Introduction

Rachid Guerraoui[‡] Michel Raynal[⋆]

[‡] School of Computer and Communication Sciences, EPFL, Switzerland
[⋆] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
rachid.guerraoui@epfl.ch raynal@irisa.fr

October 1, 2007

1 The broad picture

The field of concurrent computing has gained a huge importance after major chip manufacturers announced their switch of focus from increasing the speed of individual processors to increasing the number of processors on a chip. In the old good times, nothing needed to be done to boost the performance of programs, besides changing the underlying processors with more recent ones. Those times are over. To exploit multicore architectures, programs will have to be devised in a parallel manner. For instance, a single-threaded application can exploit at most 1/100 of the potential throughput of a 100-core chip and such a chip might be available before this book is edited. Chip manufacturers are calling for a new software revolution: the *concurrency revolution*.

Talking about a new revolution might look surprising at first glance for concurrency is almost as old as computer science. Indeed, the famous computer scientists such as Dijkstra and Hoare that shaped up the field of computing have devoted a large amount of their time studying concurrency. The first object-oriented programming language, Simula, has inherent support for concurrency.

In fact, what the revolution is about is more than concurrency alone: it is about *concurrency for the masses*. In short, concurrency is going out of the small box of specialist programmers and is conquering the masses. The challenge is to come up with abstractions that such programmers can easily use for general purpose concurrent programming. In particular, designing and implementing abstractions to enable interprocess synchronization is crucial. In a way, whereas *forking threads* is relatively easy, *synchronizing* their activities is usually much more complicated. This is precisely the topic of this book.

2 The topic

In concurrent computing, a problem is solved through a set of processes that represent each a different flow of control and execute different tasks. Except in embarrassingly parallel programs where the tasks are completely separated, the tasks of common parallel programs need sometimes to synchronize their activities through shared elements. It is good practice to view these elements as instances of abstract data types, accessible through a clean interface exporting a set of operations. This interface is itself defined by a specification that captures the semantics of the operations and the way these have to be used.

This book studies algorithms that implement such *shared* objects in a *resilient* manner. Roughly speaking, "resilience" encompasses two aspects:

- The first aspect of resilience is related to asynchrony. No process p ever prevents any other process q from making progress when q executes an object operation on shared object X. This means that, provided it remains alive and kicking, q terminates its operation on X despite the speed or the failure of any other process p. Process p could be very fast and might be permanently accessing shared object X, or could have been swapped out by the operating system while accessing X. None of these situations should prevent p from executing its operation. This property of object implementations is called *wait-freedom*. As we will explain later in this chapter, this property transforms the difficult problem of reasoning about a failure-prone concurrent system where processes can be arbitrarily delayed and swapped-out (or paged-out) by the operating system, into the simpler problem of reasoning about a failure-free concurrent system where every process progresses at its own pace and runs to completion.

- The second aspect of resilience is related to concurrency. Despite spanning overlapping intervals of time, the operations issued on each object appear as if they are executed each instantaneously, one after the other. In fact, each operation *op* on an object X appears to take effect at some indivisible instant between the invocation and the reply times of *op*. This property, called *atomicity*, transforms the difficult problem of reasoning about a concurrent system into the simpler problem of reasoning about a sequential one. In order to understand the concurrent semantics of a set of atomic objects potentially shared by a number of parallel programs, it is enough to understand the sequential specification of each individual object.

This book studies *wait-free implementations of atomic objects*. Basically, given certain shared objects of *base* types, say provided in hardware, we will study how and whether it is at all possible to wait-free implement in software an atomic object of a *more powerful* type. In fact, and strictly speaking, when we talk about implementing an object, we actually mean devising an algorithm that implements the operations exported by its type. As we shall see, ensuring each of atomicity or wait-freedom alone is trivial. To ensure atomicity alone, it is enough for the implementation to never return any response to an operation. To ensure wait-freedom alone. The challenge is to ensure both at the same time.

The material of the book is presented in an incremental manner. We first define atomicity and wait-freedom, and then we show how to implement simple shared objects from even simpler ones, and then progressively how to use the objects resulting from those implementations to build even more powerful objects.

3 Content of the book

We overview in the following the fundamental concepts mentioned above and to which this book is devoted. In particular, we argue why it is important to study these concepts.

3.1 Shared objects as concurrency abstractions

Defining and implementing programming abstractions that capture recurrent algorithmic patterns are among the main challenges of computer science. A file, a stack, a record, a list, a queue and a set are well-known examples of abstractions that have proved to be valuable in traditional sequential and centralized computing.

