Robust Concurrent Computing

Rachid Guerraoui[‡] Petr Kuznetsov[⋆]

[‡] School of Computer and Communication Sciences, EPFL
[⋆] Technische Universität Berlin, Deutsche Telekom Laboratories

June 23, 2012

Contents

Chapter 1

Introduction

1.1 A broad picture: the concurrency revolution

The field of concurrent computing has gained a huge importance after major chip manufacturers have switched their 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. A single-threaded application can for instance 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.

In short, 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. In fact, the revolution is more than about concurrency alone: it is about *concurrency for every one*. In other words, concurrency is going out of the small box of specialist programmers and is conquering the masses.

The challenge underlying this revolution is to come up with abstractions that average programmers can easily use for general purpose concurrent programming. Such abstractions need to be composed with others, possibly more sophisticated, that advanced programmers could use to fully leverage their expertise and the available underlying hardware. The aim of this book is to study how to build these abstractions and we will focus on those that are considered the most difficult: *synchronization elements*.

1.2 The topic: robust synchronization objects

In concurrent computing, a problem is solved through several processes that execute a set of tasks. Except in embarrassingly parallel programs, i.e., problems that can easily and regularly be decomposed into independent parts, the tasks usually need to synchronize their activities by accessing shared elements. These elements typically serialize the threads and reduce parallelism. According to Amdahl's law, the cost of accessing these elements significantly impacts the overall performance of concurrent computations. Devising, implementing and making good usage of such synchronization elements usually lead to intricate schemes that are very fragile and sometimes error prone.

Every multicore architecture provides such elements in hardware. Usually, these elements are very low level and their good usage is not trivial. Also, the synchronization elements that are provided in hardware differ from architecture to architecture, making it hard to port concurrent programs. Sometimes, event if the elements look the same, their exact semantics are different and some subtle details can have important consequences on the performance or the correctness of the concurrent program. Clearly, coming up with a high

level and uniform library of synchronization abstractions that could be used across multicore architectures is crucial to the success of the multicore revolution. Such a library could only be implemented in software for it is simply not realistic to require multicore constructors to agree on the same high level library to offer to their programmers.

This book shows how to design such a library in a principled manner. We assume a small set of low level synchronization primitives provided in hardware and show to build algorithms that could be used to implement higher level synchronization elements, in turn to be used by different kinds of programmers, depending on their skills and the target concurrent system, and smoothly composed within the same application. We view synchronization elements, both those available in hardware and those constructed above them, as *shared objects*, 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 precise 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 two things:

- 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 where the processes access each object one after the other.

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 synchronization abstractions

As we pointed out, this manuscript describes how to build synchronization abstractions. The quest for such abstractions can be viewed as a continuation of one of the most important challenges in computing. 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. Their definitions and effective implementations have enabled programming to become a high-level activity and made it possible to reason about algorithms without specific mention of hardware primitives.

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 abstractions we will consider are *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 individual 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 is the first main property we will require from the shared objects we seek to study. It is also called *linearizability*, and it basically 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

This is the second property we will typically require from the object implementations we will study. It can be viewed as a way to enforce a radically alternative approach to locking. Indeed, 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. With modern architectures, we might be talking about one process delaying hundreds of processors, making them completely idle and useless.

Lock-free implementations of atomic objects provide an alternative to mutual exclusion-based implementations. In particular *wait-freedom*, a strong form of lock-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

In short, this book studies how to wait-free implement high level atomic objects out of more primitive base objects. The notions of high and primitive being of course relative as we will see. The notion of implementation has to be considered here in the *algorithmic* sense (there will not be any 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 than 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 to seek general algorithms.

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.

When the answer to the reducibility question is negative, and it will be for some values of $X1$ and $X2$, then it is also interesting to ask what is needed (under some minimality metric) to add to the base objects ($X1$) in order to implement the desired high level object ($X2$). 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. We will see examples of these situations.

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 [5, 14, 18, 19]. In the context of concurrent computing, one of the earliest work was reported in [10, 17]. More information on the history concurrent programming can be found in the book [4].

The notion of register (as considered in this book) and its formalization are due to Lamport [13]. A more hardware-oriented presentation was given in [16]. The notion of atomicity has been generalized to any object type by Herlihy and Wing [9] under the name linearizability. The concept of snapshot object has been introduced in [3]. A theory of wait-free atomic objects was developed in [11].

The mutual exclusion problem has been introduced by Dijkstra [6]. 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 [22]. Various synchronization algorithms are also detailed in [23].

The notion of wait-free computation originated in the work of Lamport [12], and was then explored further by Peterson [21]. It has then generalized and formalized by Herlihy [8].

The consensus problem was introduced in [20]. Its impossibility in asynchronous message-passing systems prone to process crash failures has been proved by Fischer, Lynch and Paterson in [7]. Its impossibility in shared memory systems was proved in [15]. The universality of the consensus problem and the notion of consensus number were investigated in [8].

Part I

Correctness: safety and liveness

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 get 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 $\perp$.

$Enq\,(a)$ $Enq\,(c)$ $Enq\,(b)$ $Deq\,(a)$ $Deq\,(c)$

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?

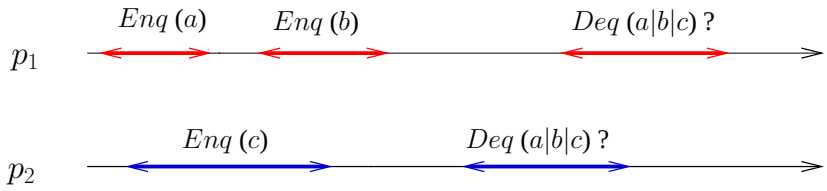

Figure 2.2: A concurrent execution on a queue

- 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, including the *initial* state; (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.

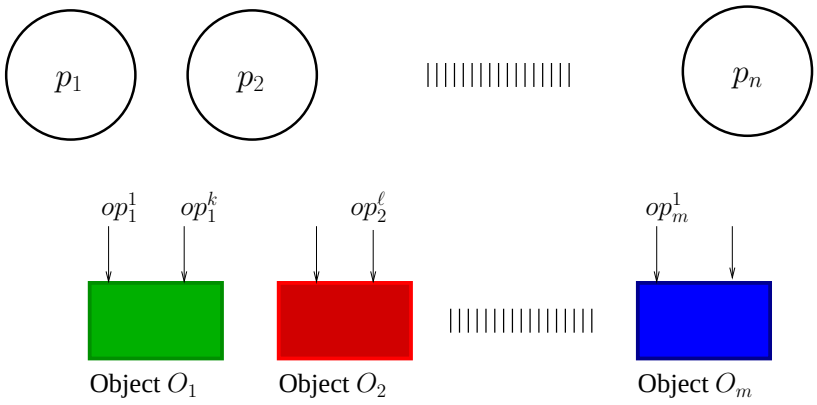

Figure 2.3: Structural view of a system

Each operation is associated with two predicates: 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 its pre-assertion associated is always satisfied, i.e., the operation is defined for every state of the object. Otherwise, the operation is *partial*. An object type is total if it has only total operations.

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 type is deterministic if it has only deterministic operations; otherwise it is non-deterministic.

Sequential specification Every obect type determines a *sequential specification*: the set of sequences of invocations immediately followed by a matching response (i.e., sequences of operations) that, starting from the initial state of the type, respect the type specification. These sequences are called *sequential* histories of the type.

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). We assume that each event in H is uniquely identified.[1] 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 sequential histories have 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.

[1] For example, we can choose the identifier of an (invocation or response) event x of a process p_i in H as (p_i, k) where k is the number of events preceding x in $H|p_i$.

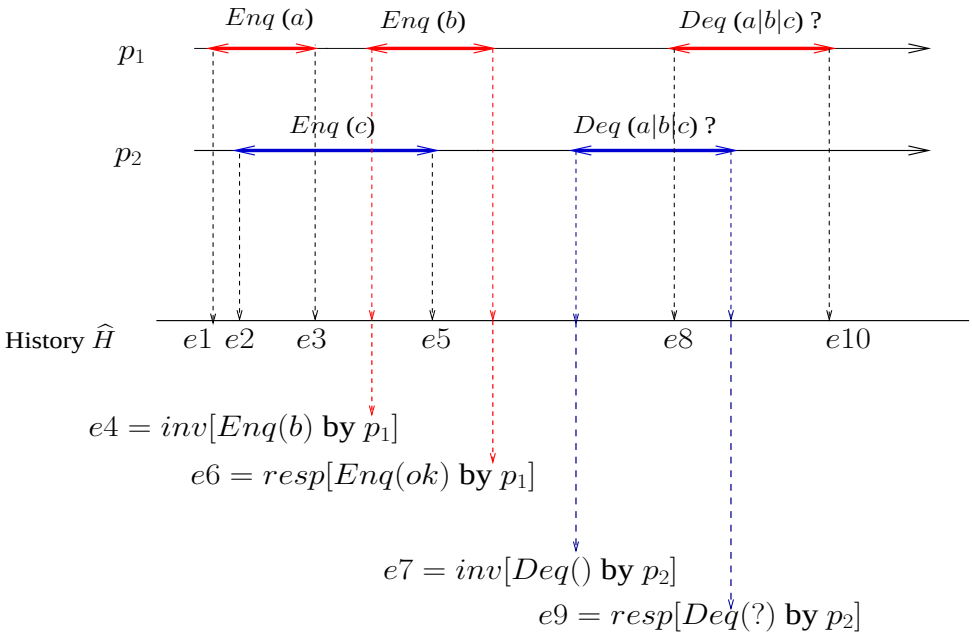

Figure 2.4: Example of a history

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 $e9 = resp[Deq(a) \text{ by } p_2]$ and $e10 = resp[Deq(b) \text{ by } p_1]$. 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 $e9 = resp[Deq(b) \text{ by } p_2]$ and $e10 = resp[Deq(a) \text{ by } p_1]$. It is easy to

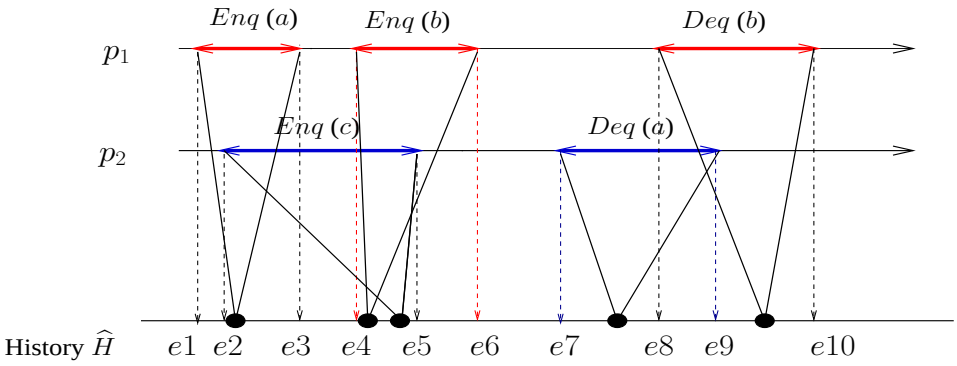

Figure 2.5: Linearization points

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 that 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. H' is obtained by adding response events to certain pending operations of H, as if these operations have indeed been completed, but also by removing invocation events from some of the pending operations of H. We require however that all complete operations of H are 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

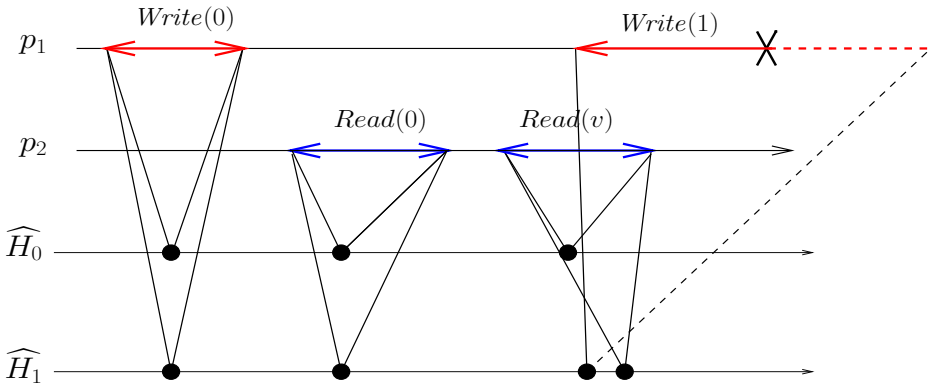

Figure 2.6: Two ways of completing a history

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 Safety and Liveness

It is convenient to reason about correctness of a concurrent object implementation by splitting the correctness properties into *safety* and *liveness*. Intuitively, safety properties ensure that nothing "bad" is ever going to happen, and liveness properties guarantee that something "good" eventually happens.

