Editor: Steve Vinoski • vinoski@ieee.org

# A Smooth Concurrency Revolution with Free Objects

**Rachid Guerraoui** • *Ecole Polytechnique Fédérale de Lausanne, Switzerland*

Chip manufacturers are currently talking about potentially doubling efficiency on a 2x core, quadrupling on a 4x core, and so forth; some programs even run 18 times faster on a 32-core machine. In the past, it was typical to talk about 5 to 10 percent improvements by parallelizing parts of an application, not 400 to 1,800 percent. For a change, this is a change. Yet, multicore is useless without concurrent programming. A single-threaded application can exploit at most 1 percent of the potential throughput of a 100-core chip, so the reigning software development advice that "processors just keep getting faster, and we can get the benefit without doing anything" is over. Multicore chip manufacturers are calling for a new software revolution: the concurrency revolution. At first glance, this might seem surprising, for concurrency is almost as old as computing, and many concurrent programming models and languages have been invented over the years. In fact, this revolution is about more than concurrency alone, but rather concurrency for the masses.

Most of the many concurrent programming models that have been invented were devised by concurrency "aristocrats" for concurrency aristocrats. These aristocrats were born and raised in a world of threads, wait-free computing, active objects, and actor models. They still leverage their educations to make their living and have the luxury of devoting their entire lives to improving specific concurrency constructs or algorithms. But not everyone is so lucky. To get concurrency to the masses, we need a simple way to exploit parallelism with languages that common developers are already used to. In particular, many developers have adopted object-oriented (OO) development over the past 15 to 20 years, so asking them to completely revise their educations to become comfortable with concurrency would be problematic. It might instead be more pragmatic to explore ways to implicitly extract parallelism from current OO programs by overloading OO constructs. I will illustrate this idea here by briefly describing one such way.

In short, I propose viewing a program as a collection of "free" objects, each of which has its own thread of control and executes its operations within atomic transactions. Such objects would communicate by asynchronous message passing with *futures* — objects that encapsulate the results of server invocations and let clients retrieve them while hiding the actual state of readiness from the client. Except for the unleashing of newly created free objects, there's no need for specific language constructs other than those already available in OO languages. After all, C++ has been successful in part because it looked like C, Java has been successful in part because it looked like C++, and C# looks like both. With free objects, the program looks exactly like a traditional OO program, with one exception: certain object creations fork new activity threads and must be distinguished as such. This column discusses how to do that.

## The Exciting Life of a Concurrency Aristocrat

Edsger Dijkstra, C.A.R. Hoare, and other famous computer scientists who shaped the field devoted much of their time to studying concurrency. Since then, many researchers, including myself, have made a living from devising programming models and algorithms for concurrent computing. Perhaps because such models and algorithms had to be published and made good cases for PhDs and faculty positions, they were (and needed to be) complicated. Yet, it's hard to imagine using any of these concurrency models for general purpose programming, at least not now.

We can very roughly classify the models that researchers have studied as

 1089-7801/07/$25.00 © 2007 IEEE 

- threads and locks,
- nonblocking data structures,
- transactions,
- actors, and
- active objects.

As I'll discuss, these models are powerful but hard to use.

### Threads and Locks

With this model, programs are structured as sets of threads that concurrently access shared data structures protected by locks. Threads are mapped to processors according to some strategy that typically depends on the actual operating system and hardware architecture. Although this is probably the most popular model for concurrent programming, it's still restricted to specific programs and raises several tricky questions, including when to fork a thread exactly, how to decompose a program into concurrent threads, and when to acquire locks. Coarse-grained locking leads to inefficient solutions, whereas fine-grained locking leads to error-prone programs. Moreover, locks are not robust: a thread that's swapped out while holding a lock prevents all other threads contending for the same data structure from making any progress. Doug Lea's excellent book, *Concurrent Programming in Java* (Addison-Wesley, 1996), provides good hints on how to handle this issue, but determining good locking strategies remains challenging.

