On the use of Smalltalk for Concurrent and Distributed Programming*

Jean-Pierre BRIOT† — Rachid GUERRAOUI

Dept. of Information Science
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku
Tokyo 113, Japan

`briot@is.s.u-tokyo.ac.jp`

Département d'Informatique
École Polytechnique
Fédérale de Lausanne
CH-1015, Lausanne, Suisse

`guerraoui@lse.epfl.ch`

Abstract

This paper studies *why* and *how* Smalltalk, although designed as a mono-user and mono-processor programming language, is a very good foundation for concurrent and distributed programming. The two main reasons behind this likely paradox are: the rich and reusable library of classes, and the extreme flexibility of Smalltalk. These two specificities complement more general object-oriented characteristics and advantages. As examples supporting our claim, we overview two different Smalltalk-based software platforms for concurrent and distributed programming.

1 Introduction

Our basic claim in this paper is that Smalltalk, although initially designed as a sequential, mono-user and mono-processor programming environment, is actually a very good basis for building concurrent and distributed programming systems. We may summarize the reasons behind our claim in the four main following points:

- Smalltalk, as an object-based [1] language, provides good *encapsulation* and *structuring* to build services and mechanisms. Meanwhile,

*In SI Informatik Journal, Special Issue on Smalltalk, Vol 1, 1996.

†Main affiliation at LAFORIA, Institut Blaise Pascal, 4 place Jussieu, 75252 Paris Cedex 05, France. E-mail: `briot@laforia.ibp.fr`.

[1] Peter Wegner has defined in [Weg87] an *object-based* language, as a language which provides encapsulation, and an *object-oriented* language, as a language which further provides class and inheritance concepts.

it provides *genericity* to match various software and hardware architectures, and *independency of physical location* through the message passing communication metaphor.

- Smalltalk, as an object-oriented language, provides *classification* and *specialization* mechanisms to classify and customize various mechanisms and protocols.
- Smalltalk, as a rich programming *environment*, provides *standard libraries* on which to build up various concurrency and remote invocation mechanisms.
- Smalltalk, as a *reflective* language (self-descriptive and modifiable), makes it easy to *customize* and *integrate* extended concurrency and distribution models and protocols.

To illustrate our analysis, after reviewing these key features of Smalltalk, we will overview the construction of two concurrent and distributed programming platforms built on top of Smalltalk: Actalk [Bri 94] and GARF [GGM 94].

Note that this paper is in spirit a position paper, and by no means an exhaustive study of pros and cons of Smalltalk or other programming languages for concurrent and distributed programming.

2 Objects for Concurrent and Distributed Programming

It is now well accepted that the notion of "object" provides good foundations for the new challenges of parallel, distributed and open computing. For instance both the Open Distributed Processing (ODP) and the Object Management Group (OMG) recent initiatives for heterogeneous distributed computing are based on object concepts [NWM 93].

Concurrency is also potentially present in the concept of object. The first object-oriented programming language, Simula-67 (Smalltalk-72 being the second one), introduced the concept of object together with coroutine abilities. Also, there are obvious similarities between the notion of object and the notion of "process": variables, persistent data, encapsulation, communication.

Integrating objects and processes, then synchronization and message passing, leads to the notion of *active* object ([Gue 95] discusses various active object models). Furthermore, message passing not only ensures the independence of the behavior of an object with its actual internal representation, but it also easily ensures the independence of its physical *location*. Objects appear as natural and potential units for program distribution. Self-containedness of objects (data plus procedures, plus possible internal activity) also eases the issue of moving (migrating) them around.

3 Smalltalk

Smalltalk [2] is often considered as one of the "purest" example of object-oriented language. This is because its "motto" is to have only a few concepts (object, message passing, class, inheritance) and to apply them *uniformly* to any aspect of the language and environment. One consequence is that the language is actually very *simple*. The richness of Smalltalk comes from it rich set of class libraries, as we will see.

3.1 Standard Libraries

Smalltalk offers a rich and standard library of classes. They describe and implement various programming constructs (control structures, data structures...), internal resources (messages, processes, compiler...), and a sophisticated programming environment with integrated tools (browser, inspector, debugger...).

