Object-Oriented Design of a Generic Scheduler

Jean-Pierre Briot

Dept. of Information Science, The University of Tokyo

Abstract

This paper discusses the reuse of software components for describing and implementing various schedulers. We advocate the design of a generic scheduler based on object-oriented methodology (namely Smalltalk-80)[1] in order to obtain optimal expressivity and reuse. It may be instantiated to match various scheduling policies. This is achieved by defining new scheduler subclasses. Scheduling primitives are proposed as building blocks to be combined to define scheduling policies. Some virtual methods represent the semantics of electing current processes and may be customized accordingly.

This scheduler has been primarily developed for Actalk, a testbed for actor languages integrated into Smalltalk-80[2], to ease the prototyping of scheduling policies for actors. It proved very useful for specifying various scheduling policies for actors (like for instance in the Actor computation model where there are at least two levels of concurrency: inter-object and intra-object).

In this paper, we first present the goal of time-sharing algorithms. Then we discuss the goals and motivations leading to the generic scheduler. We then describe its architecture, how it collaborates with the standard scheduler, and the genericity and openness of its design. We shortly discuss its implementation, focusing on some use of reflective techniques in order to open-up the fixed connection between the Smalltalk-80 virtual machine and the standard scheduler. We then shortly survey how it can be used to implement step by step various scheduling policies. We then conclude by summarizing how the full

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

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

use of object-oriented techniques allows the genericity of our scheduler, and by comparing to related work.

1 Introduction

All systems providing multiple units of activities (let's call them *processes*) include a scheduler at their execution layer in order to manage execution of these different activities on the available resources (often a single processor). In many systems, the scheduler is hidden to the programmer, as the current execution of processes is left transparent at the programming level. However different scheduling policies may be more appropriate to optimize execution for different problems. Actually many scheduling policies have been proposed to match various requirements. For instance the *Feedback* policy has been designed in order to optimize execution, that is increase priority, of short processes. We believe that it is very useful to provide ways to customize the scheduling policies to the problems, possibly dynamically during execution of the program.

The goal is to provide a scheduler which is generic enough so it can be easily extended to specify any kind of scheduling policy. The scheduler we designed and implemented makes use of object-oriented methodology and reflective techniques in order to achieve this goal. This scheduler was designed as one of the tool of the Actalk platform. The Actalk platform project [Briot 89, Lescaudron et al. 91] is a programming environment, based on Smalltalk-80, to model, classify and experiment with object-oriented concurrent programming. Most of Actalk tools are built as extension/customization of the Smalltalk-80 existing ones. This is for instance the case for the generic scheduler.

2 Scheduling for Time-Sharing

The main idea of scheduling 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 access to 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*).

Time-sharing policies may be very different one from another, according to the following features: *preemption* (if a new process entering the system may become the current process), *computation of priorities* (which may be implicit or explicit *and* static or dynamic), and *value of the quantum* (predefined amount of time to execute a process; powerful policies change it dynamically).

Numerous scheduling policies have been proposed (like Round Robin, Feedback... see [Coffman and Kleinrock 68]). No time-sharing policy is considered as the general optimal. Each has its pros and cons, because their power is highly dependent on the state of the system, and the nature of computation in progress.

3 The Generic Scheduler

3.1 Goals and Motivations

In order to be able to specify and control various scheduling policies, we designed a generic scheduler in the Smalltalk-80 environment. Actually our first initial goal was to add time-slicing facility to the Smalltalk-80 scheduler in order to get a satisfying concurrent execution of multiple actors. We then decided to expand our goals further in order to get a new generic scheduler supporting various scheduling policies. We wanted it to be both easy to use and expressive. A good integration within the Smalltalk-80 environment was also a major goal.

3.2 Design

3.2.1 Reuse and Cooperation

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

To ensure a smooth cooperation of the generic scheduler with the standard one, we need to open-up some fixed assumption between the SVM and the standard scheduler. This is achieved by using reflective techniques offered by Smalltalk-80 (see Section 3.4.2).

3.2.2 Specialization

Complex scheduling policies are usually defined as variations/specializations of more basic ones, depending on the chosen semantics of some features. In our scheduler, scheduling policies are defined as classes, allowing step by step refinements through inheritance.

3.2.3 Modularity

To preserve standard Smalltalk-80 scheduling policy and achieve modularity, we distinguish between standard Smalltalk-80 processes and processes at the generic scheduler level. We will call them *generic processes* and they will be instances of a new class `GenericProcess`, subclass of standard class `Process`.