Formally, we can represent an *execution* of an concurrent object implementation as a sequence of system states $s_0 s_1 \ldots$, where each state after s_0 results after one atomic action (e.g., an atomic access to a base object) applied to the preceding state. A *property* is then a set of (finite or infinite) executions. Now a property P is a safety property if:

- P is *prefix-closed*: if $\sigma \in P$, then for every prefix σ' of σ, $\sigma' \in P$.
- P is *limit-closed*: for every infinite sequence $\sigma_0, \sigma_1, \ldots$ of executions, where each σ_i is a prefix of σ_{i+1} and each $\sigma_i \in P$, the limit $\sigma = \lim_{i \to \infty} \sigma_i$ is in P.

To ensure that a safety property P holds for a given implementation, it is thus enough to show that every *finite* execution is in P: an execution is in P if and only if each of its prefixes is in P. It can be shown that atomicity is a safety property and, thus, for the rest of this book, we mainly focus on

Now P is a liveness property if *any* finite σ has an extension in P. To show that a liveness property P holds, we should thus show that all infinite executions are in P.

Every property can be represented as an intersection of a safety property and a liveness property. In this book, we focus on atomicity (linearizability) and wait-freedom, two fundamental properties of concurrent object implementations, where atomicity is a safety property and wait-freedom is a liveness property.

2.5 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.)

2.5.1 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).

2.5.2 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 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.

2.6 Atomicity is nonblocking

Atomicity is a *nonblocking* property: an incomplete total operation is not required to wait until another operation to complete.

Theorem 2 *Let H be a finite linearizable history, and let $inv[op]$ be a pending invocation of a total operation in H. Then there exists $r = res[op]$ such that $H \cdot r$ is linearizable.*

Proof Let H be a finite linearizable history and L be linearization of H. Let $\bar{H}$ be a completion of H such that L is equivalent to H. Recall that L is legal and respects the real-time order of H.

Let $i = inv[op]$ be a pending invocation in H, where op is total.

Now two cases are possible:

- L contains op, and let r be the matching response of op in L. Then L is also linearization of $H \cdot r$.

  Indeed, consider history $\bar{H}'$, an extension of $H \cdot r$ that is equivalent to $\bar{H}$. We obtain it by reordering responses added to H to obtain $\bar{H}$ so that r is the first such response. Then $\bar{H}'$ is a linearization of $H \cdot r$.

- L contains op. Consider $L' = L \cdot i \cdot r$, where r is a legal response matching the invocation i applied at the end of L. Since op is total, such a response exists.

  L' is a linearization of $H \cdot r$. Indeed $\bar{H}'$ obtained from $\bar{H}$ by inserting r immediately after the last event of H is a completion of $H \cdot r$ that is equivalent to L'.

In both cases, we construct a linearizable history $H \cdot r$ in which $inv[op]$ is complete. $\square_{Theorem\ 1}$

2.7 Alternatives to atomicity

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

2.7.1 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).

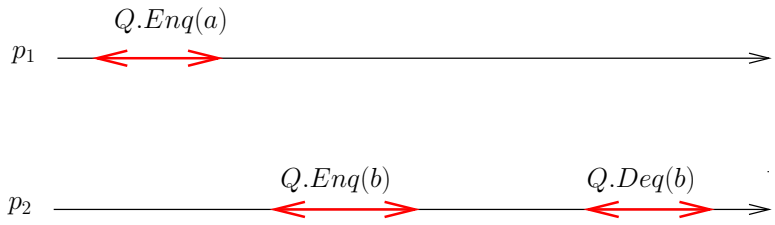

Figure 2.7: A sequentially consistent history

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.

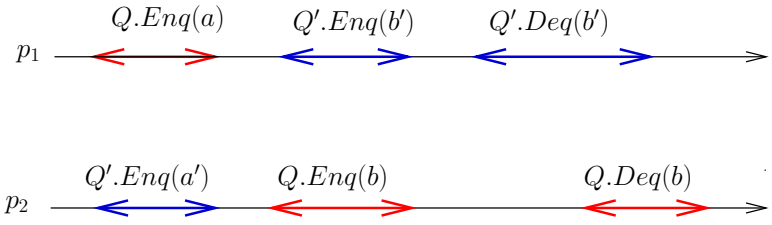

Figure 2.8: Sequential consistency is not a local property

2.7.2 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.

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.

2.8 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.

2.9 Bibliographic notes

The notion of atomic read/write objects (registers), as studied here, has been investigated and formalized by Lamport [13] and Misra [16]. The generalization of the atomicity consistency condition to objects of any sequential type has been developed by Herlihy and Wing under the name linearizability [9].

The notion of sequential consistency has been introduced by Lamport [29]. The relation between atomicity and sequential consistency was investigated in [25] and [31] 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 [24, 25, 32].

The concept of transactions is part of every textbook on database systems. Books entirely devoted to transactions are [26, 27, 28]). The theory of serializability was the main topic of the following books [27, 30].

The notions of safety and liveness were introduced by Lamport [40] and refined by Alpern and Schneider [33]. Lynch reformulated the notions for finite histories and proved that linearizability (atomicity) is a safety property [**?**].

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 2, 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.

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.

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.

Consider the example of a fetch-and-increment high-level-object implementation described in Section 3.4.2. As low-level objects, the implementations uses an infinite array $T[, \ldots, \infty]$ of TAS (test-and-set) objects and a snapshot-memory object *mi*. Here a high level history is a sequence built from invocation and reply events associated to one operation *fetch-and-increment*, while a low level history (or implementation history) is a sequence built from the primitives *read*(), *update*(), $snapshot()$ and $test - and - set()$.

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 implementing 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.

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

- 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 [8]. 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 $f()$ 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 3 *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'. $\square_{Theorem\ 3}$

3.4.2 A simple example

To give an example of a wait-free linearizable implementation, consider the algorithm in Figure 3.1. The algorithm implements a *fetch-and-increment (FAI* object using an infinite array of *TAS objects* $T[1, \ldots, \infty]$ and a *snapshot memory mi*. Below we give a brief description of these objecs.

An FAI object stores an integer value exports one operation *fetch-and-increment*() that atomically increments the value of the object and returns the previous value.

A TAS object exports one atomic operation *test-and-set*() that returns 0 or 1 and guarantees that the first invocation of *test-and-set*() on the object returns 1 and all subsequent invocations return 0. Intuitively, a TAS object allows a single process to distinguish itself from the rest of the system.

Finally, snapshot memory can be seen as an array of n registers, one for each process, such that each process p_i can atomically write a value v to its dedicated register with an operation *update*(i, v) and atomically read the content of the array using an operation *snapshot*(). [2]

The algorithm in Figure 3.1 is very simple. To increment the value of the object, a process first increments its dedicated register in the snapshot memory *my_inc*. Then it takes a snapshot of the memory and evaluates *entry* as the sum of all its elements. Then, starting from the $T[$*entry*$]$ down to 1, the process invokes operations *test-and-set*() until the first TAS object to return 1. The index of this object minus 1 is then returned by *fetch-and-increment*().

Intuitively, if a process p_i evaluates its local variable *entry* to ℓ, it means that at most ℓ processes have previously incremented their positions and, thus, at least one TAS object in the array $T[1, \ldots, \ell]$ is "reserved" for p_i (p_i is one of these ℓ processes). Every process that increments its position in *my_inc* later will obtain a strictly higher value of *entry*. Thus, eventually, every operation obtains 1 from one of the TAS objects and returns. Moreover, since a TAS object returns 1 to exactly one process, every returned value is unique. Try to see that it guarantees that every history of this implementation is linearizable.

Notice that the number of steps performed by a *fetch-and-increment*() operation is finite but in general unbounded (the implementation is not bounded wait-free). This is because an unbounded number of increments can be performed by other processes in the time lag between a process increments it position in *my_inc* and the moment it takes a snapshot of *my_inc*. It is however not difficult to modify the algorithm so that every operation performs $O(n^2)$ steps.

[2] In Chapter 6, we show that snapshot memory can be wait-free implemented using only read-write registers.

```
Shared
    T[1, ..., ∞]: n-process TAS objects
    my_inc[1, ..., ∞]: snapshot memory, initialized to 0

Local
    entry, c (initially 0), S

operation fetch-and-increment():
    c ← c + 1;
    my_inc.update(i, c);
    S ← my_inc.snapshot();
    entry ← sum(S)
    while T[entry].test-and-set() ≠ 0 do
        entry ← entry − 1;
    end_do;
    return(entry − 1)
```

Figure 3.1: Fetch-and-increment implementation: code for process p_i

3.4.3 A less simple example

Proving that a given implementation satisfies atomicity and wait-freedom can be extremely tricky sometimes. To illustrate this, consider the algorithm in Figure 3.2 that intends to implement an unbounded FIFO queue. The sequential specification of this object has been given in Section 2.1 of Chapter 2.

The algorithm is quite simple. 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 prepares 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 $\mathsf{read}()$ and $\mathsf{fetch\&add}()$. The invocation $NEXT.\mathsf{fetch\&add}(x)$ 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 "←").