In modern computing, an abstraction is usually captured by an object representing a server program that offers a set of operations to its users. These operations and their specification define the behavior of the object, also called the *type* of the object. The way an abstraction (object) is implemented is usually hidden to its users who can only rely on its operations and their specification to design and produce the program making use of that object. Such a modular approach is key to implementing provably correct software that can be reused by subsequent programmers.

The aim of the book is to study abstractions for *concurrent* computing, in the form of *shared* objects, i.e., objects that can be accessed by concurrent processes. That is, the operations exported by the shared object can be accessed by concurrent processes. Each process accesses the shared object in a sequential manner. Roughly speaking, sequentiality means here that, after it has invoked an operation on an object, a process waits to receive a reply indicating that the operation has terminated, and only then is allowed to invoke another operation on the same or a different object. The fact that a process p is executing an operation on a shared object X does not however preclude another process q from invoking an operation on the same object X.

3.2 Atomicity

Atomicity, also called *linearizability*, means that each object operation appears to execute at some indivisible point in time, also called *linearization* point, between the invocation and reply time events of the operation. Atomicity provides the illusion that the operations issued by the processes on the shared objects are executed one after the other. To program with atomic objects, the developer simply needs the *sequential specification* of each object, called also its sequential type or simply its type, which specifies how the object behaves when accessed sequentially by the processes.

Most interesting synchronization problems are best described as atomic objects. Examples of popular synchronization problems are the *reader-writer* and the *producer-consumer* problems. In the reader-writer problem, the processes need to read or write a shared data structure such that the value read by a process at a given point in time t is the last value written before t. Solving this problem boils down to implementing an atomic object exporting `read()` and `write()` operations. Such an object type is usually called an atomic variable or a `register`. It abstracts the very notions of shared file and disk storage.

In the producer-consumer problem, the processes are usually split into two camps: the producers which create items and the consumers which use the items. It is typical to require that the first item produced is the first to be consumed. Solving the producer-consumer problem boils down to implementing an atomic object type, called a FIFO `queue` (or simply a queue) that exports two operations: `enqueue()` (invoked by a producer) and `dequeue()` (invoked by a consumer).

3.3 Wait-freedom

Traditional synchronization algorithms rely on *mutual exclusion* (typically based on *locking* primitives): critical shared objects (or critical sections of code within shared objects) are accessed by processes one at a time. No process can enter a critical section if some other process is in that critical section. We also say that a process has acquired a *lock* on that object (resp., critical section). This technique is *safe* in the sense that it ensures atomicity and protects the program from inconsistencies due to concurrent accesses to shared variables.

However, coarse-grained mutual exclusion does not scale and fine-grained mutual exclusion can easily lead to violate atomicity. Indeed, atomicity is only ensured if indeed all related variables are protected by the same critical section. This significantly limits the parallelism and thus the performance of the program, unless the program is devised with minimal interference among processes. This, on the other hand, is nevertheless hard to expect from common programmers and precludes most legacy programs.

Maybe more importantly, mutual exclusion hampers progress since a process delayed in a critical section prevents all other processes from entering that critical section. Delays could be significant and especially when caused by crashes, preemptions and memory paging. For instance, a process paged-out might be delayed for millions of instructions, and this would mean delaying many other processes if these want to enter the critical section held by the delayed process.

Wait-free implementations of atomic objects provides an alternative to mutual exclusion-based implementations. In particular *wait-freedom* precludes any form of blocking. In short, wait-freedom stipulates that, unless it stops executing (say it crashes), any process that invokes an object operation eventually obtains a reply. That is, the process calling the operation on the object (to be implemented), should obtain a response for the operation, in a finite number of its own steps, independently of concurrent steps from other processes. The notion of step means here a local instruction of the process, say updating a local variable, or an operation invocation on a base object used in the implementation, say provided in hardware. Sometimes we will assume that the object to be implemented should tolerate a certain number of base object failures. That is, we will seek to implement objects that are resilient in the sense that they eventually return from process invocations, even if the underlying base objects fail and do not return, or return useless replies.

3.4 Object implementation

This book studies how to wait-free implement a certain atomic object out of a given set of base objects. The notion of implementation has to be considered here in the *algorithmic* sense (there is no promise of Java code in this book).