Actually, even basic control structures, such as loop and conditional, are not primitive language constructs, but just standard methods (of standard classes) which make use of the generic invocation of message passing. They are based on booleans and *blocks*. A block is a general sequence of expressions, enclosed within brackets (`[` and `]`), whose evaluation is delayed. Blocks (represented as instances of class `BlockContext`) are essential for building various control structures that the user may extend at his wish. They also represent one kind of execution *context*, the basis for multi-threaded concurrency. By sending message `fork` to a block, a new process is created (and started). Standard class `Process` describes their representation and its associated methods implement process management (suspend, resume, adjust priority...). The behavior of the process scheduler is itself described by a class, named `ProcessorScheduler`.

The basic synchronization primitive is the *semaphore*, represented by class `Semaphore`. Standard libraries also include higher abstractions: class `SharedQueue` to manage communication between processes, and class `Promise` for representing the eager evaluation (value computed by a concurrently executing process). These two abstractions are the basis for implementing buffered (asynchronous) and eager reply ("future" type) communications, for (object-oriented) concurrent programming. More generally speaking, it is easy to build up on this basic standard library of concurrency classes to construct more sophisticated abstractions, as for example in the Actalk platform (see in Section 4.1).

Smalltalk also offers libraries for remote communication (Unix sockets, RPC...), and standard libraries for storage and exchange of object structures (the *Binary Object Streaming Service (BOSS)* library) as a basic support for persistency, transaction checkpoint, and marshalling for remote communications (see for instance in Section 4.2).

[2] In this text, "Smalltalk" will implicitly refer to "Smalltalk-80" [GR 83], and its most recent marketing denomination: "VisualWorks."

3.2 Flexibility and Integration

The fact that most of Smalltalk language constructs (classes [3], methods, messages, control structures, contexts, processes...), and mechanisms (compiler, scheduler...), are "first-class" objects, and accordingly represented through standard class libraries, makes the language self-descriptive and also extensible. In order to also open up the semantics/implementation of its basic behavior (sending a message, referencing an object...), Smalltalk provides "hooks" to be able to customize it. The combination of self-description of Smalltalk and such hooks (reflective primitives) is a key aspect in the flexibility of Smalltalk and the ability to transparently integrate customizations.

We will quickly describe some of the key reflective facilities of Smalltalk, while specially focusing in this paper on their possible use for embedding concurrent and distributed computing systems into Smalltalk.

changing semantics of message passing – If a message is not understood by some receiver object, the message `doesNotUnderstand:` is sent to it with the unknown message as its argument. Default semantics is to open an error notifier, but any class may locally customize the behavior by redefining the corresponding method. This is specially useful for locally adapting the semantics of message passing to various contexts/mechanisms (asynchronism, multicast, transaction, e.g. in Section 4).

changing references to objects – Method `become:` exchanges (swaps) references between two objects. This primitive is also useful for preserving references when refactoring objects (as for instance when encapsulating them as explained above).

current context – Pseudo-variable `thisContext` references current context of execution (a context being a Smalltalk object). This actually provides an entry point to the execution stack. For instance, it is useful for customizing the debugger for concurrent asynchronous message passing.

changing the class of an object – Method `changeClassToClassOf:` changes the class of an object, and thus its behavior. This is another very useful primitive for evolution and integration of protocols (see in Section 4.1).

4 Experiences

Various frameworks for concurrent, parallel, and distributed programming have been developed with Smalltalk. They include: libraries, platforms,

[3] A class, being a "first-class" object, is itself instance of a class, named a *metaclass*. Metaclasses may be used to hold information about classes and basic representation of their instances. A more fine-grained meta-description may be individualized at the object-level instead (such object meta-description is called a *meta-object* [Mae87]). These are various degrees of the general concept of a *reflective* system, that is a computational system which may describe (and thus adapt) its own behavior.

and full systems/products. They all make use of the flexibility and richness of Smalltalk, and help at embedding or/and reusing standard programs with specific libraries and mechanisms for concurrency and distribution management. Below we will briefly describe two academic examples of such frameworks, namely Actalk: a platform for concurrent programming, and GARF: a platform for distributed and reliable programming.