The algorithm does not use locks, so no process can block other processes forever by crashing. 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).

```
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

Therefore, the implementation is wait-free: every process completes every its operation in a finite number of its own steps: the number of steps performed by *Enq*() is two, and the number of steps performed by *Deq*() is proportional to the queue size as evaluated in the first line of its pseudocode.

But is the implementation linearizable? Superficially, yes: it seems that if every dequeue operation returns a non-$\perp$ value, we can order operation based on the time when the corresponding operation on $Q[]$ (a write performed by *Enq*() or a successful swap performed by *Deq*()) takes place.

However, if a dequeue operation may return $\perp$ it is not always possible to find the right place for it in a legal linearization. For example, consider the following scenario:

1. Process p_1 performs *Enq*(x). As a result, the value of *NEXT* is 1, and $Q[0]$ stores x.
2. Process p_2 starts executing *Deq*() and reads 1 in $NEXT$.
3. Process p_1 performs *Enq*(y). The value of *NEXT* is now 2, $Q[0]$ stores x, and $Q[1]$ stores y.
4. Process p_3 performs *Deq*(), reads 2 in $NEXT$, finds x in $Q[0]$ and returns x. The value of $Q[0]$ is $\perp$ now.
5. Finally, p_2 reads $\perp$ in $Q[0]$ and completes *Deq*() by returning $\perp$.

In this execution: we have the following partial order on operations: p_1.*Enq*(x) $\rightarrow$ p_1.*Enq*(y) $\rightarrow$ p_3.*Deq*(x), and p_1.*Enq*(x) $\rightarrow$ p_2.*Deq*($\perp$). Thus, there are only three possible ways to linearize p_2.*Deq*($\perp$): right after p_1.*Enq*(x), right after p_1.*Enq*(y) or right after p_3.*Deq*(). In all three possible linearizations, the queue is non-empty when p_2 invokes *Deq*(), and thus $\perp$ cannot be returned.

How to fix this problem? One solution is to sacrifice linearizability and not to consider operations returning $\perp$ in a linearization.

Another solution is to sacrifice wait-freedom and instead of returning $\perp$ in the last line of the *Deq*(), repeat the same procedure (evaluating $NEXT$ and going through the first $NEXT$ elements in $Q[]$) over and over until a non-$\perp$ value is found in $Q[]$. As long as a producer keeps adding items to the queue, every *Deq*() operation is guaranteed to eventually return.

3.4.4 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.

3.4.5 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

The notion of wait-freedom originated in the work of Lamport [12]. An associated theory was developed by Herlihy [8].

The notion of solo-termination was presented implicitly in [35]. It has been introduced as a progress property in [38] under the name *obstruction-free* synchronization. That notion has been formalized in [34]. More developments on obstruction-freedom can be found in [36]. The minimal knowledge on process failures needed to transform any solo-terminating implementation into a wait-free one was investigated in [37]. Other liveness properties (also called progress conditions) are presented in [39].

The algorithms in Figures 3.1 and 3.2 were proposed by Afek et al. [1]. A blocking variant of the algorithm in Figure 3.2 in which $\perp$ is never returned was given and proved correct by Herlihy and Wing [9].

Part II

Read-Write Memory

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.

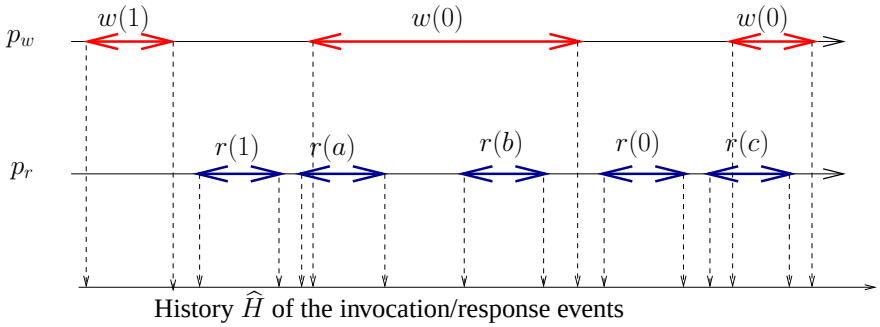

Figure 4.1: History of a register

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.

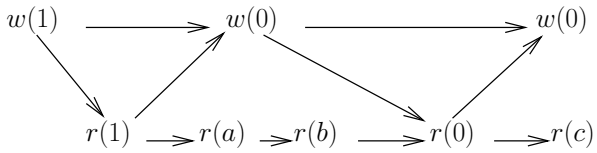

Figure 4.2: History of a safe register

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.

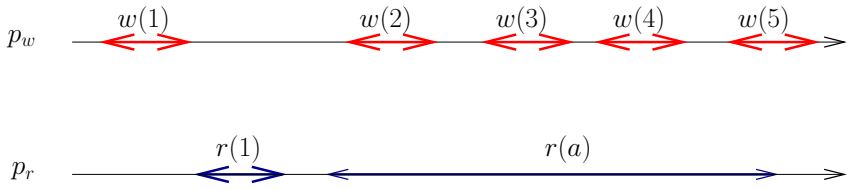

Figure 4.3: History of a regular register

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 4 *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 5 *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 4 and 5 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 6 *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. $\square_{Theorem\ 6}$

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

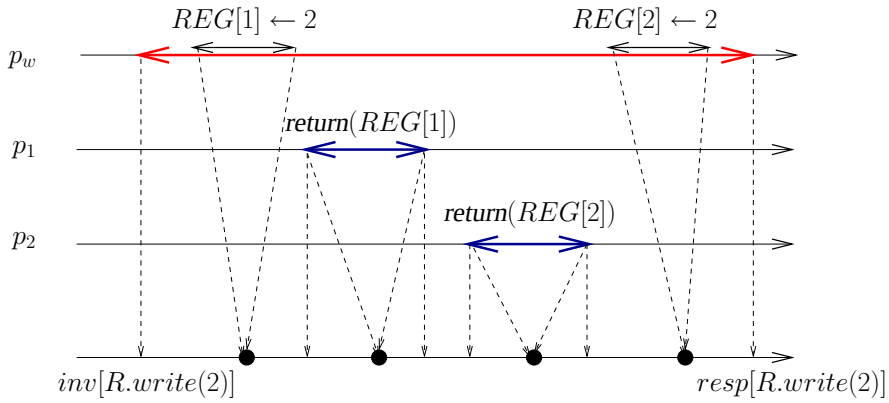

Figure 4.5: A counter-example

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.

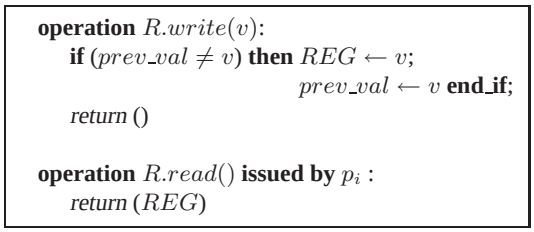

Figure 4.6: From a binary safe to a binary regular register (bounded transformation)

Theorem 7 *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. $\square_{Theorem\ 7}$

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 8 *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. $\square_{Theorem\ 8}$

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 9 *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.

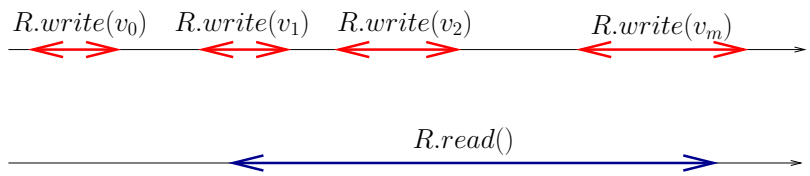

Figure 4.9: A read with concurrent writes

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.

$\square_{Theorem\ 9}$

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).

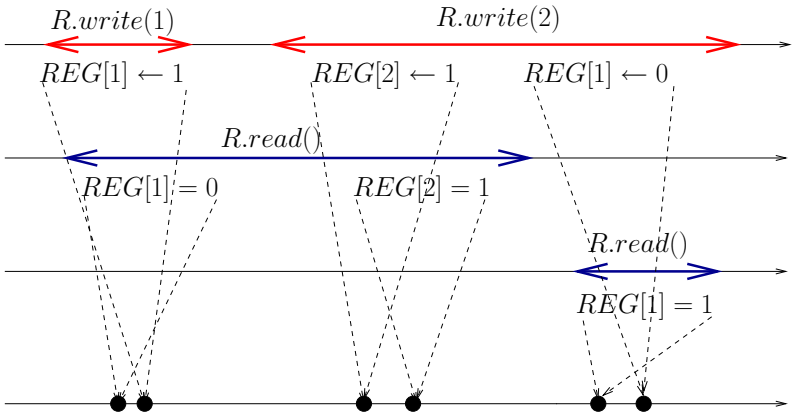

Figure 4.10: A counter-example for atomicity

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 10 *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 11 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 9 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.

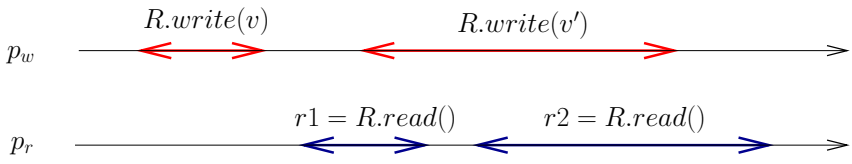

Figure 4.12: There is no new/old inversion

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.

$\square_{Theorem\ 10}$

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 11, 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 11 *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 11. 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).

$\square_{Theorem\ 11}$

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

Proof As for Theorem 11, 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 11. 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 11, but based on the sequence numbers provided by the arrays $REG[1 : n]$ and $HELP[1 : n, 1 : n]$, shows that S respects $\rightarrow_H$. $\square_{Theorem\ 12}$

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 13 *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 11. 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 11 but based on timestamps shows that S respects $\rightarrow_H$. $\square_{Theorem\ 13}$

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 [13].

Theorem 11, and the algorithms described in Figure 4.4, Figure 4.6, Figure 4.7 and Figure 4.8 are due to Lamport [13]. The algorithm described in Figure 4.11 is due to Vidyasankar [44]. The algorithms described in Figure 4.14 and 4.15 are due to Vityani and Awerbuch [48].

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 [49, 50, 51, 52, 53, 41, 42, 43, 45, 46, 47].

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 14 *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

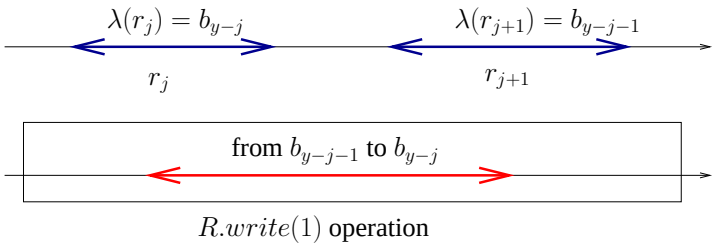

Figure 5.1: Two read operations r_j and $r_j + 1$ concurrent with $R.write(1)$

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. $\square_{Theorem\ 14}$

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 15 *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. $\square_{Theorem\ 15}$

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 15 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 ≠ 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 ≠ RR then change RR end_ if; % Strive to establish WR = RR %
4  val ← 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 ← 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.

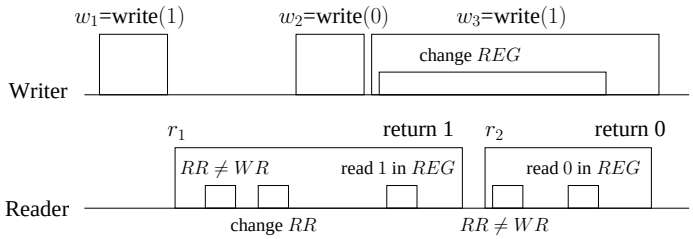

Figure 5.3: Counter example to step 2 of the construction: new-old inversion for r_1 and r_2

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.

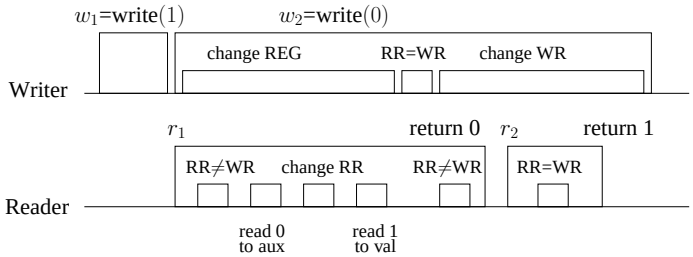

Figure 5.4: Counter example to step 3 of the construction: new-old inversion for r_1 and r_2

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.

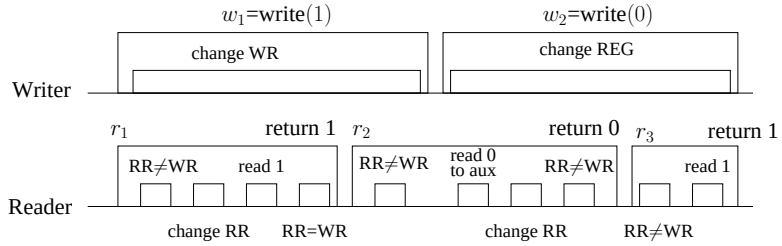

Figure 5.5: Counter example to step 4 of the construction: new-old inversion for r_2 and r_3

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 16 *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 5, 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 corresponding execution containing, additionally, the action invocation and response events is denoted L. Let $\rightarrow_L$ denote the corresponding partial relation on the actions.

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:

- 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.

Let ϕ be any regular reading function on REG. Thus, for each read action ρ_r we can define the corresponding write action $\phi(\rho_r)$ that writes the value returned by r. The write operation that contains $\phi(\rho_r)$ determines $\pi(r)$. If there is no such write operation, i.e., ρ_r returns the initial value of REG, we assume that $\pi(r)$ is the (imaginary) initial write operation that writes the initial value and precedes all actions in H.