### Nonblocking Data Structures

In this model, programs are structured as sets of threads that synchronize their activities using nonblocking data structures. This characteristic circumvents locking's robustness issue. The magic lies in making use of (or rather, assuming) low-level atomic objects, such as registers and `compare&swap` operations, typically provided in hardware. Devising nonblocking data structures is nontrivial; indeed, efficient implementations of such structures are publishable as results in conferences such as ACM's Conference on Principles of Distributed Computing (PODC) and the International Symposium on Distributed Computing (DISC). Whereas it's reasonable to assume that expert programmers can devise and use libraries of nonblocking data structures — Maurice Herlihy and Nir Shavit's soon-to-be-published *The Art of Multiprocessor Programming* will certainly help — but it's unrealistic to ask common developers to turn all their potentially shared classes into nonblocking ones.

### Transactions

Transactions have been around for some time, and not just in databases — Barbara Liskov incorporated them in her Argus language two decades ago, for example. Yet, transactions are generating a lot of interest today because they address some of the problems I've mentioned. In fact, one way to view a transaction is as a simple way to make a block of code atomic and nonblocking. Basically, a transaction is a logical unit of computation that can be safely aborted. Hence, we can implement transactions without underlying locks, simply aborting and retrying them in case strong suspicions indicate that a deadlock has occurred or a thread participating in the transaction has been swapped out. Given that this can lead to *live-locks*, in which transactions mutually abort each other, interesting ongoing research is exploring how to devise contention managers that can adopt strategies to retry aborted transactions at the right time or with the right priorities, based on the transaction's age or how many objects it has actually accessed — a transaction that started earlier or accessed a larger number of objects should have a higher priority. Transaction models, however, leave open the questions of how to decompose an application into a set of threads and how to demarcate transaction boundaries.

**One way to view a transaction is as a simple way to make a block of code atomic and nonblocking. A transaction is a logical unit of computation that can be safely aborted.**

### Actors

Following an anthropomorphic direction initiated by Agha and Hewitt more than two decades ago, developers using this approach build concurrent programs as sets of communicating actors. This activity includes no explicit thread forking; instead, each actor implicitly has its own thread and communicates asynchronously with other actors by exchanging messages. When it receives a message, an actor executes some behavior, possibly sends a message to another actor, and then becomes a new actor. Making the last operation the unique point of state change significantly simplifies concurrency control within an actor: the actor is, in this sense, stateless. However, programming an application in terms of asynchronous messaging and stateless objects isn't trivial — at least, the gap between this model and a typical OO program isn't clear. How do you transform a program made of stateful sequential objects interacting in a synchronous manner into a system of asynchro-

nously communicating stateless actors? How do you group several actors inside the same logical entity without systematically creating bigger actors to encompass the entire computation — for example, programming an atomic transaction that withdraws and then deposits money inside two different actors? There are no simple answers to these questions.

### Active Objects

Unlike an actor, an active object is typically stateful. I say "typically" because there have been almost as many active object models as papers on the subject — and there were indeed many papers on this topic 10 years ago at the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) and the European Conference on Object-Oriented Programming (ECOOP). Moving from a sequential OO program to a program of (stateful) active objects is quite natural with few exceptions. Because the active object is stateful, having it explicitly control its concurrency to avoid hampering encapsulation is tempting. For instance, we might want to enrich an object's interface to express patterns such as "certain operations cannot execute concurrently" or "some operation cannot execute in a certain state." We might also express the fact that a `get` operation can't be executed on an empty buffer. As Satoshi Matsuoka and Akinori Yonezawa pointed out,[1] this poses further problems when extending interfaces. For example, to add a `gget` operation (say, through inheritance) that can be executed only after exactly one `get`, we must modify the original `get` to recall exactly when it was executed. On the other hand, although an object operation's atomicity seems easy to ensure if the operation is purely local, the situation is more complicated if the operation involves several other active objects.

## A Smooth Revolution with Free Objects

To smoothly integrate concurrency within an object model, I propose unifying the concepts of object, thread, and transaction in what I call here a free object. To explain how such a model would look, let's consider how to address two questions.