An object to be implemented is typically called *high-level*, in comparison with the objects used in the implementation, considered at a *lower-level*. This notion of level is clearly relative as the same object can be a different levels in different implementations. It is common to talk about *emulations* of the high-level object using the low-level ones. Unless explicitly stated otherwise, we will by default mean *wait-free implementation* when we write *implementation*, and *atomic object* when we write *object*.

It is often assumed that the underlying system model provides some form of *registers* as base objects. These provide the abstraction of read-write storage elements. Message passing systems can also, under certain conditions, emulate such registers. Sometimes the base registers that are supported are atomic but sometimes not. As we will see in this book, there are algorithms that implement atomic registers out of non-atomic base registers that might be provided in hardware.

Some multiprocessor machines do also provide objects that are more powerful that registers like *test&tet* objects or *compare&swap* objects. Intuitively, these are more powerful in the sense that the writer process does not systematically overwrite the state of the object, but specifies the conditions under which this can be done. Roughly speaking, this enables more powerful synchronization schemes than with a simple register object. We will capture the notion of "more powerful" more precisely later in the book.

Not surprisingly, a lot of work has been devoted to figure out whether certain objects can implement other objects. As we have seen, focusing on wait-free implementations clearly excludes mutual exclusion based approaches, with all its drawbacks. From the application perspective, and as we pointed out, there is a clear gain because relying on wait-free implementations makes it less vulnerable to failures and deadlocks. However, the desire for wait-freedom makes the design of atomic object implementations subtle and difficult. This is particularly so when we assume that processes have no information about their respective interleaving: this is the model we will assume by default in this book.

3.5 Reducibility

In its abstract form, the question we address in this book, namely of implementing high level objects using lower level objects, can be stated as a general *reducibility* question in the parlance of the classical theory of computing. Given two object types $X1$ and $X2$, can we implement $X2$ using any number of instances of $X1$ (we simply say using $X1$)? In other words, is there an algorithm that implements $X2$ using $X1$? The specificity of concurrent computing here lies in the very fact that, under the term "implementing", lies the notions of atomicity and wait-freedom: we want to implement atomic objects and in a wait-free manner. These notions encapsulate the smooth handling of concurrency and failures.

If the answer to the reducibility question is negative, then it is also interesting to ask what is needed (under some minimality metric) to add to the base objects in order to implement the desired high level object. For instance, if the base objects provided by a given multiprocessor machine are not enough to implement a particular object, knowing that extending the base objects with another specific object (or many of such objects) is sufficient, might give some useful information to the designers of the new version of the multiprocessor machine in question.

4 Content and organization

The book is organized in an incremental way, going from implementing simple objects from even simpler ones, to implementing more powerful objects. After precisely defining the notions of atomicity and wait-freedom, we go through the following steps.

1. We first study how to implement atomic registers objects out of non-atomic base registers. Roughly speaking, assuming as base objects registers that provide weaker guarantees than atomicity, we show how to wait-free implement atomic registers. Furthermore, we also show how to implement registers that can contain an arbitrary large range of values, and be read and written by any process in the system, from single bit registers (i.e., that contain only 0 or 1) that can be accessed by only one writer process p and only one reader process q.

2. We then discuss how to use registers to implement seemingly more sophisticated objects than registers, like *counters* and *snapshot* objects. We contrast this with the inherent limitation of registers to implement more powerful objects like *queues*. This limitation is highlighted through the seminal *consensus impossibility* result.

3. We then discuss the importance of consensus as an object type, by explaining its *universality*. In particular, we describe a simple algorithm that uses registers and consensus objects to implement any other object. Then, we turn to the question on how to implement a consensus object from other objects. In particular, we describe an algorithm to implement a consensus object in a system of two processes, using registers and either test&set or queue objects, as well as an algorithm that implements a consensus object using a compare&swap object in a system with an arbitrary size. The difference between these implementations is highlighted to introduce the notion of *consensus number*.

4. We then study a complementary way of implementing consensus: using registers and some additional assumptions about the way processes access these registers. More precisely, we make use of an oracle that reveals information about the operational status of the processes accessing the shared registers. We discuss how even an oracle that is unreliable most of time can help devise a consensus algorithm, and hence any other object. We also discuss the implementation of such an oracle assuming that the computing environment satisfies additional assumptions about the scheduling of the processes. This may be viewed as a slight weakening of the wait-freedom requirement which requires progress no matter how processes interleave their steps.

