Lectures Notes on Concurrent Computing

October 17, 2011

Contents

Chapter 1

Introduction

1.1 A 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. The old good times where nothing needed to be done to boost the performance of programs, besides changing the underlying processors, are over. To exploit multi-core architectures, programs 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*.

This might look surprising at first glance for concurrency is almost as old as computer science. The famous computer scientists that shaped up the field of computing have devoted a large amount of their time studying concurrency, including mainly Dijkstra and Hoare. In fact, the revolution is more than about 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 inter-process synchronization is crucial. Moreover, given that the contribution of synchronization mechanisms to the costs of concurrent computations is deciding [1], these implementations should also be efficient. 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.

1.2 The topic

In concurrent computing, a problem is solved through a set of processes that execute relatively independent tasks. Except in embarrassingly parallel programs, the tasks 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 some 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 *robust* manner. Roughly speaking, "robustness" means the following:

- 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 aspect of robustness 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), into the simpler problem of reasoning about a failure-free concurrent system where every process progresses at its own pace and runs to completion.

- Despite concurrency, the operations issued on each object appear as if they are executed sequentially. 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 robustness property is called *atomicity*.

  In short, this property transforms the difficult problem of reasoning about a concurrent system into the simpler problem of reasoning about a sequential one.

This book focuses mainly on *wait-free implementations of atomic objects*. Basically, given certain shared objects of *base* types, say provided in hardware, we study how and whether it is at all possible to wait-free implement (i.e., 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 implementing its type. As we shall see, ensuring each of atomicity or wait-freedom alone is trivial. The challenge is to ensure both.

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 more progressively how to use the resulting objects to build even more powerful objects.

1.3 Content of the book

1.3.1 Shared objects as concurrency abstractions

Defining and implementing appropriate programming abstractions are among the main challenges of computer science. A file, a stack, a record, a list, 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 upper layer software, i.e., software using 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 other processes q from invoking an operations on the same object X.

1.3.2 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 read-write 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).

1.3.3 Wait-freedom

Traditional synchronization algorithms rely on *mutual exclusion* (typically based on some *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 automatically ensured only if 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.

Lock-free implementations of atomic objects provide 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. 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.

1.3.4 Object implementation

This book studies how to wait-free implement certain atomic objects out of certain base objects. The notion of implementation has to be considered here in the *algorithmic* sense (there is no promise of C or 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*. 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 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 wait-free 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, there is a clear gain because relying on wait-free implementations makes it less vulnerable to failures and dead-locks. 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 *a priori* information about the interleaving of their steps: this is the model we will assume by default in this book.

1.3.5 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 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.

1.4 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, and we show how to wait-free implement atomic registers from these weak 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 in implementing 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 a test&set or a 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 algorithms 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.

1.5 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 [4, 13, 17, 18]. In the context of concurrent computing, one of the earliest work was reported in [9, 16]. More information on the history concurrent programming can be found in the book [3].

The notion of register (as considered in this book) and its formalization are due to Lamport [12]. A more hardware-oriented presentation was given in [15]. The notion of atomicity has been generalized to any object type by Herlihy and Wing [8] under the name linearizability. The concept of snapshot object has been introduced in [2]. A theory of wait-free atomic objects was developed in [10].

The mutual exclusion problem has been introduced by Dijkstra [5]. 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 [21]. Various synchronization algorithms are also detailed in [22].

The notion of wait-free computation originated in the work of Lamport [11], and was then explored further by Peterson [20]. It has then generalized and formalized by Herlihy [7].

The consensus problem was introduced in [19]. Its impossibility in asynchronous message-passing systems prone to process crash failures has been proved by Fischer, Lynch and Paterson in [6]. Its impossibility in shared memory systems was proved in [14]. The universality of the consensus problem and the notion of consensus number were investigated in [7].

Chapter 2

Atomicity: A Correctness Property for Shared Objects

2.1 Introduction

Before diving into how to implement shared sequential objects, we first address in this chapter the following questions:

- What is a sequential object?
- What does it mean for a shared-object implementation to be correct? In particular, how to evaluate correctness even when one or more processes stop their execution in the middle of an operation?

To give a flavor of the questions we address, let us consider an unbounded FIFO (first in first out) queue. This is an object of the type queue defined by the following two operations:

- $Enq(v)$: Add the value v at the end of the queue,
- $Deq()$: Return the first value of the queue and suppress it from the queue; if the queue is empty, return the default value $\perp$.

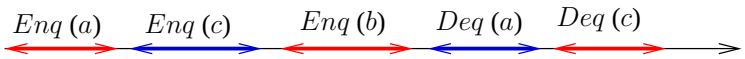

Figure 2.1: A sequential execution an a queue

Figure 2.1 describes a sequential execution of a system made up of a single process using the queue. The time-line, going from left to right, describes the progress of the process when it enqueues first the value a, then the value c, and finally the value b. According to the expected semantics of a queue, and as depicted by the figure, the first invocation of $Deq()$ returns the value a, the second returns the value c, etc.

Figure 2.2 depicts an execution of a system made up of two processes sharing the same queue. Now, process p_1 enqueues a and then b whereas process p_2 concurrently enqueues c. On the figure, the execution of $Enq(c)$ by p_2 overlaps both $Enq(a)$ and $Enq(b)$ by p_1. Such execution raises the following questions:

- What values are dequeued by p_1 and p_2?

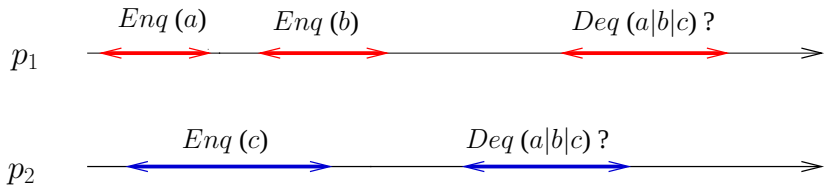

Figure 2.2: A concurrent execution on a queue

- What values can be returned by a process if the other process has failed while executing an operation?
- What happens if p_1 and p_2 share several queues instead of a single one? Etc.

Addressing these and related questions goes first through defining more precisely our model of computation.

2.2 Model

2.2.1 Processes and operations

The system we consider consists of a finite set of n *processes*, denoted $p_1, \ldots, p_n$. The processes execute some common distributed computation and, while doing so, cooperate by accessing *shared objects*.

Processes synchronize their activities by executing operations exported by shared objects. An execution by a process of an operation on an object X is denoted $X.op(arg)(res)$ where arg and res denote respectively the input and output parameters of the invocation. The output corresponds to the response to the invocation. Sometimes we simply write $X.op$ when the input and output parameters are not important. The execution of an operation $op()$ on an object X by a process p_i is modeled by two events, namely, the events denoted $inv[X.op(arg)$ by $p_i]$ that occurs when p_i invokes the operation (invocation event), and the event denoted $resp[X.op(res)$ by $p_i]$ that occurs when the operation terminates. (When there is no ambiguity, we talk about *operations* where we should be talking about *operation executions*.) We say that these events are generated by the process p_i and associated with the object X. Given an operation $X.op(arg)(res)$, the event $resp[X.op(res)$ by $p_i]$ is called the response event matching the invocation event $inv[X.op(arg$ by $p_i]$.

An execution of a distributed system induces a sequence of interactions between the processes of the system and the shared objects. Every such interaction corresponds to a computation *step* and is represented by an *event*: the visible part of a step, i.e., the invocation or the reply of an operation. A sequence of events is called a *history* and this is precisely how we model executions. We will detail this later in this chapter.

As we pointed out in the introduction of the book, we generally assume that processes are *sequential*: a process executes (at most) one operation of an object at a time. That is, the algorithm of a sequential process stipulates that after an operation is invoked on an object and until a matching response is received, the process does not invoke any other operation. The fact that processes are individually sequential does not preclude them from concurrently invoking operations on the same shared object. Sometimes, we will focus on *sequential executions* (modeled by *sequential histories*) which precisely preclude such concurrency; that is, only one process at a time invokes an operation on an object in a sequential execution.

2.2.2 Objects

An object has a name and a type. A type is defined by (1) the set of possible values for (the states of) objects of that type; (2) a finite set of operations through which the objects of that type can be manipulated; and (3) a specification describing, for each operation, the condition under which that operation can be invoked, and the effect produced after it has been executed. Figure 2.3 presents a structural view of a set of n processes sharing m objects.

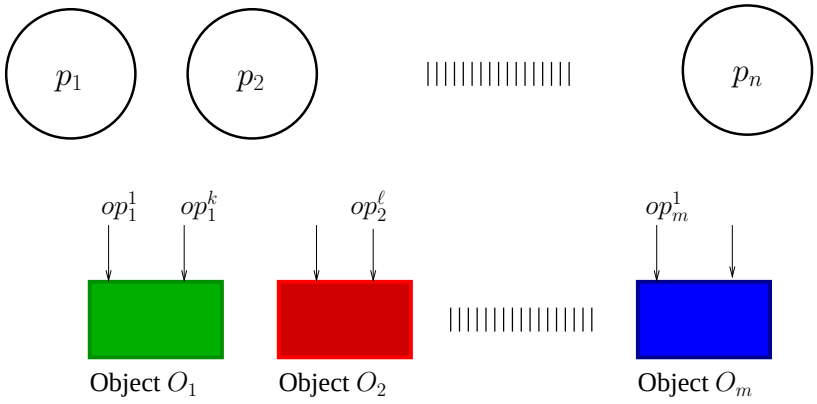

Figure 2.3: Structural view of a system

Sequential specification The object types we consider do generally have a sequential specification. We talk both about the specification of the object or the specification of the type. A sequential specification depicts the behavior of the object when accessed sequentially, i.e., in a sequential execution.

One can describe a sequential specification by associating two predicates with each operation. These predicates are called pre-assertion and post-assertion. Assuming the pre-assertion is satisfied before executing the operation, the post-assertion describes the new value of the object and the result of the operation returned to the calling process. We say that an object operation is *total* if it is defined for every state of the object; otherwise it is *partial*. This means that, differently from the pre-assertion associated with a partial operation, the pre-assertion associated with a total operation is always satisfied.

We also say that an object operation is *deterministic* if, given any state of the object that satisfies the pre-assertion and input parameters, the output parameters and the final state of the object are uniquely defined. An object (resp., type) that has only deterministic operations is said to be deterministic; otherwise we say that the object (resp., type) is non-deterministic.

Example 1: a read/write object (register) To illustrate the notion of sequential specification, we consider here three examples object types. The first type (called the type register) is a simple read/write abstraction, that models objects such as a shared memory word, a shared file or a shared disk.

It has two operations:

- The operation $read()$ has no input parameter. It returns a value of the object.
- The operation $write(v)$ has an input parameter, v, a new value of the object. The result of that operation is a value ok indicating to the calling process that the operation has terminated.

The sequential specification of the object is defined by all the sequences of read and write operations in which each read operation returns the value of the last preceding write operation (i.e., the last value written). Clearly, the read and write operations are always defined: they are total operations.

The problem of implementing a concurrent read/write object is a classical synchronization problem known under the name *reader/writer* problem.

Example 2: a FIFO queue with total operations The second example is the unbounded (FIFO) queue described in Section 2.1. Such object has the following sequential specification: every dequeue returns the first element enqueued and not dequeued yet. If there is not such element (i.e., the queue is empty), a specific default value $\perp$ is returned. This definition never prevents an enqueue or a dequeue operation to be executed: both enqueue and dequeue operations are total.

Example 3: a FIFO queue with a partial operation Let us consider now the previous queue definition modified as follows: a dequeue operation can be executed only when the queue is not empty. The sequential specification of this object is then a restriction of the previous specification; all situations where a dequeue operations returns $\perp$ have to be precluded. The enqueue operation can always be executed, so it remains a total operation. On the other hand, the pre-assertion of the dequeue operation states that it can only be executed when the queue is not empty; consequently, that operation is a partial operation.

The two FIFO queues examples (2 and 3) are two variants of a the classical *producer/consumer* synchronization problem.

Example 4: an object with no sequential specification Not all object types have a sequential specification. To illustrate this, let us consider a *rendezvous* object that can be accessed by two processes p_1 and p_2. Such an object provides the processes with a single operation $meeting()$ with the following semantics: after it has been invoked by a process, the operation terminates only when the other process has also invoked the operation. In other words, the key property of this object is that no process can terminate an operation without a concurrent invocation. It is easy to see that such a rendezvous object has no sequential specification: the behavior of the object cannot be described simply by stating what happens when the operation invocations by p_1 and p_2 would be totally ordered. (A rendezvous object is a typical example of an object that has no sequential specification. In this book, we are mainly interested in objects that have a sequential specification.)

2.2.3 Histories

An execution of a set of processes accessing a set of shared objects is captured through the notion of a *history*.

Representing an execution as a history of events Processes interact with shared objects via *invocation* and *response* events. We assume that simultaneous (invocation or response) events do not affect each other. This is generally the case, in particular for events generated by sequential processes accessing objects with a sequential specification. Therefore, without loss of generality, we can arbitrarily order simultaneous events.

This makes it possible to model the interaction between processes and objects as an ordered sequence of events H, called a *history* (sometimes also called a *trace*). The total order relation on the set of events induced by H is denoted $<_H$. A history abstracts the real-time order in which the events do actually occur.

Recall that an event includes the name of an object, the name of a process, the name of an operation and input -or output- parameters). The objects and processes associated with events of H are said to be involved in H.

A *local* history of p_i, denoted $H|p_i$, is a projection of H on process p_i: the subsequence H consisting of the events generated by p_i.

Equivalent histories Two histories H and H' are said to be *equivalent* if they have the same local histories, i.e., for each p_i, $H|p_i = H'|p_i$. That is, equivalent histories cannot be distinguished by any process.

Well-formed histories As we are interested only in histories generated by sequential processes, we restrict our attention to the histories H such that, for each process p_i, $H|p_i$ (the local history generated by p_i) is sequential: it starts with an invocation, followed by a response, called the matching response and associated with the same object, followed by another invocation, etc. We say in this case that H is *well-formed*.

Complete vs incomplete histories An operation is said to be *complete* in a history if the history includes both the event corresponding to the invocation of the operation and its response. Otherwise we say that the operation is *pending*. A history without pending operations is said to be *complete*. A history with pending operations is said to be *incomplete*. Note that, being sequential, a process can have at most one pending operation in a given history.

Partial order on operations A history H induces an irreflexive partial order on its operations as follows. Let $op = X.op1()$ by p_i and $op' = Y.op2()$ by p_j be two operations. Informally, operation op precedes operation op', if op terminates before op' starts, where "terminates" and "starts" refer to the time-line abstracted by the $<_H$ total order relation. More formally:

$$\big(op \rightarrow_H op'\big) \stackrel{\text{def}}{=} \big(resp[op] <_H inv[op']\big).$$

Two operations op and op' are said to *overlap* (we also say are *concurrent*) in a history H if neither $resp[op] <_H inv[op']$, nor $resp[op'] <_H inv[op]$. Notice that two overlapping operations are such that $\neg(op \rightarrow_H op')$ and $\neg(op' \rightarrow_H op)$. As a sequential history has no overlapping operations, it follows that $\rightarrow_H$ is a total order if H is a sequential history.

Illustrating histories Figure 2.4 depicts a well-formed history H. The history comprises ten events $e1 \ldots e10$ ($e4,\ e6,\ e7$ and $e9$ are explicitly detailed). As all the events in H are on the same object, its name is omitted. The enqueue operation issued by p_2 overlaps both enqueue operations issued by p_1. Notice that the operation $Enq(c)$ by p_2 is concurrent with both $Enq(a)$ and $Enq(b)$ issued by p_1. Moreover, the history H has no pending operations, and is consequently complete.

To illustrate the notions of incomplete and complete histories, let us again consider Figure 2.4. The sequence $e1 \ldots e9$ is an incomplete history where the dequeue operation issued by p_1 is pending. The sequence $e1 \ldots e6\ e7\ e8\ e10$ is another incomplete history in which the dequeue operation issued by p_2 is pending. Finally, the history $e1 \ldots\ e8$ has two pending operations. Now we are ready to define what we mean by a *sequential* history.

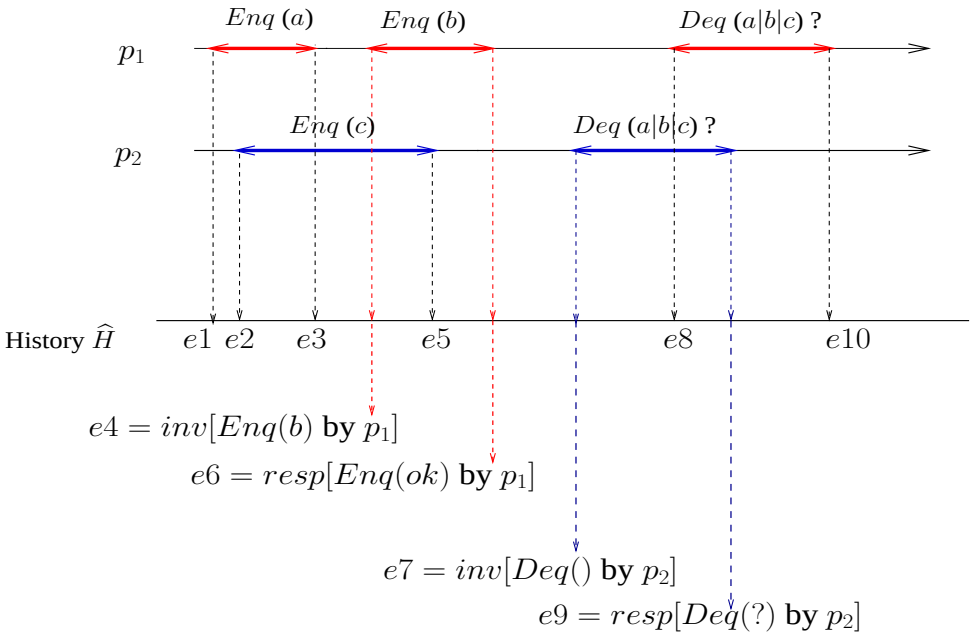

Figure 2.4: Example of a history

2.2.4 Sequential history

Definition A history is *sequential* if its first event is an invocation, and then (1) each invocation event, except possibly the last, is immediately followed by the matching response event, and (2) each response event, except possibly the last, is immediately followed by an invocation event. The sentence "except possibly the last" associated with an invocation event is due to the the fact that a history can be incomplete. A complete sequential history always ends with a response event. A history that is not sequential is *concurrent*.

A sequential history models a sequential multiprocess computation (there are no overlapping operations in such a computation), while a concurrent history models a concurrent multiprocess computation (there are at least two overlapping operations in such a computation). Given that a sequential history S has no overlapping operations, the associated partial order $\rightarrow_S$ defined on its operations is actually a total order.

Strictly speaking, the sequential specification of an object is a set of sequential histories involving solely that object. Basically, the sequential specification represents all possible sequential accesses to the object.

Example Considering Figure 2.4, H is a complete concurrent history. On the other hand, the complete history

$$H_1 = e1\ e3\ e4\ e6\ e2\ e5\ e7\ e9\ e8\ e10$$

is sequential: it has no overlapping operations. We can thus highlight its sequential nature by separating its operations using square brackets as follows:

$$H_1 = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e7\ e9]\ [e8\ e10].$$

The following histories H_2 and H_3

$$H_2 = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e8\ e10]\ [e7\ e9],$$

$$H_3 = [e1\ e3]\ [e4\ e6]\ [e8\ e10]\ [e2\ e5]\ [e7\ e9].$$

are also sequential. Let us also notice that histories H, H_1, H_2, H_3 are equivalent. Let H_4 be the history defined as follows

$$H_4 = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e8\ e10]\ [e7.$$

H_4 is an incomplete sequential history. All these histories have the same local history for process p_1: $H|p_1 = H_1|p_1 = H_2|p_1 = H_3|p_1 = H_4|p_1 = [e1\ e3]\ [e4\ e6]\ [e8\ e10]$, and, as far p_2 is concerned, $H_4|p_2$ is a prefix of $H|p_2 = H_1|p_2 = H_2|p_2 = H_3|p_2 = [e2\ e5]\ [e7\ e9]$.

So far, we defined the notion of a history as an abstract way to depict the interaction between a set of processes and a set of shared objects. In short, a history is a total order on the set of invocation and response events generated by the processes on the objects. We are now ready to define what we mean by a correct shared-object implementation, based on the notion of a *atomic* (or *linearizable*) history.

2.3 Atomicity

This section introduces the correctness condition called *atomicity* (or *linearizability*). The aim of atomicity is to transform the difficult problem of reasoning about a concurrent execution into the simpler problem of reasoning about a sequential one.

Intuitively, atomicity states that a history is correct if its invocation and response events could have been obtained, in the same order, by a single sequential process. In an atomic (also called linearizable) history, each operation has to appear as if it has been executed alone and instantaneously at some point between its invocation event and its response event. This section defines formally the atomicity concept and presents its main properties.

