of interferences: display of Life's game.

3 Actalk MVC

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

the *model* which holds the applications state and behavior,

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

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

Consequently we used the MVC as the basis to develop our programming environment for concurrent applications.

3.1 Lacks of the Standard MVC Model

The MVC paradigm provides, at the abstract level, convenient methodology for modular interface construction. However MVC is not directly usable to visualize concurrent objects because it makes several assumptions about the nature of objects to be visualized (they are supposed to be passive objects, i.e., no activity on their own). The use of the MVC interface may become frustrating and error generating for the end-user because of the interferences that non-passive objects (actors) generate on the screen. In the snapshot shown in Figure 2, the user is currently interacting with the system browser, while some Actalk simulation of Conway's Life's game is concurrently displaying successive generations.

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

- Models are supposed to be passive. In the standard MVC framework, the state modification of a model is controlled by the user (via the controller part of the interface). The view (after the model's change notification occurs) can display information directly onto the screen because this view is automatically one of the sub-components of the active window (which is always displayed in front of the screen).

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

  With the existence of concurrent activities, the screen must be considered as a shared resource for actors. The displaying of informations in a window must take into account that the window may be inactive and overlapped by other windows. Actors should also consider that a window is a shared resource. No change notifications should be taken into account while the user is changing its size or location.

- The MVC of ST80 uses bitmap images in order to restore back contents of windows when they are moved, resized or closed. Cached images[2] are also used to refresh contents of a user activated window (in order to make the window and its sub-view redrawing faster). The problem is that a window cached image is updated in only one case: when the user disactivates it[3]. A window image (accordingly to the ST80 MVC paradigm) cannot change while the window is inactive because models are passive objects.

  This assumption does not stand for actors because they can change their state at any time. Activating a window in such a case may have the effect of displaying the cached image which stores an obsolete representation of the model state.

  Another side effect of using caches when visualizing active models is noticeable when interferences occur where the active window is displayed. When the user disactivates the window, it updates its caches by reading the images from the screen. Interferences are stored in the cached images, and when the user reactivates it, the cache is displayed with the interferences.

  Then the only way to go back to a clean screen representation is to redraw the whole screen. This is a time consuming operation since all views in the system have to be redrawn.

3.2 A New MVC Dedicated to Actors

3.2.1 Goals and Design

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

- the user is able to take control over the representation (view) of an actor at any time: while the actor is computing, and even if it attempts to update its view,

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

[2]These images are stored in the window controller's instances variables named `labelForm` and `viewForm`.
[3]A window is disactivated when the user clicks on another window.

Figure 3: Subdivision of windows.

We chose the following strategy: giving higher priority to control (interaction) 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).

The MVC extension should also allow:

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

3.2.2 Managing Visualization in Overlapping Windows

The solution we adopted to manage overlapping windows is to slightly extend the MVC. In this solution, views associated to an actor must display information on the cache of the window. Each time the contents of the cache is updated, the window is asked to display it in its visible parts, without affecting the contents of windows that may overlap it (its obscured parts). This type of windows is subdivided into visible parts (parts that are not overlapped by any window) and obscured parts (invisible parts), see figure 3.

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

In order to keep a high level of concurrency, windows that may want to update themselves on the screen while visibility information is not available are not suspended: they inform the window manager that their contents have changed, and let their models (actors) continue execution. The window manager automatically redraws those windows as soon as it finishes computing visibility information.

Figure 4: Interface generated for prime numbers algorithm.

Implementation Issues

To implement our extension to the standard MVC model, we defined the class `ActorStandardSystemView` as a subclass of standard MVC class `StandardSystemView`. `ActorStandardSystemView` defines two instance variables holding visibility information for the view and the label of the window, and defines a class variable that synchronizes all windows that display actors on the availability of those informations. Our implementation preserves the standard MVC and achieves our compatibility goal.

Efficiency Issues

We developed our solution before the availability of the new release of ST80 (4.0). Release 4.0 delegates the window management to the underlying environment (X for workstations), and consequently manages visualization in partially overlapped windows.

Meanwhile, we noticed that our solution was more efficient than the release 4.0. The reason is that the X window system sends exposure events to windows which have been exposed (made partially visible). Applications (models) associated to those windows must compute the image of exposed parts to refresh them. In our solution, we used image caches[4] which are always up to date. When a window is exposed, the only thing it has to do is to redisplay properly the cache. No further interaction with the application (the model) is needed.