There are also commercial products such as: the HP Distributed Smalltalk environment for developing (CORBA-compliant) distributed applications, and the OTI Envy Developer team development environment (see [Man 94] for related informations).

4.1 Actalk

Actalk [4] is a generic software platform/testbed for describing and classifying various (object-oriented) concurrent programming (OOCP) language constructs and mechanisms [Bri 94]. It has been used as a prototype foundation by various projects, e.g., simulation of software engineering process models, multi-agent systems applied to natural language processing, genome sequencing, knowledge acquisition...

The architecture of Actalk includes a kernel which models basic OOCP semantics (that is active/serialized objects which communicate by asynchronous/unidirectional message passing). The kernel is composed of a set of kernel component classes, each *component* describing a different aspect of an active object. Main components are: *behavior* (defining the actual behavior/program of the active object), *activity/synchronization* (defining the way method invocations are selected, scheduled, synchronized and computed), and *communication* (defining the way message transmissions will be interpreted, e.g., with or without reply, asynchronous or synchronous...). Each component class is itself further decomposed through "parameter methods", specifically intended to be redefined/specialized in subclasses in order to model alternative language designs. This relative independence of components allows the user to independently model or/and associate various aspects (activity, synchronization, communication) of the computation model/context to a given object/program.

Several extensions (subclasses) of the different kernel component classes have been developed to implement various *language models and constructs*, *communication models*, and *synchronization schemes* described in very progressive refinement steps [Bri 96]).

The implementation of the Actalk kernel relies at first on the redefinition of the `doesNotUnderstand:` method to transparently implement asynchronous buffered message passing (which may then be further specialized, as for instance to express eager reply, thanks to the standard class `Promise`). Buffering of incoming messages is expressed as a specialization of the standard class `SharedQueue`. Class `Message` may also be extended in order to include further information (e.g., sender of the message, or arrival time as to express priority-based synchronization algorithms [Bri 96]).

[4] *Actalk* stands for "*act*ive objects, or *act*ors, in Small*talk*-80."

Within the Actalk project, a generic time-slicing scheduler has been developed and integrated with standard Smalltalk-80 scheduler. In order to integrate this new scheduler with standard Smalltalk-80 virtual machine, and more specifically with the management of process suspension onto semaphores, the implementation may change dynamically and temporarily the behavior of semaphores when needed (thanks to the reflective primitive `changeClassToClassOf:`).

4.2 GARF

GARF [5] is an object oriented system aimed to support the design and the programming of reliable distributed applications on top of a network of workstations [GGM 94, GGM 95]. The specificity of GARF resides in its incremental programming model, and its extensible library of generic components. GARF has been written in Smalltalk and was first implemented on top of the Isis toolkit [BvR 93], then ported on top of Phoenix [6].

GARF promotes software modularity by clearly separating the behavioral features that concern concurrency, distribution and reliability, from functional ones than concern traditional sequential and centralized aspects. First, the GARF programmer may design and implement application components in a centralized environment, focusing only on their functionalities. In further steps, without modifying the previously written code, the programmer may turn to behavioral features, by expressing and controlling concurrency, distributing the application over a network, and replicating its critical components to increase the reliability. Of course, all these steps, including the first one, may be performed concurrently, but the code written for different steps is separated.

GARF handles two kinds of objects: *data* objects and *behavioral* objects. *Data objects* are used to describe the functional aspects of the application (i.e., sequential and centralized aspects). These objects are passive entities (i.e., standard Smalltalk program objects) that communicate in a *point-to-point, synchronous, request/reply* manner. *Behavioral objects* can be viewed as *"meta data objects"* [Mae 87], used to describe behavioral features (i.e., concurrency, distribution, and reliability) of data objects.