2.3.1 Legal history

As we pointed out earlier, shared objects that are usually considered in programming typically have a sequential specification defining their semantics. Not surprisingly, a definition of what is a "correct" history has to refer in one way or another to sequential specifications. The notion of *legal* history captures this idea.

Given a sequential history S, let $S|X$ (S at X) denote the subsequence of S made up of all the events involving object X. We say that a sequential history S is *legal* if, for each object X, the sequence $S|X$ belongs to the sequential specification of X. In a sense, a history is legal if it could have been generated by processes sequentially accessing objects.

2.3.2 The case of complete histories

We first define in this section atomicity for complete histories H, i.e., histories without pending operations: each invocation event of H has a matching response event in H. The section that follows will extend this definition to incomplete histories.

Definition A complete history H is *atomic* (or *linearizable*) if there is a "witness" history S such that:

1. H and S are equivalent,
2. S is sequential and legal, and

3. $\rightarrow_H \subseteq \rightarrow_S$.

The definition above states that for a history H to be linearizable, there must exist a permutation of H, S (witness history), which satisfies the following requirements. First, S has to be indistinguishable from H to any process [item 1]. Second, S has to be sequential (interleave the process histories at the granularity of complete operations) and legal (respect the sequential specification of each object) [item 2]. Notice that, as S is sequential, $\rightarrow_S$ is a total order. Finally, S has also to respect the real-time occurrence order of the operations as defined by $\rightarrow_H$ [item 3]. S represents a history that could have been obtained by executing all the operations, one after the other, while respecting the occurrence order of non-overlapping operations. Such a sequential history S is called a *linearization* of H.

When proving that an algorithm implements an atomic object, we need to prove that all histories generated by the algorithm are linearizable, i.e., identify a linearization of its operations that respects the "real-time" occurrence order of the operations and that is consistent with the sequential specification of the object.

It is important to notice that the notion of atomicity includes inherently a form of nondeterminism. A history H, may allow for several linearizations.

Linearization: an example Let us consider the history H described in Figure 2.4 where the dequeue operation invoked by p_1 returns the value b while the dequeue operation invoked by p_2 returns the value a. This means that we have $e9 = resp[Deq(a)$ by $p_2]$ and $e10 = resp[Deq(b)$ by $p_1]$. To show that this history is linearizable, we have to exhibit a linearization satisfying the three requirements of atomicity. The reader can check that history $H_1 = [e1\ e3]\ [e4\ e6]\ [e2\ e5]\ [e7\ e9]\ [e8\ e10]$ defined in Section 2.2.4 is such a witness. At the granularity level defined by the operations, witness history H_1 can be represented as follows

$$[Enq(a) \text{ by } p_1][Enq(b) \text{ by } p_1][Enq(c) \text{ by } p_2][Deq(a) \text{ by } p_2][Deq(b) \text{ by } p_1].$$

This formulation highlights the intuition that underlies the definition of the atomicity concept.

Linearization points The very existence of a linearization of an atomic history H means that each operation of H could have been executed at an indivisible instant between its invocation and response time events (while providing the same result as H). It is thus possible to associate a *linearization point* with each operation of an atomic history. This is a point of the time-line at which the corresponding operation could have been "instantaneously" executed according to its legal linearization.

To respect the real time occurrence order, the linearization point associated with an operation has always to appear within the interval defined by the invocation event and the response event associated with that operation.

Example Figure 2.5 depicts the linearization point of each operation. A triangle is associated with each operation, such that the vertex at the bottom of a triangle (bold dot) represents the associated linearization point. A triangle shows how atomicity allows shrinking an operation (the history of which takes some duration) into a single point of the time-line.

In that sense, atomicity reduces the difficult problem of reasoning about a concurrent system to the simpler problem of reasoning about a sequential system where the operations issued by the processes are instantaneously executed.

As a second example, let us consider the complete history depicted in Figure 2.5 where the response events $e9$ and $e10$ are such that $e9 = resp[Deq(b)$ by $p_2]$ and $e10 = resp[Deq(a)$ by $p_1]$. It is easy to

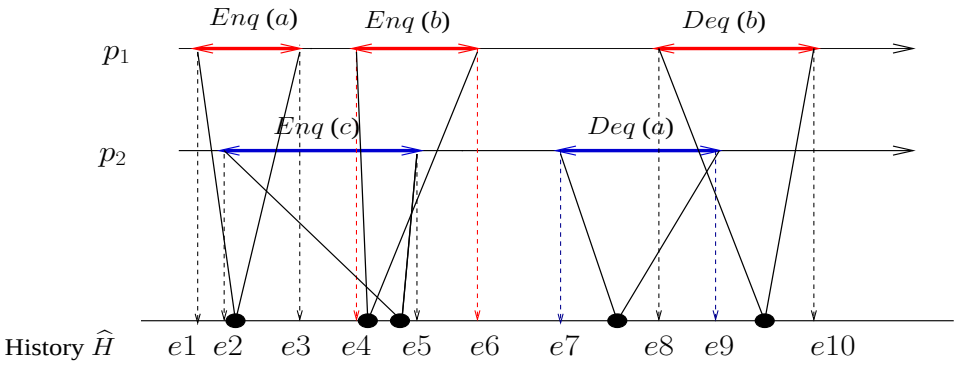

Figure 2.5: Linearization points

see that this history is linearizable: the sequential history H_2 described in Section 2.2.4 is one possible linearization. Similarly, the history where $e9 = resp[Deq(c)$ by $p_2]$ and $e10 = resp[Deq(a)$ by $p_1]$ is also linearizable. It has the following sequential witness history:

$$[Enq(c) \text{ by } p_2][Enq(a) \text{ by } p_1][Enq(b) \text{ by } p_1][Deq(c) \text{ by } p_2][Deq(a) \text{ by } p_1].$$

On the other hand, the history in which the two dequeue operations would return the same value is not linearizable: it does not have any witness history which respects the sequential specification of the queue.

2.3.3 The case of incomplete histories

We show here how to extend the definition of atomicity to partial histories. As we explained, these are histories with at least one process whose last operation is pending: the invocation event of this operation appears in the history while the corresponding response event does not. The history H_4 described in Section 2.2.4 is such a partial history. Extending atomicity to partial histories is important as it allows to cope with process crashes.

Definition A partial history H is linearizable if H can be *completed*, i.e., modified in such a way that every invocation of a pending operation is either removed or completed with a response event, so taht the resulting (complete) history H' is linearizable.

Basically, we reduce the problem of determining whether an incomplete history H is linearizable to the problem of determining whether a complete history H', extracted from H, is linearizable. We obtain H' by adding response events to certain pending operations of H, as if these operations have indeed been completed, but also removing invocation events from some of the pending operations of H. We require however that all complete operations of H be preserved in H'. It is important to notice that, given a history H, we can extract several histories H' that satisfy the required conditions.

Example Consider Figure 2.6 where we depict two processes accessing a shared register. Process p_1 first writes the value 0. The same process later issues a write for the value 1, but p_1 crashes during this second write (this is indicated by a cross on its time-line). Process p_2 executes two consecutive read operations. The first read operation lies between the two write operations of p_1 and returns the value 0. A different value

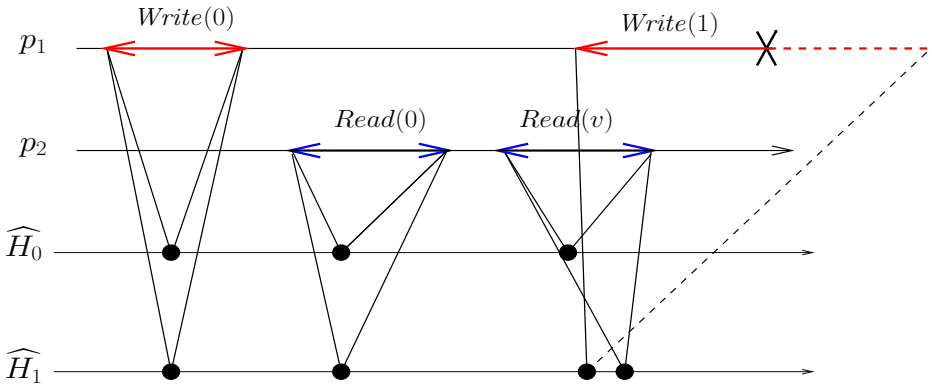

Figure 2.6: Two ways of completing a history

would clearly violate atomicity. The situation is less obvious with the second value and it is not entirely clear what value v has to be returned by the second read operation in order for the history to be linearizable?

As we now explain, both values 0 and 1 can be returned by that read operation while preserving atomicity. The second write operation is pending in the incompletehistory H modeling this execution. This history H is made up of 7 events (the name of the object and process names are omitted as there is no ambiguity), namely:

$$inv[write(0)]\ resp[write(0)]\ inv[read(0)]\ resp[read(0)]\ inv[read(v)]\ inv[write(1)]\ resp[read(v)].$$

We explain now why both 0 and 1 can be returned by the second read:

- Let us first assume that the returned value v is 0.
  We can associate with history H a legal sequential witness history H_0 which includes only complete operations and respects the partial order defined by H on these operations (see Figure 2.6). To obtain H_0, we construct history H' by removing from H event $inv[write(1)]$: we obtain a complete history, i.e., without pending operations.

  History H with $v = 0$ is consequently linearizable. The associated witness history H_0 models the situation where p_1 is considered as having crashed before invoking the second write operation: everything appears as if this write has never been issued.

- Let us now assume that the returned value v is 1.
  Similarly to the previous case, we can associate with history H a witness legal sequential history H_1 that respects the partial order on the operations. We actually derive H_1 by first constructing H', which we obtain by adding to H the response event $res[write(1)]$. (In Figure 2.6, the part added to H in order to obtain H' -from which H_1 is constructed- is indicated by dotted lines).

  The history where $v = 1$ is consequently linearizable. The associated witness history H_1 represents the situation where the second write is taken into account despite the crash of the process that issued that write operation.

2.4 Locality

This section presents an inherent property of atomicity that makes it particularly attractive. (Another important property of atomicity, namely *non-blockingness*, is discussed in the next chapters.)

2.4.1 Local properties

Let P be any property that is on a set of objects. The property P is said to be *local* if the set of objects as a whole satisfies P whenever each object taken alone satisfies P.

Locality is an important concept that promotes modularity. Consider some local property P. To prove that an entire set of objects satisfy P, we only have to ensure that each object -independently from the others satisfies P. As a consequence, property P can be implemented on a per object basis. At one extreme, it is even possible to design an implementation where each object has its own algorithm implementing P. At another extreme, all the objects (whatever their types) might use the same algorithm to implement P (each object using its own instance of the algorithm).

2.4.2 Atomicity is a local property

Intuitively, the fact that atomicity is local comes from the fact that (1) it considers that each operation is on single object, and (2) it involves the real-time occurrence order on non-concurrent operations whatever the objects and the processes concerned by these operations. We will rely on these two aspects in the proof of the following theorem.

Theorem 1 *A history H is atomic (linearizable) if and only if, for each object X involved in H, $H|X$ is atomic (linearizable).*

Proof The "$\Rightarrow$" direction (only if) is an immediate consequence of the definition of atomicity: if H is linearizable then, for each object X involved in H, $H|X$ is linearizable. So, the rest of the proof is restricted to the "$\Leftarrow$" direction. We also restrict the rest of the proof to the case where H is complete, i.e., H has no pending operation. This is without loss of generality, given that the definition of atomicity for an incomplete history is derived from the definition of atomicity for a complete history.

Given an object X, let S_X be a linearization of $H|X$. It follows from the definition of atomicity that S_X defines a total order on the operations involving X. Let $\rightarrow_X$ denote this total order. We construct an order relation $\rightarrow$ defined on the whole set of operations in H as the union $\{\bigcup_X \rightarrow_X\} \cup \rightarrow_H$, i.e.:

1. For each object X: $\rightarrow_X \subseteq \rightarrow$,
2. $\rightarrow_H \subseteq \rightarrow$.

Basically, "$\rightarrow$" totally orders all operations on the same object X, according to $\rightarrow_X$ (item 1), while preserving $\rightarrow_H$, i.e., the real-time occurrence order on the operations (item 2).
Claim. $\rightarrow$ is acyclic.

The claim implies that a transitive closure of $\rightarrow$ indeed defines a partial order on the set of all the operations of H. Since any partial order can be extended to a total order, we construct a sequential history S including all events of H and respecting $\rightarrow$. By construction, we have $\rightarrow \subseteq \rightarrow_S$ where $\rightarrow_S$ is the total order on the operations defined from S. We have the three following conditions: (1) H and S are equivalent (2) S is sequential (by construction) and legal (due to item 1 above); and (3) $\rightarrow_H \subseteq \rightarrow_S$ (due to item 2 above and

the fact that $\rightarrow \subseteq \rightarrow_S$). It follows that H is linearizable.

Proof of the claim. We show (by contradiction) that $\rightarrow$ is acyclic. Assume first that $\rightarrow$ induces a cycle involving the operations on a single object X. Indeed, as $\rightarrow_X$ is a total order, in particular transitive, there must be two operations op_i and op_j on X such that $op_i \rightarrow_X op_j$ and $op_j \rightarrow_H op_i$. But $op_i \rightarrow_X op_j \Rightarrow inv[op_i] <_H resp[op_j]$ because X is linearizable. Given that $op_j \rightarrow_H op_i \Rightarrow resp[op_j] <_H inv[op_i]$, which establishes the contradiction as $<_H$ is a total order on the whole set of events.

It follows that any cycle must involve at least two objects. To obtain a contradiction we show that, in that case, a cycle in $\rightarrow$ implies a cycle in $\rightarrow_H$ (which is acyclic). Let us examine the way the cycle could be obtained. If two consecutive edges of the cycle are due to just some $\rightarrow_X$ or just $\rightarrow_H$, then the cycle can be shortened as any of these relations is transitive. Moreover, $op_i \rightarrow_X op_j \rightarrow_Y op_k$ is not possible for $X \neq Y$, as each operation is on only one object ($op_i \rightarrow_X op_j \rightarrow_Y op_k$ would imply that op_j is on both X and Y). So let us consider any sequence of edges of the cycle such that: $op1 \rightarrow_H op2 \rightarrow_X op_3 \rightarrow_H op4$. We have:

- $op1 \rightarrow_H op2 \Rightarrow resp[op1] <_H inv[op2]$ (definition of $op1 \rightarrow_H$),
- $op2 \rightarrow_X op3 \Rightarrow inv[op2] <_H resp[op3]$ (as X is linearizable),
- $op3 \rightarrow_H op4 \Rightarrow resp[op3] <_H inv[op4]$ (definition of $op1 \rightarrow_H$).

Combining these statements, we obtain $resp[op1] <_H inv[op4]$ from which we can conclude that $op1 \rightarrow_H op4$. It follows that any cycle in $\rightarrow$ can be reduced to a cycle in $\rightarrow_H$. A contradiction as $\rightarrow_H$ is an irreflexive partial order. *End of the proof of the claim.* $\square_{Theorem\ 1}$

Considering an execution of a set of processes that access concurrently a set of objects, atomicity allows reasoning as as the operations issued by the processes on the objects were executed one after the other. The previous theorem is fundamental. It states that when one has to reason on sequential processes that access concurrent atomic objects, one can reason on a per object basis, without loosing the atomicity property on the whole computation.

2.5 Alternatives to atomicity

This section discusses alternatives to atomicity, namely, *sequential consistency* and *serializability*.

2.5.1 Sequential consistency

Overview Atomicity stipulates that the witness sequential history S for a given history H should respect the partial order relation $\rightarrow_H$ on operations in H (also called the real-time order). Any two operations op and op' such $op \rightarrow_H e'$ should appear in that order in the witness history S, irrespective of the processes invoking them and the objects on which they are performed.

A relaxation of atomicity, called *sequential consistency* only requires that the real-time order is preserved if the operations are invoked by the same process, i.e., S is only supposed to respect the *process-order*.

Definition The definition of the sequential consistency correctness condition reuses the notions of history, sequential history, complete history, as in Section 2.2. To simplify the presentation and without loss of generality, we only consider complete histories (with no pending operations).

A history H is *sequentially consistent* if there is a "witness" history S such that:

1. H and S are equivalent,
2. S is sequential and legal. respect process-order).

To illustrate sequential consistency, let us consider Figure 2.7. There are two processes p_1 and p_2 that share a queue Q. At the operation level, the local history of p_1 comprises a single operation, $Q.Enq(a)$, while the local history of p_2 comprises two operations, first $Q.Enq(b)$ and then $Q.Deq(b)$. The reader can easily verify that this history is not atomic: as all the operations are totally ordered according to real-time, the $Q.Deq()$ operation issued by p_2 should return the value a whose enqueuing was terminated before the enqueuing of a has started. However, the history is sequentially consistent: The sequential history (described at the operation level)

$$S = [Q.Enq(b) \text{ by } p_2][Q.Enq(a) \text{ by } p_1][Q.Deq(b) \text{ by } p_2]$$

is legal and respects the process-order relation.

Both consistency criteria, atomicity and sequential consistency, require a witness sequential history, but sequential consistency has no requirement related to the occurrence order of operations issued by different processes (and captured by the real-time order). It can be seen as based only on a logical time (the one defined by the witness history).

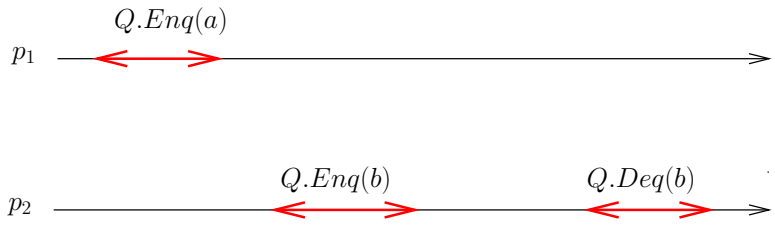

Figure 2.7: A sequentially consistent history

Atomicity vs sequential consistency Clearly, any linearizable history is also sequentially consistent. As shown by the example of Figure 2.7 however, the contrary is not true. It is then natural to ask whether sequential consistency is not good enough to reason about correctness of concurrent implementations.

A drawback of sequential consistency is that it is not a local property. To illustrate this, consider the counter-example described in Figure 2.8. History H involves two processes accessing two concurrent queues Q and Q'. It is easy to see that, when we consider each object in isolation, we obtain the histories $H|Q$ and $H|Q'$ that are sequentially consistent. Unfortunately, there is no way to witness a legal total order S that involves the six operations: if p_1 dequeues b' from Q', $Q'.enq(a')$ has to be ordered after $Q'.enq(b')$ in a witness sequential history. But this means that (to respect process-order) $Q.enq(a)$ by p_1 is necessarily ordered before $Q.enq(b)$ by p_2: consequently $Q.Deq()$ by p_2 should returns a for S to be legal. A similar reasoning can be done starting from the operation $Q.Deq(b)$ by p_2. It follows that there can be no legal witness total order: even though $H|Q$ and $H|Q'$ are sequentially consistent, the whole history H is not.

2.5.2 Serializability

Overview Both atomicity and sequential consistency guarantee that operations appear to execute instantaneously at some point of the time line. The difference is that atomicity requires that, for each operation, this instant lies between the occurrence times of the invocation and response events associated with the operation, which is not the case for sequential consistency.

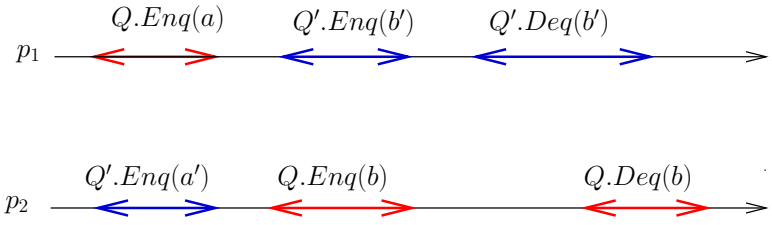

Figure 2.8: Sequential consistency is not a local property

Sometimes, it is important to ensure that *groups* of operations appear to execute as if they have been executed without interference with any other group of operations. The concept of *transaction* is then the appropriate abstraction that allows grouping operations.

A transaction is a sequence of operations that might complete successfully (commit) or abort. In short, the execution of a set of concurrent transactions is correct if committed transactions appear to execute at some indivisible point in time and aborted transactions do not appear to have been executed at all. This correctness criteria is called *serializability* (sometimes it is also called atomicity). The point (again) is to reduce the difficult problem of reasoning about concurrent transactions into the easier problem of reasoning about transactions that are executed one after the other. For instance, if some invariant predicate on the set of shared objects is preserved by every individual committed transaction, then it will be preserved by a serializable execution of transactions.

Definition To define serializability, the notion of history needs to revisited. Events are now associated with objects and transactions. In short, processes are replaced by transactions. For each transaction, in addition to the invocation and response events, two new events come into the picture: *commit* and *abort* events. These are associated with transactions. At most one such event is associated with every transaction in a history. A transaction without such event is called pending; otherwise the transaction is said to be complete (committed or aborted). Adding a commit (resp., abort) event after all other events of a pending transaction is called committing (resp., aborting) the transaction. A sequential history is a sequence of committed transactions. We say that a history is complete if all its transactions are complete.