5. We then consider the question of implementing objects out of base objects that can fail. This issue can be of practical relevance in a distributed multi-core architecture where it is reasonable to assume that certain base objects might fail. It also abstracts the problem of implementing a highly available storage abstraction in a storage area network where basic units (files or disks) can fail. Not surprisingly, the general way to achieve resilience is replication, but the underlying approach depends on the failure model. We distinguish two canonical failure models. First, we consider a failure model where a base object that fails keeps on returning a specific value $\perp$ whenever it is invoked. This model is called the *responsive* failure model. Then we look at another failure model where a base object that fails stops replying. This model is called the *non-responsive* failure model. As we will see, algorithms that tolerate the first form of failures are usually sequential whereas those that tolerate the second form of failures are usually parallel ones.

6. Finally, we revisit some of the implementations given in the book by giving up the assumption that processes do have unique identities. We study here *anonymous* implementations. We give anonymous implementations of a weak counter object and a snapshot object based on registers.

5 Bibliographical notes

The fundamental notion of abstract object type has been developed in various textbooks on the theory or practice of programming. Early works on the genesis of abstract data types were described in [3, 12, 16, 17]. In the context of concurrent computing, one of the earliest work was reported in [8, 15] and more recently in [2].

The notion of register (as considered in this book) and its formalization are due to [11]. A more hardware-oriented presentation was given in [14]. The notion of atomicity has been generalized to any object type in [7] under the name linearizability. The concept of snapshot object has been introduced in [1]. A theory of wait-free atomic objects was developed in [9].

The mutual exclusion problem has been introduced in [4]. The problem constituted a basic chapter in nearly all textbooks devoted to operating systems. There was also an entire monograph solely devoted to the mutual exclusion problem [20]. Various synchronization algorithms are also detailed in [21].

The notion of wait-free computation originated in [10], and was then explored in [19]. It has then been generalized and formalized in [6].

The consensus problem was introduced in [18]. Its impossibility in asynchronous message-passing systems prone to process crash failures has been proved in [5]. Its impossibility in shared memory systems was proved in [13]. The universality of the consensus problem and the notion of consensus number were investigated in [6].

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890, 1993.

[2] Brinch Hansen P. (Editor), The Origin of Concurrent Programming. *Springer Verlag*, 534 pages, 2002.

[3] Dahl O.-J., Dijkstra E.W.D. and Hoare C.A.R., Structured Programming. *Academic Press*, 220 pages, 1972.

[4] Dijkstra E.W.D., Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[5] Fisher M.J., Lynch N.A. and Paterson M.S., Impossibility of distributed consensus with one faulty-process. *Journal of the ACM*, 32(2): 374-382, 1985.

[6] Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

[7] Herlihy M.P. and Wing J.M, Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[8] Hoare C.A.R.,, Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549-557, 1974.

[9] Jayanti P., Chandra T. and Toueg S., Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451-500, 1998.

[10] Lamport L., Concurrent reading and writing. *Communications of the ACM*, 20(11):806-811, 1977.

[11] Lamport. L., On interprocess communication, part 1: basic formalism, Part II: algorithms. *Distributed Computing*, 1(2):77-101, 1986.

[12] Liskov B. and Zilles S., Specification Techniques for Data Abstraction. *IEEE Transactions on Software Engineering*, SE1:7-19, 1975.

[13] Loui M.C. and Abu-Amara H.H. Memory requirements for agreement among unreliable synchronous processes. *Advances in Computing Research*, 163-183, 1987.

[14] Misra J., Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142-153, 1986.

[15] Owicki S. and Gries D., Verifying Properties of Parallel Programs: An Axiomatic Approach. *Commununications of the ACM*, 19(5): 279-285, 1976.

[16] Parnas D.L., A Technique for Software Modules with Examples. *Commununications of the ACM*, 15(2):220-336, 1972.

[17] Parnas D.L., On the Criteria to be Used in Decomposing Systems in to Modules. *Commununications of the ACM*, 15(12):1053-1058, 1972.

[18] Pease L., Shostak R. and Lamport L., Reaching agreement in presence of faults. *Journal of the ACM*, 27(2):228-234, 1980.

[19] Peterson G.L., Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46-55, 1983.

[20] Raynal M., Algorithms for mutual exclusion. *The MIT Press*, ISBN 0-262-18119-3, 107 pages, 1986.

[21] Taubenfeld G., Synchronization algorithms and concurrent programming. *Pearson Prentice-Hall*, ISBN 0-131-97259-6, 423 pages, 2006.