Design of a Generic and Reusable Scheduler for Smalltalk-80

Position Paper Presented at the

Ecoop'92 Workshop on Object-Based Concurrency and Reuse

Jean-Pierre BRIOT & Loïc LESCAUDRON*

LITP, Institut Blaise Pascal
(* Equipe Mixte Rank Xerox France - LITP)
4 place Jussieu, 75252 Paris Cedex 05, France

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

Abstract

This paper discusses the reuse of software components for describing and implementing various schedulers. We advocate the design of a generic and general purpose scheduler. This generic scheduler is based on object-oriented methodology (namely Smalltalk-80) and may be instantiated to match various scheduling strategies. This scheduler has been primarily developed for Actalk, a testbed for actor languages integrated into Smalltalk-80, to ease the prototyping of scheduling policies for actors. Our generic scheduler is particularly efficient to help the design of time-sharing algorithms that are not supported by the standard Smalltalk-80 scheduler. Our starting point was the Smalltalk-80 scheduler. We extend it to take time-sharing into account. In spite of the fact that our generic scheduler is used for Actalk, its purpose is far more general: it may be used for simulation for example.

In this paper, we firstly present the goal of time-sharing algorithms and a few examples. Secondly, we explain why and how we designed our own generic scheduler on top of the Smalltalk-80 one. We illustrate the use of our scheduler by simulating the algorithms firstly presented as examples. We discuss how object-oriented techniques allow us to build-up and reuse scheduling policies.

Keywords

object, concurrency, actor, reusability, genericity, Smalltalk-80[1], Actalk[2], scheduling, scheduling policy.

1 Introduction

This paper is concerned with the design of a generic scheduler primarily used with Actalk. The goal of the initial Actalk project [Briot 89] was to develop a testbed to simulate, experiment and classify actor languages. Actalk is based on Smalltalk-80. Actalk fulfills its goal, more especially for pedagogical uses (description of the Actor model of computation or ABCL/1), and for the design of distributed AI systems.

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

[2] Actalk [Briot 89] stands for *act*ors in Small*talk*-80.

This success led us to extend the purpose of Actalk by adding a programming environment dedicated to actors languages. This second step of the Actalk project was then to offer programming environment tools for actor languages designed within the Actalk system [Lescaudron et al. 91]. Most of those tools are built as an extension/customization of the Smalltalk-80 existing ones. In this paper we describe and discuss the tool for scheduling actors. This scheduler is generic and we will discuss how it helps to reuse and extend scheduling strategies. We will first start with a short survey of scheduling policies.

2 Time-Sharing Scheduling

Time-sharing scheduling became an important research topic in the 60's, triggered by a new way of using computers resources (mainly processors). Before those days, computers were used in batch-processing mode: the computer processed each request (i.e., a process) in turn till completion. Those requests were previously ordered according to what is called a *scheduling policy*. The most important advantage of this way of using a computer was the simplicity of the scheduling policy, but to the detriment of interactivity. Research on this topic led to new scheduling policies allowing better interactivity. Those new scheduling policies are known as *time sharing policies*.

2.1 What is Time-Sharing Made Of?

The main idea of those new policies is to share the processor among numerous processes, giving the illusion to each of them that it is the only process using the processor. This means that the processor manages a list of processes expecting to access the processor (those processes are called *eligible processes*). Each process in this list is executed by the processor (and then called the *current process*) for a predefined amount of time (called a *quantum*). If the current process terminates before the end of the quantum, a new current process is elected, according to the scheduling policy, from the eligible processes list. On the other hand, if the current process is not terminated, it goes back to the eligible processes list, and another process is elected and becomes the current process for a quantum of time.

Nevertheless time-sharing policies may be very different one from another, according to the following features:

- *preemption* The preemption defines the behavior of the scheduler when a new process enters the system. If the scheduler forbids preemption, this new process is just added to the eligible processes list. If preemption is allowed, the new process may become the current process, and the actual current process goes back to the eligible processes list. Generally, the preemption is performed if the new process has a higher priority than the current process. But, in some particular cases, the current process may be preempted for other reasons such as error recovery or fault tolerance.

- *computation of priorities* The priority of processes is used to select the next current process among the eligible processes: the process with the highest priority is elected. This priority may be implicit (the order of arrival in the system is used), or explicit. When the priority is explicit, each process has its own priority. This priority can be static (defined when the process is created), or dynamic which may change (depending on the policy) during its lifetime.

- *value of the quantum* Most of the time, the quantum is fixed during the design of the scheduler. The smallest the value, the biggest is the process turn over. The designers of a scheduler have to pay attention to *overhead* induced by changing the current process too often. Changing the current process is time consuming; if this time is too important according to the value of the quantum, this quantum must be increased. Powerful policies allow to change dynamically the value of the quantum accordingly to the state of the system.

