Issues in Object-Oriented Parallel and Distributed Programming

Jean-Pierre Briot

LAFORIA, IBP, Case 169
CNRS - Univ. Paris-VI
4 place Jussieu, 75252 Paris Cedex 05
briot@laforia.ibp.fr

Abstract: This position paper discusses object orientation in parallel and distributed programming. One fundamental issue is: "How should we use object concepts to address parallelism and distribution concerns and mechanisms (e.g., concurrency control, distribution, and fault tolerance) ?". We may consider three main approaches for answering that question. The "applicative approach" applies object concepts, as they are, to structure parallel and distributed computing systems through libraries. The "integrative approach" aims at merging object concepts with parallelism and distribution concepts, e.g., by merging object and activity, or/and message passing and transaction. The "reflexive approach" may be considered as a mid way between the two previous approaches as it helps at transparently integrating various protocol libraries (to manage parallelism and distribution) within a programming language/system. We finally conjecture that, in the long run the reflexive approach is the most promising way to build future parallel and distributed programming systems.

Keywords: object, message passing, parallelism, distribution.

Objects as a Good Foundation

It is now well accepted that the notion of "object" provides good foundations for the new challenges of parallel, distributed and open computing. The notion of "object" (and of "message passing") helps at structuring and encapsulation of modules of computation, while it is flexible enough to match various granularities of software and hardware architectures. 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 [NWM93].

However, development of current object-oriented technology has until now mostly focused on sequential processing, for both technological and cultural reasons. Meanwhile the seminal concept of object, as introduced by Simula-67, was already showing promises for potential concurrency through its support for coroutines. Moreover, similarity between the notion of object and the notion of "process" (variables, persistent data, encapsulation, communication means) has been noted (e.g., in [MEY93]). Object concepts also exhibit great and natural potential for distributed applications. Message passing brings independence of the localization of an object. Self-containedness of objects (data plus procedures, plus possible internal activity) also eases the issue of moving (migrating) them around. Also note that the client-server architecture currently the basis of most of distributed systems and human-computer interfaces provides some of the characteristics of object-orientation. (However in the object world, the client/server dichotomy is not static, as every object may be alternatively considered as a client or a server while it sends or receives messages.)

Matching Parallelism and Distribution Concerns

Although object concepts provide a good foundation, current object-oriented technology (and methodology) does not directly address concurrency, coordination and distribution issues. As noted above, object-oriented technology has been mostly restricted to sequential computation. Meanwhile various communities (parallel programming, operating systems, distributed computing, data-bases...) have designed various abstractions and mechanisms (e.g., synchronization schemes, capabilities, atomicity, transactions...) to address parallelism and distribution concerns (concurrency control, resources management, fault-tolerance...). A fundamental question is: "How should we use object concepts to address parallelism and distribution concerns and mechanisms ?". In other words: "How should we articulate (apply, integrate, expand) object concepts with current parallelism and distribution technology ?".

By analyzing current experience and trends (as for instance reported in [GNR94]), we may classify three main approaches for answering this question [BG95]. The "applicative approach" applies object concepts, as they are, to structure parallel and distributed computer systems through class libraries. The "integrative approach" aims at merging object concepts with concurrency and distribution concepts and mechanisms, notably in unifying object and activity, or/and message passing with transactions. The "reflexive approach" may be considered as a mid way between the two previous approaches as it helps at transparently integrating various computing protocol libraries within a programming language/system.

The Applicative Approach

The basic idea of the "applicative approach" is to apply object concepts as a structuring tool to build parallel and distributed computing systems. A representative example is the Choices object-oriented generic and distributed operating system. It defines abstract notions of components (such as file format, communication network, and memory model) as abstract classes, which may then be specialized as concrete subclasses. The Smalltalk-80 programming language and environment is another good example of this approach where a basic and simple object concept is uniformly applied to model and structure the whole system through class libraries. For instance concurrency aspects are defined as classes (process, scheduler, semaphore...). (But note that the notion of object and process are kept distinct.) HP Distributed Smalltalk provides a OMG (CORBA) compliant set of distributed services, themselves implemented in Smalltalk-80.

The Integrative Approach

The main limitation of the applicative approach is that, although it helps at structuring concepts and mechanisms for parallel and distributed programming, it keeps them disjoint from the objects structuring the application programs. In other words, the programmer still needs to have at least two different tasks: programming with objects, and managing parallelism and distribution of the program, also through objects, but not the same objects ! Rather than keeping these dimensions orthogonal, the "integrative approach" aims at merging these various concerns into the same objects for the programmer.