**To smoothly integrate concurrency within an object model, I propose unifying the concepts of object, thread, and transaction in what I call here a free object.**

### Which Objects Are Free?

We could imagine three different ways to distinguish between objects that are free and those that are not:

- *All objects of certain classes or types are free.* The programmer would have to highlight which classes/types of objects are free — say, by making them inherit from some specific ancestor class/type.
- *The creator of the object decides whether the latter is free.* The creator can decide whether to "unleash" the new object or not using a specific `new` construct, say `fnew`, provided as a primitive of the language.
- *In an ideal world, the newly created object would decide whether it was free or not.* We could use specific constructors to achieve this.

To understand the free object model, another question is in order.

### What Does a Free Object Do with Its Freedom?

When a free object is instantiated, the run-time system automatically creates a new thread and a new transaction, both of which are associated with the object. The transaction terminates when the object constructor terminates. The thread can then serve incoming requests, if any. These requests are executed in the context of other transactions.

An embarrassingly trivial parallel program would simply consist of a set of free objects that didn't communicate. Things would get slightly more complicated if they needed to synchronize, so my proposal is to have such objects interact (and hence synchronize) via message passing, as in any OO language. The major twist is to overload the message-passing semantics using atomicity, asynchrony, and futures. (This is similar to the idea underlying the Argus language, except that the latter had atomic synchronous invocations. I later extended that concept in the ACS system to account for asynchrony; my proposal for interacting free objects is along those lines.[2])

Every invocation creates two concurrent subtransactions of the transaction running on the client object. The first subtransaction executes the invoked operation while the second continues the computation at the client. Unless a reply is expected from the invoked operation, the two subtransactions execute concurrently and each terminates at the end of its operation. If a reply is expected, the runtime system uses a future object to keep the invocation transparent. The synchronization between the client and the server occurs exactly when the

value of the reply is retrieved — right before accessing this value.

Disclaimer: I include no code examples because the point isn't to introduce new language constructs. Indeed, we could apply the idea of free objects to any OO language as is. Behind the scenes, of course, the language implementer should provide a transactional system and an asynchronous messaging system, which developers have worked hard at implementing in popular OO languages.

Very roughly speaking, we can view the model I propose here as the result of the following straightforward unification as a key to leveraging the increasing parallelism provided by the hardware:

```
data + functions + thread +
transaction = free object
```

Unifying data and function was the key principle underlying abstract data types and object-oriented languages. Further encompassing threads led to active objects and actor models. As discussed earlier, such models are appealing for their anthropomorphism but leave the synchronization problem to the developer. Encompassing transactions provides the developer with a simple solution to that. After all, Simula, the first OO language, had implicit concurrency as a first-class citizen.

Of course, it doesn't take long to figure out examples in which such unification would be restrictive. The compiler could help to figure out which operations don't really need to remain within the same transaction and which objects should actually be free. We could even imagine linguistic constructs that would let a concurrency aristocrat fork several threads within a single free object or break transactions into pieces.

But the bottom-line question is really whether to propose to the programmer a simple unified default model, with the ability for concurrency aristocrats to use advanced constructs to enhance performance, or a rich model enabling a priori all kinds of concurrency schemes. Trying to address this question might not be very productive, but exploring models in both directions seems interesting.

**Acknowledgments**

Anne-Marie Kermarrec (IRISA), Michal Kapalka (EPFL), and Steve Vinoski were kind enough to provide useful comments in a very (very) short period of time.

**Reference**

1. S. Matsuoka and A. Yonezawa, "Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages," *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993, pp. 107–150.
2. R. Guerraoui et al., "Nesting Actions through Asynchronous Message Passing: the ACS Protocol," *Proc. European Conf. Object-Oriented Programming* (ECOOP 92), LNCS 615, Springer, 1992, pp. 170-184.

**Rachid Guerraoui** is a professor of computer science at Ecole Polytechnique Fédérale de Lausanne, Switzerland. His research interests include distributed algorithms and programming languages. Guerraoui has a PhD in computer science from the University of Orsay and has been affiliated with HP Labs and MIT. He is coauthor of *Introduction to Reliable Distributed Programming* (Springer-Verlag, 2006). Contact him at rachid.guerraoui@epfl.ch.