3.2.4 Genericity

We provide an open-ended data structure to contain the eligible processes. For instance, depending of scheduling policies, this could be a simple list (`OrderedCollection` in Smalltalk), or a sorted collection where processes will be sorted according to their priorities (see examples in Section 4.1). Some virtual methods are associated to define (generic) manipulation of this data structure (see Section 3.3, method `schedule`).

3.2.5 Building Blocks

Definition of scheduling policies cannot be based on low level manipulations of the standard and generic eligible process list. It is too low level constraints for the users of our system, and pretty dangerous as well. Therefore we have defined a set of *scheduling primitives*, used as building blocks to specify various scheduling policies. Some of these scheduling primitives are further decomposed in terms of virtual methods in order to parameterize the semantics of process election. They may be redefined in order to specialize the structure of the set of eligible processes and the way to choose among them.

3.3 Architecture

The generic scheduler is defined by an abstract class named `GenericProcessorScheduler` as a subclass of class `ProcessorScheduler` (which describes the standard scheduler). The generic scheduler applies the scheduling policy as defined by its method `run`. This method is defined by combining among a set of scheduling primitives. The two main scheduling primitives currently proposed are the following:

- `yield`

  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 scheduling primitive `yield`[3] ensures a time-shared execution of the standard processes and the generic processes.

- `schedule`

  The scheduling primitive `schedule` is the entry point to schedule generic processes. Its behavior is decomposed/parameterized among several methods. Some are primitive, some are *virtual* (à la C++) and they are to be redefined in subclasses. Methods `addToGenericTable` and `searchForProcess` are the two virtual methods. They respectively describe how to add a new process to the (generic) table of eligible generic processes (thus exposing the structure of this table), and the way of choosing the next current process.

[3]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.

3.4 Implementation

3.4.1 The Generic Scheduler Process

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.

3.4.2 Use of Reflective Techniques to Open-Up Standard Scheduler Entry Points

The generic scheduler makes use and relies on the standard Smalltalk-80 scheduler. This standard scheduler is tightly coupled with the Smalltalk-80 virtual machine. This may lead to problems because the virtual machine makes assumptions about the scheduler which could bypass our generic scheduler. The use of reflective techniques enables to open-up this otherwise inflexible implementation semantics (as discussed by Gregor Kiczales [Kiczales 92]). More precisely we change dynamically and temporarily the behavior of some standard Smalltalk-80 objects (namely semaphores) to expose the SVM/scheduler relationship (namely connection between semaphores and scheduler) to the generic scheduler.

Let's explain the possible problem. When the scheduling primitive `schedule` is executed, the current generic process is removed from the table of standard Smalltalk-80 eligible processes, and a new one will be elected. But it may happen that the current generic process is not in the table of standard processes. If this process is waiting on a semaphore, it is not in the table (but it is linked to the semaphore) and the generic scheduler does not know about it! This is because when a process suspends onto a semaphore, the standard scheduler is informed and possibly updated. But the virtual machine does not know about our generic scheduler.

To handle this problem the intuitive solutions are following:

- define a new class of semaphores,

which will inform the generic scheduler. The problem is that we cannot ensure not using any standard semaphore while programming applications (which could make use of standard Smalltalk-80 code).

- redefine methods for semaphores,
  (they are named `wait` and `signal` in Smalltalk-80). But we need to ensure their atomicity and we decrease the efficiency of the whole system.

Preventing the problem is too expensive, so we prefer to cure it when necessary. (Such a situation is not standard.) The basic idea is to first trap the error, then change temporarily the semantics of the semaphore. The changed semantics will ensure that when signaled, the semaphore will place the resuming process back to the eligible list of generic processes (the generic scheduler) and not to the standard scheduler. Eventually the semaphore regains its initial semantics. This dynamic and temporary change of semantics is achieved by changing dynamically the class of the semaphore (one of the reflective operations offered by Smalltalk-80).

4 Implementing Various Policies with our Generic Scheduler

4.1 Simulating the Round Robin Policies

The simulation of the basic Round Robin policy is immediate. We define a subclass of the class `GenericProcessorScheduler` 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` primitive. The (instance) variable[4] `quiescentProcessList` refers to the processes list of the generic scheduler. Remember that the Round Robin policy is based on a FIFO algorithm:

[4]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 its semantics is generic (open-ended). It acts as a list for the Round Robin scheduler. Thus it is implemented by an `OrderedCollection` in order to easily manipulate it as a FIFO.

```
addToGenericTable: 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.

4.2 A Case Study: 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]. The idea of the model is to describe various policies through variations of two parameters: α and β. α represents the variation rate at which the current process changes its priority. β represents the variation rate at which the generic eligible processes change their priorities. Depending on the values of α and β, Kleinrock's model behaves as a Round Robin model, a First-Come-First-Served model, or others... (cf [Kleinrock 70] 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 the generic scheduler 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
```

5 Related and Further Work

5.1 Related Work

The implementation of the Portable Concurrent Smalltalk (CST) system also extends the standard Smalltalk-80 scheduler to support time-slicing. But there is no distinction between CST processes and Smalltalk-80 processes, thus reflecting the integration achieved. Our goal is different because we want to achieve a finer grain and generic control of scheduling of actors.