Various levels of integration are possible. A first level of integration is the notion of active object, unifying object with activity. Actor languages are a seminal example. A second level of integration is the notion of synchronized object, expressing synchronization of programs at the level of objects (e.g., by attaching guards to methods) and message passing (e.g., by supporting both synchronous and asynchronous invocations). Guide is an example of programming language based on the notion of synchronized object but which keeps object and activity distinct. A third level of integration is the notion of distributed object, seeing objects as the unit of distribution and possible replication, and using the message passing metaphor as a transparent way of invoking either local or remote objects. The Emerald distributed programming language is an example of that last kind. One can also further integrate message passing with the transaction concept as to support inter-object synchronization and fault-tolerance.

Although the integrative approach is very appealing by its unification goals, it unfortunately also has its limitations. One reason behind is that object-oriented technology have been mostly developed in a sequential and not distributed computing context. Some of the mechanisms earlier developed may show their limits once put in a wider context where previous assumptions may not hold anymore. As for instance the use of inheritance as a tool to inherit synchronization specifications proved to be not trivial (the so-called "inheritance anomaly phenomenon"). Also the notion of class variable, as introduced in several systems, following Smalltalk-80, makes it difficult to be distributed. Another possible limitation of the integrative approach is that one could lose some flexibility and moreover efficiency through some too systematic unification of concepts. For instance, stating that every object is active, or/and every message transmission is a transaction, may be inappropriate for some applications not necessarily requiring such protocols and their associated computational load. In other words it would be interesting to keep the unification/simplification advantages of the integrative approach while retaining the flexibility of the applicative/library approach.

The Reflexive Approach

The "reflexive approach" appears as a promising step in that direction. "Reflection" is a general methodology to control the behavior of a computational system. The basic idea is to provide a representation of the important characteristics/parameters of the system in terms of the system itself. By writing programs changing these characteristics/representations, one may then change the behavior of the system itself. Such "meta-programs" are used to control the behavior of application programs by changing the computing characteristics. Examples are in transparently changing policies for data representation, communication, scheduling, load balancing, adding protocols, optimization... with no or very little change to the application programs. Note that the reflective approach is close to the applicative approach in that (meta)-libraries are designed to express and change various aspects of parallelism (computation model, communication) and distribution (protocols, resources management) aspects. It is also close to the integrative approach in that these libraries are intimately integrated with the programming language or system so that they may transparently change its behavior.

Reflection fits particularly well with object concepts, as objects encapsulate local specializations (meta-programs associated to a specific meta-object are thus called "meta-objects"). Also class and inheritance allow to organize various specializations. One example of application is to add fault-tolerant protocols, such as atomic transactions, by changing meta-objects in charge of communication, and only local to objects where needed. Note that reflection is also good at expressing resources control not only at the level of one object, but in a larger context (as for instance scheduling activities on one processor, load balancing objects onto various nodes...). Such policies are open for customization with the whole expressive power of the programming language. As for instance it is possible to control the decision for migrating or not an object (and to which node) along various considerations (various processor loads, ratio of remote communications...). Apertos is a pioneering example of adaptive distributed operating system based on reflection.

In summary reflection is a very effective methodology for decoupling application programs with the specification of their computational context and protocols. Reflection is also a promising methodology for addressing the evolutive aspects of computing components for future distributed open systems. These object components should become more flexible and autonomous as to be able to constantly readjust their acquaintances and coordination with other evolutive components (this is related to the recent concept of "software agents"). One current problem with reflection is efficiency, as meta-interpretation tends to add extra computation load. Partial evaluation is currently being investigated as a promising technique to reduce such extra interpretation costs. Meanwhile, we now need more experience in the practical use of reflection to be able to find good tradeoffs between the flexibility required, the architecture complexity, and the resulting efficiency.

Conclusion

In the long run, we conjecture that the reflective approach (combined with object concepts) provides the most promising framework for developing future parallel and distributed computing systems. By allowing high-level and transparent mapping of various mechanisms and protocols onto application programs, it helps at designing and adapting parallel and distributed computing systems, taken in their globality (blurring the frontier between programming language, operating system, communication network, and data-base).

Acknowledgement

This short position paper has been based on a joint article/work with Rachid Guerraoui [BG96].

References

[BG96] J.-P. Briot and R. Guerraoui. "Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances." (in french.) 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, 15(6):765–800, June 1996.

[GNR94] Proceedings of the "ECOOP'93 Workshop on Object-Based Distributed Programming." Edited by R. Guerraoui, O. Nierstrasz and M. Riveill, LNCS, No 791, Springer-Verlag, 1994.

[MEY93] Special Issue on "Concurrent Object-Oriented Programming." Edited by B. Meyer, CACM, Vol. 36, No 9, September 1993.

[NWM93] J.R. Nicol, C.T. Wilkes and F.A. Manola. "Object Orientation in Heterogeneous Distributed Computing Systems." IEEE Computer, pages 57-67, June 1993.