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, $resp[r] <_L inv[\pi(\rho_r)]$)—a contradiction.

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 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. Recall that $\phi(\rho_r) \in \pi(r)$ (by definition). Let ω be the "change REG" action of the operation w ($\omega \in w$). By the case assumption, 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 by a read operation in line 3 *before* the read operation r starts but *after or concurrently with* the read action on RR performed by w. 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 that $r'' \neq r'$. 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),
- $inv[\rho_r] <_L resp[\omega]$ (since ϕ is a regular reading function and $\phi(rho_r)$ precedes ω);
  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, then it completes the write before w starts reading RR. But r' writes to RR r' after or concurrently with w reading RR. Therefore, $r'' \neq r'$ and, thus, $\rho_r \notin r'$. *End of the proof of the claim.*
But since 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 to which ρ_r is the last action "$val \leftarrow REG$" executed before r starts), which completes the proof of the assertion $A1$.

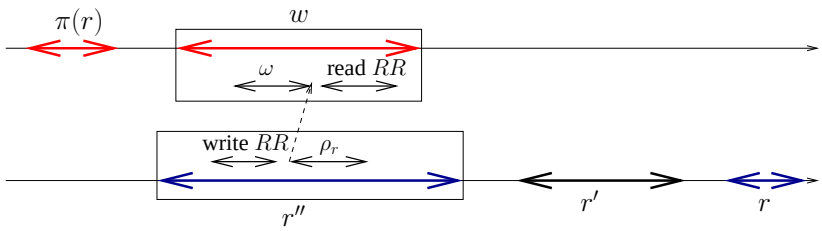

Figure 5.7: ρ_r belongs neither to r nor to r'

Proof of $A2$**.** 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, we assume that if $r1$ returns at line 1, then ρ_{r1} is the read action in line 6 in the immediately preceding read operation. Since $\pi(r2) \neq \pi(r1)$, we have $\rho_{r1} \neq \rho_{r2}$. Thus, 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 $resp[\rho_{r1}] <_L inv[r2]$. Combined with the case assumption, the assertion implies $\rho_{r2} \rightarrow_L \rho_{r1} \rightarrow_L r2$, 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.

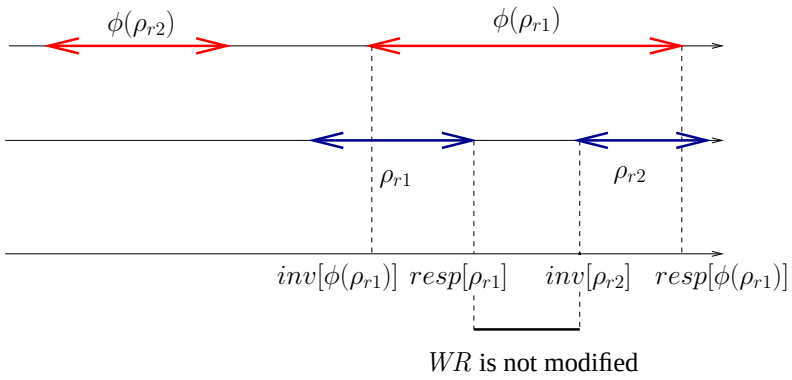

Figure 5.8: A new/old inversion on the regular register REG

- $\rho_{r1} \rightarrow_L \rho_{r2}$.
  By definition $\phi(\rho_{r1}) \in \pi(r1)$ and $\phi(\rho_{r2}) \in \pi(r2)$. As $\pi(r2) \rightarrow_H \pi(r1)$, we have $\phi(\rho_{r2}) \rightarrow_L \phi(\rho_{r1})$.
  Thus, we have $\phi(\rho_{r2}) \rightarrow_L \phi(\rho_{r1})$ and $\rho_{r1} \rightarrow_L \rho_{r2}$ (Figure 5.8) which implies a new/old inversion on the base regular register REG. But since ϕ is a regular reading function on REG, we have $\neg(\rho_{r1} \rightarrow_L \phi(rho_{r1})$ and $\neg(\phi(\rho_{r1}) \rightarrow_L \rho_{r2})$. Thus, both ρ_{r1} and ρ_{r2} have to overlap $\pi(\rho_{r1})$ (Figure 5.8): $inv[\phi(\rho_{r1})] <_L resp[\rho_1]$ and $inv[\rho_2] <_L resp[\phi(\rho_{r1})]$. As $\phi(\rho_{r1})$ is a base action that updates REG, and as REG and WR are both updated by the writer, the "value" of the base register WR does not change while the writer is updating REG or, more formally:

  Property P: all read actions on WR performed between $resp[\rho_{r1}]$ and $inv[\rho_{r2}]$ return the same value.

  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.
    - After having performed ρ_{r1}, $r1$ reads WR and if $WR \neq RR$, it sets $RR = WR$ in line 3. But $r1$ returns in line 7, after having seen RR different from WR in line 5 (otherwise, it would have returned in line 5). Thus, $r1$ reads different values of WR after ρ_{r1} (line 2 of $r1$) and before ρ_{r2} (line 6 of $r1$ or later). 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. Indeed, if $r2$ returns in line 1, the property P implies that the last read on REG preceding line 1 of $r2$ is line 4 of $r1$, i.e., $\rho_{r1} = \rho_{r2}$. Thus, $r2$ sees $RR \neq WR$ in line 1, before performing ρ_{r2} is in line 2 or line 4 of $r2$. But $r1$ has seen $WR = RR$ in line 5, after having performed ρ_{r1} in line 4—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 $r1$. Again, since $\rho_{r1} \rightarrow_L \rho_{r2}$, $r2$ does not return in line 1, from which we conclude that, before performing ρ_{r2}, $r2$ sees

$RR \neq WR$ in line 1. On the other hand, $r1$ sees $RR = WR$ in line 1 after having performed ρ_{r1} which contradicts property P and concludes the proof.

Thus, π is an atomic reading function. $\square_{Theorem\ 16}$

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)

Chapter 6

Collect and Snapshot objects

In many applications, processes cooperate in the following way. Periodically, a process posts a value based on its current state in the shared memory so that the other processes can read and use it. The last value stored by a process overwrites the value it has previously stored (if any). To compute the new value, a process accesses the shared memory to get the "latest" values stored by the other processes. This kind of cooperation is usually used in asynchronous systems: processes compute, post and read values at their own paces, and we assume no bounds on the relative processing speeds. Thus, we expect the cooperation to be *wait-free*.

In a concurrent system, it may be hard however to say which value is the latest one. We discuss multiple ways of defining such abstractions, starting from the weakest *store-collect* object, to stronger *snapshot* and immediate snapshot objects.

6.1 Store-collect object

A *store-collect* object exports the operation $store()$ that is used to post values and the operation $collect()$ that is used to obtain get the values that have been posted so far. The set of collected values defines what is called a *view*: a view V is an n-vector, with one value per process. A $store(v)$ is invoked by process p_i to replace the value in position i of the view with v. The view returned by a $collect()$ operation contains $\perp$ at position i if no value posted by p_i has been found.

6.1.1 Definition

Let H be a history of events on a store-collect object: $inv[store()]$, $resp[store()]$, $inv[collect()]$ $resp[collect()]$ issued by the processes. Recall that $<_H$ denotes the total order on the events in H and $\rightarrow_H$ denoted the real-time order on the operations in H. As usual, we assume that H is well-formed: no process invokes a new operation on the store-collect object before its previous operation returns. Thus, for any two operations invoked by a given process, we can say which one *precedes* the other, i.e., all operation of a given process are related by $\rightarrow_H$. (See Chapter 2.)

Let V_1 and V_2 be two views returned by collect operations in a history H. We say that $V_1 \leq V_2$ if, for every process p_i, $V_1[i] = V_2[i]$ or $store(V_1[i])$ precedes $store(V_2[i])$ (notice that both $store()$ operations are issued by p_i). Respectively, $V1 < V2$ if $V1 \leq V2$ and $V1 \neq V2$.

Now a store-collect object satisfies, for every its history H, the following properties:

- *Validity*. Let col be a collect operation issued by p_j that returns a view V such that $V[i] = v$. Then there is a $store(v)$ operation st by p_i that started before col terminates, i.e., $col \nrightarrow_H st$.

  In other words, a $collect()$ operation cannot output values that have not been yet stored.

- *Up-to-dateness*. Let $st = store(v)$ be issued by p_i and $col = collect()$ be issued by p_i. If $st \rightarrow_H col$, then col returns a view V such that $V[i]$ is v or a value posted by p_i after v.

  The views returned by the collect operations are thus "up to date" in the sense that, as soon as a value has been posted, it cannot be ignored by future collect operations. Note that Validity and Up-to-dateness imply that each position i in the view returned by col contains the value written by the last store by p_i that precedes col or the value written by a store operation issued by p_i that is concurrent to col.

- *Partial Consistency*. Let col_1 and col_2 be two $collect()$ operations that obtain views V_1 and V_2, respectively. If $col_1 \rightarrow_H col_2$, then $V1 \leq V2$.

  This property requires that non-concurrent collect operations are mutually consistent, namely, a collect operation cannot obtain values older than the values obtained by a preceding collect operation. Note that there are no constraints on the views returned by concurrent collect operations.

6.1.2 A trivial implementation

A store-collect object has a trivial implementation from 1WMR atomic registers. This implementation is based on an array $REG[1..n]$ with one entry per process. Only p_i can write $REG[i]$, while all the processes can read it. For collecting values, a process reads the entries of the array $REG[1..n]$ in any order. Each entry is initialized to a default value $\perp$ ($\perp$ cannot be posted by a process). The construction is described in Figure 6.1.