2.2 Examples of Time-Sharing Policies

To illustrate the principles of time-sharing policies, we are going now to present a few scheduling policies based on time-sharing. More information can be found in [Coffman and Kleinrock 68].

Round Robin

The Round Robin model is the most primitive time-sharing model. The value of the quantum is predefined and no priority is supported. The current process is elected as in a FIFO list: first-come-first-served. When a new process enters the system, it is added at the end of the eligible processes list. The scheduler always selects the process at the very beginning of this list when a new current process should be elected.

This model is very fair since each process is elected in turn for the same amount of time. But, it is unable to provide short times of execution for the small and highly interactive processes. In fact, since processes are executed without distinction, *small* processes may spend to much time till completion because many *big* processes[3] are present in the system at the same time and are elected at the same rate.

Feedback

The Feedback model (also called Foreground-Background) is an extension of the Round Robin model. It tries to solve the problem explained in the previous section by introducing explicit and dynamic priorities, decreasing in accordance with the number of time the process is elected. So that, *big* processes have a smaller priority than *small* processes; *small* processes are elected more often and terminate more quickly.

Cycle-Oriented Round Robin

Schedulers based on Cycle-Oriented Round Robin policy compute dynamically the value of the quantum to be able to execute once each process of the eligible processes list during a *cycle* period T. Roughly speaking, the quantum is computed as follow: $quantum = \frac{T}{number\ of\ eligible\ processes}$. When the number of eligible processes is too important, the computed value may be too small (in spite of overhead constraints explained previously) to be considered.

Time-sharing policies currently offer the best compromises for scheduling. But, as we can notice from the previous section, no time-sharing policy is considered as the best one. All of them have their pros and cons, because their power is highly dependent on the state of the system, and the nature of computation in progress.

2.3 Towards a Better Interaction between the Scheduler and the Application

We remark that many of the concurrent languages don't consider much control on scheduling, or/and delegate scheduling to an underlying scheduler (like Un*x). This may lead to limitations and conflicts (for example the Un*x scheduler limitates the number of available processes). We firmly believe that the scheduling policy has to be more explicit and connected with the application. Thus, a "good" scheduler should be able to change its policy accordingly to the application currently in progress. For

[3] We call a process a *small* process or a *big* process accordingly to the amount of time (i.e., the amount of quanta) it requires to finish its execution.

example, the scheduling of agents in a distributed AI system involves a fair time-sharing among them. On the other hand, database requests may be scheduled, on a time-shared basis, in accordance with their priority.

3 Our Generic Scheduler

3.1 Goals and Motivations

The design of our generic scheduler is motivated by our argument explained in section 2.3: the running application and the scheduler are connected. The purpose of our generic scheduler is to offer a generic tool in Smalltalk-80 to help the designer of any application to implement the appropriate scheduling policy, itself based on the standard Smalltalk-80 scheduler (this point will be explained in section 3.2). More precisely we can define our goals as follows:

1. *Extending the power of the Smalltalk-80 scheduler*
   Extending a scheduler (namely the standard Smalltalk-80 scheduler) for our purpose requires that it is preemptive (it is) and that it is based on time-sharing (it isn't[4]) — the reasons are explained in the following sections. Preemption is sufficient to implement time-sharing policies of our generic scheduler. But without time-sharing at the standard scheduler level, scheduling of standard Smalltalk-80 processes may be unabled. Consequently extending the standard Smalltalk-80 scheduler for time-sharing is necessary if we want to preserve the scheduling of standard Smalltalk-80 processes.

2. *A good integration within the Smalltalk-80 environment*
   By preserving the execution of standard Smalltalk-80 processes, we don't touch the standard Smalltalk-80 interface architecture (MVC) which relies on controllers (relying themselves on processes). Another goal is to reuse the MVC to build various visualizations tools for the scheduler (described in [Lescaudron et al. 91]).

3. *Genericity*
   This is probably the most important and innovative point. Our generic scheduler should be considered as a testbed to simulate various scheduling policies (as Actalk with OOCP). Thus, our scheduler should offer the basic mechanisms to express any kind of scheduling policy (in the context of time-sharing).

4. *Easy to use and highly expressive*
   Since the users of our generic scheduler are not necessary aware of scheduling, our tool should be easy to use. Nevertheless, it should be very expressive too, because scheduling algorithms are getting more and more complex and difficult to express.