Let H be a complete history. H is *serializable* if there is a "witness" history S such that:

1. For each transaction T, $S|T = H|T$.

2. S is sequential and legal, and

Let H be a history that is not complete. H is *serializable* if we can derive from H a complete serializable history H' by completing or removing pending transactions from H.

Atomicity vs serializability Again, correctness is defined according to the equivalence to a witness sequential history. No real-time ordering is required. In this sense, serializability can be viewed as the extension of sequential consistency to several operations. Like sequential consistency, serializability is not a local property either. Replacing in Figure 2.8 processes with transactions gives a counter-example that proves that.

2.6 Summary

We introduced in this chapter the basic elements that are needed to reason about executions of a distributed system made up of concurrent processes interacting through shared objects. More specifically, we introduced the elements that are needed to introduce the atomicity concept.

The fundamental element is that of a history: a sequence of events depicting the interaction between processes and objects. An event represents the invocation of an object or the return of a response. A history is atomic if, despite concurrency, it appears as if processes access the objects of the history in a sequential manner. In this sense, the correctness of a concurrent computation is judged with respect to a sequential behavior, itself determined by the sequential specification of the objects.

2.7 Bibliographic notes

The notion of atomic read/write objects (registers), as studied here, has been investigated and formalized by Lamport [12] and Misra [15]. The generalization of the atomicity consistency condition to objects of any sequential type has been developed by Herlihy and Wing under the name linearizability [8].

The notion of sequential consistency has been introduced by Lamport [28]. The relation between atomicity and sequential consistency was investigated in [24] and [30] where it was shown that, from a protocol design point of view, sequential consistency can be seen as a lazy linearizability. Examples of protocols implementing sequential consistency can be found in [23, 24, 31].

The concept of transactions is part of every textbook on database systems. Books entirely devoted to transactions are [25, 26, 27]). The theory of serializability was the main topic of the following books [26, 29].

Chapter 3

Wait-freedom: A Progress Property for Shared Object Implementations

3.1 Introduction

The previous chapter focused on the *atomicity* property of shared objects.This property requires operations to appear as if executed one after the other. Basically, atomicity stipulates that certain behaviors should be precluded, namely those that do not hide concurrent operation executions on the same object.

Atomicity is a *safety* property. It states what should *not* happen in an execution involving processes and shared objects (namely, an execution that is not linearizable must never happen). Clearly, this could be achieved by a trivial algorithm that would not return any operation of the object to be implemented, i.e., one that would never return any result: an empty history would be a trivial linearization of every execution of this algorithm.

However, one would also require the implemented shared object to also perform its operations when it is asked to do so by an application process making use of that object. In other words, one would also require that the algorithm implementing the object satisfies some *progress* property. Not surprisingly, this also depends on the process invoking the operation and in particular on how this process is scheduled by the operating system.

For instance, if a process invokes an operation and immediately crashes or is paged out by the operating system, then it makes little sense to require that the process obtains a reply matching its invocation. In fact, one might require that the shared object satisfies some progress property, provided that the process invoking an operation on the shared object is scheduled by the underlying operating system to execute enough steps of the algorithm implementing that operation. Performing such steps reflect the ability of the process to invoke primitives objects used in the implementation in order to eventually obtain a reply to the high-level operation being implemented.

One might, for example, require that if a process invokes an operation and keeps executing steps of the algorithm implementing the operation, then the operation eventually terminates and the process obtains a reply. Sometimes one might even require that, after invoking an operation, the process should obtain a response to the operation within b steps of the process.

To express such requirements, we need to carefully define the notion of object *implementation* and zoom into the way processes execute the algorithm implementing the object, in particular how their steps are scheduled by the operating system. We will in particular introduce the notion of *implementation history*: this is a *lower level* notion than that describing the interaction between the processes and the object being

implemented (previous chapter). Accordingly, the first is called a *high level history* whereas the second is called a *low level history*. This will be used to introduce progress properties of shared object implementations, the strongest of these being *wait-freedom*.

We will focus on a *single* object implementation. As discussed in Chapter 1, when implementing atomic objects, it is enough to consider each object separately from the other objects. This is because the atomicity consistency criterion is a local property (i.e. the composition of objects, that individually satisfy atomicity, provides a system that, as a whole, does satisfy the atomicity consistency criterion).

3.2 Implementation

3.2.1 High Level Object and Low Level Object

To distinguish the shared object to be implemented from the underlying objects used in the implementation, we typically talk about a *high level* object and underlying *low level* objects. (The latter are sometimes also called *base* objects.) Similarly, to disambiguate, we will talk about *primitives* instead of operations as far as low level objects are concerned. That is, a process invokes *operations* on a high level object and the implementation of these operations requires the process to invoke *primitives* of the underlying low level objects. When a process invokes such a primitive, we say that the process performs a *step*.

The very notions of high level and low level are relative and depend on the actual implementations. An object type might be considered high level in a given implementation and low level in another one. The object to be implemented is the high level one and the objects used in the implementation are the low level ones. In general, the intuition is that the low level objects might typically capture basic synchronization abstractions provided in hardware whereas the high level ones are those we want to emulate in software (what we call *implement*). Such emulations are strongly motivated by the desire to facilitate the programming of concurrent applications, i.e. to provide the programmer with powerful synchronization abstractions encapsulated by high-level objects. Another motivation is to reuse programs initially devised with the high level object in mind in a system that does not provide such an object in hardware. Indeed, multiprocessor machines do not all provide the same basic synchronization abstractions. For instance, some modern machines provide compare&swap as a base object in hardware. Others do not and might provide instead test&set or simply some form of registers. Providing an implementation of compare&swap using test&set would, for instance, make it possible to directly reuse, within an old machine, an application written for a modern machine.

An example is detailed in Figure 3.2 (Section 3.4.2) that describes the implementation of a FIFO queue (the high level object) from atomic low level objects denoted $NEXT$, $Q[1]$, $Q[2]$, etc. The low level object $NEXT$ provides the processes with the primitives fetch&add() and read(), while each $Q[i]$ low level object provides the processes with the primitives swap() and write(). So, a high level history is made up of $Enq()$ and $Deq()$ operations, while a low level history is made up of invocations of the primitives fetch&add() and read() on the low level object $NEXT$, and swap() and write() on the low level objects $Q[x]$. This is schematically represented in the Figure 3.1.

Sometimes the low level objects are assumed to be atomic, and sometimes not. As shown later in the book, it is sometimes useful to first implement intermediate objects that are not atomic, then implement the desired high level atomic objects on top of them.

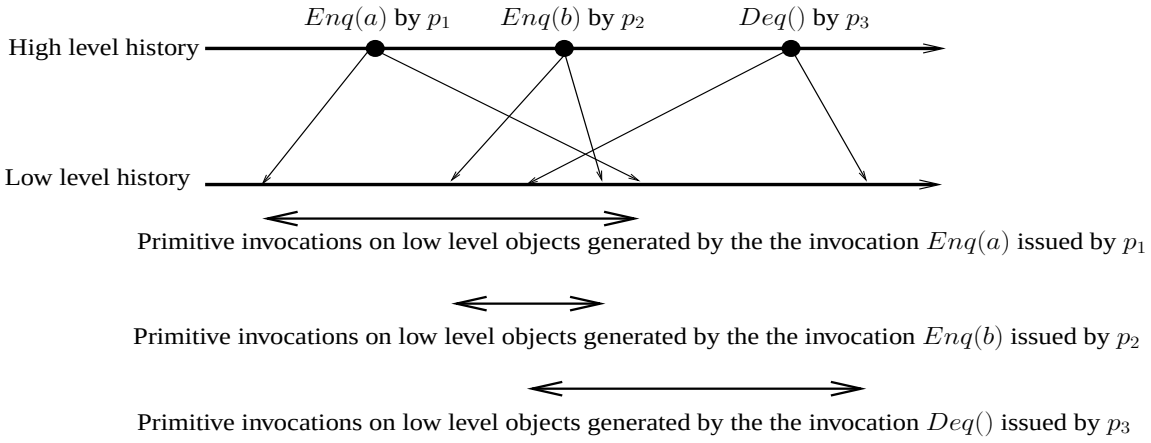

Figure 3.1: High level and low level histories

3.2.2 Zooming into histories

When reasoning about the atomicity of an object to be implemented, i.e., a high-level object, the executions of the processes accessing the object are represented with histories. As defined in the previous chapter, a history is a sequence of events, each representing an invocation or a reply on the high-level object in question.

History of an implementation Reasoning about progress properties requires to zoom into the invocations and replies of the lower level objects on top of which the high level object is built. Hence, *lower level* histories are needed that depict events at the interface between the processes and the low level objects *used* in the implementation, i.e., the primitive events. Without such a zooming, it is not possible, for instance, to distinguish a process that crashes right after invoking a high level object operation and stops invoking low-level objects, from one that keeps executing the algorithm implementing that operation and invoking primitives of low level objects. We might want to require that the later obtains a matching reply and exempt the former from having to obtain a reply. So, when we talk about the *history of an implementation*, we implicit assume such a low level history, which is a refinement of the higher level history involving only the invocations and replies of the high level object to be implemented.

Considering the example of a FIFO high level object developed in Section 3.4.2, a high level history is a sequence built from invocation and reply events associated to operations $Enq()$ and $Deq()$, while a low level history (or implementation history) is a sequence built from the primitives fetch&add() and read() associated with the $NEXT$ low level object, and the primitives swap() and write() associated with each $Q[i]$ low level object, that are generated from the invocations of the $Enq()$ and $Deq()$ operations.

The two faces of a process To better understand the very notion of a low level history, it is important to distinguish the two roles of a process. On the one hand, a process has the role of a *client* that sequentially invokes operations on the high level object and receives replies. On the other hand, the process has also the role of a *server* implementing the operations. While doing so, the process invokes primitives on lower level objects in order to obtain a reply to the high-level invocation.

It might sometimes be convenient to think of the two roles of a process as executed by different entities. As a client, a process invokes object operations but does not control the way the low level primitives imple-

menting these operations are executed. It even does not know how an object operation is implemented. Differently, as a server, a process executes the algorithm (made up of invocations of low level object primitives) associated with the high level object operation. Such an algorithm is typically described by an automaton (possibly with an unbounded number of states). The execution of a low level object primitive is called a *step* and it typically represents an atomic unit of computation.

Scheduling and asynchrony The interleaving of steps in an implementation is specified by a *scheduler* (itself part of an operating system). This is outside of the control of processes and, in our context, it is convenient to think of a scheduler as an *adversary*. This is because, when devising a distributed algorithm, one has to cope with worst-case strategies of a scheduler that could defeat the algorithm.

Strictly speaking, a process is said to be *correct* in a low-level history when it executes an infinite number of steps, i.e., when the scheduler schedules an infinite steps of that process. This "infinity" notion models the fact that the process executes as many steps as needed by the implementation. Otherwise, the process is said to be *faulty*. Sometimes it is convenient to see a faulty processes as a process that crashes and prematurely quits the computation. In the context of this book, we assume that processes might indeed crash and permanently stop performing steps but do not deviate from the algorithm assigned to them. In other words, they are not malicious (we also say they are not Byzantine).

Unless explicitly stated otherwise, the system is assumed to be *asynchronous* which means that the relative speeds of the processes are unbounded: for any Φ there is an execution in which a process takes Φ steps while another not crashed process takes only one step. Basically, an asynchronous system progresses is controlled by a very weak scheduler, i.e., a scheduler whose only restriction lies in the fact that it cannot prevent forever a correct (never crashing) process from executing steps.

3.3 Progress properties

As pointed out above, a trivial way to ensure atomicity would be to do nothing, i.e., not return any reply to any operation invocation. This would preclude any history that violates linearizability by simply precluding any history with a reply.

Besides this (clearly, meaningless) approach, a popular way to ensure atomicity is to use *critical sections* (say using *locks*), preventing concurrent accesses to the same object. In the simplest case, every operation on a shared object is executed as a critical section. When a process invokes an operation on an object, it first requests the corresponding lock, and the algorithm of the operation is executed by the process only when the lock is acquired. If the lock is not available, the process waits until the lock is released. After a process obtains the reply to an operation, it releases the corresponding lock.

As we discussed in the introduction of this book, such an implementation of a shared object has an inherent drawback: the crash of the process holding the lock on an object prevents any other . In practice, this might correspond to the situation where the process holding the lock is preempted for a long period of time, and all processes contending on the same object are blocked. When processes are asynchronous (i.e., the scheduler can arbitrarily preempt processes) which is the default assumption we consider, there is no way for a process to know whether another process has crashed (or was preempted for a long while) or is only very slow.

This book focuses on shared object implementations with progress properties that preclude the use of critical sections and locks. Informally, we say an implementation is lock-based if it allows for a situation in which a process running in isolation from some point on is never able to complete its operation. Taking a negation of this property, we state that an implementation does not employ locks if starting after any finite

execution, every process can complete its operation in a finite number of its solo steps. Intuitively, this property, called *obstruction-freedom* (or *solo termination*), must be satisfied by any implementation where the crash of some processes does not prevent other processes from making progress. Several such progress properties, including obstruction-freedom, are presented below.

3.3.1 Solo, partial and global termination

- Obstruction-freedom (also called *Solo termination*). An implementation of a concurrent object is obstruction-free, if any of its operations is guaranteed to terminate if it is eventually executed without concurrency (assuming that the invoking process does not crash[1]).

  An operation is "eventually executed without concurrency" if there is a time after which the only process to take steps is the process that invoked that operation. Note that this does not prevent other processes from having started and not yet finished operations on the same object (this is for example the case of a process that crashed in the middle of an operation on the object).

  Note that obstruction-freedom allows executions in which several processes invoking operations on the same object forever contend on the internal representation of the object without terminating.

  As we observed earlier, obstruction-freedom precludes the use of locks.

- Non-blockingness. This is a *partial termination* property that is strictly stronger than obstruction-freedom. It states the following: despite asynchrony and process crashes, if several processes execute operations on the same object and do not crash, at least one of them terminates its operation.

  So, non-blocking means *deadlock-freedom* despite asynchrony and crashes.

- Wait-freedom. This is a *global termination* property that states the following: despite asynchrony and process crashes, any process that executes an operation on the object (and does not crash), terminates its operation [7]. Wait-freedom is strictly stronger than non-blockingness.

  So, wait-freedom means *starvation-freedom* despite asynchrony and crashes.

3.3.2 Bounded termination

Wait-freedom, the strongest among the liveness properties considered above, does not stipulate a bound on the number of steps that a process needs to execute before obtaining a matching reply when it invokes a high level object operation. Typically, this number can depend on the behavior of the other processes. For example, it can be small if no other process performs any step, and increases when all processes perform steps (or the opposite), while remaining always finite, regardless of the number and timing of crashes.

- An implementation satisfies the **bounded wait-free** property if there is a bound B such that in any low level history every process p that invokes an operation receives a matching reply within B of its own steps. (The B steps of p are not required to be consecutive.)

  In other words, there is no prefix of a low level history in which a process invokes an operation and executes B steps without obtaining a matching reply.

Showing that an implementation is bounded wait-free consists in exhibiting an upper bound on the number of steps needed to return from any operation. That upper bound is usually defined by a function $f()$

[1] Let us recall that "a process does not crash" means that "it executes an infinite number of steps".

on the number of processes (e.g., $O(n^2)$). One can similarly define notions like bounded solo or bounded partial termination.

3.4 Atomicity and wait-freedom

Just as it is meaningless to ensure atomicity alone, without any progress guarantee, it is also meaningless to ensure any progress guarantee alone. Meaningful implementations are those that ensure both: ideal ones are those that ensure atomicity and wait-freedom.

Before diving into such implementations in the next chapters, it is important to ask whether every atomic object has an implementation that ensures wait-freedom and atomicity. In fact, it is easy to see that this would not be the case for objects with *partial* operations (previous chapter). By definition, the progress of such operations may depend on concurrent invocations of other operations. That is, if an object's specification requires that a process does not return from an operation unless some other process completes some other operation first, then it would be impossible to come up with even a solo-terminating implementation of this object, regardless of how powerful underlying base objects are. As we discuss below however, an implementation that ensures wait-freedom and atomicity is always possible for objects with *total* operations.

3.4.1 Operation termination and atomicity

Besides being a *local* property, which we discussed in the previous chapter, atomicity is also *non-blocking*, meaning that a pending invocation of a total operation is never required to wait for another operation to complete and yet preserve atomicity. This property has a fundamental consequence. It means that, per se, atomicity never forces a pending total operation to block. In other words, atomicity, *per se*, cannot prevent wait-freedom. Blocking (or even deadlock and starvation) can occur as an artifact of a particular implementation of atomicity, but is not inherent to atomicity. The following theorem captures this idea by stating that any (linearizable) history with a pending operation invocation can be extended with a reply to that operation.

Theorem 2 *Let $inv[op(arg)]$ be the invocation event of a total operation that is pending in a linearizable history H. There exist a matching reply event $resp[op(res)]$ such that the history $H' = H.resp[op(res)]$ is linearizable.*

Proof Let S be a linearization of the partial history H. By definition of a linearization, S has a matching reply to every invocation. Assume first that S includes a reply event $resp[op(res)]$ matching the invocation event $inv[op(arg)]$. In this case, the theorem trivially follows as then S is also a linearization of H'.

If S does not include a matching reply event, then S does not include $inv[op(arg)]$ either. Because the operation $op()$ is total, there is a reply event $resp[op(res)]$ matching the invocation event $inv[op(arg)]$ in every state of the shared object. Let S' be the sequential history S with the invocation event $inv[op(arg)]$ and a matching reply event $resp[op(res)]$ added in that order at the end of S. S' is trivially legal. It follows that S' is a linearization of H'. $\Box_{Theorem\ 2}$

3.4.2 Example

To illustrate the inherent non-blocking feature of atomicity, and indirectly illustrate implementations that ensure atomicity and wait-freedom, consider a simple FIFO queue that can contain an unbounded number of items. The sequential specification of this object has been given in Section 2.1 of Chapter 2.

The example is simple. (More sophisticated examples will be given in the next chapters.) The system we consider here is made up of producers (clients) and consumers (servers) that cooperate through an unbounded FIFO queue. A producer process repeats forever the following two statements: it first computes a new item v, and then invokes the operation $Enq(v)$ to deposits v in the queue. Since we assume that the queue is unbounded, the operation $Enq(v)$ is total.

Similarly, a consumer process repeats forever the following two statements: it first withdraws an item from the queue by invoking the operation $Deq()$, and then consumes that item. If the queue is empty, then the default value $\perp$ is returned to the invoking process. (This default value that cannot be deposited by a producer process.) Since we do not preclude the possibility of returning $\perp$, the $Deq()$ operation also is total. We assume that no processing by the consumer is associated with the $\perp$ value.

The algorithm implementing the shared queue relies on an array $Q[0..\infty)$ used to store the items of the queue. Each entry of the array is initialized to $\perp$.

To enqueue an item to the queue, the producer first locates the index of the next empty slot in the array Q, reserves it, and then stores the item in that slot. To dequeue a value, the consumer first determines the last entry of the array Q that has been reserved by a producer. Then, it scans the array Q in ascending order until it finds an item different from the default value $\perp$. If it finds one, it returns it. Otherwise, the default value is returned.

The algorithm is given in Figure 3.2. The *return* () statement terminates the operation (it corresponds to the reply event associated with that operation). Lowercase letters are used for identifiers of local variables. Uppercase letters are used for shared variables. The implementation uses the following shared variables: $NEXT$ (initialized to 1) and the array Q, used to contain the values that have been produced and not yet consumed. The variable $NEXT$ is a pointer to the next slot of the array Q that can be used to deposit a new value. (This implementation could be optimized by reclaiming the slots from which items have been dequeued. But this is not the point here.)

The variable $NEXT$ is provided with two primitives denoted read() and fetch&add(). The invocation $NEXT$.fetch&add(x) returns the value of $NEXT$ before the invocation and adds x to $NEXT$. Similarly, each entry $Q[i]$ of the the array is provided with two primitives denoted write() and swap(). The invocation $Q[i]$.swap(v) writes v in $Q[i]$ and returns the value of $Q[i]$ before the invocation.

The execution of the read(), write(), fetch&add() and swap() primitives on the shared base objects ($NEXT$ and each variable $Q[i]$) are assumed to be atomic. The primitives read() and write() are implicit in the code of Figure 3.2 (they are in the assignment statements denoted "$\leftarrow$").

The algorithm does not use locks, so no process can block other processes forever by crashing inside a critical section. Furthermore, each value deposited in the array by a producer will be withdrawn by a swap() operation issued by a consumer (assuming that at least one consumer is correct).

To better understand the algorithm, let us explore the two following situations.