The reflective OOCP system RbCl [Ichisugi et al. 92] reifies the scheduler as active object(s) which may be modified to change the scheduling policy. The AL-1 system [Ishikawa 91] also provides a way to selectively modify the scheduler. As opposed to such systems, we started from an existing system, namely Smalltalk-80, mainly because of its wonderful and integrated programming environment. Although Smalltalk-80 is not a "fully" reflective language (for instance the scheduler is an object, but the virtual machine makes some assumptions about the scheduler which lower its openness), we showed that it provides reflective techniques powerful enough so we could open-up these assumptions and provide a smooth cooperation between our generic scheduler and standard Smalltalk-80 scheduler. Our main goal was to provide an architecture to help expressing and reusing various scheduling policies.

5.2 Further Work

We believe that the architecture of our generic scheduler may be expanded further to achieve time-depending computation. One way is to add a new slot to generic processes about expected time of end of computation, and to add a new scheduling primitive to control preemption in order to ensure time constraints.

Also note that, although this generic scheduler is implemented in Smalltalk-80, 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++, as well as for other OOCP systems.

6 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 scheduling policies. A few examples showed how this generic scheduler can be instantiated to implement *and reuse* various scheduling policies. This genericity is achieved by using object-oriented methodology: that is classes, inheritance, decomposition, virtual methods, and reflection.

Note that, although not described in this paper (see [Lescaudron et al. 91]), we also developed visualization graphic tools to show the activity of the scheduler. An instantaneous view, based on a pie menu, displays activation of scheduled processes. A buffered history view (called a chronogram) summarizes scheduling and life-lines of processes.

Acknowledgements This paper describes and elaborates on the work of Loïc Lescaudron who was the main designer and implementer of this generic scheduler, as part of the Actalk project conducted at LITP, CNRS - Université Paris VI, France. We also would like to express our sincere thanks to Jean Bézivin who drew our attention to Kleinrock's model.

References

[Briot 89] Briot J.-P., "Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment," *ECOOP'89*, Cambridge University Press, pages 109-129, 1989.

[Briot and Lescaudron 92] Briot J.-P. and Lescaudron L., "Design of a Generic and Reusable Scheduler for Smalltalk-80," (Position Paper), *ECOOP'92 Workshop on Object-Based Concurrency and Reuse*, Utrecht, Netherlands, July 1992.

[Coffman and Kleinrock 68] Coffman E.G. and Kleinrock L., "Computer Scheduling Method and Their Countermeasures," *Spring Joint Computer Conference*, AFIPS Press, Vol. 32, page 11, 1968.

[Ichisugi et al. 92] Ichisugi, Y., Matusoka, S., and Yonezawa, A., "RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel," *IMSA'92 Workshop on Reflection and Meta-Level Architectures*, pages 24-35, Tokyo, Japan, November 1992.

[Ishikawa 91] Ishikawa, Y., "Reflection Facilities and Realistic Programming," Sigplan Notices, Vol. 26, No 8, pages 101–110, August 1991.

[Kiczales 92] Kiczales G., "Towards a New Model of Abstraction in Software Engineering," *IMSA'92 Workshop on Reflection and Meta-Level Architectures*, pages 1–11, Tokyo, Japan, November 1992.

[Kleinrock 70] Kleinrock L., "A Continuum of Time-Sharing Scheduling Algorithms," *Spring Joint Computer Conference*, AFIPS Press, page 453, 1970.

[Lescaudron et al. 91] Lescaudron L., Briot J.-P., and Bouabsa M., "Prototyping Programming Environments of Object-Oriented Languages: a Smalltalk-Based Experience," *Tools 5 = Tools Usa'91*, Prentice Hall, page 449–462, 1991.

[Lescaudron 92] Lescaudron L., "Prototypage d'Environnements de Programmation pour les Langages à Objets Concurrents : une Réalisation en Smalltalk-80 pour Actalk," Thèse d'Université, *Université Paris VI*, Paris, France, May 1992.

[Okamura and Tokoro 90] Okamura H. and Tokoro M., "ConcurrentSmalltalk-90," *Tools 3 = Tools Pacific'90*, pages 231–244, November 1990.