3.2 Design

We cannot create a new scheduler from scratch in Smalltalk-80. In fact, the Smalltalk-80 virtual machine (SVM) refers to the Smalltalk-80 scheduler (the global Smalltalk object `Processor`) to access the eligible processes list. If we change the Smalltalk-80 scheduler (i.e., the reference to `Processor`), the SVM loses this access since it doesn't know anymore how this scheduler is implemented. Thus, the generic scheduler cannot schedule processes without the Smalltalk-80 scheduler. The generic scheduler has to *cooperate* with the standard scheduler. More precisely, the generic scheduler, responsible of the scheduling policy, forces the standard scheduler to act accordingly to the policy it defines[5] by

[4] Processes must make explicit calls to the scheduler (by evaluating `Processor yield`) to give chance to others.

[5] After a deep study of the standard scheduler [Lescaudron 92] and the standard scheduling policy, we know exactly how it behaves and we can indeed force it to act as we want (i.e., to schedule a given process).

modifying the standard eligible processes list. In other words, the standard scheduler is slaved to (or controlled by) the generic scheduler.

The Smalltalk-80 eligible processes are not directly concerned with the generic scheduling policy. Hence, it's better to memorize the concerned processes (we will call them *generic* (scheduler) *processes* to distinct them from the standard (scheduler) processes) in a different eligible list than the standard one. The generic scheduler manages this new list itself. When the generic scheduler wants to schedule a given generic process from its list, it takes benefit of the knowledge of the standard scheduler to insert this process at the right place to ensure that this process is going to be elected.

Nevertheless, scheduling policies cannot be based on low level manipulations of the standard and generic eligible processes lists. It is too low level constraints for the users of our system, and pretty dangerous as well. We have defined a set of scheduling primitives used by the users to build their own scheduling policies. The implementation of those primitives implies manipulations of the two lists, but this is hidden to the users. Those primitives are composed by the user, to create the algorithm in charge of the scheduling policy, in a method called `run`.

The generic scheduler is implemented with a standard Smalltalk-80 process. This infinite process (called the *generic scheduler process*) awakes periodically to execute the `run` method and goes back to sleep for a while. Thus, to enable the correct execution of this process, we should be able to ensure that it will be correctly awaken, and that it will effectively become the current process. This explains why we required that the underlying scheduler is preemptive. Thus we just have to give a high priority to the generic scheduler process to be sure that it will become the current process when it is awaken.

Note that the generic scheduler is the one to elect the next current process. The standard scheduler is "fooled" to execute the process chosen by the generic scheduler.

3.3 Implementation

The Generic Scheduler

The generic scheduler `GenericProcessor`[6] is an instance of the class `GenericProcessorScheduler`. This class is defined as a subclass of `ProcessorScheduler` because we partially reuse the structure of the eligible processes list and a few methods. The `GenericProcessor` creates and refers to the generic scheduler process in that way:

```
GenericProcessorScheduler methodsFor: 'initialization'
setSchedulerProcess
    schedulerProcess:=
        [[true]
           whileTrue:[
                   "execution of the run method"
              self run.
                   "the generic scheduler process sleeps for a duration
                    specified by the method suspendingDelay"
              (Delay forMilliseconds: self suspendingDelay) wait
           ]
        ] newProcess.
    schedulerProcess priority: 8     "high priority to ensure preemption
                                      when the process awakes"
```

The abstract class `GenericProcessorScheduler` doesn't define the `run` method. The users just have to define subclasses of this abstract class to implement their own version of `run` (i.e., of scheduling policy). On the other hand, the scheduling primitives are defined in the abstract class. We are going to explain the two main ones:

[6] We named it this way because the standard scheduler is named `Processor`.

The yield Primitive

As explained in section 3.1, in order for standard Smalltalk-80 processes and generic processes to execute together in a fair way, the standard Smalltalk-80 scheduler needs to be extended to support time-sharing. The primitive `yield`[7] ensures a time-shared execution of the standard processes and the generic processes. The implementation of this primitive is very simple. It just has to change the order of the eligible processes to ensure scheduling. Let's say that the first process (normally elected) is going to be removed from the list, and put back to the last position:

```
yield
    | l |
    l _ (Processor processesListAt: UserSchedulingPriority).
    l isEmpty ifTrue:[^self].
    (l add: l removeFirst).
    Processor processesListAt: UserSchedulingPriority put: l
```

Note that defining the `run` method as `self yield` and defining a fixed returned value for method `suspendingDelay` leads to a Round Robin policy for standard Smalltalk-80 processes.