3.3 Automatic MVC Generator

In order to ease the programmer with the construction of MVC-based interfaces for actors, we offer an automated interactive generation of the interface (namely the view and controller classes). Figure 4 shows an example of interface automatically generated from each of the prime number filter composing the algorithm described in section 2.2.2.

3.3.1 Design

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

[4] In fact, we reused the caches defined in the class `StandardSystemView` because it was the only mechanism provided by ST80 to restore contents of windows.

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

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

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

- *interactive description:* The user can choose, at generate time, what instance variables will be visualized. We intend to provide also non-textual (graphical) representations of numerical variables (for example, a gauge).
- *modularity:* The generator takes the hierarchy into account. When defining a hierarchy of models, their respective views and controllers automatically match this hierarchy.
- *genericity:* The generator is not limited to a specific actor model. For instance, this generator supports Agha's Actor computation model [Agha 86]. If the behavior is replaced by a structurally different one (e.g., whose instance variables differ), the view (and the controller) change accordingly.
- *debugging tools:* The user can dynamically control the way an actor computes messages (step by step, sleep...). Those tools are well suited to spy and monitor the activity of actors.

4 Scheduling Actors

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

4.1 The Standard Smalltalk-80 Scheduler

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

4.2 A First Step: Generic Scheduling Methods

In order to give a more modular and abstract level to specify scheduling, we first have proposed a set of generic event-driven methods. Each is associated with some event (e.g., receive a message, compute a message...). Classes `ExtendedActor` and `ExtendedActorBehavior`, subclasses of the Actalk kernel, redefine implementations of events to trigger these event-driven methods. For instance asynchronous message passing is redefined (in class `ExtendedActor`) to trigger the "*after receiving a message*" event-driven method, named `genericEventAfterReceivingAMessage:`

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

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

Generic event-driven methods are defined in class `ExtendedActorBehavior` with a default empty behavior. The user may redefine them in applications, for instance in class `PrimeFilter`, usually by calling `Processor yield` to enforce scheduling:

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

genericEventAfterReceivingAMessage: aMessage
    Processor yield! !
```

But such scheduling can be done only if events occur. If some actor keeps computing without any sending/receiving of asynchronous message, no scheduling will take place.

4.3 A Preemptive Scheduler for Actalk

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

4.3.1 Architecture

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

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

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

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

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

4.3.2 Time-slicing

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

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

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

A naive example of time-slicing method is:

```
!PreemptiveProcessorScheduler methodsFor: 'scheduling'!

yield
   "an example of time-slicing for processes with UserSchedulingPriority"
   | l |
   l := Processor processesListAt: UserSchedulingPriority.
   l isEmpty ifTrue: [^self].
   l addLast: l removeFirst.
   Processor processesListAt: UserSchedulingPriority put: l! !
```

4.3.3 Scheduling

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

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

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

4.3.4 Combining Time-Slicing and Scheduling

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

Figure 5: Pie menu: instant visualization of scheduling.

```
!PreemptiveProcessorScheduler methodsFor: 'scheduling'!