- The first situation is when a producer crashes after $NEXT$ is increased by 1 and before the corresponding item is deposited in the array Q. That is, the producer reserves an index without ever using the corresponding slot. This conveys the fact that $NEXT$ represents an upper bound (and not a tight bound) on the number of items that are deposited in Q.) This can force consumers to explore more entries of Q than necessary. However, all these additional entries that are visited are equal to $\perp$. This means that, when a producer crashes in such a scenario, everything appears as if the producer has not issued the enqueue operation at all.

- The second situation is when during a dequeue operation, a consumer crashes right after having executed the statement $Q[i]$.swap(aux). If $aux = \perp$, from an external observer point of view, everything

```
operation Enq(v):
    in ← NEXT.fetch&add (1);
    Q[in] ← v;
    return ()

operation Deq():
    last ← NEXT − 1;
    for i from 0 until last do
        aux ← Q[i].swap (⊥);
        if (aux ≠ ⊥) then return (aux) end_if
    end_do;
    return (⊥)
```

Figure 3.2: Enqueue and dequeue implementations

appears as if the dequeue operation has never been issued. If $aux = v \neq \perp$, everything appears as if that value v has been obtained by the dequeue operation, the consumer crashing just after using it.

3.4.3 On the power of low level objects

The previous example shows that a FIFO queue, shared by an arbitrary number of processes, can be wait-free implemented from two types of base atomic objects, namely, an atomic object $NEXT$ whose type is defined by then pair of primitives fetch&add() and read(), as well as an array Q of atomic objects, the type of these objects being defined by the pair of primitives write() and swap().

This means that these base types are "powerful enough" to wait-free implement a FIFO queue shared by any number of processes. The investigation of the power of base object types to wait-free implement *any* shared object constitutes the topic addressed in the third part of the book.

3.4.4 Non-determinism

Before concluding this chapter, it is worthwhile to highlight some sources of non-determinism in a concurrent system that need to be considered when devising a shared object implementation.

1. The scheduler of a concurrent system can orchestrate the steps of the processes in all kinds of ways and this is a source of non-determinism that any wait-free implementation has to cope with.

2. Finally, when seeking for a linearization of a concurrent history, we can also choose among several possible sequential histories. First, there might indeed be several ways of completing the original history, especially when non-deterministic objects are involved. Second, there might be several ways of ordering concurrent operations in the equivalent linearization.

3.5 Summary

We defined in this chapter three progress properties: solo-termination, partial-termination and wait-freedom. A wait-free implementation of an atomic object is inherently robust in the following sense.

- It is inherently starvation-free. (This is an immediate consequence of the definition of the wait-free property.)
- It is $(n-1)$-resilient. This expresses the fact it naturally copes with up to $(n-1)$ process crashes.

Bibliographic notes

Atomicity is a safety property whereas wait-freedom is a liveness property. The notions of safety and liveness were introduced by Lamport [39] and refined by Alpern and Schneider [32]. A safety property (such as mutual exclusion or never deciding on conflicting values) stipulates that "nothing bad happens" and is consequently expressed as a property on all the prefixes of a computation. In contrast, a liveness property (such as eventually entering a critical section or eventually reaching agreement) stipulates that "something eventually happens". A liveness property involves consequently the whole computation. More on the liveness notion can be found in [32].

The notion of wait-freedom originated in the work of Lamport [11]. An associated theory was developed by Herlihy [7].

The notion of solo-termination was presented implicitly in [34]. It has been introduced as a progress property in [37] under the name *obstruction-free* synchronization. That notion has been formalized in [33]. More developments on obstruction-freedom can be found in [35]. The minimal knowledge on process failures needed to transform any solo-terminating implementation into a wait-free one was investigated in [36]. Other liveness properties (also called progress conditions) are presented in [38].

Chapter 4

Safe, regular and atomic registers

4.1 Introduction

This part of the book is devoted to the construction of the simplest linearizable objects that are usually considered, namely shared *storage* objects that provide their users with two basic operations: *read* and *write*. These objects are usually called *registers*, and linerarizable registers are called *atomic registers*. In particular, we focus on how to wait-free implement such atomic registers using "weaker" registers. Again, the picture to have in mind is one where the weak registers are provided in hardware and the strongest registers are emulated in software to facilitate the job of the application's developer.

This chapter defines different sorts of registers and these differences depend on three dimensions: (a) the capacity of a register, (b) the access pattern to a register and (c) the behavior of a register in face of concurrency. The capacity of a register conveys the range of values it can store and we will in particular distinguish registers that can store a binary value from those than can store any number (possibly an infinite number) of values. The access pattern or a register conveys the number of processes that can read (resp. write) in a register. Finally, we will distinguish registers that do not provide any guarantee if accessed concurrently at one extreme, from those that ensure linearizability at the other extreme (i.e., atomic registers).

The weakest kind of a shared register is one that can only store one bit of information, can be read by a single process p, can be written by a single process q, and does not ensure any guarantee on the value read by p when p and q access it concurrently. We will show how, using multiple such registers, we can construct an atomic register that can store an arbitrary number of values and be read and written by any number of processes. This construction will be presented incrementally, going through intermediate kinds of registers, interesting in their own right.

An algorithm used to implement a register of a given kind from one of another kind is sometimes called *transformation* or *reduction*, the first high-level register being "reduced" to the second register used as a base object in the implementation. We also say that the high-level register is emulated by the second one.

4.2 The many faces of registers

The capacity of a register According to the operations on a register issued by the processes, read operations on the register can return different values at different times. So, the first dimension that characterizes a register is related to its capacity, i.e., how much information it can contain.

The simplest kind of register is the *binary* register: it can only store a single bit 0 or 1. We talk about a *shared bit*, or simply a *bit*.

More generally, a *multi-valued* register may store two or more distinct values. A multi-valued register can be bounded or unbounded. A *bounded* register is one whose value range contains exactly b distinct values (e.g., the values from 0 until $b-1$) where b is typically a constant known by the processes. Otherwise the register is said to be *unbounded*.

A register that can contain b distinct values is said to be b*-valued*. Its binary representation requires $B = \lceil \log_2 b \rceil$ bits. Its unary representation is more expensive as it requires b bits (the value v being then represented with a bit equal to 1 followed by $v-1$ bits equal to 0).

Access patterns This dimension concerns the sets of processes that can read from or write into the register. A register is called *single-writer*, denoted 1W, (resp., *single-reader*, noted 1R) if only one specific process, known in advance, and called the *writer* (resp., the *reader*) can invoke the write (resp., read) operation on the register. A register that can be written (resp., read) by any process is called a *multi-writer* (resp., *multi-reader*) register. Such a register is denoted MW (resp., MR).

For instance, a binary 1WMR register is a register that (a) can contain only 0 or 1, (b) can be read by all the processes but (c) written by a single process.

The concurrent behavior of a register When accessed sequentially, the behavior of a register is simple to define: a read invocation returns the last value written. When accessed concurrently, the semantics is more involved and several variants have been considered. We overview these variants in the following.

4.3 Safe, regular and atomic registers

We consider three kinds of registers that vary according to their behavior in the presence of concurrent accesses. The differences are depicted in the value returned by a read operation invoked on the register concurrently with a write operation. When there is no concurrency, the behavior is the same in all cases. For the one-writer case, all the registers defined below preserve the following invariant:

- A read that is not concurrent with a write (i.e., their executions do not overlap) returns the last written value.

4.3.1 Safe registers

A *safe* register is the weakest traditionally considered in distributed computing. It has a single writer, and, since we assume that every process is sequential, allows for no concurrent writes. A safe register only guarantees that:

- A read that is concurrent with one or several writes returns any element of the value range of the register.

It is important to see that, in the presence of concurrency, the value returned by a read operation can possibly be a value that has never been written. The only constraint is that the value needs to be in the range of the register. To illustrate this, consider a safe register that can contain only $b = 3$ values, e.g., $1, 2$ and 3. The register is bounded. Assuming that the current value is 1, consider a write of value 2 that is concurrent with a read operation. That read operation can returns 1, 2 or even 3. It cannot return 5 as that value is not in the range of the safe register.

An interesting particular case is the binary 1WMR (one-writer-one-reader) safe register. This can contain only 0 and 1 and can be seen as modeling a flickering bit. Whatever its previous value, the value of the

register can flicker during a write operation and only when the write finishes the register stabilizes to its final value (the value just written) and keep that value until the next write.

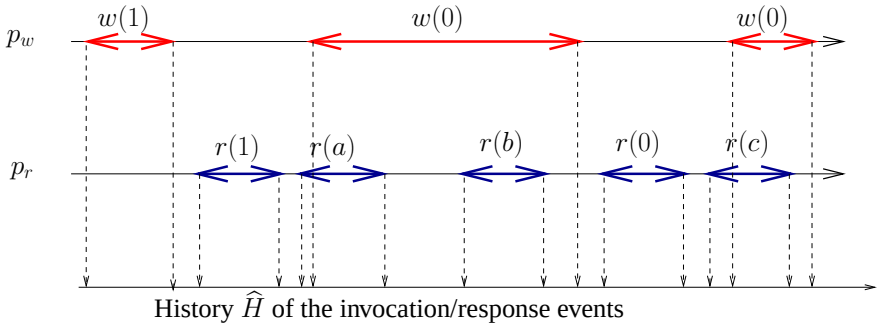

Figure 4.1: History of a register

Value returned	a	b	c
Safe	$1/0$	$1/0$	$1/0$
Regular	$1/0$	$1/0$	0
Atomic	1	$1/0$	0
	0	0	0

Table 4.1: Safe, regular and atomic registers

An example of the behavior of a binary safe register (i.e., a safe bit) is depicted in Figure 4.1 and Table 4.1. We consider there a 1W1R safe register: only one reader is involved. The writer process is denoted p_w whereas the reader process is denoted p_r ($w(v)$ stands for a write operation that writes the value v; similarly, $r(v)$ stands for a read operation that obtains the value v). As the first and the fourth read operations do not overlap a write, they return the last written value namely, 1 for the first read and 0 for the fourth one. The values returned by the other read operations are denoted a, b and c. All these read operations overlap a write and can consequently return any of the values that the register can contain (this is denoted $1/0$ as the register is binary in Table 4.1). So, the last read can return 1 even if the previous value was 0 and the concurrent operation writes the very same value 0. This gives 8 possible correct executions, assuming indeed a binary safe register.

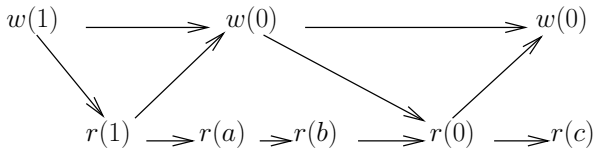

Figure 4.2: History of a safe register

Figure 4.2 depicts the corresponding history at the operation level (i.e., the partial order on the operations denoted $\rightarrow_H$). The transitive dependencies are not indicated. The unordered operations (e.g., the second $w(0)$ operation issued by p_w and $r(c)$ issued by p_r) are concurrent.

4.3.2 Regular registers

A *regular* register is also defined for the case of a single writer. It is a safe register that satisfies the following additional property:

- A read that is concurrent with one or several writes returns the value written by a concurrent write or the value written by the last preceding write.

To illustrate the regular register notion, let us again consider Figure 4.1. The values that can be possibly returned by a regular register are described in Table 4.1. The second read operation can return either the previous value or the value of the concurrent write, namely, 0 or 1. It is the same for the third read operation. In contrast, as the last write does not change the value of the register, the last read can return only the value 0. This means that 4 possible correct executions can be determined for Figure 4.1.

It is important to see that a read that overlaps several write operations can return any value among the values written by these writes as well as the value of the register before these writes. This is depicted in Figure 4.3 where value a returned by the second read can be any of 1, 2, 3, 4 or 5.

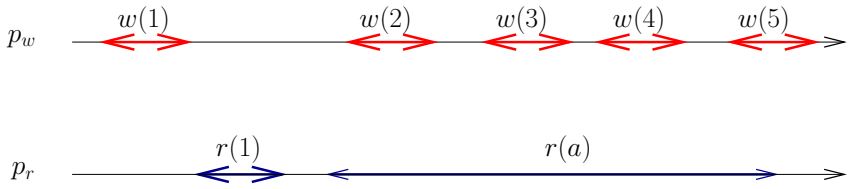

Figure 4.3: History of a regular register

4.3.3 Atomic registers

An *atomic* register is a MWMR register whose execution histories are linearizable. This means that it is possible to totally order all its read and write operations in such a way that this total order $\widehat{S}$ respects their real-time occurrence order and each read returns the value written by the last write operation that precedes it in $\widehat{S}$ (legality property).

Again, Figure 4.1 illustrates the atomicity notion for the specific case of a register. The second read $r(a)$ is concurrent with the $w(0)$ operation. Given that the previous value of the register is 1, the returned value a can be either 1 or 0. If it returns 1 (the value written by the last preceding write), then the third read can return either 1 or 0. In contrast, if the second read returns 0 (the value written by the concurrent write), only value 0 can be returned by the third read as the second read indicates that the value 1 is now overwritten by the "new" value 0. Finally, the last read can only return the value 0. It is easy to see that there are 3 possible executions when the registers are binary and atomic. As previously, the possible values returned by the three read operations concurrent with a write operation are summarized in Table 4.1.

4.3.4 Regularity and atomicity: a reading function

One important difference between regularity and atomicity is that a regular register allows for *new/old inversion*: in case two read operations are concurrent with a write, the first read may return the concurrently written value while the second read may still return the value written by a preceding write. Such a history is

not allowed by an atomic register, since the second read must succeed the first one in any linearization, and thus must return the same or a "newer" value.

For example, the history depicted in Figure 4.1 and Table 4.1, the history is correct with $a = 0$ and $b = 1$ with respect to regularity and incorrect with respect to atomicity. In that history, and considering the two consecutive read operations $r(a)$ and $r(b)$, the first', namely $r(a)$, obtains the "new" value ($a = 0$) while the second', namely $r(b)$, obtains the "old" value ($b = 1$).

Formally, we capture the difference bwtween (one-writer) regular and atomic registers using the notion of a *reading function*. A reading function is associated with a given history and maps every returned read operation $r(x)$ to some $w(x)$ in that history. Without loss of generality, we assume that every history starts with a sequential operation $w(x_0)$ that writes the initial value x_0.

We say that a reading function π associated with a history H is *regular* if (here r and w with indices denote read and write operations in H):

$A0$: $\forall\, r$: $\neg(r \rightarrow_H \pi(r))$. *(No read returns a value not yet written.)*

$A1$: $\forall\, r, w$ in H: $(w \rightarrow_H r) \Rightarrow \big(\pi(r) = w \ \vee\ w \rightarrow_H \pi(r)\big)$. *(No read obtains an overwritten value.)*

We say that a reading function is *atomic* if it is regular and satisfied the following additional property:

$A2$: $\forall\, r1, r2$: $(r1 \rightarrow_H r2) \Rightarrow \big(\pi(r1) = \pi(r2) \ \vee\ \pi(r1) \rightarrow_H \pi(r2)\big)$. *(No new/old inversion.)*

We show now determining a regular reading function is exactly what we need to show that a history can be produced by a regular register.

Theorem 3 *Let H be an execution history of a 1WMR register. H can be a history of a regular register if and only if it allows for a regular a reading function π.*

Proof Suppose that H is a history of a regular register. We define π as follows: for any read For any $r(x)$, a complete read operation in H, we define $\pi(r)$ as the last write operation $w(x)$ in H such that $\neg(r(x) \rightarrow_H w(x)$. It is easy to see that π is a regular reading function.

Now suppose that H allows for a regular reading function. Let $r(x)$ be a complete read operation in H. Then there exists a write $w(x)$ in H that either precedes or is concurrent with $r(x)$ in H ($A0$) and is not succeeded by a write that precedes $r(x)$ in H ($A1$). Thus, $r(x)$ returns either the last written or a concurrently written value. $\square_{Theorem}$ **??**

Theorem 4 *Let H be an execution history of a 1WMR register. H is linearizable if and only if it allows for an atomic a reading function π.*

Proof Given a linearizable history H, it is straightforward to construct an atomic reading function: take any S, a linearization of H and define $\pi(r)$ as the last write that precedes r in S. By construction, $\pi(r)$ satisfies properties $A0$, $A1$ and $A2$.

Now suppose that H allows for an atomic reading function π. We use π to construct S, a linearization of H, as follows.

We first construct S as the sequence of all writes that took place in H in the order of appearance. Since we have only one writer, the writes are totally ordered. (In case the last write is incomplete, we add to S its complete version.) Then we put every complete operation r immediately after $\pi(r)$, making sure that:

$$\textit{if } \pi(r1) = \pi(r2) \textit{ and } r1 \rightarrow_H r2, \textit{ then } r1 \rightarrow_S r2.$$

Clearly, S is legal: the reading function guarantees that $\pi(r)$ writes the value read by r, and thus every read in S returns the last written value.

To show that $\rightarrow_H \subseteq \rightarrow_S$, we consider the following four possible cases ($w1$ and $w2$ (resp., $r1$ and $r2$) denote here write (resp., read) operations):

- $w1 \rightarrow_H w2$. By the very construction of S (that considers the order on write operations performed by the writer), we have $w1 \rightarrow_S w2$.

- $r1 \rightarrow_H r2$. By $A2$, we have $\pi(r1) = \pi(r2)$ or $\pi(r1) \rightarrow_H \pi(r2)$.

  If $\pi(r1) = \pi(r2)$, as $r1$ started before $r2$ (case assumption), due to the way S is constructed, $r1$ is ordered before $r2$ in S, and we have consequently $r1 \rightarrow_S r2$.

  If $\pi(r1) \rightarrow_H \pi(r2)$, as (1) $r1$ and $r2$ are placed just after $\pi(r1)$ and $\pi(r2)$, respectively, and (2) $\pi(r1)1 \rightarrow_S \pi(r2)$ (see the first item), the construction of S ensures $r1 \rightarrow_S r2$.

- $r1 \rightarrow_H w2$. By $A0$, either $\pi(r1)$ is concurrent with $r1$ or $\pi(r1) \rightarrow_H r1$. Since $r1 \rightarrow_H w2$ and all writes are totally ordered, we have $\pi(r1) \rightarrow_H w2$. By construction of S, since $\pi(r1)$ is the last write preceding $r1$ in S, $r1 \rightarrow_S w2$.

- $w1 \rightarrow_H r2$. By $A1$ we have $\pi(r2) = w1$ or $w1 \rightarrow_H \pi(r2)$.

  Case $\pi(r2) = w1$. As $r2$ is placed just after $\pi(r2)$ in S, we have $\pi(r2) = w1 \rightarrow_S r2$.

  Case $w1 \rightarrow_H \pi(r2)$. As (2) $w1 \rightarrow_H \pi(r2) \Rightarrow w1 \rightarrow_S \pi(r2)$ (first item), and (2) $\pi(r2) \rightarrow_S r2$ ($r2$ is ordered just after $\pi(r2)$ in S), we obtain (by transitivity of $\rightarrow_S$) $w1 \rightarrow_S r2$.

Finally, since S contains all complete operations of H and preserves $\rightarrow_H$, H is indistinguishable from S for every reader, modulo the last incomplete read operation (if any).

Thus, S is a legal sequential history that is equivalent to a completion of H and preserves $\rightarrow_H$. $\square_{Theorem}$ **??**

Now we can say that a history of a regular register suffers from new/old inversion if it allows for no atomic reading function. Theorems 3 and 4 imply that an atomic register can be seen as a regular register that never suffers from new/old inversion.

It follows from the fact that atomicity (linearizability) is a local property that a set of 1WMR regular registers behave atomically if each of them *independently from the others* is written by a single process and satisfies the "no new/old inversion" property.

4.3.5 From very weak to very strong registers

To summarize, there are different kinds of registers and the differences depend on several dimensions. It is appealing to ask whether registers of strong kinds can be constructed in software (emulated) using registers of weak kinds. As pointed out in the introduction of this chapter, and this might look surprising, it is indeed possible to emulate a multi-valued MWMR atomic register using binary 1W1R safe registers. Next sections are devoted to proving that.

In general, what we call a (register) *transformation* is here an algorithm that builds a register R with certain properties, called a high-level register, from other registers featuring different properties. These registers, used in the implementation, are called low-level or base registers. These low level registers are called *base registers*. Of course, for a transformation to be interesting, the base registers it uses have to provide weaker properties than the high level register we want to construct. Typically:

- The base registers are safe (resp., regular) while the high level register is regular (resp., atomic).
- The base registers are 1W1R (resp., 1WMR) while the constructed register is 1WMR (resp., MWMR).
- The base registers are binary whereas the high level register is multi-valued.

The transformations also vary according to the number and size of base registers considered. Basically:

- The number of base registers needed to build the high level register might or not depend on the total number of processes in the system, i.e., readers and writers.
- The amount of information used to build the high level register might be bounded or not. Sometimes, the transformation algorithm uses sequence numbers that can arbitrarily grow and is inherently unbounded. In general, and except for few constructions, bounded transformations are much more difficult to design and prove correct than unbounded ones. From a complexity point of view, bounded ones are better.