The schedule Primitive

The `schedule` primitive implements scheduling of processes in the generic scheduler. Its implementation relies on *virtual* methods (à la C++) defined in subclasses:

```
schedule
    | prev |
    "remove the current generic process (if it exists) from the standard
     eligible processes list and put it back to the generic one"
    (prev := activeProcess) notNil
        ifTrue:[....stuff deleted....
           "
           self removeActiveProcessFromST80Table.
           self toGenericTable: prev].

    "put the next current generic process (eventually the same)
     in the standard list"
    (activeProcess := self searchForProcess) isNil
        ifFalse:[self activeProcessToST80Table]
        ifTrue:[...stuff deleted...]
```

The methods `removeActiveProcessFromST80Table` and `activeProcessToST80Table` are primitives. They hide the manipulation of the two electible processes lists to the users and won't be addressed here. The methods `toGenericTable` and `searchForProcess` are virtual. They are defined in subclasses accordingly to respectively, the structure of the generic eligible processes list, and the way of choosing the next current process.

3.4 Implementing Various Policies with our Generic Scheduler

Simulating the Round Robin Policies

The simulation of the basic Round Robin policy is immediate. We define a subclass of the class `GenericSchedulerProcess` called `RRScheduler`, to specify a value for the quantum in the method `suspendingDelay` (let's say 50 milliseconds), and to define the virtual methods used by the `schedule`

[7] Note that this primitive of the generic scheduler (class `GenericProcessorScheduler`) is distinct from standard primitive `yield` (which does not need to be used anymore) and is also more efficient.

primitive. The (instance) variable[8] `quiescentProcessList` refers to the processes list of the generic scheduler. Remember that the Round Robin policy is based on a FIFO algorithm:

```
toGenericTable: aProcess
    quiescentProcessList add: aProcess

searchForProcess
    ^quiescentProcessList isEmpty
        ifTrue:[nil]
        ifFalse:[quiescentProcessList removeFirst]
```

Then, the `run` method can be defined easily. For this paper, we give a very naive version of the implementation that doesn't take into account the time-sharing of Smalltalk-80 processes and the current generic process:

```
run
    self schedule
```

Simulating feedback or Cycle-Oriented Round Robin is very simple too. We are not going to explain their implementations here.

A Case Study: the Kleinrock's Continuum of Time-Sharing Scheduling Algorithms

In this last example, we are going to simulate the model introduced by Kleinrock in [Kleinrock 70]. His idea was to define two parameters called α and β. α represents the variation rate at which the current process changes its priority. β represent the variation rate at which the generic eligible processes change their priorities. Depending on the values of α and β his model behaves as if it was a Round Robin model, a First-Come-First-Served model, or others... (cf [Kleinrock 70] or [Lescaudron 92] for more details). To simulate this model in our system, we define a new class:

```
RRScheduler subclass: #AlphaBetaScheduler
    instanceVariableNames: 'alpha beta '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Actalk-Scheduler'!
```

We change the initialization of `GenericProcessor` in such a way that `quiescentProcessList` is now an instance of `SortedCollection` to sort processes accordingly to their priorities. And, last but not the least, we overwrite the definition of `run`:

```
run
    self computeNewPrioritiesWithAlphaAndBeta.
    self schedule
```

4 Conclusion

In this paper we described a generic scheduler implemented in Smalltalk-80. This scheduler makes full use of OOP methodology (and reuses the standard Smalltalk-80 scheduler) to help defining and reusing various scheduling policies. A few examples show how this generic scheduler can be instantiated to implement standard scheduling policies. This scheduler is implemented in Smalltalk-80. However we believe that our experience could be transposed: for instance light weight processes (LWP) are a good starting point to transpose this work in a compiled language such as C++.

We would like to express our sincere thanks to Jean Bézivin who drew our attention to Kleinrock's model and initiated the present development.

[8] This instance variable is inherited from the standard scheduler (`ProcessorScheduler` class). In the standard scheduler, this list of processes is an eight entries table, according to priorities. In the generic scheduler this is a one-level list, implemented by an `OrderedCollection` in order to easily manipulate it as a FIFO.

References

[Briot 89] 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.

[Coffman and Kleinrock 68] Coffman E.G. and Kleinrock L., Computer scheduling method and their countermeasures, *Spring Joint Computer Conference*, Vol. 32, page 11, AFIPS press, Montvale N.J., 1968.

[Kleinrock 70] Kleinrock L., A Continuum of Time-Sharing Scheduling Algorithms, *Proceedings of the AFIPS SJCC*, page 453, 1970.

[Lescaudron et al. 91] Lescaudron L., Briot J.-P., Bouabsa M., "Prototyping Programming Environments of Object-Oriented Languages: a Smalltalk-Based Experience," *TOOLS 5*, Prentice Hall, page 449, 1991.

[Lescaudron 92] Lescaudron L., "Prototypage d'environnements de programmation pour les langages à objets concurrents : une réalisation en Smalltalk-80 pour Actalk," *Université Paris VI*, Thèse d'Université, May 1992.