Dealing with the behavioral features of an application actually comes down to (dynamically) bind behavioral objects to data objects. GARF supports two types of behavioral objects: *encapsulators* and *mailers*. *Encapsulators* are used to *wrap* data objects by controlling the way they treat incoming and outcoming requests. *Mailers* are used to perform (remote) communications between encapsulators. GARF provides a library of encapsulator and mailer classes. For example, the encapsulator class `Replica` enables to create multiple replicas of a data object, and the mailer class `Abcast` ensures that all replicas of an encapsulator receive concurrent

[5] *GARF* stands for "*G*énération *A*utomatique d'*A*pplications *R*ésistantes aux *F*autes".

[6] Phoenix is an Isis-like toolkit, designed and implemented at EPF Lausanne, with large scale distribution in mind.

requests in the same order.

The "transparent" interception of messages was the key issue in order to reuse, as such, the functional code (i.e., the data objects) into a concurrent, distributed and reliable context (i.e., with behavioral objects). The interception of creation messages was achieved using the Smalltalk `Dictionary` and the `doesNotUnderstand:` operation. Another issue was the marshalling and unmarshalling of messages for remote communication. This was achieved using the BOSS (Binary Object Streaming Service) classes provided by Smalltalk.

5 Conclusion

This paper described the pros of using Smalltalk as a foundation for developing concurrent and distributed programming systems. Although Smalltalk was initially designed for mono-processor and mono-user environments, the combination of standard object-oriented features (encapsulation, genericity, reuse) with Smalltalk-specific richness of its libraries and the extreme flexibility of the language, makes Smalltalk a very good foundation. We quickly reviewed two concurrent and distributed programming systems developed in Smalltalk as examples for supporting our claim. We believe there will be a growing number of Smalltalk-based concurrent/distributed programming projects and products, as already witnessed by commercial products such as HP Distributed Smalltalk and Envy Developer.

References

[BvR 93] K. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Tookit*, IEEE Computer Society Press, 1993.

[Bri 94] J.-P. Briot, "Modélisation et Classification de Langages de Programmation Concurrente à Objets : l'expérience Actalk," *Colloque Langages et Modèles à Objets (LMO'94)*, also published as Rapport de Recherche LITP, n° 94.59, Paris, France, October 1994.

[Bri 96] J.-P. Briot, "An Experiment in Classification and Specialization of Synchronization Schemes," to appear in *2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, edited by K. Futatsugi and S. Matsuoka, LNCS, Springer-Verlag, March 1996.

[BG 96] J.-P. Briot and R. Guerraoui, "Objets pour la Programmation Parallèle et Répartie : Intérêts, Evolutions et Tendances," to appear in *Special Issue on "Systèmes à objets : tendances actuelles et évolution,"* edited by A. Napoli and J.-F. Perrot, *Technique et Science Informatiques (TSI)*, AFCET - Hermès, France, 1996.

[GGM 94] B. Garbinato, R. Guerraoui, and K.R. Mazouni, "Distributed Programming in GARF," In *Object-Based Distributed*

Programming, edited by R. Guerraoui, O. Nierstrasz, and M. Riveill, LNCS, n° 791, Springer-Verlag, 1994, pages 225–239.

[GGM 95] B. Garbinato, R. Guerraoui, and K.R. Mazouni, "Implementation of the GARF Replicated Objects Platform," (2) *Distributed Systems Engineering Journal*, 1995.

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

[Gue 95] R. Guerraoui, "Les Langages Concurrents à Objets" : *Technique et Science Informatiques (TSI)*, AFCET - Hermès, France, October 1995, pages 945–973.

[Mae 87] P. Maes, "Concepts and Experiments in Computational Reflection," *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Special Issue of ACM Sigplan Notices*, Vol. 22, n° 12, December 1987, pages 147–155.

[Man 94] S. Mann (edited by), *The Smalltalk Resource Guide*, Creative Digital Systems, San Francisco CA, USA, (`cds@netcom.com`), 1994.

[NWM 93] J. Nicol, T. Wilkes, and F. Manola, "Object-Orientation in Heterogeneous Distributed Computing Systems," *IEEE Computer*, Vol. 26, n° 6, June 1993, pages 57–67.

[Weg 87] P. Wegner, "Dimensions of Object-Based Language Design," *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87), Special Issue of ACM Sigplan Notices*, Vol. 22, n° 12, December 1987, pages 168–182.