```
when store(v) is invoked by p_i:
    REG[i] := v;
    return ()

when collect() is invoked:
    for_all j ∈ {1, ..., n} do
        V[j] := REG[j]
    end_for;
    return (V)
```

Figure 6.1: Simple implementation of a store-collect object

6.1.3 A store-collect object has no sequential specification

An abstraction A has a sequential specification S, if its behavior of can be expressed through a set of sequential histories of S. Intuitively, A behaves like an atomic implementation of S. Formally:

- Every implementation of A is an atomic implementation of S, and
- Every atomic implementation of S is an implementation of A.

Note that the second property implies that *every* sequential history of S should be a history of A. If an abstraction A has a sequential implementation, we say that A is an *atomic object*.

Lemma 4 *Store-collect is not an atomic object.*

Proof Suppose, by contradiction, that the store-collect abstraction has a sequential specification S.

Consider the execution history in Figure 6.2. Here the $collect()$ issued by p_1 operation is concurrent with two store operations issued by p_2 and p_3. The history could have been exported by an execution of the algorithm in Figure 6.1, where p_1, within its $collect()$ operation, reads $REG[2]$ before any write on $REG[2]$ performed by p_2 and $REG[3]$ after the write on $REG[3]$ performed by p_3.

By our assumption, the history should be atomic with respect S. We recall that any linearization of H should respect the real-time order on operations and, thus, should put $[store(v)$ by $p_2]$ before $[store(v')$ by $p_3]$. We establish a contradiction by showing that there is no way to find a place for the $collect()$ operation in any linearization of H.

Suppose that S allows placing the $collect()$ operation *before* $[store(v')$ by $p_3]$. Thus, S contains a sequential history that violates the Validity property of store-collect (the collect operation returns a value which is not written yet)!

Now suppose that S allows placing the $collect()$ operation after $[store(v')$ by $p_3]$. This results in a history that violates the Up-to-dateness of store-collect (the collect operation returns an overwritten value)!

$\square_{Lemma\ 4}$

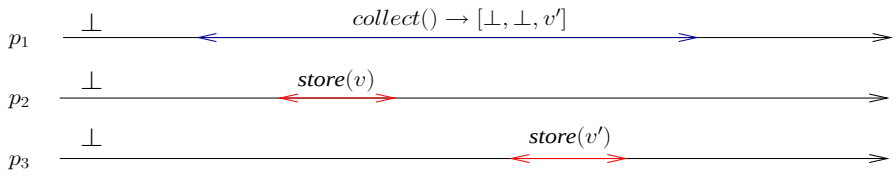

Figure 6.2: A store-collect object has no sequential specification

6.2 Snapshot object

We have seen that a store-collect object provides the processes with two operations, $store(v)$ that allows a process to define v as its last posted data, and $collect()$ that allows a process to obtain the last data posted by each process (if any). We have also seen that a store-collect object cannot be captured by a sequential specification. One of the reasons is that the partial consistency property allows concurrent collect operations to be ordered arbitrarily.

In this chapter, we introduce an "atomic restriction" of store-collect: a *snapshot* object. A snapshot object is accessed with operations $update()$ and $snapshot()$. The $snapshot()$ operation returns a vector of n values (one per process). The value in position i of the vector is affected by last preceding or concurrent $update(v)$ operation executed by process p_i.

A snapshot object satisfies the properties of store-collect (Section 6.1.1), where *store* and *collect* are replaced with $update$ and $snapshot$, plus the following property:

- *Snapshot order*. For any two views S_1 and S_2 obtained by snapshot operations, either $S_1 \leq S_2$ or $S_2 \leq S_1$

  The property implies that all views obtained by a snapshot operation can be totally ordered respecting the $\leq$ relation.

- *Update order*. For any two updates $update(v)$ and $update(v')$ performed by p_i and p_j, respectively, and any view S obtained by a snapshot operation, if the response of $update(v)$ precedes the invocation of $update(v')$ and S contains v' in position j, then S contains v or a later value in position i.

  in other words, non-concurrent updates cannot be observed by snapshot operations in the opposite order.

The sequential specification of type snapshot defines the set of allowed sequential histories of $update$ and $snapshot$ operations. In every such sequential history, each position i of the vector returned by every $snapshot$ operation contains the argument of last preceding $update$ operation of p_i (if any, or the initial value otherwise). Intuitively, a snapshot operation allows to view update and snapshot operations as taking place instantaneously. We show that this type indeed captures the behavior of a snapshot object.

Lemma 5 *A snapshot abstraction is atomic.*

Proof We show below that any snapshot implementation is atomic with respect to the snapshot type, and any atomic implementation of the snapshot type is a snapshot implementation. Consider a finite history H of an implementation that satisfies the properties of collect (Section 6.1.1), plus the Snapshot Order and Update Order properties.

We construct a linearization L of H as follows. First we order all complete snapshot operations in H, based on the $\leq$ relation, so that the real-time order is respected. This is possible by the Total Order and the Partial Consistency properties. Indeed, if a snapshot operation returns S_1 and precedes a snapshot operation that returns S_2, then $S_1 \leq S_2$.

Let $update(v) = U$ be an operation performed by p_i. U is then inserted in L just before the first snapshot operation that returns v or a later value in position i, or at the end of the sequence if there is no such a snapshot. After having done this for every update, we obtain a sequence $[U_0], S_1, [U_1], S_2, [U_2], \ldots, S_k, [U_k]$, where each $[U_j]$ is a (possibly empty) sequence of update operations U such that snapshot S_j contains values older that written by U and S_{j+1} contains the value written by U or a later value. Now we rearrange elements of each $[U_j]$ so that the real-time order is respected. This is possible since the real-time order is acyclic.

Now we observe that the resulting linearization L does not violate the real-time order. Indeed, suppose that an operation op precedes op' in H. Three cases are possible:

- Both op and op' are update operations. Let op and op' belong to $[U_m]$ and $[U_\ell]$, respectively. If $\ell = k$ (op' belongs to the last subsequence of updates in L), then, by construction op precedes op' in L.

  Now suppose that $\ell < k$ and consider $S_{\ell+1}$. By the construction of L, S_{m+1} is the first snapshot that contains the value written by op' or a later value at the corresponding position. By the Update Order property, S_{m+1} also contains the value written by op or a later value at the corresponding position and, thus, $\ell \leq m$. Thus, op precedes op' in L.

- Both op and op' are snapshot operations that return views S and S', respectively. If op' is incomplete, then it does not appear in L. If op' is complete, then, by the Partial Consistency property (Section 6.1.1), $S \leq S'$. Thus, by construction, op precedes op' in L.

- op is an update and op' is a snapshot. By the Up-to-dateness property (Section 6.1.1), op' returns the value written by op or a later value, and, by the construction of L and the Snapshot Order property, op precedes op' in L.

- op is a snapshot and op' is an update. By the Validity property (Section 6.1.1), the value written by op' does not appear in the result of op. By the construction of L and the Snapshot Order property, op precedes op' in L.

Thus, any snapshot object is indeed a atomic implementation of the snapshot type.

Now consider a history H of a atomic implementation of the snapshot type. Let L be the linearization of H, i.e., L respects the real-time order in H, L is legal with respect to the snapshot type, and L is equivalent to a completion of H. Recall that, in particular, L contains every complete operation in H.

- Suppose that a snapshot operation S returns a value $v \neq \perp$ at position i in H. Since L is legal, v is the value written by the last update U of p_i that precedes S in L. Since L respects the real-time order, S cannot precede U in H, and, thus, Validity is ensured in H.

- Suppose an update U precedes a snapshot S in H. Since L respects the real-time order of H, U precedes S also in L. Since L is legal, $S_1 \leq S_2$ and, thus, Partial Consistency is ensured in H.

- Suppose a snapshot S_1 precedes a snapshot S_2 in H. Since L respects the real-time order of H, S_1 precedes S_2 also in L. Since L is legal, S returns the value written by U or a later value at the corresponding position and, thus, Up-to-dateness is ensured in H.

- All complete snapshot operations appear in L and, since L is legal, are related by $\leq$: Snapshot Order is ensured in H.

- Suppose an update U_1 precedes an update U_2 and a snapshot S returns the value written by U_2. Thus, U_1 precedes U_2 also in L. Since L is legal, U_2 precedes S in L. Thus, U_1 precedes S in L and, since L is legal, S returns the value written by U_1 or a later value: Update Order is ensured in H.

Thus, any atomic implementation of the snapshot type is indeed is a snapshot object. $\square_{Lemma\ 5}$

By default, when we refer to a snapshot object, we mean a *wait-free* atomic implementation of the snapshot type. But we start with a simple *non-blocking* implementation of atomic snapshot that only guarantees that at least one correct process completes every its operation. The construction assumes that the underlying base registers can store values of arbitrary size, i.e., we may associated ever-growing sequence numbers with every stored value. Then we turn the construction into a wait-free one. Finally, we present a wait-free snapshot implementation that uses bounded memory.

6.2.1 Non-blocking snapshot