In the following, we proceed as follows.

1. We illustrate the notion of transformation algorithm by presenting first two simple (bounded) algorithms. The first constructs a 1WMR safe register out of a number of 1W1R safe registers. The second builds a binary 1WMR regular register out of a binary 1WMR safe register. The combination of these algorithms already shows that we can implement a binary 1WMR regular register using a number of binary 1W1R safe registers.

2. We then show how to transform a binary 1WMR register that provides certain semantics (safe, regular or atomic) into a multi-valued 1WMR register that preserves the same semantics. The three transformations we present here are all bounded. The combination of the second of these with those above shows that we can implement a multi-valued 1WMR regular register using a number of binary 1W1R safe registers.

3. We finally show how to transform a 1W1R regular register into a MWMR atomic register. We go through three intermediate (unbounded) transformations: from a 1W1R regular register into a 1W1R atomic register, then to a 1WMR atomic register, and finally to a MWMR register. These, with the combination pointed out above, shows that we can construct a multi-valued MWMR atomic register using binary 1W1R safe registers.

4.4 Two simple bounded transformations

This section describes two very simple bounded transformations. We focus on safe and regular registers. (Recall that these kinds of registers are defined for systems with a single writer for each register.) The first transformation extends a single-reader register, being safe or regular, to multiple readers. The second transformation transforms a shared safe bit into a regular one.

4.4.1 Safe/regular registers: from single reader to multiple readers

We present here an algorithm that implements a 1WMR safe (resp., regular) register using 1W1R safe (regular) registers. In short, the transformation allows for multiple readers instead of single readers. Not surprisingly, the idea is to emulate the multi-reader register using several single-reader registers.

The transformation, described in Figure 4.4, is very simple. The constructed register R is built from n 1W1R base registers, denoted $REG[1:n]$, one per reader process. (We consider a system of n processes and all are potential readers.) A reader p_i reads the base register $REG[i]$ it is associated with, while the single writer writes all the base registers (in any order).

It is important to see that this transformation is bounded: it uses no additional control information beyond the actual value stored, and each base register can be of the same size (measured in number of bits) as the multi-readers register we want to build.

Interestingly, with the same algorithm, if the base 1W1R registers are regular, than the resulting 1WMR register we then obtain is regular.

```
operation R.write(v):
    for_all j in {1, ..., n} do REG[i] ← v end_do;
    return ()

operation R.read() issued by p_i :
    return (REG[i])
```

Figure 4.4: From 1W1R safe/regular to 1WMR safe/regular (bounded transformation)

Theorem 5 *Given one base safe (resp., regular) 1W1R register per reader, the algorithm described in Figure 4.4 implements a 1WMR safe (resp., regular) register.*

Proof Assume first that base registers are safe 1W1R registers. It follows directly from the algorithm that a read of R (i.e., $R.read()$) that is not concurrent with a $R.write()$ operation obtains the last value deposited in the register R. The obtained register R is consequently safe while being 1WMR.

Let us now consider the case where the base registers are regular. We will argue that the high-level register R constructed by the algorithm is a 1WMR regular one. The fact that R allows for multiple readers is by construction. Because a regular register is safe, and by the argument above (for the case where the base registers are safe), we only need to show that a read operation $R.read()$ that is concurrent with one or more write operations $R.write(v)$, $R.write(v')$, etc., returns one of the values $v, v', ...$ written by these concurrent write operations, or the value of R before these write operations.

Let p_i be any process that reads some value from R. When p_i reads the base register $REG[i]$, while executing operation $R.read()$, p_i obtains (a) the value of a concurrent write on this base register $REG[i]$ (if any) or (b) the last value written on $REG[i]$ before such concurrent write operations. This is because the base register $REG[i]$ is itself regular. In case (a), the value v obtained is from a $R.write(v)$ that is concurrent with the $R.read()$ of p_i. In case (b), the value v obtained can either be (b.1) from a $R.write(v)$ that is concurrent with the $R.read()$ of p_i , or (b.2) from the last value written by a $R.write()$ before the $R.read()$ of p_i. It follows that the constructed register R is regular. $\Box_{Theorem\ 5}$

It is important to see that the algorithm of Figure 4.4 does not implement a 1WMR atomic register even when every base register $REG[i]$ is a 1W1R atomic register. Roughly speaking, this is because the transformation can cause a new/old inversion problem, even if the base registers preclude these. To show

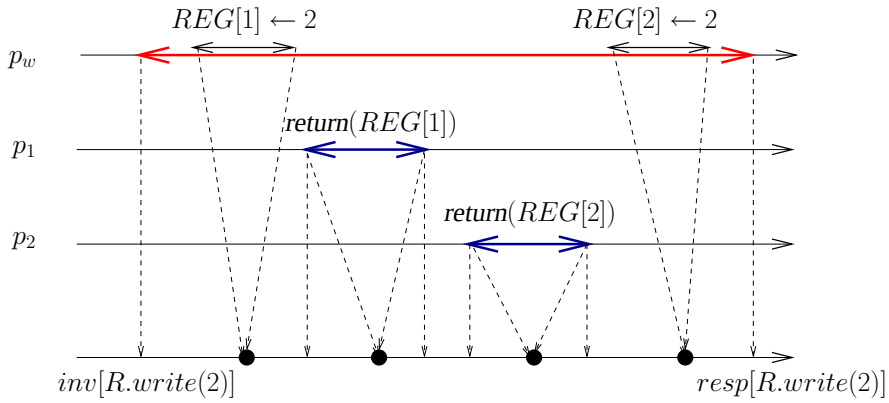

Figure 4.5: A counter-example

this, let us consider the counter-example described in Figure 4.5. The example involves one writer p_w and two readers p_1 and p_2. Assume the register R implemented by the algorithm contains initially the value 1 (which means that we initially have $REG[1] = REG[2] = 1$). To write value 2 in R, the writer first executes $REG[1] \leftarrow 2$ and then $REG[2] \leftarrow 2$. The duration of these two write operations on base registers can be arbitrarily long. (Remember that processes are asynchronous and there is no bound on their execution speed). Concurrently, p_1 reads $REG[1]$ and returns 2 while later (as indicated on the figure) p_2 reads $REG[2]$ and returns 1. The linearization order on the two base atomic registers is depicted on the figure (bold dots). It is easy to see that, from the point of view of the constructed register R, there is a new/old inversion as p_1 reads first and obtains the new value, while p_2 reads after p_1 and obtains the old value. The constructed register is consequently not atomic.

4.4.2 Binary multi-reader registers: from safe to regular

The aim is here to build a regular binary register from a safe binary register, i.e., to construct a regular bit out of a safe one. As we shall see, the algorithm is very simple, precisely because the register to be implemented, R, can only contain one out of two values (0 or 1).

The difference between a safe and a regular register is only visible in the face of concurrency. That is, the value to be returned in the regular case has to be a value concurrently written or the last value written, whereas no such restriction exists for safe semantics. The fact that we only consider a shared bit means however that the issue to be addressed is restricted: only one out of two values can be returned anyway. To illustrate the issue, assume that the regular register is directly implemented using a safe base register: every read (resp. write) on the high-level register is directly translated into a read (resp. write) on the base (safe) register. Assume this register has value 0 and there is a write operation that writes the very same value 0. As the base register is only safe, it is possible that a concurrent read operation obtains value 1, which might have never been written.

The way to fix this problem is to preclude the actual writing in the base register if the value to be written in the high-level register is the *same* as the value previously written. If the value to be written is *different* from the previous value, then it is okay to write the value in the base register: a concurrent read can obtain the other value (remember that only two values are possible) and this is fine with the regularity semantics. With this strategy, a read operation concurrent with one or more write operations obtains the value before these write operations or the value written by one of these operations.

The transformation algorithm is described in Figure 4.6. Besides a safe register REG shared between the reader and the writer, the algorithm requires that the writer uses a local register $prev\_val$ that contains the previous value that has been written in the base safe register REG. Before writing a value v in the high-level regular register, the writer checks if this value v is different from the value in $prev\_val$ and, only in that case, v is written in REG.

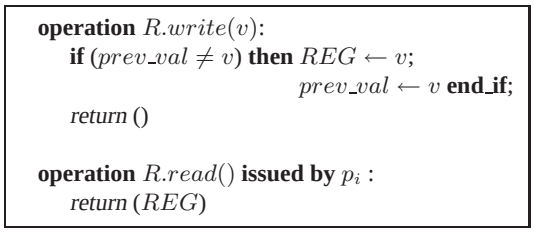

Figure 4.6: From a binary safe to a binary regular register (bounded transformation)

Theorem 6 *Given a 1WMR binary safe register, the algorithm described in Figure 4.6 implements a 1WMR binary regular register.*

Proof The proof is an immediate consequence of the following facts. (1) As the underlying base register is safe, a read that is not concurrent with a write obtains the last written value. (2) As the underlying base register always alternate between 0 and 1, a read concurrent with one or more write operations obtains the value of the base register before these write operations or one of the values written by such a write operation. $\square_{Theorem\ 6}$

As we pointed out in the overview description above, the transformation heavily exploits the fact that the constructed register R can only contain one out of two possible values (0 or 1). It does not work for multi-valued registers. The transformation does not implement an atomic register either as it does not prevent a new/old inversion. Notice also that If the safe base binary register is 1W1R, then the algorithm implements a 1W1R regular binary register.

4.5 From binary to b-valued registers

This section presents three transformations from binary registers to multi-valued registers. These are called b-valued registers in the sense that their value range contains b distinct values; we assume that $b > 2$. Our transformations have three characteristics.

1. Although we assume that the base binary registers (bits) are 1WMR registers and we transform these into 1WMR b-valued registers, our algorithms can also be used to transform 1W1R bits into 1W1R b-valued registers; i.e., if the base bits allow only a single reader, then the same algorithm implements a b-valued bit.

2. Our transformations preserve the semantics of the base registers in the following sense: if the base bits have semantics X (safe, regular or atomic), then the resulting high-level (b-valued) registers also have semantics X (safe, regular or atomic).

3. The transformations are bounded. There is a bound on the number of base registers used, as well as on the amount of memory needed within each register.

4.5.1 From safe bits to safe b-valued registers

Overview The first algorithm we present here uses a number of safe bits in order to implement a b-valued register R. We assume that b is an integer power of 2, i.e., $b = 2^B$ where B is an integer. It follows that (with a possible pre-encoding if the b distinct values are not the consecutive values from 0 until $b-1$) the binary representation of the b-valued register R we want to build consists of exactly B bits. In a sense, any combination of B bits defines a value that belongs to the range of R (notice that this would not be true if b was not an integer power of 2).

The algorithm relies on this encoding for the values to be written in R. It uses an array $REG[1:B]$ of 1WMR safe bit registers to store the current value of R. Given $\mu_i = REG[i]$, the binary representation of the current value of R is $\mu_1 \dots \mu_B$. The corresponding transformation algorithm is given in Figure 4.7.

```
operation R.write(v):
    let μ1 ... μB be the binary representation of v;
    for_all j in {1, ..., B} do REG[j] ← μj end_do;
    return ()

operation R.read() issued by pi:
    for_all j in {1, ..., B} do μj ← REG[j] end_do;
    let v be the value whose binary representation is μ1 ... μB;
    return (v)
```

Figure 4.7: Safe register: from bits to b-valued register

Space complexity As $B = \log_2(b)$, the memory cost of the algorithm is logarithmic with respect to the size of the value range of the constructed register R. This follows from the binary encoding of the values of the high level register R.

Theorem 7 *Given $b = 2^B$ and B 1WMR safe bits, the algorithm described in Figure 4.7 implements a 1WMR b-valued safe register.*

Proof A read of R that does not overlap a write of R obtains the binary representation of the last value that has been written into R and is consequently safe to return. A read of R that overlaps a write of R can obtain any of b possible values whose binary encoding uses B bits. As every possible combination of the B base bit registers represents one of the the b values that R can potentially contain (this is because $b = 2^B$), it follows that a read concurrent with a write operation returns a value that belongs to the range of R. Consequently, R is a b-valued safe register. $\square_{Theorem\ 7}$

It is interesting to notice that this algorithm does not implement a regular register R even when the base bits are regular. A read that overlaps a write operation that changes the value of R from $0 \dots 0$ to $1 \dots 1$ (in binary representation) can return any value, i.e., even one that was never written. The reader (the human, not the process) can check that requiring a specific order according to which the array $REG[1:B]$ is accessed does not overcome this issue.

4.5.2 From regular bits to regular b-valued registers

Overview A way to build a 1WMR regular b-valued register R from regular bits is to employ unary encoding. Considering an array $REG[1:b]$ of 1WMR regular bits, the value $v \in [1..b]$ is represented by 0s in bits 1 through $v-1$ and 1 in the vth bit.

The algorithm is described in Figure 4.8. The key idea is to write into the array $REG[1 : b]$ in one direction, and to read it in the opposite direction. To write v, the writer first sets $REG[v]$ to 1, and then "cleans" the array REG. The cleaning consists in setting the bits $REG[v-1]$ until $REG[1]$ to 0. To read, a reader traverses the array $REG[1 : b]$ starting from its first entry ($REG[1]$) and stops as soon as it discovers an entry j such that $REG[j] = 1$. The reader then returns j as the result of the read operation. It is important to see that a read operation starts reading first the "cleaned" part of the array. On the other hand, the writing is performed in the opposite direction, from $v - 1$ until 1.

It is also important to notice that, even when no write operation is in progress, it is possible that several entries of the array be equal to 1. The value represented by the array is then the value v such that $REG[v] = 1$ and for all the entries $1 \leq j < v$ we have $REG[j] = 0$. Those entries are then the only meaningful entries. The other entries can be seen as a partial evidence on past values of the constructed register.

The algorithm assumes that the register R has an initial value, say v. The array $REG[1 : b]$ is accordingly initialized, i.e., $REG[j] = 0$ for $1 \leq j < v$, $REG[v] = 1$, and $REG[j] = 0$ or 1 for $v < j \leq b$.

```
operation R.write(v):
    REG[v] ← 1;
    for j from v − 1 step −1 until 1 do REG[j] ← 0 end_do;
    return ()

operation R.read() issued by p_i:
    j ← 1;
    while (REG[j] = 0) do j ← j + 1 end_do;
    return (j)
```

Figure 4.8: Regular register: from bits to b-valued register

Two observations are in order:

1. In the writer's algorithm, once set to 1, the "last" base register $REG[b]$ keeps that value forever. In a sense, setting this register to 1 makes it useless: the writer never writes in it again, and when it has to read it, a reader might by default consider its value to be 1.

2. The reader's algorithm does not write base registers. This means that the algorithm handles any number of readers. Of course, the base registers have to be 1WMR if there are several readers (as each reader reads the base registers), and can be 1W1R when there a single reader is involved.

Space complexity The memory cost of the transformation algorithm is b base bits, i.e., it is linear with respect to the size of the value range of the constructed register R. This is a consequence of the unary encoding of these values[1].

Lemma 1 *Consider the algorithm of Figure 4.8. Any $R.read()$ or $R.write()$ operation terminates. Moreover, the value v returned by a read belongs to the set $\{1, \ldots, b\}$.*

Proof A $R.write()$ operation trivially terminates (as by definition the **for** loop always terminates). For the termination of the $R.read()$ operation, let us first make the following two observations:

[1] Let B be the number of bits required to obtain a binary representation of a value of R. It is important to see that, as $B = \log_2(b)$, the cost of the construction is exponential with respect to this number of bits.

- At least one entry of the array REG is initially equal to 1. Then, it follows from the write algorithm that each time the writer changes the value of a base register $REG[x]$ from 1 to 0, it has previously set to 1 another entry $REG[y]$ such hat $x < y \leq b$.

  Consequently, if the writer updates $REG[x]$ from 1 to 0 while concurrently the reader reads $REG[x]$ and obtains the new value 0, we can conclude that a higher entry of the array has the value 1.

- If, while its previous value is 1, the reader reads $REG[x]$ and concurrently the writer updates $REG[x]$ to the same value 1, the reader obtains value 1, as the base register is regular [2].

It follows from these observations that a sequential scanning of the array REG (starting at $REG[1]$) necessarily encounters en entry $REG[v]$ whose reading returns 1. As the running index of the **while** loop starts at 1 and is increased by 1 each time the loop body is executed, it follows that the loop always terminates, and the value j it returns is such that $1 \leq j \leq b$. $\square_{Lemma\ 1}$

Remark The previous lemma relies heavily on the fact that the high-level register R can contain only b distinct values. The lemma would no longer be true if the value range of R was unbounded. A $R.read()$ operation could then never terminate in case the writer continuously writes increasing values. To illustrate that, consider the following scenario. Let $R.write(x)$ be the last write operation terminated before the operation $R.read()$, and assume there is no concurrent write operation $R.write(y)$ such that $y < x$. It is possible that, when it reads $REG[x]$, the reader finds $REG[x] = 0$ because another $R.write(y)$ operation (with $y > x$) updated $REG[x]$ from 1 to 0. Now, when it reads $REG[y]$, the reader finds $REG[y] = 0$ because another $R.write(z)$ operation (with $z > y$) updated $REG[y]$ from 1 and so on. The read can then never terminate.

Theorem 8 *Given b 1WMR regular bits, the algorithm described in Figure 4.8 implements a 1WMR b-valued regular register.*

Proof Let us first consider a read operation that is not concurrent with any write, and let v the last written value. It follows from the write algorithm that, when $R.write(v)$ terminates, the first entry of the array that equals 1 is $REG[v]$ (i.e., $REG[x] = 0$ for $1 \leq x \leq v - 1$). Because a read traverses the array starting from $REG[1]$, then $REG[2]$, etc., it necessarily reads until $REG[v]$ and returns accordingly the value v.

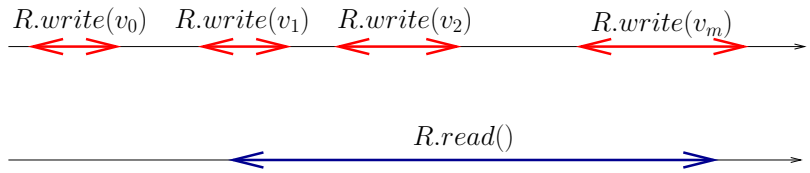

Figure 4.9: A read with concurrent writes

Let us now consider a read operation $R.read()$ that is concurrent with one or more write operations $R.write(v_1)$, ..., $R.write(v_m)$ (as depicted in Figure 4.9). Moreover, let v_0 be the value written by the last write operation that terminated before the operation $R.read()$ starts (or the initial value if there is no such write operation). As a read operation always terminates (Lemma 1), the number of write operations

[2] If the base register was only safe, the reader could obtain value 0.

concurrent with the $R.read()$ operation is finite. We have to show that the value v returned by $R.read()$ is one of the values v_0, v_1, ..., v_m. We proceed with a case analysis.

1. $v < v_0$.
   No value that is both smaller than v_0 and different from v_x ($1 \leq x \leq m$) can be output. This is because (1) $R.write(v_0)$ has set to 0 all entries from $v_0 - 1$ until the first one, and only a write of a value v_x can set $REG[v_x]$ to 1; and (2) as the base registers are regular, if $REG[v']$ is updated by a $R.write(v_x)$ operation from 0 to the same value 0, the reader cannot concurrently reads $REG[v'] = 1$. It follows from that observation that, if $R.read()$ returns a value v smaller than v_0, then v has necessarily been written by a concurrent write operation, and consequently $R.read()$ satisfies the regularity property.

2. $v = v_0$.
   In this case, $R.read()$ trivially satisfies the regularity property. Notice that it is possible that the corresponding write operation be some $R.write(v_x)$ such that $v_x = v_0$.

3. $v > v_0$.
   From $v > v_0$, we can conclude that the read operation obtained 0 when it read $REG[v_0]$. As $REG[v_0]$ was set to 1 by $R.write(v_0)$, this means that there is a $R.write(v')$ operation, issued after $R.write(v_0)$ and concurrent with $R.read()$, such that $v' > v_0$, and that operation has executed $REG[v'] \leftarrow 1$, and has then set to 0 at least all the registers from $REG[v' - 1]$ until $REG[v_0]$. We consider three cases.

   (a) $v_0 < v < v'$.
   In this case, as $REG[v]$ has been set to 0 by $R.write(v')$, we can conclude that there is a $R.write(v)$, issued after $R.write(v')$ and concurrent with $R.read()$, that updated $REG[v]$ from 0 to 1. The value returned by $R.read()$ is consequently a value written by a concurrent write operation. The regularity property is consequently satisfied by $R.read()$.

   (b) $v_0 < v = v'$.
   The regularity property is then trivially satisfied by $R.read()$, as $R.write(v')$ and $R.read()$ are concurrent.