run
   (nbTimeSlice >= 5)
      ifTrue:[nbTimeSlice := 0.
              self schedule]
      ifFalse:[nbTimeSlice := nbTimeSlice + 1.
               self yield]! !
```

4.3.5 Visualization

In order to visualize scheduling between actors, we designed a graphic tool. It is based on *Pie menus* described in [Lalonde and Pugh 89]. Figure 5 gives a sample of its use. Each actor is represented by a slice of the pie. The slice is highlighted when the actor is scheduled and removed when the actor dies. This tool allows dynamic visualization of scheduling events.

A complementary tool, called chronogram, records scheduling events (creation, destruction, activation ... of actors), and is used to display a graphical overview (see Figure 6).

The user can easily find informations about:

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

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

5 Future Work

5.1 Debugger

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. In a debugging

Figure 6: Chronogram: overview visualization of scheduling.

mode, any time an asynchronous message is sent, it also transparently conveys the context of the sender. The debugger is extended in order to look for the sender context in the message (and not in the current context) in case of asynchronous message passing. Consequently this hybrid debugger may reconstruct the complete history of message sendings in any mixing between objects and actors.

We already have implemented a prototype compiler for higher level language (with bidirectional message passing) which generates explicit continuations. We need to provide information to the debugger so that it can match a request and the reply to a continuation (this is called transaction pairing in [Manning 87]) to restructure history of computation. This is a first step towards a higher level debugger.

5.2 Stepper

The current prototype environment provides a stepper, as part of the Actalk MVC. The user can step handling of messages by an actor. This tool is simple, but very useful to monitor and spy execution of programs. We are currently extending it in order to provide stepping of the computation of a message, in a way analog to stepping execution in the debugger.

5.3 Application Window

When developing/testing large programs in the Actalk programming environment, there is a tendency for contention of the space screen (always limited). This causes the actor windows to be overstacked when a lot of actors are visualized.

A further work is to supply an application window integrating all the tools, and offering a sub-window (considered as a virtually infinite space) where actors related to this application would be displayed.

Advantages of this approach are:

- to minimize space contention of the screen,

- to put together actors related to the same concurrent application in a unique window,
- to provide a unified global vision of a concurrent application.

Afterwards, we will manage the lay-out of actors (taking into account the virtually infinite space sub-window and structural connections between actors). This tricky issue has been discussed in [Kahn and Saraswat 90].

We plan to improve the application window by providing other tools of visualization. This can be done easily since all informations are now centralized in the application window. For example one of those tools will permit to visualize message sending: actors would be represented as a graph (nodes are actors and links are connections between them), message sending could be dynamically visualized as an icon moving all along a path (a structural connection) between the sender and the receiver.

6 Related Work

Friend projects are ConcurrentSmalltalk [Yokote and Tokoro 87], Actra [Lalonde et al. 86], and Mushroom [Hopkins et al. 89]. Actra focuses on multi-processing, Mushroom on distributed processing, and ConcurrentSmalltalk on programming language aspects. All systems conducted a small (ConcurrentSmalltalk)[5] or deep (Mushroom) evolution from the Smalltalk-80 language and virtual machine in order to fulfill their goals. As opposed to these systems, we don't intend to change the semantics of the current Smalltalk-80 language and virtual machine. We came to focus on the programming environment aspects and we expect our prototype programming environment to be easily reusable for other Smalltalk based concurrent systems, such as ConcurrentSmalltalk. We also hope that our experience can be of some interest for designing programming environments for other OOCP systems.

7 Conclusion

In this paper we discussed how to extend standard Smalltalk-80 programming environment to prototype programming environments for object-oriented concurrent programming languages. We based our work on the Actalk platform, integration of actors into the Smalltalk-80 environment. We described an interactive interface framework and generator for actors, based on MVC, and a generic preemptive scheduler of actors. We also prospected further tools for visualizing and controlling concurrent objects. This prototype programming environment indeed involves a lot of Smalltalk-80 programming. Meanwhile we believe that the issues we discussed are generic to most of the OOCP systems, and that part of our experience and prospects could be transposed to other OOCP systems.

One of our current activities is porting the system onto parallel hardware with the support of the Esprit Parallel Computing Action. We thank our colleagues of the RXF-LITP team working with us on porting the platform on multiprocessors. We thank the other teams working on concurrent Smalltalk systems for inspiration and interaction. We thank the Laforia team for building fancy DAI systems on top of our platform.

[5]The newest version, named ConcurrentSmalltalk-90 [Okamura and Tokoro 90], is simulated on top of Smalltalk-80 in order to regain better portability.

References

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

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

[Bouron et al. 90] T. Bouron, J. Ferber and F. Samuel, "A Multiagent Testbed for Heterogeneous Agents," *2nd European Workshop on Modelizing Autonomous Agents and Multi-Agent Worlds (MAAMAW'90)*, also published as a Laforia Research Report, No 20/90, Institut Blaise Pascal, Paris, France, July 1990.

[Briot 89] J.-P. Briot, "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, July 1989.

[Briot 90] J.-P. Briot, "OOCP = OOP + C," *3rd Conference on the Technology of Object-Oriented Languages and Systems (Tools 3 = Tools Pacific'90), Sydney, Australia*, pages 417-421, November 1990.

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

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

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

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

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

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

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

[Manning 87] C.R. Manning, "Traveler: the Apiary Observatory," *European Conference on Object-Oriented Programming (ECOOP'87), Lecture Notes in Computer Science*, No 276, pages 89-97, Springer-Verlag, 1987.

[Okamura and Tokoro 90] "ConcurrentSmalltalk-90," *3rd Conference on the Technology of Object-Oriented Languages and Systems (Tools 3 = Tools Pacific'90), Sydney, Australia*, pages 231-244, November 1990.

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

[Voyer 89] R. Voyer, "A Tool for Generating Knowledge Representation in Smalltalk-80," *3rd Conference on the Technology of Object-Oriented Languages and Systems (Tools 3 = Tools Pacific'90), Sydney, Australia*, pages 37-45, November 1990.

[Yokote and Tokoro 87] T. Yokote and M. Tokoro, "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.