In this section, we assume we can use base registers of unbounded size. Our n-process implementation of snapshot uses an array of atomic registers $REG[]$. Each value that can be stored in a register $REG[i]$ is associated with a sequence number that is incremented each time a new value is stored. So each $REG[i]$ consists of two fields, denoted $REG[i].sn$ and $REG[i].val$. The implementation of $update()$ is presented in Figure 6.3. Here sn_i is a local variable that p_i uses to generate sequence numbers.

To maintain consistency across the results of snapshot operations, each snapshot operation is implemented using the "double scan" technique: the process keeps reading registers $REG[1, \ldots, n]$ until two consecutive collects return identical results. The result of the last scan is then returned by the snapshot operation.

The $scan()$ function asynchronously reads the last (sequence number, data) pairs posted by each process:

function $REG.scan()$: **for** $j \in \{1, \ldots, n\}$ **do** $r[j] := REG[j]$ **end_do**; *return* (r).

operation $update\ (v)$ **invoked by** p_i:

$sn_i := sn_i + 1$ { *local sequence number generator* }

$REG[i] := [v, sn_i]$ { *store the pair* }

Figure 6.3: Update operation

operation $snapshot()$:

1 $aa := REG.scan()$;

2 **repeat forever**

3 $bb := REG.scan()$;

4 **if** $(\forall j :\ aa = bb)$ **then** *return* $(aa.val)$ **end_if**; { *return the vector of read values* }

5 $aa := bb$

6 **end_while**.

Figure 6.4: Snapshot operation

Theorem 17 *The algorithm in Figures 6.3 and 6.4 is a non-blocking atomic snapshot implementation.*

Proof To prove that the implementation is non-blocking, we observe first that the update operation contains only one base-object step. Consider an infinite execution of the algorithm, and suppose that a snapshot operation performed by a correct process p_i never terminates. By the algorithm, p_i thus executes infinitely many scans of REG. The only reason not to return in line 4 is to find out that one of the positions in REG has changed since the last scan. Thus, for every two consecutive scan operations S_1 and S_2 executed by p_i, another process p_j executes an update operation U such that write to $REG[j]$ in U takes place between the read of $REG[j]$ in S_1 and the read of $REG[j]$ in S_2.

Since there are only finitely many processes, at least one process performs infinitely update operations concurrently with the snapshot operation of p_i. Thus, in every infinite execution of the algorithm, at least one correct process completes every its operation. So the implementation is indeed non-blocking.

Now we prove atomicity. Let E be any finite execution of the algorithm and H be the corresponding history. Consider any complete $snapshot()$ operation in E. Let C_1 and C_2 be its last two scans. By the algorithm, C_1 and C_2 return the same result. Now we choose the linearization point of the snapshot operation to be any point in E between the response of C_1 and the invocation of C_2 (see example in Figure 6.5). Otherwise, if a snapshot operation does not return in E, we do not include it in the linearization (or, equivalently, we remove the operation from our complete extension of the corresponding history H).

Consider now an $update(v)$ operation executed by a process p_i in E. We linearize the operation at the point when it performs a write on $REG[i]$ in E (if it does not, we remove it from the completion of H).

Let L be the resulting *linearization* of H, i.e., the sequential history where operations appear in the order of their linearization points in E. By the construction, L is equivalent to a completion of H. Also, since each operation is linearized within its interval in E, L respects the real-time order of H. We show that L is legal, i.e., at every position i, every snapshot operation in L returns the value written by the latest preceding update of p_i.

Let S be a snapshot operation in L, and let C_1 and C_2 be the two last scans of S. For each p_i, let U_i be the last update operation of p_i preceding S in L. If there is no such an update operation in L, we artificially add $update(\perp)$ by p_i at the beginning of L and call it U_i. Recall that U_i is linearized at the write on $REG[i]$ and S is linearized between the response of C_1 and the invocation of C_2. Since, by the algorithm, C_1 and C_2

read the same value in $REG[i]$, no write on $idREG[i]$ takes place between the read of $REG[i]$ performed within C_1 and the read of $REG[i]$ performed within C_2. Thus, since the write operation performed within U_i is the last write on $REG[i]$ to precede the linearization point of S in E, we derive that it is also the last write on $REG[i]$ to precede the read of $REG[i]$ performed within C_1.

Therefore, for each p_i, the value of p_i returned by C_1 and, thus, by S is the value written by U_i. Hence, L is legal and the algorithm in Figure. $\square_{Theorem\ 17}$

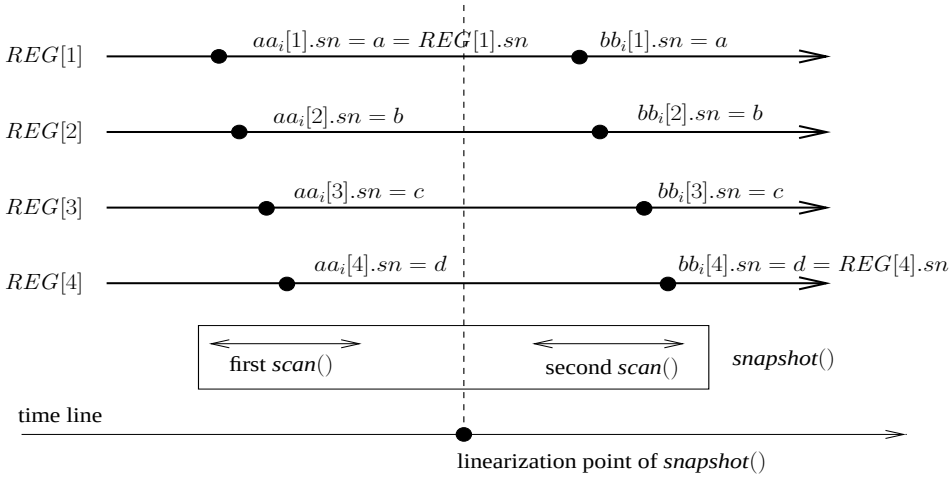

Figure 6.5: Linearization point of a $snapshot()$ operation

6.2.2 Wait-free snapshot

In the non-blocking snapshot implementation in Figures 6.3 and 6.4, update operations may starve a snapshot operation out by "selfishly" updating REG. This implementation can be turned into a wait-free one using *helping*: an update operations helps concurrent snapshot operations to terminate. An update operation may itself take a snapshot of and store the result together with the new value in REG. Of course, for this helping mechanism to work, we need to make sure that the intertwined snapshot and update operations do not prevent each other from terminating. Below we present an exact argument for this elegant idea.

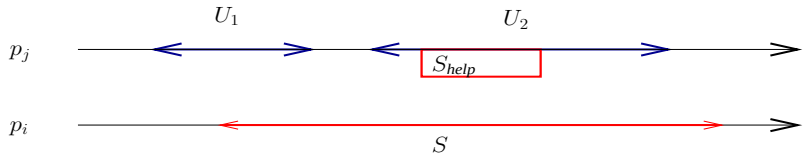

Figure 6.6: Each $update()$ operation includes a $snapshot()$ operation

First we can make the two following observations on the non-blocking snapshot implementation:

- If two consecutive scans performed within a snapshot operation are not identical (and, thus, the snapshot operation cannot return), then at least one process has concurrently performed an update operation.

- If a snapshot operation S issued by a process p_i witnesses that the value of $REG[j]$ has changed twice, i.e., p_j concurrently executed two update operations U_1 and U_2, then the second of these updates was entirely performed within the interval of S (see Figure 6.6). This is because the update by p_j of the base atomic register $REG[j]$ is the last operation executed in an $update()$ operation.

As the second update is entirely executed during S, we may use it to "help" S:

- Within U_2, p_j takes a snapshot itself (using the algorithm in Figure 6.4) and writes the result $help$ to $REG[j]$.
- Within S, p_i uses the result read in $REG[j]$ as the response of S. This is going to be a valid result, since the execution of U_2 (and, thus, of the snapshot performed by U_2) takes place entirely within the interval of S, so S can simply "borrow" the snapshot result $help$ from U_2.

Note that for this kind of helping to work, S must witness at least two concurrent updates of the same process. For example, even though the write on $REG[j]$ performed within U_1 takes place within the interval of S, the snapshot written by U_1 together with its value may have taken place way before the invocation of S. Thus, adopting the result of U_1's snapshot as the result of S may violate linearizability, since it may miss updates executed *after* the snapshot taken by U_1 but *before* the invocation of S. This is why, before adopting the snapshot taken by p_j, p_i should wait until it observes the second change in $REG[j]$.

The algorithms for resulting $update()$ and $snapshot()$ are described in Figure 6.7.

The atomic register $REG[i]$ consists now of three fields, $REG[i].val$ and $REG[i].sn$ as before, plus the new field $REG[i].help\_array$ that contains the result of the snapshot taken by p_i in the course of its latest update operation.

The new local variable $idcould\_help_i$ is used by process p_i when it executes $snapshot()$. Initially $\emptyset$, $idcould\_help_i$ contains the set of the processes that terminated update operations concurrently with the snapshot operation currently executed by p_i (lines 11-15). When p_i observes that a process $p_j \in could\_help$ updated its value in REG, i.e., p_i finds out that $aa_i[j].sn \neq bb_i[j].sn$, p_i returns $REG[j].help\_array$ as the result of its snapshot operation.

6.2.3 The snapshot object construction is bounded wait-free

Theorem 18 *Each $update()$ or $snapshot()$ operation returns after at most $O(n^2)$ operations on base registers.*

Proof Let us first observe that an $update()$ by a correct process always terminates as long as the $snapshot()$ operation it invokes always returns. So, the proof consists in showing that any $snapshot()$ issued by a correct process p_i terminates.

Suppose, by contradiction, that a snapshot operation executed by p_i has not returned after having executed n times the **while** loop (lines 5-19). Thus, each time it has executed the loop, p_i has found out that for some new $j \notin could\_help_i$, $aa_i[j].sn \neq bb_i[j].sn$ (line 11), i.e., p_j has executed a new $update()$ operation since the last $scan()$ of p_i. After this j is added to the set $could\_help_i$ in line 14.

Note that $i \notin could\_help_i$ (p_i does not change the value of $REG[i]$ while executing $snapshot()$). Thus, after $n-1$ iterations, $could\_help_i$ contains all other $n-1$ processes $\{1, \ldots, i-1, i+1, \ldots, n\}$. Therefore, when p_i executes the while loop for the nth time, for any p_j such that $aa_i[j].sn \neq bb_i[j].sn$ (line 11), it finds $j \in idcould\_help_i$ in line 12. By the algorithm, p_i returns in line 13, after having executed n iterations in lines 5-19—a contradiction.