   (c) $v_0 < v' < v$.
   In this case, $R.read()$ missed the value 1 in $REG[v']$. This can only be due to a $R.write(v'')$ operation, issued after $R.write(v')$ and concurrent with $R.read()$, such that $v'' > v'$, and that operation has executed $REG[v''] \leftarrow 1$, and has then set to 0 at least all the registers from $REG[v'' - 1]$ until $REG[v']$.
   We are now in the same situation as the one described at the beginning of item 3, where v_0 and $R.write(v')$ are replaced by v' and $R.write(v'')$. As (a) the number of values between v_0 and b is finite and (b) the read operation $R.read()$ terminates, if follows that this operation eventually terminates in 3a or 3b, which completes the proof of the theorem.

$\square_{Theorem\ 8}$

A counter-example for atomicity Figure 4.10 shows that, even if all base registers are atomic, the algorithm we just presented (Figure 4.8) does not implement an atomic b-valued register.

Let us assume that $b = 5$ and the initial value of the register R is 3, which means that we initially have $REG[1] = REG[2] = 0$, $REG[3] = 1$ and $REG[4] = REG[5] = 0$. The writer issues first $R.write(1)$

and then $R.write(2)$. There are concurrently two read operations as indicated on the figure. The first read operation returns value 2 while the second one returns value 1: there is a new/old inversion. The last line of the figure depicts a linearization order S of the read and write operations on the base binary registers. (As we can see, each base object taken alone is linearizable. This follows from the fact that linearizability is a local property, see the first chapter).

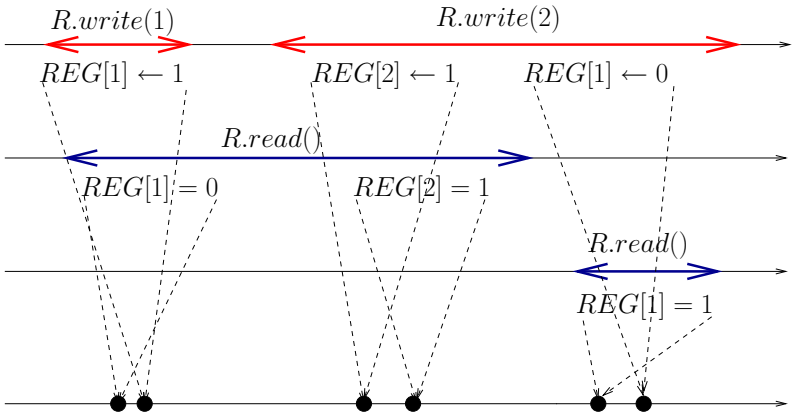

Figure 4.10: A counter-example for atomicity

4.5.3 From atomic bits to atomic b-valued registers

As just seen, the algorithm of Figure 4.8 does not work if the goal is to build a b-valued atomic register from atomic bits. Interestingly, a relatively simple modification of its read algorithm makes that possible by preventing the new/old inversion phenomenon.

Overview The idea consists in decomposing a $R.read()$ operation in two phases. The first phase is the same as in the read algorithm of Figure 4.8 : base registers are read in ascending order, until an entry equal to 1 is found; let j be that entry. The second phase traverses the array in the reverse direction (from j to 1), and determines the smallest entry that contains value 1: this is then returned. So, the returned value is determined by a double scanning of a "meaningful" part of the REG array.

The new algorithm is given in Figure 4.11. To understand the underlying idea, let us consider the first $R.read()$ operation depicted in Figure 4.10. After it finds $REG[2] = 1$, the reader changes its scanning direction. The reader then finds $REG[1] = 1$ and returns consequently value 1. In the figure, the second read obtains 1 in $REG[1]$ and consequently returns 1. This shows that, in the presence of concurrency, this construction does not strive to eagerly return a value. Instead, value v returned by a read operation has to be "validated" by an appropriate procedure, namely, all the "preceding" base registers $REG[v-1]$ until $REG[1]$ have to be found equal to 0 when rereading them.

Theorem 9 *Given b 1WMR atomic bits, the algorithm described in Figure 4.11 implements a 1WMR atomic b-valued register.*

Proof The proof consists in two parts: (1) we first show that the implemented register is regular, and then (2) we show that it does not allow for new/old inversions. Applying Theorem 10 proves then that the con-

```
operation R.write(v):
    REG[v] ← 1;
    for j from v − 1 step −1 until 1 do REG[j] ← 0 end_do;
    return ()

operation R.read() issued by p_i:
    j_up ← 1;
(1) while (REG[j_up] = 0) do j_up ← j_up + 1 end_do;
(2) j ← j_up;
(3) for j_down from j_up − 1 step −1 until 1 do
(4)                                  if (REG[j_down] = 1) then j ← j_down end_if
    end_do;
    return (j)
```

Figure 4.11: Atomic register: from bits to b-valued register

structed register is a 1WMR atomic register.

Let us first show that the implemented register is regular. Let $R.read()$ be a read operation and j the value it returns. We consider two cases:

- $j = j\_up$ (j is determined at line 2).
  The value returned is then the same as the one returned by the algorithm described in Figure 4.8. It follows from theorem 8 that the value read is then either the value of the last preceding write or the new value of an overlapping write.

- $j < j\_up$ (j is determined at line 4; let us observe that, due to the construction, the case $j > j\_up$ cannot happen).
  In that case, the read found $REG[j] = 0$ during the ascending loop (line 1), and $REG[j] = 1$ during the descending loop (line 4). Due to the atomicity of the base $REG[j]$ register, this means that a write operation has written $REG[j] = 1$ between these two readings of that base atomic register. It follows that the value j returned has been written by a concurrent write operation.

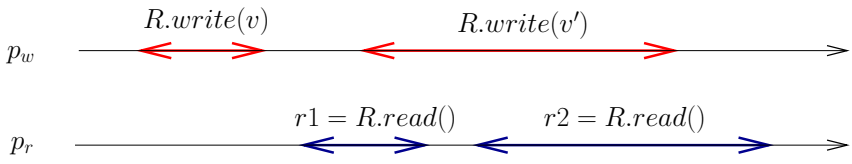

Figure 4.12: There is no new/old inversion

To show that there is no new/old inversion, let us consider Figure 4.12. There are two write operations, and two read operations $r1$ and $r2$ that are concurrent with the second write operation. (The fact that the read operations are issued by the same process or different processes is unimportant for the proof.) As the constructed register R is regular, both read operations can return v or v'. If the first read operation $r1$ returns v, the second read can return either v or v' without entailing a new/old inversion. So, let us consider the case where $r1$ returns v'. We show that the second read $r2$ returns v'', where v'' is v' or a value written by a more recent write concurrent with this read. If $v'' = v'$, then there is no new/old inversion. So, let us

consider $v'' \neq v'$. As $r1$ returns v', $r1$ has sequentially read $REG[v'] = 1$ and then $REG[v' - 1] = 0$ until $REG[1] = 0$ (lines 2-4). Moreover, $r2$ starts after $r1$ has terminated ($r1 \rightarrow_H r2$ in the associated history H).

1. $v'' < v'$. In that case, a write operation has written $REG[v''] = 1$ after $r1$ has read $REG[v''] = 0$ (at line 4) and before $r2$ reads $REG[v''] = 1$ (at line 2 or 4) with $1 \leq v'' < v'$. It follows that this write operation is after $R.write(v')$ (there is a single sequential writer, and $r1$ returns v'). Consequently, $r2$ obtains a value newer than v', hence newer than v: there is no new/old inversion.

2. $v'' > v'$. In that case, $r2$ has read 1 from $REG[v'']$ and then 0 from $REG[v']$ (line 4). As $r1$ terminates (reading $REG[v'] = 1$ and returning v') before $r2$ starts, and write operations are sequential, it follows that there is a write operation, issued after $R.write(v')$, that has updated $REG[v']$ from 1 to 0.

   (a) If that operation is $R.write(v'')$, we conclude that the value v'' read by $r2$ is newer than v', and there is no new/old inversion.

   (b) If that operation is not $R.write(v'')$, it follows that there is another operation $R.write(v''')$, such that $v''' > v'$, that has updated $REG[v']$ from 1 to 0, and that update has been issued after $R.write(v')$ (that set $REG[v']$ to 1), and before $r2$ reads $REG[v'] = 0$.

   Moreover, $R.write(v''')$ is before $R.write(v'')$ (otherwise, the update of $REG[v']$ from 1 to 0 would have been done by $R.write(v'')$).

   It follows that $R.write(v''')$ is after $R.write(v')$ and before $R.write(v'')$, from which we conclude that v'' is newer than v', proving that there is no new/old inversion.

$\square_{Theorem\ 9}$

4.6 Three (unbounded) atomic register implementations

So far, none of our algorithms implements an atomic register out of a non-atomic register. (Moreover, the one that implements atomic registers implements a b-valued register out of a binary atomic on.) In the following, we present algorithms that implement *unbounded* atomic registers. Such registers can contain any number of distinct values.

We present three algorithms. All use the notion of *sequence number*. In short, this notion represents a concept of logical time. The sequence numbers are associated with each write operation and induce a total order on these operations: the total order is then exploited to ensure atomicity. These numbers are written in the base registers, which also means that such registers are unbounded, because the space of sequence numbers is the set of natural numbers.

More specifically, in the algorithms presented in this section, a base register is made up of several fields, namely:

- A data part intended to contain the value v of the constructed high-level register R.
- A control part containing a sequence number and possibly a process identity. The sequence number values increase proportionally with the number of write operations, and is consequently bounded.

Two observations are in order before diving into the details of these algorithms.

1. As we pointed out, the high-level register we construct can be unbounded and might contain, at different points in time, an arbitrary number of distinct values. In fact, the fact that the high-level register can be unbounded does not mean it has to. This depends on the application that uses this high-level register and the operations that access it.

2. The base registers are unbounded for they contain arbitrarily large sequence numbers. In fact, there are techniques to recycle sequence numbers and bound these constructions. These techniques are however pretty involved and we do not present them here.

4.6.1 1W1R registers: From unbounded regular to atomic

We show in the following how to implement an 1W1R atomic register using a 1W1R regular register. The use of sequence numbers make such a construction easy and helps in particular prevent the new/old inversion phenomenon. Preventing this, while preserving regularity, means, by Theorem 10, that the constructed register is atomic.

The algorithm is described in Figure 4.13. Exactly one base regular register REG is used in the implementation of the high-level register R. The local variable sn at the writer is used to hold sequence numbers. It is incremented for every new write in R. The scope of the local variable aux used by the reader spans a read operation; it is made up of two fields: a sequence number ($aux.sn$) and a value ($aux.val$).

Each time it writes a value v in the high-level register, R, the writer writes the pair $[sn, v]$ in the base regular register REG. The reader manages two local variables: $last\_sn$ stores the greatest sequence number it has even read in REG, and $last\_val$ stores the corresponding value. When it wants to read R, the reader first reads REG, and then compares $last\_sn$ with the sequence number it has just read in REG. The value with the highest sequence number is the one returned by the reader and this prevents new/old inversions.

```
operation R.write(v):
    sn ← sn + 1;
    REG ← [sn, v];
    return ()

operation R.read():
    aux ← REG;
    if (aux.sn > last_sn) then last_sn ← aux.sn;
                               last_val ← aux.val end_if;
    return (last_val)
```

Figure 4.13: From regular to atomic: unbounded construction

Theorem 10 *Given an unbounded 1W1R regular register, the algorithm described in Figure 4.13 constructs a 1W1R atomic register.*

Proof The proof is similar to the proof of Theorem 10. We associate with each read operation r of the high-level register R, the sequence number (denoted $sn(r)$) of the value returned by r: this is possible as the base register is regular and consequently a read always returns a value that has been written with its sequence number, that value being the last written value or a value concurrently written -if any-. Considering an arbitrary history H of register R, we show that H is atomic by building an equivalent sequential history S that is legal and respects the partial order on the operations defined by $\rightarrow_H$.

S is built from the sequence numbers associated with the operations. First, we order all the write operations according to their sequence numbers. Then, we order each read operation just after the write operation that has the same sequence number. If two reads operations have the same sequence number, we order first the one whose invocation event is first. (Remember that we consider a 1W1R register)

The history S is trivially sequential as all the operations are placed one after the other. Moreover, S is equivalent to H as it is made up of the same operations. S is trivially legal as each read follows the corresponding write operation. We now show that S respects $\rightarrow_H$.

- For any two write operations $w1$ and $w2$ we have either $w1 \rightarrow_H w2$ or $w2 \rightarrow_H w1$. This is because there is a single writer and it is sequential: as the variable sn is increased by 1 between two consecutive write operations, no two write operations have the same sequence number, and these numbers agree on the occurrence order of the write operations. As the total order on the write operations in S is determined by their sequence numbers, it consequently follows their total order in H.

- Let $op1$ be a write or a read operation, and $op2$ be a read operation such that $op1 \rightarrow_H op2$. It follows from the algorithm that $sn(op1) \leq sn(op2)$ (where $sn(op)$ is the sequence number of the operation op). The ordering rule guarantees that $op1$ is ordered before $op2$ in S.

- Let $op1$ be a read operation, and $op2$ a write operation. Similarly to the previous item, we then have $sn(op1) < sn(op2)$, and consequently $op1$ is ordered before $op2$ in S (which concludes the proof).

$\square_{Theorem\ 10}$

One might think of a naive extension of the previous algorithm to construct a 1WMR atomic register from base 1W1R regular registers. Indeed, we could, at first glance, consider an algorithm associating one 1W1R regular register per reader, and have the writer writes in all of them, each reader reading its dedicated register. Unfortunately, a fast reader might see a new concurrently written value, whereas a reader that comes later sees the old value. This is because the second reader does not know about the sequence number and the value returned by the first reader. The latter stores them locally. In fact, this can happen even if the base 1W1R registers are atomic. The construction of a 1WMR atomic register from base 1W1R atomic registers is addressed in the next section.

4.6.2 Atomic registers: from unbounded 1W1R to 1WMR

We presented in Section 4.4.1 an algorithm that builds a 1WMR safe/regular register from similar 1W1R base registers. We also pointed out that the corresponding construction does not build a 1WMR atomic register even when the base registers are 1W1R atomic (see the counter-example presented in Figure 4.5).

This section describes such an algorithm: assuming 1W1R atomic registers, it shows how to go from single reader registers to a multi-reader register. This algorithm uses sequence numbers, and requires unbounded base registers.

Overview As there are now several possible readers, actually n, we make use of several (n) base 1W1R atomic registers: one per reader. The writer writes in all of them. It writes the value as well as a sequence number. The algorithm is depicted in Figure 4.14.

We prevent new/old inversions (Figure 4.5) by having the readers "help" each other. The helping is achieved using an array $HELP[1:n, 1:n]$ of 1W1R atomic registers. Each register contains a pair (sequence number, value) created and written by the writer in the base registers. More specifically, $HELP[i, j]$

is a 1W1R atomic register written only by p_i and read only by p_j. It is used as follows to ensure the atomicity of the high-level 1WMR register R that is constructed by the algorithm.

- *Help the others.* Just before returning the value v it has determined (we discuss how this is achieved in the second bullet below), reader p_i helps every other process (reader) p_j by indicating to p_j the last value p_i has read (namely v) and its sequence number sn. This is achieved by having p_i update $HELP[i, j]$ with the pair $[sn, v]$. This, in turn, prevents p_j from returning in the future a value older than v, i.e., a value whose sequence number would be smaller than sn.

- *Helped by the others.* To determine the value returned by a read operation, a reader p_i first computes the greatest sequence number that it has ever seen in a base register. This computation involves all 1W1R atomic registers that p_i can read, i.e., $REG[i]$ and $HELP[j, i]$ for any j. p_i. Reader p_i then returns the value that has the greatest sequence number p_i has computed.

The corresponding algorithm is described in Figure 4.14. Variable aux is a local array used by a reader; its jth entry is used to contain the (sequence number, value) pair that p_j has written in $HELP[j, i]$ in order to help p_i; $aux[j].sn$ and $aux[j].val$ denote the corresponding sequence number and the associated value, respectively. Similarly, reg is a local variable used by a reader p_i to contain the last (sequence number, value) pair that p_i has read from $REG[i]$ ($reg.sn$ and $reg.val$ denote the corresponding fields).

Register $HELP[i, i]$ is used only by p_i, which can consequently keep its value in a local variable. This means that the 1W1R atomic register $HELP[i, i]$ can be used to contain the 1W1R atomic register $REG[i]$. It follows that the protocol requires exactly n^2 base 1W1R atomic registers.

```
operation R.write(v):
    sn ← sn + 1;
    for_all j in {1, ..., n} do REG[i] ← [sn, v] end_do;
    return ()

operation R.read() issued by p_i:
    reg ← REG[i];
    for_all j in {1, ..., n} do aux[j] ← HELP[j, i] end_do;
    let sn_max be max(reg.sn, aux[1].sn, ..., aux[n].sn);
    let val be reg.val or aux[k].val such that the associated seq number is sn_max;
    for_all j in {1, ..., n} do HELP[i, j] ← [sn_max, val] end_do;
    return (val)
```

Figure 4.14: Atomic register: from one reader to multiple readers (unbounded construction)

Theorem 11 *Given n^2 unbounded 1W1R atomic registers, the algorithm described in Figure 4.14 implements a 1WMR atomic register.*

Proof As for Theorem 10, the proof consists in showing that the sequence numbers determine a linearization of any history H.

Considering an history H of the constructed register R, we first build an equivalent sequential history S by ordering all the write operations according to their sequence numbers, and then inserting the read operations as in the proof of Theorem 10. This history is trivially legal as each read operation is ordered just after the write operation that wrote the value that is read. A similar reasoning similar as the one used in Theorem 10, but based on the sequence numbers provided by the arrays $REG[1 : n]$ and $HELP[1 : n, 1 : n]$, shows that S respects $\rightarrow_H$. $\square_{Theorem\ 11}$

4.6.3 Atomic registers: from unbounded 1WMR to MWMR

This section shows how to use sequence numbers to build a MWMR atomic register from n 1WMR atomic registers (where n is the number of writers). The algorithm is simpler than the previous one. An array $REG[1:n]$ of n 1WMR atomic registers is used in such a way that p_i is the only process that can write in $REG[i]$, while any process can read it. Each register $REG[i]$ stores a (sequence number, value) pair. Variables $X.sn$ and $X.val$ are again used to denote the sequence number field and the value field of the register X, respectively. Each $REG[i]$ is initialized to the same pair, namely, $[0, v_0]$ where v_0 is the initial value of R.

The problem we solve here consists in allowing the writers to totally order their write operations. To that end, a write operation first computes the highest sequence number that has been used, and defines the next value as the sequence number of its write. Unfortunately, this does not prevent two distinct concurrent write operations from associating the same sequence number with their respective values. A simple way to cope with this problem consists in associating a *timestamp* with each value, where a timestamp is a pair made up of a sequence number plus the identity of the process that issues the corresponding write operation.

The timestamping mechanism can be used to define a total order on all the timestamps as follows. Let $ts1 = [sn1, i]$ and $ts1 = [sn2, j]$ be any two timestamps. We have:

$$ts1 < ts2 \stackrel{\text{def}}{=} \big((sn1 < sn2) \vee (sn1 = sn2 \wedge i < j)\big).$$

The corresponding construction is described in Figure 4.15. The meaning of the additional local variables that are used is, we believe, clear from the context.

```
operation R.write(v) issued by p_i:
    for_all j in {1,...,n} do reg[j] ← REG[j] end_do;
    let sn_max be max(reg[1].sn,...,reg[n].sn) + 1;
    REG[i] ← [sn_max, v];
    return ()

operation R.read() issued by p_i:
    for_all j in {1,...,n} do reg[j] ← REG[j] end_do;
    let k be the process identity such that [sn, k] is the greatest times-tamp
        among the n time-stamps [reg[1].sn, 1], ... and [reg[n].sn, n];
    return (reg[k].val)
```

Figure 4.15: Atomic register: from one writer to multiple writers (unbounded construction)

Theorem 12 *Given n unbounded 1WMR atomic registers, the algorithm described in Figure 4.15 implements a MWMR atomic register.*

Proof Again, we show that the timestamps define a linearization of any history H.

Considering an history H of the constructed register R, we first build an equivalent sequential history S by ordering all the write operations according to their timestamps, then inserting the read operations as in Theorem 10. This history is trivially legal as each read operation is ordered just after the write operation that wrote the read value. Finally, a reasoning similar to the one used in Theorem 10 but based on timestamps shows that S respects $\rightarrow_H$. $\square_{Theorem\ 12}$

4.7 Concluding remark

We have shown in Section 4.6 how to build a MWMR atomic register from unbounded 1W1R regular registers. All these use sequence numbers. The only transformation from safe to regular that has been presented concerns the case of binary registers (Section 4.4.2). At least three questions are natural to ask:

- How to implement a 1W1R atomic bit from a bounded number of 1W1R safe bits? This question is of independent interest and is addressed in Chapter 4.
- How to implement a 1W1R atomic register from a bounded number of 1W1R safe bits? This question is also of independent interest and is addressed in Chapter 5.
- How to implement a MWMR atomic register from bounded 1W1R atomic registers. This question is not addressed in this book.