```
operation update(v) invoked by p_i:
(1)  help_array_i := snapshot();
(2)  sn_i := sn_i + 1;
(3)  REG[i] := (v, sn_i, help_array_i)

operation snapshot():
(4)  could_help_i := ∅;
(5)  aa_i := scan();
(6)  while true do
(7)    bb_i := scan();
(8)    if (∀j ∈ {1,...,n} :  aa_i[j].sn = bb_i[j].sn)
(9)      then return (aa_i.val)
(10)     else  for_each j ∈ {1,...,n} do
(11)                  if (aa_i[j].sn ≠ bb_i[j].sn) then
(12)                    if (j ∈ could_help_i)
(13)                      then return (bb_i[j].help_array)
(14)                      else  could_help_i := could_help_i ∪ {j}
(15)                  end_if end_if
(16)           end_for
(17)   end_if;
(18) aa_i := bb_i
(19) end_while
```

Figure 6.7: Atomic snapshot object construction

Thus, every snapshot operation returns after having executed at most n **while** loops in lines 5-19. Since every loop involves exactly n base-object reads (in the scan operation on registers $REG[1], \ldots, REG[n]$), every snapshot terminates in $O(n^2)$ base-object steps. Same holds for an update operation, since it additionally executes only one base-object write. $\square_{Theorem\ 18}$

6.2.4 The snapshot object construction is atomic

Theorem 19 *The object built by the algorithms described in Figure 6.7 is atomic with respect to the snapshot type.*

Proof Let E be an execution of the algorithm and H be the corresponding history of E. To prove that the algorithm is indeed an atomic snapshot implementation, we construct a linearization of H, i.e., a total order L on the operations in H such that: (1) L is equivalent to a completion of H, (2) L respects the real-time order of H, and (3) L is legal, i.e., each $snapshot()$ operation S in L returns, for each process p_j, the value written by the last $update()$ operation of p_j that precedes S in L.

The desired linearization L is built as follows. The linearization point of a complete $update()$ operation in E is the write in the corresponding 1WMR register (line 3). Incomplete update operations are not included to L. The linearization point of a $snapshot()$ operation S issued by a process p_i depends on the line at which it returns.

(i) The linearization point of a S operation that terminates in line 9 (successful double $scan()$) is at any time time between the end of the first $scan()$ and the beginning of the second $scan()$ (see the proof of Theorem 17 and Figure 6.5).

(ii) The linearization point of a S operation that terminates in line 13 (i.e., p_i terminates with the help of another process p_j) is defined inductively as follows (see Figure 6.8). The arrows show the direction in

which snapshot results are adopted by one operation from another.

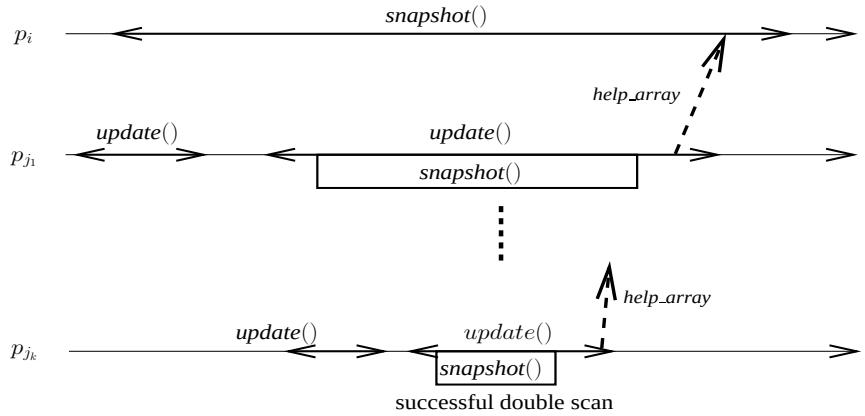

Figure 6.8: Linearization point of a $snapshot()$ operation (case ii)

Since S returns in line 13, the array (say $help\_array$) returned by p_i has been provided by an $update()$ operation executed by some process p_{j_1}. As we observed earlier, this $update()$ has been entirely executed within the interval of S. Indeed, $help\_array$ is the result of the second update operation of p_j that is observed by p_i to be concurrent with S. Thus, this update started after the invocation of S and its last event (the write in $REG[j]$ in line 8) before the response of S.

Recursively, $help\_array$ has been obtained by p_{j_1} from a successful double scan, or from another process p_{j_2}. As there are at most n concurrent processes, it follows by induction that there is a process p_{j_k} that has executed a $snapshot()$ operation within the interval of S and has obtained $help\_array$ from a successful double scan.

The linearization point of the $snapshot()$ operation issued by p_i is thus defined as the linearization point of $snapshot()$ operation of p_{j_k} whose double scan determined $help\_array$.

This association of linearization points to the operations in H results in a linearization L that puts the operation in the order their linearization points appear in E. L trivially satisfies properties (1) and (2) stated at the beginning of the proof. Reusing the proof of Theorem 17, we observe that, for every p_j, every snapshot operation S (be it a standalone snapshot or a part of an update) returns the value written to $REG[j]$ by the last update of p_j to precede the linearization point of S in E. Thus, L also satisfies (3), and the algorithm in Figure 6.7 is an atomic implementation of snapshot. $\square_{Theorem\ 19}$

6.2.5 Bounded snapshot object

Dolev-Shavit's bounded timestamps

Bibliographic notes

Afek et al. JACM

Aguilera 04

Attiya-Fouren

Borowsky-Gafni 93

Masuzawa 94, MWMR and $O(n)$

Exercises

One-shot snapshot from renaming (attiya)

Chapter 7

Immediate Snapshot and Iterated Immediate Snapshot

7.1 Immediate snapshot object

7.1.1 Immediate snapshot and participating set problem

One-shot immediate snapshot object A one-shot *immediate snapshot* object is a snapshot object where the $update()$ and $snapshot()$ are fused in a single operation denoted $update\_snapshot()$, and such that each process invokes at most once that operation. When a process p_i invokes $update\_snapshot(v)$, it deposits v as its last value and obtains a set V_i made up of (process identity, value) pairs. From an external observer point of view, everything has to appear as if the operation was executed instantaneously. (An analogous operation has been seen in the context of store-collect objects, where the $store\_collect()$ operation fuses $store()$ and $collect()$.)

More formally, a one-shot immediate snapshot object is defined by the following properties. Let V_i denote the set returned by p_i when it invokes $update\_snapshot(v_i)$.

- Liveness. An invocation of $update\_snapshot(v)$ by a correct process terminates.
- Self-inclusion. $(i, v_i) \in V_i$.
- Set inclusion. $\forall i, j : V_i \subseteq V_j$ or $V_j \subseteq V_i$.
- Immediacy. $\forall i, j :$ if $(j, v_j) \in V_i$ then $V_j \subseteq V_i$.

The first three properties are satisfied by a snapshot object where the $update\_snapshot()$ is implemented by a $update(v_i)$ invocation followed by a $snapshot()$ invocation. The last immediacy property is not satisfied by this implementation, which shows the fundamental difference between snapshot and immediate snapshot. This is illustrated in the Figures 7.1 and 7.2.

Figure 7.1 shows three processes p_1, p_2 and p_3. Each process executes an $update()$ followed by a $snapshot()$. The process identity appears as a subscript in the operation invoked. Moreover, the value written by p_i is i. According to the specification of the snapshot object, $snapshot_1()$ returns $[1, 2, \perp]$ (where $\perp$ is the value initially placed in the register associated with each process), while $snapshot_2()$ and $snapshot_3()$ return $[1, 2, 3]$. This means that it is possible to associate with this execution the following sequence of operations $\widehat{S}$

$$update_1(1)\ \ update_2(2)\ \ snapshot_1()\ \ update_3(3)\ \ snapshot_2()\ \ snapshot_3(),$$

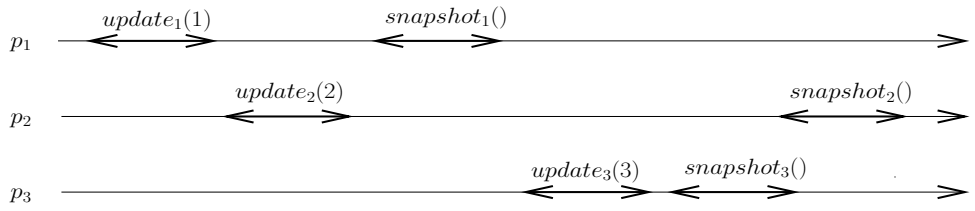

Figure 7.1: $update()$ and $snapshot()$ operations

thereby showing the atomicity of this execution.

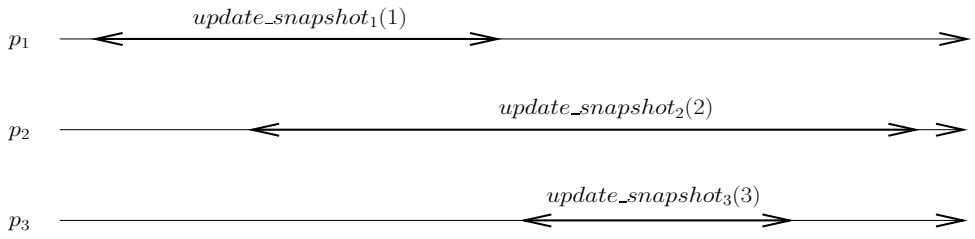

Figure 7.2: $update\_snapshot()$ operations

Figure 7.2 shows the same processes where the operations $update_i()$ and $snapshot_i()$ issued by p_i are replaced by a single $update\_snapshot_i()$ (that starts at the same time as $update_i()$ starts, and terminates at the same time as $snapshot_i()$ terminates). As $update\_snapshot_1(1)$ terminates before $update\_snapshot_3(3)$ starts, it is not possible for the latter to return the value 1 written by p_1. Let V_i be the set of pairs returned by p_i. The following results are possible:

- $V_1 = \{(1,1)\}$, $V_2 = \{(1,1),(2,2)\}$, and $V_3 = \{(1,1),(2,2),(3,3)\}$,
- $V_1 = V_2 = \{(1,1),(2,2)\}$, and $V_3 = \{(1,1),(2,2),(3,3)\}$,
- $V_2 = \{(2,2)\}$, $V_1 = \{(1,1),(2,2)\}$, and $V_3 = \{(1,1),(2,2),(3,3)\}$,
- $V_1 = \{(1,1)\}$, and $V_2 = V_3 = \{(1,1),(2,2),(3,3)\}$, and
- $V_1 = \{(1,1)\}$, $V_3 = \{(1,1),(3,3)\}$, and $V_2 = \{(1,1),(2,2),(3,3)\}$.

When V_1, V_2 and V_3 are all different, everything appears as if the $update\_snapshot()$ operations have been executed sequentially (and consistently with their realtime occurrence order). When two of them are equal, e.g., $V_1 = V_2 = \{(1,1),(2,2)\}$ (second case), everything appears as if $update\_snapshot_1(1)$ and $update\_snapshot_2(2)$ have been executed at the very same time, both before the $update\_snapshot_3()$ operation. This possibility of simultaneity is the very essence of the "immediate" snapshot abstraction. It also shows that an immediate snapshot object is not an atomic object.

Theorem 20 *A one-shot immediate object satisfies the following property: if* $(i,-) \in V_j$ *and* $(j,-) \in V_i$, *then* $V_i = V_j$.

Proof If $(j,-) \in V_i$ (theorem assumption), we have $V_j \subseteq V_i$, due to the immediacy property. Similarly, $(i,-) \in V_j$ implies that $V_i \subseteq V_j$. It trivially follows that $V_i = V_j$ when $(j,-) \in V_i$ and $(i,-) \in V_j$. $\square_{Theorem\ 20}$

This theorem states that, while its operations appear as if they were executed instantaneously, an immediate snapshot object is not an atomic object. This is because it is not always possible to totally order

all its operations. The immediacy property states that, from a logical time point of view, it is possible that operations occur simultaneously (they then return the same result), making impossible to consider that one occurred before the other. Differently from atomic snapshot objects, the specification of an immediate snapshot object allows for concurrent operations. It requires that these operations return the very same result. Stated another way, this means that an immediate snapshot object has no sequential specification.

The participating set problem The *participating set* problem is a particular instance of the one-shot immediate snapshot problem. It considers the case where the value v_i deposited by a process p_i is its own identity. The corresponding operation is consequently denoted $participate()$. The properties a set V_i returned by $participate()$ are then:

- Self-inclusion. $i \in V_i$.
- Set inclusion. $\forall i, j : V_i \subseteq V_j$ or $V_j \subseteq V_i$.
- Immediacy. $\forall i, j :$ if $j \in V_i$ then $V_j \subseteq V_i$.

7.1.2 A one-shot immediate snapshot construction

This section describes a very simple one-shot immediate snapshot algorithm based on an algorithm solving the participating set problem. (The section that follows provides a solution to that problem.)

The algorithm is described in Figure 7.3. It uses an array $REG[1:n]$ of 1WMR atomic registers, and a participating set object denoted $PART$. $REG[i]$ is the register where p_i deposits its value. Its initial value is $\perp$.

```
operation update_snapshot(v) invoked by p_i:
(1) REG[i] ← v;
(2) present ← PART.participate();
(3) result ← ∅;
(4) for_each j ∈ present do result ← result ∪ {(j, REG[j])} end_do;
(5) return (result)
```

Figure 7.3: Atomic snapshot object construction

Theorem 21 *The algorithm described in Figure 7.3 is a bounded wait-free implementation of a one-shot immediate snapshot object.*

Proof Let us first observe that the algorithm is bounded wait-free as soon as the algorithm implementing the underlying participating set object $PART$ is bounded wait-free. We will see in Theorem 22 that there is a bounded wait-free implementation of $PART$.

As a process p_i that invokes $update\_snapshot(v)$, first updates its register $REG[i]$, and then invokes $PART.participate()$, it follows that a participating process has always deposited a value. The rest of the proof follows directly from the specification of the object $PART$. The set of process identities returned to p_i is the set from which it builds its result. As this set satisfies the self-inclusion, set inclusion and immediacy properties associated with the object $PART$, the set of pairs computed satisfies the corresponding properties of the one-shot immediate snapshot specification. $\square_{Theorem\ 21}$

7.1.3 A participating set algorithm

Underlying data structure A participating set algorithm is described in Figure 7.4. This algorithm uses an array of 1WMR atomic registers $LEVEL[1:n]$, where $LEVEL[i]$ can be written only by p_i. A process p_i uses also a local array $level_i[1:n]$ to keep the last values it has (asynchronously) read from $LEVEL[1:n]$. A register $LEVEL[i]$ contains at most n distinct values (from $n+1$ until 1), which means that it requires $b = \lceil \log_2(n) \rceil$ bits. It is initialized to $n+1$.