4.8 Bibliographic notes

The notions of safe, regular and atomic registers have been introduced by Lamport [12].

Theorem 10, and the algorithms described in Figure 4.4, Figure 4.6, Figure 4.7 and Figure 4.8 are due to Lamport [12]. The algorithm described in Figure 4.11 is due to Vidyasankar [43]. The algorithms described in Figure 4.14 and 4.15 are due to Vityani and Awerbuch [47].

The wait-free construction of stronger registers from weaker registers has always been an active research area. The interested reader can consult the following (non-exhaustive!) list where numerous algorithms are presented an analyzed [48, 49, 50, 51, 52, 40, 41, 42, 44, 45, 46].

Chapter 5

From safe bits to atomic bits: an optimal construction

5.1 Introduction

In the previous chapter, we introduced the notions of safe, regular and atomic (linearizable) read/write objects (also called registers). In the case of 1W1R (one writer one reader) register, assuming that there is no concurrency between the reader and the writer, the notions of safety, regularity and atomicity are equivalent. This is no longer true in the presence of concurrency. Several bounded constructions have been described for concurrent executions. Each construction implements a stronger register from a collection of weaker base registers. We have seen the following constructions:

- From a safe bit to a regular bit. This construction improves on the quality of the base object with respect to concurrency. Contrarily to the base safe bit, a read operation on the constructed regular bit never returns an arbitrary value in presence of concurrent write operations.
- From a bounded number of safe (resp., regular or atomic) bits to a safe (resp., regular or atomic) b-valued register. These constructions improve on the quality of each base object as measured by the number of values it can store. They show that "small" base objects can be composed to provide "bigger" objects that have the same behavior in the presence of concurrency.

To get a global picture, we miss one bounded construction that improves on the quality in the presence of concurrency, namely, a construction of an atomic bit from regular bits. This construction is fundamental, as an atomic bit is the simplest nontrivial object that can be defined in terms of *sequential* executions. Even if an execution on an atomic bit contains concurrent accesses, the execution still appears as its sequential *linearization*.

In this chapter, we first show that to construct a 1W1R atomic bit, we need at least three regular bits, two written by the writer and one written by the reader. Then we present an optimal three-bit construction of an atomic bit.

5.2 A Lower Bound Theorem

In Section 4.6.1 of Chapter 4, we presented the construction of a 1W1R atomic register from an *unbounded* regular register. The base regular register had to be unbounded because the construction was using sequence

numbers, and the value of the base register was a pair made up of the data value of the register and the corresponding sequence number. The use of sequence numbers makes sure that new/old inversions of read operations never happen.

A fundamental question is the following: Can we build a 1W1R atomic register from a finite number of regular registers that can store only finitely many values, and can be written only by the writer (of the atomic register)?

This section first shows that such a construction is impossible, i.e., the reader must also be able to write. In other words, such a construction must involve two-way communication between the reader and the writer. Moreover, even if we only want to implement one atomic bit, the writer must be able to write in *two* regular base bits.

5.2.1 Digests and Sequences of Writes

Let A be any finite sequence of values in a given set. A *digest* of A is a shorter sequence B that "mimics" A: A and B have the same first and last elements; an element appears at most once in B; and two consecutive elements of B are also consecutive in A. B is called a *digest* of A.

As an example let $A = v_1, v_2, v_1, v_3, v_4, v_2, v_4, v_5$. The sequence $B = v_1, v_3, v_4, v_5$ is a digest of A. (there can be multiple digests of a sequence).

Every finite sequence has a digest:

Lemma 2 *Let $A = a_1, a_2, \ldots, a_n$ be a finite sequence of values. For any such sequence there exists a sequence $B = b_1, \ldots, b_m$ of values such that:*

- $b_1 = a_1 \ \wedge \ b_m = a_n$,
- $(b_i = b_j) \Rightarrow (i = j)$,
- $\forall j : \ 1 \leq j < m : \exists i : \ 1 \leq i < n : b_j = a_i \ \wedge \ b_{j+1} = a_{i+1}$.

Proof The proof is a trivial induction on n. If $n = 1$, we have $B = a_1$. If $n > 1$, let $B = b_1, \ldots, b_m$ be a digest of $A = a_1, a_2, \ldots, a_n$. A digest of $a_1, a_2, \ldots, a_n, a_{n+1}$ can be constructed as follows:
- If $\forall j \in \{1, \ldots, m\} : b_j \neq a_{n+1}$, then $B = b_1, \ldots, b_m, a_{n+1}$ is a digest of $a_1, a_2, \ldots, a_n$.
- If $\exists j \in \{1, \ldots, m\} : b_j = a_{n+1}$, there is a single j such that $b_j = a_{n+1}$ (this is because any value appears at most once in $B = b_1, \ldots, b_m$). It is easy to check that $B = b_1, \ldots, b_j$ is a digest of $a_1, \ldots, a_n, a_{n+1}$.

$\square_{Lemma\ 2}$

Consider now an implementation of a bounded atomic 1W1R register R from a collection of base *bounded* 1W1R regular registers. Clearly, any execution of a write operation w that changes the value of the implemented register must consist of a sequence of writes on base registers. Such a sequence of writes triggers a sequence of state changes of the base registers, from the state before w to the state after w.

Assuming that R is initialized to 0, let us consider an execution E where the writer indefinitely alternates $R.write(1)$ and $R.write(0)$. Let w_i, $i \geq 1$, denotes the i-th $R.write(v)$ operation. This means that $v = 1$ when i is odd and $v = 0$ when i is even. Each prefix of E, denoted by E', unambiguously determines the resulting *state* of each base object X, i.e., the value that the reader would obtain if it read X right after E', assuming no concurrent writes. Indeed, since the resulting execution is sequential, there exists exactly one reading function and we can reason about the state of each object at any point in the execution.

Each write operation $w_{2i+1} = R.write(1)$, $i = 0, 1, \ldots$, contains a sequence of writes on the base registers. Let $\omega_1, \ldots, \omega_x$ be the sequence of base writes generated by w_{2i+1}. Let A_i be the corresponding

sequence of base-registers states defined as follows: its first element a_0 is the state of the base registers before ω_1, its second element a_2 is the state of the base registers just after ω_1 and before ω_2, etc.; its last element a_x is the state of the base registers after ω_x.

Let B_i be a digest derived from A_i (by Lemma 2 such a digest sequence exists).

Lemma 3 *There exists a digest $B = b_0, \ldots, b_y$ ($y \geq 1$) that appears infinitely often in $B_1, B_2, \ldots$.*

Proof First we observe that every digest B_i ($i = 1, 2, \ldots$) must consists of at least two elements. Indeed if B_i is a singleton b_0, then the read operation on R applied just before w_i and the read operation on R applied just after w_i observe the same state of base registers b_0. Therefore, the reader cannot decide when exactly the read operation was applied and must return the same value—a contradiction with the assumption that w_i changes the value of R.

Since the base registers are bounded, there are finitely many different states of the base registers that can be written by the writer. Since a digest is a sequence of states of the registers written by the writer in which every state appears at most once, we conclude that there can only be finitely many digests. Thus, in the infinite sequence of digests, $B_1, B_2, \ldots$, some digest B (of two or more elements) must appear infinitely often. $\Box_{Lemma\ 3}$

Note that there is no constraint on the number of *internal* states of the writer. Since there may be no bound on the number of steps taken within a write operation, all the sequences A_i can be different, and the writer may never perform the same sequence of base-register operations twice. But the evolution of the base-register states in the course of A_i can be reduced to its digest B_i.

5.2.2 The Impossibility Result and the Lower Bound

Theorem 13 *It is not possible to build a 1W1R atomic bit from a finite number of regular registers that can take a finite number of values and are written only by the writer.*

Proof By contradiction, assume that it is possible to build a 1W1R atomic bit R from a finite set S of regular registers, each with a finite value domain, in which the reader does not update base registers.

An operation $r = R.read()$ performed by the reader is implemented as a sequence of read operations on base registers. Without loss of generality, assume that r reads *all* base registers. Consider again the execution E in which the writer performs write operations $w_1, w_2, \ldots$, alternating $R.write(1)$ and $R.write(0)$.

Since the reader does not update base registers, we can insert the complete execution of r between every two steps in E without affecting the steps of the writer. Since the base registers are regular, the value read in a base register X by the reader performing r after a prefix of E is unambiguously defined by the latest value written to X before the beginning of r. Let $\lambda(r)$ denote the state of all base registers observed by r.

By Lemma 3, there exists a digest $B = b_0, \ldots, b_y$ ($y \geq 1$) that appears infinitely often in $B_1, B_2, \ldots$, where B_i is a digest of w_{2i+1}. Since each state in $\{b_0, \ldots, b_y\}$ appears in E infinitely often, we can construct an execution E' by inserting in E a sequence of read operations $r_0, \ldots, r_y$ such that for each $j = 0, \ldots, y$, $\lambda(r_j) = b_{y-j}$. In other words, in E', the reader observes the states of base registers evolving downwards from b_y to b_0.

By induction, we show that in E', each r_j ($j = 0, \ldots, y$) must return 1. Initially, since $\lambda(r_0) = b_y$ and b_y is the state of the base registers right after some $R.write(1)$ is complete, r_0 must return 1. Inductively, suppose that r_j (for some j, $0 \leq j \leq y - 1$) returns 1 in E'.

Consider read operations r_j and r_{j+1} ($j = 0, \ldots, y - 1$). Recall that $\lambda(r_j) = b_{y-j}$ and $\lambda(r_{j+1}) = b_{y-j-1}$. Since digest B appears in $B_1, B_2, \ldots$ infinitely often, E' contains infinitely many base-register

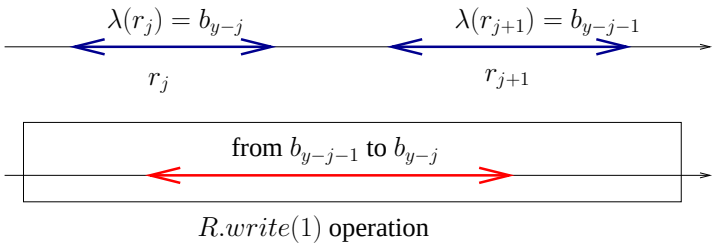

Figure 5.1: Two read operations r_j and $r_j + 1$ concurrent with $R.write(1)$

writes by which the writer changes the state of base registers from b_{y-j-1} to b_{y-j}. Let X be the base register changed by these writes.

Since X is regular, we can construct an execution E'' which is indistinguishable to the reader from E', where r_j are concurrent with a base-register write performed within $R.write(1)$ in which the writer changes the state of the base registers from b_{y-j-1} to $b_y - j$ (Figure 5.1).

By the induction hypothesis, r_j returns 1 in E' and, thus, in E''. Since the implemented register R is atomic and r_j returns the concurrently written value 1 in E'', r_{j+1} must also return 1 in E''. But the reader cannot distinguish E' and E'' and, thus, r_{j+1} returns 1 also in E'.

Inductively, r_y must return 1 in E'. But $\lambda(r_y) = b_0$, where b_0 is the state of base registers right after some $R.write(0)$ is complete. Thus, r_y must return 0—a contradiction. $\square_{Theorem\ 13}$

Therefore, to implement a 1W1R atomic register from bounded regular registers, we must establish two-way communication between the writer and the reader. Intuitively, the reader must inform the writer that it is aware of the latest written value, which requires at least one base bit that can be written by the reader and read by the writer. But the writer must be able to react to the information read from this bit. In other words:

Theorem 14 *In any implementation a 1W1R atomic bit from regular bits, the writer must be able to write to at least 2 regular bits.*

Proof Suppose, by contradiction, that there exists an implementation of a 1W1R atomic bit R in which the writer can write to exactly one base bit X.

Note that every write operation on R that changes the value of X and does not overlap with any read operation must change the state of X. Without loss of generality assume that the first write operation $w_1 = R.write(1)$ performed by the writer in the absence of the reader changes the value of X from 0 to 1 (the corresponding digest is $0, 1$).

Consider an extension of this execution in which the reader performs $r_1 = R.read()$ right after the end of w_1. Clearly, r_1 must return 1. Now add $w_2 = R.write(0)$ right after the end of r_1. Since the state of X at the beginning of w_2 is 1, the only digest generated by w_2 is $1, 0$.

Now add $r_2 = R.read()$ right after the end of w_2, and let E be the resulting execution. Now r_2 must return 0 in E. But since X is regular, E is indistinguishable to the reader from an execution in which r_1 and r_2 take place within the interval of w_1 and thus both must return 1—a contradiction. $\square_{Theorem\ 14}$

As we have seen in the previous chapter, there is a trivial bounded algorithm that constructs a regular bit from a safe bit. This algorithm only requires one additional local variable at the writer. The combination of this algorithm with Theorem 14 implies:

Corollary 1 *The construction of a 1W1R atomic bit from safe bits requires at least 3 1W1R safe bits, two written by the writer and one written by the reader.*

As the construction presented in the next section uses exactly 3 1W1R regular bits to build an atomic bit, it is optimal in the number of base safe bits.

5.3 From three safe bits to an atomic bit

Now we present an optimal construction of a high level 1W1R atomic bit R from three base 1W1R safe bits. The high level bit R is assumed to be initialized to 0. It is also assumed that each $R.write(v)$ operation invoked by the writer changes the value of R. This is done without loss of generality, as the writer of R can locally keep a copy v' of the last written value, and apply the next $R.write(v)$ operation only when it modifies the current value of R.

The construction of R is presented in an incremental way.

5.3.1 Base architecture of the construction

The three base registers are initialized to 0. Then, as we will see, the read and write algorithms defining the construction, are such that, any write applied to a base register X changes its value. So, its successive values are 0, then 1, then 0, etc. Consequently, to simplify the presentation, a write operation on a base register X, is denoted "change X". As any two consecutive write operations on a base bit X write different values, it follows that X behaves as regular register.

The 3 base safe bits used in the construction of the high level atomic register R are the following:

- REG: the safe bit that, intuitively, contains the value of the atomic bit that is constructed. It is written by the writer and read by the reader.
- WR: the safe bit written by the writer to pass control information to the reader.
- RR: the safe bit written by the reader to pass control information to the writer.

5.3.2 Handshaking mechanism and the write operation

As we saw in the previous section, the reader should inform the writer when it read a new value v in the implemented register. Otherwise, the uninformed writer may subsequently repeat the same digest of state transitions executing $R.write(v)$ so that the reader would be subject to new/old inversion. Therefore, whenever the writer is informed that a previously written value is read by the reader, it should change the execution so that critical digests are not repeated.

The basic idea of the construction is to use the control bits WR and RR to implement the *handshaking* mechanism. Intuitively, the writer informs the reader about a new value by changing the value of WR so that $WR \neq RR$. Respectively, the reader informs the writer that the new value is read by changing the value of RR so that $WR = RR$. With these conventions, we obtain the following handshaking protocol between the writer and the reader:

- After the writer has changed the value of the base register REG, if it observes $WR = RR$, it changes the value of WR.

  As we can see, setting the predicate $WR = RR$ equal to false is the way used by the writer to signal that a new value has been written in REG. The resulting is described in Figure 5.2.

operation $R.write(v)$: %Change the value of R %
i change REG;
ii **if** $WR = RR$ **then** change WR **end_ if**; % Strive to establish $WR \neq RR$ %
return ()

Figure 5.2: The $R.write(v)$ operation

- Before reading REG, the reader changes the value of RR, if it observes that $WR \neq RR$. This signaling is used by the writer to update WR when it discovers that the previous value has been read.

As we are going to see in the rest of this chapter, the exchange of signals through WR and RR is also used by the reader to check if the value it has found in REG can be returned.

5.3.3 An incremental construction of the read operation

The reader's algorithm is much more involved than the writer's algorithm. To make it easier to understand, this section presents the reader's code in an incremental way, from simpler versions to more involved ones. In each stage of the construction, we exhibit scenarios in which a simpler version fails, which motivates a change of the protocol.

The construction: step 1 We start with the simplest construction in which the reader establishes $RR = WR$ and returns the value found in REG.

3 **if** $WR \neq RR$ **then** change RR **end_ if**; % Strive to establish $WR = RR$ %
4 $val \leftarrow REG$;
5 *return* (val)

We can immediately see that this version does not really use the control information: the value returned by the read operation does not depend on the states of RR and WR. Consequently, this version is subject to new/old inversions: suppose that while the writer changes the value of REG from 0 to 1 (line ii in Figure 5.2), the reader performs two read operations. The first read returns 1 (the "new" value of R) and the second read returns 0 (the "old" value), i.e., we obtain a new/old inversion.

The construction: step 2 An obvious way to prevent the new/old inversion described in the previous step is to allow the reader to return the current value of REG only if it observes that the writer has updated WR to make $WR \neq RR$ since the previous read operation.

1 **if** $WR = RR$ **then** *return* (val) **end_ if**;
3′ change RR; % Strive to establish $WR = RR$ %
4 $val \leftarrow REG$;
5 *return* (val)

Here we assume that the local variable val initially contains the initial value of R (0). Checking whether $WR \neq RR$ before changing RR in line 3′ looks unnecessary, since the reader does not touch the shared memory between reading WR in line 1 and in line 3, so we dropped it for the moment.

Unfortunately, we still have a problem with this construction. When a read is executed concurrently with a write, it may happen that the read returns a concurrently written value but a subsequent read finds $RR \neq WR$ and returns an old value found in REG.

Indeed, consider the following scenario (Figure 5.3):

1. $w_1 = R.write(1)$ completes.
2. r_1 reads WR, finds $WR \neq RR$ and changes RR.
3. $w_2 = R.write(0)$ begins, changes REG to 0, reads RR, finds $WR = RR$, changes WR, restoring the predicate $WR \neq RR$, and completes.
4. $w_3 = R.write(1)$ begins and starts changing REG from 0 to 1.
5. r_1 concurrently reads REG and returns the new value 1
6. $r_2 = R.read()$ begins, finds $RR \neq WR$, reads REG and returns the old value 0 (which is perfectly possible since the write operation on REG performed by w_3 is not yet finished).

In other words, we obtain ta new-old inversion for read operations r_1 and r_2.

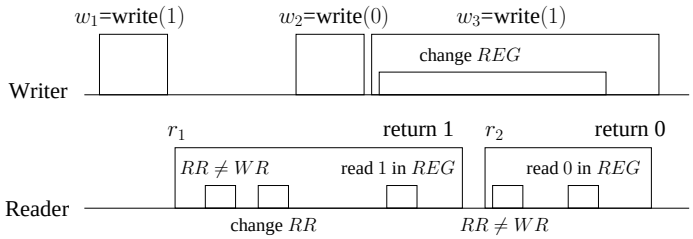

Figure 5.3: Counter example to step 2 of the construction: new-old inversion for r_1 and r_2

The construction: step 3 The problem with the scenario above is that a read operation is too quick to return the new value of REG without noticing that the writer has meanwhile changed WR. A subsequent read operation may observe $RR = WR$ and thus return the value read in REG (line 4) which may, in case of a slow concurrent write, still be the old value.

One solution to circumvent this is to evaluate REG before changing RR. If the predicate $RR = WR$ does not hold after RR was changed (line 3′) and REG was read again (line 4), then the reader returns the older (conservative) value of REG.

```
1  if WR = RR then return (val) end_ if;
2  aux ← REG; % Conservative value %
3′ change RR; % Strive to establish WR = RR %
4  val ← REG;
5  if WR = RR then return (val) end_ if
7  return (aux)
```

Unfortunately, there is still a problem here. The variable val evaluated in line 4 may be too conservative to be returned by a subsequent read operation that finds $RR = WR$ in line 1.

Again, suppose that $w_1 = R.write(1)$ is followed a concurrent execution of $r_1 = R.read()$ and $w_2 = R.write(0)$ as follows (Figure 5.4):

1. $w_1 = R.write(1)$ completes.
2. $w_2 = R.write(0)$ begins and starts changing REG from 1 to 0.
3. r_1 finds $WR \neq RR$, reads 0 from REG and stores it in aux (line 2), changes RR, reads 1 from REG and stores it in val (the write operation on REG performed by w_2 is still going on).
4. w_2 completes its write on REG, finds $RR = WR$ and starts changing WR.
5. r_1 finds $WR \neq RR$ (line 5), concludes that there is a concurrent write operation and returns the "conservative" value 0 (read in line 2).
6. $r_2 = R.read()$ begins, finds $RR = WR$ (the write operation on WR performed by w_2 is still going on), and returns 1 previously evaluated in line 4 of r_1.

That is, r_1 returned the new (concurrently written) value 0 while r_2 returned the old value 1.

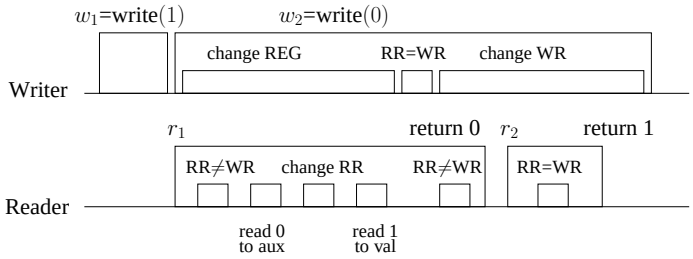

Figure 5.4: Counter example to step 3 of the construction: new-old inversion for r_1 and r_2

The construction: step 4 Intuitively, the problem with the algorithm above is that r_1 did not realize that the "conservative" value evaluated in line 2 is in fact the concurrently written value, while the "new" value evaluated in line 4 is outdated. To fix this, before the reader decides to be conservative and return in line 7, we add one more read of REG to update the local variable val. This way, a subsequent read would also return the new value.

operation $R.read()$:
1 **if** $WR = RR$ **then** *return* (val) **end_ if**;
2 $aux \leftarrow REG$;
3′ change RR;
4 $val \leftarrow REG$;
5 **if** $WR = RR$ **then** *return* (val) **end_ if**;
6 $val \leftarrow REG$;
7 *return* (aux)

But still there is a problem here. Changing RR in line 3′ without previously checking if $WR = RR$ may create an illusion to the reader that it has established the predicate $RR = WR$, while in fact the predicate was invalidated by a concurrent write.

Consider the following execution (Figure 5.5):

1. $w_1 = R.write(1)$ begins, changes REG to 1, finds $RR = WR$, and starts changing WR to 1.
2. $r_1 = R.read()$ begins, observes $RR \neq WR$ in line 1, reads 1 from REG in line 2, changes RR to 1, reads 1 from REG in line 4, and returns 1 in line 5.
3. $r_2 = R.read()$ begins and finds $WR = RR$ (the write on WR performed by w_1 is still going on).
4. w_1 finishes changing WR to 1 and completes.
5. $w_2 = R.write(0)$ begins and starts changing REG to 0.
6. r_2 reads 0 in REG, (unconditionally) changes RR back to 0, finds $WR \neq RR$, reads 1 in REG in line 6 (the write on REG performed by w_2 is still going on), and returns the conservative value 0 in line 7.
7. $r_3 = R.read()$ begins, observes $RR \neq WR$ in line 1, reads 1 REG, changes RR to 1, reads 1 from REG again (recall that the write on REG performed by w_2 is still going on) and returns the old value 1 in line 5.

Again, we have a new-old inversion: r_2 returns the value concurrently written by w_2 while r_3 returns the old value.

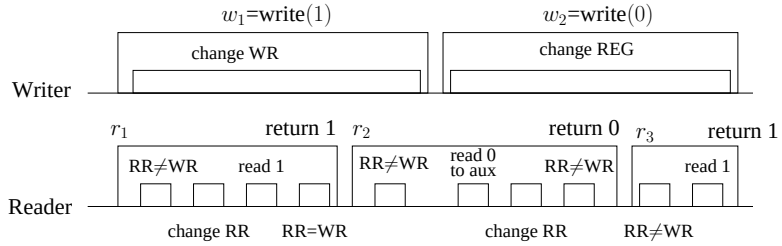

Figure 5.5: Counter example to step 4 of the construction: new-old inversion for r_2 and r_3

The construction: last step The complete read algorithm is presented in Figure 5.6. As we saw in this chapter, safe base registers allow for a multitude of possible execution scenarios, so an intuitively correct implementation could be flawed because of an overlooked case. To be convinced that our construction is indeed correct, we provide a rigorous proof below.

5.3.4 Proof of the construction

Theorem 15 *Let H be an execution history of the 1W1R register R constructed by the algorithm in Figures 5.2 and 5.6. Then H is linearizable.*

```
operation R.read():
 1  if WR = RR then return (val) end_ if;
 2  aux ← REG;
 3  if WR ≠ RR then change RR end_ if;
 4  val ← REG;
 5  if WR = RR then return (val) end_ if;
 6  val ← REG;
 7  return (aux)
```

Figure 5.6: The $R.read()$ operation

Proof Let H be an execution history. By Theorem 4,to show that H is linearizable (atomic), it is sufficient to show that there exists a reading function π satisfying the assertions $A0$, $A1$ and $A2$.

In order to distinguish the operations $R.read()$ and $R.write(v)$, denoted by r and w, from the read and write operations on the base registers (e.g., "change RR", "$aux \leftarrow REG$", etc.), the latter ones are called *actions*. The history defined from the action invocation and response events is denoted L ($<_L$ denotes the total order on its events and $\rightarrow_L$ the corresponding relation induced on its operations; without loss of generality, $<_L$ is assumed to contain all the invocation and response events defining H).

Moreover, r being a read operation and loc the local variable (aux or val) whose value is returned by r (in line 1, 5 or 7), ρ_r denotes the last read action "$loc \leftarrow REG$" executed before r returns. More explicitly, we have:

- If r returns in line 7, ρ_r is the read action "$aux \leftarrow REG$" executed in line 2 of r,
- If r returns in line 5, ρ_r is is the read action "$val \leftarrow REG$" executed in line 4 of r, and finally
- If r returns in line 1, ρ_r is is the read action "$val \leftarrow REG$" executed in line 4 or 6 of some previous read operation.

For each read action ρ_r we can determine the corresponding write action, denoted $\phi(\rho_r)$ and defined as the latest write action that writes the value returned by r and *does not* succeed ρ_r in L.

Finally, given a read operation r and its associated read action ρ_r, we define $\pi(r)$ to be the write operation that includes the write action $\phi(\rho_r)$. This means that the value returned by the read operation r has been written in the base register REG by the "change REG" action of the write operation $\pi(r)$. For notational convenience we write $a \in A$ when a is an action of the operation A.

Proof of $A0$.

Let r be a complete read operation in H. By the definition of π, the invocation of the write action $\phi(\rho_r)$ occurs before the response of ρ_r and, thus, the response of r in L, i.e., $inv[\pi(\rho_r)] <_L resp[r]$. Thus, $inv[\pi(r)] <_L inv[\pi(\rho_r)] <_L resp[r]$ and $\neg(resp[r] <_L inv[\pi(r)])$.

By contradiction, suppose that $A0$ is violated, i.e., $r \rightarrow_H \pi(r)$. Thus, at the action event level, $resp[r] <_L inv[\pi(\rho_r)]$)—a contradiction. Consequently, π satisfies $A0$.

Proof of $A1$.
Since there is only one writer, all writes are totally ordered and $w \rightarrow_H \pi(r)$ is equivalent to $\neg(\pi(r) \rightarrow_H w)$.

By contradiction, suppose that there is a write operation $w \neq \pi(r)$ such that $\pi(r) \rightarrow_H w \rightarrow_H r$. If there are several such write operations, let w be the last one before r, i.e., $\nexists\, w'$: $w \rightarrow_H w' \rightarrow_H r$.

We first claim that, in such a context, ρ_r cannot be a read action of the read operation r (i.e., $\rho_r \notin r$).
Proof of the claim. (see Figure 5.7). Recall that $\phi(\rho_r) \in \pi(r)$ (by definition). Let ω be the "change REG" action of the operation w ($\omega \in w$). Combined with the case assumption $\pi(r) \rightarrow_H w$, we obtain $\phi(\rho_r) \rightarrow_L \omega$. By the definition of $\phi(\rho_r)$, we have $\neg(\phi(\rho_r) \rightarrow_L \rho_r)$ and, thus, $\neg(\omega \rightarrow_L \rho_r)$. Therefore, $inv[\rho_r] <_L resp[\omega]$. As $\omega \in w$ and $w \rightarrow_H r$, we have $inv[\rho_r] <_L resp[w] <_L inv[r]$. As ρ_r started before r, and both are executed by the same process, we have $\rho_r \notin r$. *End of the proof of the claim.*

Since $\rho_r \notin r$, by the algorithm in Figure 5.6, the read operation r returns a value in line 1, which means that it has previously seen $WR = RR$. On the other hand, after the writer has executed ω within $\pi(r)$, it read RR in order to set WR different from RR if they were seen equal. As $w \rightarrow_H r$ and $\nexists\, w'$: $w \rightarrow_H w' \rightarrow_H r$ (assumption), it follows that RR has been modified before the read operation r starts. Moreover, RR can only be modified by a read operation in line 3. Let r' be that read operation; as there is a single process executing $R.read()$, we have $r' \rightarrow_H r$.
Now we claim that $\rho_r \notin r'$.
Proof of the claim: Let r'' be the read operation that contains ρ_r. We show $r'' \neq r'$. We observe that (Figure 5.7):
- If r'' updates RR, it does it in line 3, i.e., before executing ρ_r (in line 4 or 6),
- $inv[\rho_r] <_L resp[\omega]$ (this has been shown above; it is indicated by a dotted arrow in Figure 5.7),
- w reads RR after having executed ω (code of the write operation).

It follows from these observations that, if r'' writes into RR, it does it before w reads RR. Hence, r'' cannot change the value of RR (to establish $RR = WR$) after w has read RR or while it is reading it (to establish $RR \neq WR$). Therefore, $r'' \neq r'$ and, thus, $\rho_r \notin r'$. *End of the proof of the claim.*

As the reader modifies RR within r', it also executes line 4 of r' ($val \leftarrow REG$) before executing r (this follows from the code of the read operation). But, as $\rho_r \notin r'$, this read of REG action within r' contradicts the definition of ρ_r (according which ρ_r is the last action "$val \leftarrow REG$" executed before r starts), which completes the proof of the assertion $A1$.

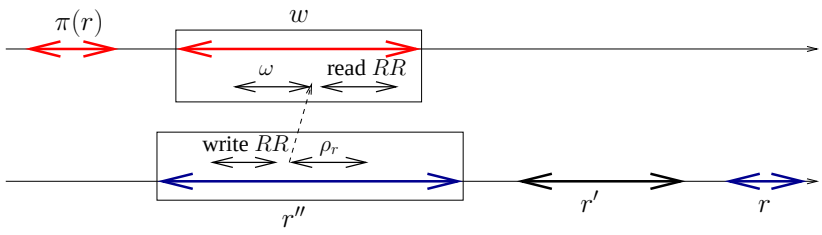

Figure 5.7: ρ_r belongs neither to r nor to r'

Proof of $A2$.
The proof is again by contradiction. Suppose that there exist $r1$ and $r2$, two complete read operations in H, such that $r1 \rightarrow_H r2$ and $\pi(r2) \rightarrow_H \pi(r1)$. Without loss of generality, let us assume that, for a given $r2$, $r1$ is the first such read operation. This means that if $r1$ returns in line 5 or 7, ρ_{r1} is a read action belonging to $r1$, and if $r1$ returns at line 1, then ρ_{r1} is a read action in the immediately preceding read operation. Moreover, as $\pi(r2) \neq \pi(r1)$, we have $\rho_{r1} \neq \rho_{r2}$. So, we have either $\rho_{r1} \rightarrow_L \rho_{r2}$ or $\rho_{r2} \rightarrow_L \rho_{r1}$.

- $\rho_{r2} \rightarrow_L \rho_{r1}$.
  As ρ_{r1} precedes or belongs to $r1$, and $r1 \rightarrow_H r2$, we have $resp[\rho_{r1}] <_L inv[r2]$. Combining this

with the case assumption we obtain $\rho_{r2} \rightarrow_L \rho_{r1} \rightarrow_L r2$, which contradicts the fact that ρ_{r2} is the last "$loc \leftarrow REG$" action executed before $r2$ started, where loc is val or aux. So, the case $\rho_{r2} \rightarrow_L \rho_{r1}$ is not possible.

- $\rho_{r1} \rightarrow_L \rho_{r2}$.
  By definition $\pi(\rho_{r1}) \in \pi(r1)$ and $\pi(\rho_{r2}) \in \pi(r2)$. As $\pi(r2) \rightarrow_H \pi(r1)$, we have $\pi(\rho_{r2}) \rightarrow_L \pi(\rho_{r1})$.

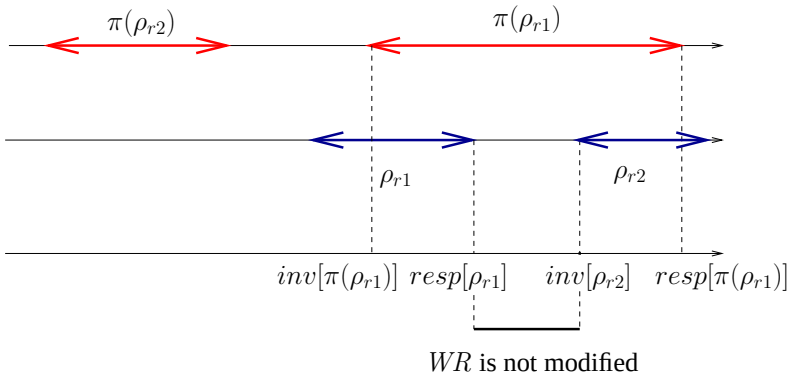

Figure 5.8: A new/old inversion on the regular register REG

Thus we obtain $\pi(\rho_{r2}) \rightarrow_L \pi(\rho_{r1})$ and $\rho_{r1} \rightarrow_L \rho_{r2}$ (Figure 5.8) which implies a new/old inversion for the base regular register REG. Therefore, both ρ_{r1} and ρ_{r2} have to overlap $\pi(\rho_{r1})$ in order to have a new/old inversion. Figure 5.8): $inv[\pi(\rho_{r1})] <_L resp[\rho_1]$ and $inv[\rho_2] <_L resp[\pi(\rho_{r1})]$. As $\pi(\rho_{r1})$ is a base action that updates REG, and as it is the same process that updates REG and WR, this means that the value of the base register WR does not change while it is updating REG, from which we conclude that:

Property P: WR does not change between $resp[\rho_1]$ and $inv[\rho_2]$

We consider three cases according to the line at which $r1$ returns.

- $r1$ returns in line 7.
  Then, ρ_{r1} is "$aux \leftarrow REG$" in line 2 of $r1$. We have the following:
  - Since $\rho_{r1} \rightarrow_L \rho_{r2}$ and $r1$ returns in line 7, ρ_{r2} can only be the read in line 6 of $r1$ or a later read action.
  - Since $r1$ executes all the actions of the read operation, in line 3 it makes RR equal to WR if they were not. On another side, as it returns in line 7, $r1$ necessarily sees RR different from WR in line 5 (otherwise, it would have returned in line 5).
  It follows from these two observations that WR has been modified between line 2 (execution of ρ_{r1}) and line 6 of $r1$ (that is or precedes ρ_{r2}). This contradicts property P above.
- $r1$ returns in line 5.
  Then, ρ_{r1} is "$val \leftarrow REG$" in line 4 of $r1$, and $r1$ sees $RR = WR$ in line 5. Since $\rho_{r1} \rightarrow_L \rho_{r2}$, $r2$ does not return in line 1 (for $r2$ to return in line 1, we need to have $RR = WR$ at line 1 of $r2$, which means that we would then have $\rho_{r1} = \rho_{r2}$). Thus, $r2$ sees $RR \neq WR$ when it executes line 1, and ρ_{r2} is in line 2 or line 4 of $r2$. It follows that WR has been modified between ρ_{r1} and ρ_{r2}—a contradiction with property P.

– $r1$ returns in line 1.
In that case, ρ_{r1} is line 4 or line 6 of the read operation that precedes $r1$. The reasoning is the same as in the previous case. Since $\rho_{r1} \rightarrow_L \rho_{r2}$, $r2$ does not return in line 1, from which we conclude that it sees $RR \neq WR$ when it executed line 1. It follows that WR has been modified between ρ_{r1} and ρ_{r2}, which contradicts property P and concludes the proof.

$\square_{Theorem\ 15}$

5.3.5 Cost of the algorithms

The cost of the $R.read()$ and $R.write(v)$ operations is measured by the the maximal and minimal numbers of accesses to the base registers. Let us remind that the writer (resp., reader) does not read WR (resp., RR) as it keeps a local copy of that register.

- $R.write(v)$: maximal cost: 3; minimal cost: 2.
- $R.read()$: maximal cost: 7; minimal cost: 1.

The minimal cost is realized when the same type of operation (i.e., read or write) is repeatedly executed while the operation of the other type is not invoked.

Let us remark that we have assumed that if $R.write(v)$ and $R.write(v')$ are two consecutive write operations, we have $v \neq v'$. This means that if the upper layer issues two consecutive write operations with $v = v'$, the cost of the second one is 0, as it is skipped and consequently there is no accesses to base registers.

5.4 Bibliographic notes

Tromp 1989

Lamport 86 (1W2R, but very inefficient)

Bibliography

[1] Gene Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings* (30): 483485, 1967.

[2] 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.

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

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

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

[6] 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.

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

[8] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] Afek Y., Brown G. and Merritt M., Lazy Caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182-205, 1993.

[24] Attiya H. and Welch J.L., Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91-122, 1994.

[25] Bernstein A., Haqzilacos V. and Goodman N., Concurrency Control and Recovery in Database Systems. *Addison Wesley*, 1986.

[26] Fekete A., Lynch N., Merritt M. and Weihl W., Atomic Trasanctions. *Morgan Kaufmann Publishing*, 1994.

[27] Gray J. and Reuter A., Transactions Procesing: Concepts and Techniques, *Morgan Kaufmann Publishing*, 1070 pages, 1992.

[28] Lamport L., How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C28(9):690-691, 1979.

[29] Papadimitriou C., The Theory of Database Concurrency Control. *Computer Science Press*, 1988.

[30] Raynal M., Sequential Consistency as Lazy Linearizabilty. *Proc. 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*, pp. 151-152, Winnipeg, 2002.

[31] Raynal M., Token-Based Sequential Consistency. *Int'l Journal of Computer Systems Science and Engineering*, 17(6):359-366, 2002.

[32] Alpern B. and Schneider F.B., Defining liveness. *Information Processing Letters*, 21(4):181-185, 18985.

[33] Attiya H., Guerraoui R. and Kouznetsov P., Computing with reads and writes in the absence of step contention. *Proc. 19rd Int'l Symposium on Distributed Computing (DISC'05)*, Springer-Verlag #3724, pp. 122-136, 2005.

[34] Fich F., Herlihy H. and Shavit N., On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843-862. 1998.

[35] Fich F., Luchangco V., Moir M. and Shavit N., Obstruction-free algorithms can be practically wait-free. *Proc. 23rd Int'l Symposium on Distributed Computing (DISC'05)*, Springer-Verlag #3724, pp. 78-92, 2005.

[36] Guerraoui R., Kapalka M. and Kouznetsov P., The weakest failure detector to boost obstruction-freedom. *Proc. 20rd Int'l Symposium on Distributed Computing (DISC'06)*, Springer-Verlag #4167, pp. 399-412, 2006.

[37] Herlihy M., Luchangco V., Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23rd IEEE Int'l Conference on Distributed Computing Systems I(CDCS '03)*, IEEE Computer Press, pp. 522-529, 2003.

[38] Herlihy M. and Shavit N., The Art of Multiprocessor Programming. *Morgan Kaufmann Pubs*, Elsevier, 508 pages, 2008.

[39] Lamport. L., Proving the correctness of multiprocess programs. *IEEE Transaction on Software Engineering*, SE-3(2):125-143, 1977.

[40] Jayanti P., Burns J. and Peterson G., Almost optimal single reader single writer atomic register. *Journal of Parallel and Distributed Computing*, 60:150-168, 2000.

[41] Li M., Tromp J. and Vityani P., How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723-746, 1996.

[42] Singh A.K., Anderson J.H. and Gouda M., The elusive atomic register. *Journal of the ACM*, 41(2):331-334, 1994.

[43] Vidyasankar K., Converting Lamport's Regular Register to Atomic Register. *Information Processing Letters*, 28(6):287-290, 1988.

[44] Vidyasankar K., An elegant 1-writer multireader multivalued atomic register. *Information Processing Letters*, 30(5):221-223, 1989.

[45] Vidyasankar K., A very simple cosntruction of 1-writer multireader multivalued atomic variable. *Information Processing Letters*, 37:323-326, 1991.

[46] Vityani P., Simple wait-free multireader registers. *Proc. 16th Int'l Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 118-132, 2002.

[47] Vityani P. and Awerbuch B., Atomic shared register access by asynchronous hardware. *Proc. 27th IEEE Symposium on Foundations of Computer Science (FOCS'87)*, IEEE Computer Press, 223-243, 1986.

[48] Bloom B., Constructing two-writer atomic regsiters. *IEEE Transactions on Computers*, 37:1506-1514, 1988.

[49] Burns J. and Peterson G., Constructing multireaders atomic values from non-atomic values. *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC'87)*, ACM Press, pp. 222-231, 1987.

[50] Chaudhuri S., Kosa M.J. and Welch J., One-write algorithms for multivalued regular and atomic registers. *Acta Informatica*, 37:161-192, 2000.

[51] Chaudhuri S. and Welch J., Bounds on the cost of multivalued register implementations. *SIAM Journal of Computing*, 2”(2):33(-354, 1994.

[52] Haldar S. and Vidyasankar K., Constructing 1-writer multireader multivalued atomic variables from regular variables. *Journal of the ACM*, 42(1):186-203, 1995.