```
operation participate() invoked by p_i:
      % initially: ∀j : LEVEL[j] = n + 1 %
(1)   repeat LEVEL[i] ← LEVEL[i] − 1;
(2)         for_each j ∈ {1, ..., n} do level_i[j] ← LEVEL[j] end_do;
(3)         set_i ← {x | level_i[x] ≤ level_i[i]}
(4)   until (|set_i| ≥ level_i[i]);
(5)   return (set_i)
```

Figure 7.4: A participating set algorithm

Underlying principles of the algorithm let us consider the image of a stairway made up of n stairs. Initially all the processes stand at the highest stair (i.e., the stair whose, number is $n+1$). (This is represented in the algorithm by the initial values of the $LEVEL$ array, namely, for any process p_j, we have $LEVEL[j] = n+1$.)

The algorithm is based on the following idea. When a process p_i invokes $participate()$, it descends along the stairway, going from the step $LEVEL[i]$ to the step $LEVEL[i]-1$ (line 1), until it attains a step k such that there are k processes (including itself) stopped on the steps 1 to k . It then returns the identities of these k processes.

To catch the underlying intuition and understand how this idea works, let us consider two extremal cases in which k processes invoke the $participate()$ operation.

- Sequential case.
  In this case, the k processes invokes the operation sequentially, i.e., the next invocation starts only after the previous one has returned. It is easy to see that the first process p_{i_1} that invokes the $participate()$ operation proceeds from the step $n+1$ until the step number 1, and stops at this step. Then, the process p_{i_2} starts and descends from the step $n+1$ until the step number 2, etc., and the last process p_{i_k} stops at the step k.

  Moreover, the set returned by p_{i_1} is $\{i_1\}$, the set returned by p_{i_2} is $\{i_1, i_2\}$, etc., the set returned by p_{i_k} being $\{i_1, i_2, \ldots, i_k\}$. These sets trivially satisfy the inclusion property.

- Synchronous case.
  In this case, the k processes proceed synchronously. They all, simultaneously, descend from the step $n+1$ to the step n, and then from the step n to the step $n-1$, etc., and they all stop at the step number k, as there are then k processes at the steps from 1 to k (they all are on the same kth step).

  It follows that all the processes return the very same set of participating processes, namely, the set including all of them $\{i_1, i_2, \ldots, i_k\}$.

Other cases, where the processes proceed asynchronously and some of them crash, can easily be designed.

The main question is now: how to make operational this idea? This is done by three statements (Figure 7.4). Let us consider a process p_i:

- First, when it is standing on a given step $LEVEL[i]$, p_i reads the steps at which the other processes are (line2). The aim of this asynchronous reading is to allow p_i to compute an approximate global state of the stairway. Let us notice that as a process p_j can go only downstairs, $level_i[j]$ is equal or smaller to the step $k = LEVEL[i]$ on which p_j currently is. It follows that, despite the fact the global state obtained by p_i is approximate, set_i can be safely used by p_i.

- Then (line3), p_i uses the approximate global state it has obtained, to compute a set set_i of processes that are standing at a step comprised between $LEVEL][1]$ and $LEVEL][i]$, the step where p_i currently is.

- Finally (line 4), if set_i is contains $k = LEVEL][i]$ or more processes, p_i returns it as its result to the participating set problem. Otherwise, it descends to the next stair $LEVEL][i] - 1$ (line 1).

Proof the algorithm Two preliminaries lemmas are proved before the main theorem.

Lemma 6 *Let $set_i = \{x \mid level_i[x] \leq LEVEL[i]\}$ (as computed at line 3). For any process p_i, the predicate $|set_i| \leq LEVEL[i]$ is always satisfied at line 4.*

Proof Let us first observe that $level_i[i]$ and $LEVEL[i]$ are always equal at lines 3 and 4. Moreover, any $LEVEL[j]$ register can only decrease, and for any (i, j) pair we have $LEVEL[j] \leq level_i[j]$.

The proof is by contradiction. Let us assume that there is at least one process p_i such that $|set_i| = |\{x \mid level_i[x] \leq LEVEL[i]\}| > LEVEL[i]$. Let k the current value of $LEVEL[i]$ when this occurs. $|set_i| > k$ and $LEVEL[i] = k$ mean that at least $k + 1$ processes have progressed at least to the stair k. Moreover, as any process p_j descends one stair at a time (it proceeds from the stair $LEVEL[j]$ to the stair $LEVEL[j] - 1$ without skipping stairs), at least $k + 1$ processes have proceeded from the stair $k + 1$ to the stair k.

Among the $\geq k+1$ processes that are on stairs $\leq k$, let p_ℓ be the last process that updated its $LEVEL[\ell]$ register to $k + 1$ (due to the atomicity of the base registers, there is such a last process). When p_ℓ was on the stair $k + 1$ (we then had $LEVEL[\ell] = k + 1$), it obtained at line 3 a set set_ℓ such that $|set_\ell| = |\{x \mid level_\ell[x]| \leq LEVEL[\ell]\} \geq k + 1$ (this is because $\geq k + 1$ processes have proceeded to the stair $k + 1$ and, as p_ℓ is the last of them, it has read a value $\leq k + 1$ from its own $LEVEL[\ell]$ register and the ones of those processes). As $|set_\ell| \geq k + 1$, p_ℓ stopped descending the stairway at line 4, at the stair $k + 1$. It then returned, contradicting the initial assumption stating that it progresses until the stair k. $\square_{Lemma\ 6}$

Lemma 7 *If p_i halts at the stair k, we then have $|set_i| = k$. Moreover, set_i is composed of the processes that are at a stair $k' \leq k$.*

Proof Due to Lemma 6, we always have $|set_i| \leq LEVEL[i]$, when p_i executes line 4. If it stops, we also have $|set_i| \geq LEVEL[i]$ (test of line 4). It follows that $|set_i| = LEVEL[i]$. Finally, if k is p_i's current stair, we have $LEVEL[i] = k$ (definition of $LEVEL[i]$ and line 1). Hence, $|set_i| = k$.

The fact that set_i is composed of the identities of the processes that are at a stair $\leq k$ follows from the very definition of set_i (namely, $set_i = \{x \mid level_i[x] \leq LEVEL[i]\}$), the fact that, for any x, $level_i[x] \leq LEVEL[x]$, and the fact that a process never climbs the stairway (it either halts on a stair, line 4, or descends to the next one, line 1). $\square_{Lemma\ 7}$

Theorem 22 *The algorithm described in Figure 7.4 is a bounded wait-free implementation of a participating set object.*

Proof Let us observe that (1) $LEVEL[i]$ is monotonically decreasing, and (2), at any time, set_i is such that $|set_i| \geq 1$ (because it contains at least the identity i). It follows that the *repeat* loop always terminates (in the worst case when $LEVEL[i] = 1$). It follows that the algorithm is wait-free. Moreover, p_i executes the *repeat* loop at most n times, and each computation inside the loop includes n read of atomic base registers. It follows that $O(n^2)$ is an upper bound on the number of read/write operations on base registers involved in a $participate()$ operation. The algorithm is consequently bounded wait-free.

The self-inclusion property is a direct consequence of the way set_i is computed (line 3): trivially, the set $\{x \mid level_i[x] \leq level_i[i]\}$ contains always i.

For the set inclusion property, let us consider two processes p_i and p_j, that stop at stairs k_i, and k_j, respectively. Without loss of generality, let $k_i \leq k_j$. Due to Lemma 7, there are exactly k_i processes on the stairs 1 to k_i, and k_j processes on the stairs 1 to $k_j \leq k_i$. As no process backtracks on the stairway (a process descends or stops), the set of k_j processes returned by p_j includes the set of k_1 processes returned by p_i.

It follows from the lines 3 and 4 that, if a a process p_j stops at a stair k_j and then $i \in set_j$, then p_i stopped at a stair $k_i \leq k_j$. It follows from Lemma 7 that the set set_j returned by p_j includes the set set_i returned by p_i, which proves the immediacy property. $\square_{Theorem\ 22}$

7.2 A connection between (one-shot) renaming and snapshot

7.2.1 A weakened version of the immediate snapshot problem

Let us consider a weakened version of the (one-shot) immediate snapshot problem without the immediacy property. This means that, when a process p_i invokes $update\_snapshot(v_i)$ it obtains a set V_i, and the sets returned satisfy the following properties:

- Self-inclusion. $(i, v_i) \in V_i$.
- Set inclusion. $\forall i, j : V_i \subseteq V_j$ or $V_j \subseteq V_i$.

This section shows that a one-shot snapshot algorithm can be obtained from a simple modification of a renaming algorithm.

7.2.2 The adapted algorithm

We consider here the renaming algorithm, based on reflector base objects, that has been described in chapter 7. The idea, to adapt it to solve the previous specification, comes from the following observation. In addition to routing processes, reflectors can be used to help processes to collect their final view V_i.

Instead of being boolean atomic registers, the base atomic objects $VISITED[0..1]$ contains now sets of pairs (i, v_i), i.e., they are views. They are initialized to $\emptyset$. (The meaning of $VISITED[y] = \emptyset$ is the same as the meaning of $\neg\, VISITED[y]$ in the base implementation of a reflector object.)

The operation $reflect()$ is modified accordingly to take into account the computation of views. In addition to an entrance number (0 or 1), it takes a view V as additional parameter. Let us remind that (1) a process that enters a reflector on the entrance labeled y, leaves it on an exit with the same label (up_y or $down_y$), and (2) the network is designed in such a way that, for each reflector, each of its entrance is used by at most one process.

The modified $reflect(V, y)$ is as follows (Figure 7.5). The input parameter V is the current estimate of the final view of the invoking process. Initially, $V = \{(i, v_i)\}$. The aim of a $reflect()$ invocation is to enrich

V in order it converges to a final value that satisfies self-inclusion and set inclusion. When a process enters the reflector on the entrance y, it writes its current local view in $VISITED[y]$, and then reads the other register $VISITED[1-y]$. If it empty, its local view V does not change, and the process exits on $down_y$. Otherwise the process adds the view in $VISITED[1-y]$ to V and exits on up_y.

```
function reflect (V, y):
(1)  VISITED[y] ← V;
(2)  if (VISITED[1 − y] = ∅) then return (V, down_y)
(3)                          else return (V ∪ VISITED[1 − y], up_y) endif
```

Figure 7.5: Adapting the reflector base object

The algorithm that directs the progress of a process in the network of reflectors is exactly the same as in the renaming algorithm. The only difference is in the returned value. Instead of a row number, the view obtained after the process has visited its last reflector (that reflector belongs to the last column) is returned as its final view to the invoking process. The corresponding $update\_snapshot()$ operation is described in Figure 7.6. Let us remind that the reflector object whose coordinates are (r, c) is denoted $R[r, c]$. The algorithm can be trivially modified to solve both one-shot renaming and one-shot snapshot.

```
operation update_snapshot (v_i):
(1)  V_i ← {(i, v_i)}; c_i ← id_i; r_i ← id_i;
(2)  while (c_i = id_i) do
(3)        (V_i, exit) ← R[r_i, c_i].reflect (V_i, 1);
(4)        if (exit = up_1) then c_i ← c_i + 1
(5)                         else r_i ← r_i − 1;
(6)                              if r_i < −c_i then c_i ← c_i + 1 endif
(7)        endif
(8)  endwhile;
(9)  while (c_i < N) do
(10)       (V_i, exit) ← R[r_i, c_i].reflect (V_i, 0);
(11)       c_i ← c_i + 1;
(12)       if (exit = up_0) then r_i ← r_i + 1
(13)                        else r_i ← r_i − 1
(14)       endif
(15) endwhile;
(16) return (V_i)      %  0 ≤ r_i + N ≤ 2(n − 1)  %
```

Figure 7.6: From renaming to snapshot

Let us remind that the new name of a process in the original renaming algorithm is the number of the row where it is when it attains the last column. It follows from that algorithm and the modified $reflect()$ operation that, if two processes p_i and p_j are such that the new name of p_i is smaller than the new name of p_j, we have $V_i \subseteq V_j$. The self-inclusion property follows directly from the $reflect()$ operation, as the set it returns always includes its input parameter set, and the set V_i, whose final value it returned to p_i, is initialized to $\{(i, v_i)\}$.

7.3 Iterated immediate snapshot

Iterated memory and nonblocking simulation.

Bibliographic notes

Afek et al. JACM

Aguilera 04

Attiya-Fouren

Borowsky-Gafni 93

Masuzawa 94, MWMR and $O(n)$

Gafni and Rajsbaum, OPODIS 2010

Exercises

One-shot snapshot from renaming (attiya)

Part III

General Memory

Chapter 8

Consensus and universal construction

In the first part of this book, we considered multiple powerful abstractions that can be wait-free implemented using read-write registers. A natural question: can any object type be implemented this way? We show in this chapter that the answer is no: for example, a queue cannot be wait-free implemented even when shared by two processes. More generally, we address the following fundamental question:

> Given object types T and T', is there a wait-free implementation of an object of type T from objects of type T'?

Recall that an object operation can be either total or partial (Section 2.2.2). A pending partial operation may not always be able to complete. Indeed, there are executions in which the partial operation cannot be linearized, and, thus, it must be forced to wait until the value of the object allows it to proceed. In contrast (Chapter 2.6), a pending total operation can always be completed by a process, regardless the behavior of the other processes. Thus, only total operations can be wait-free implemented. In this chapter we assume *total* object types.

8.1 What cannot be read-write implemented

To warm up, let us consider a *queue* object type that exports two operations *enqueue*() and *dequeue*(). In a sequential execution, $enqueue(v)$ adds v to the end of the queue and $dequeue()$ returns the first element in the queue and removes it from the queue. If the queue is empty the default value $\perp$ is returned.

8.1.1 The case of one dequeuer

Let us assume only one process is allowed to invoke $dequeue()$ on the concurrent implementation of the queue. Such a restricted queue allows for a simple read-write wait-free implementation.

Each enqueuer p_i maintains a register R_i which stores the sequence of values enqueued by p_i so far, each value equipped with a "timestamp." Each time p_i enqueues a value, it scans all the registers to find a the highest timestamp t used so far and updates R_i equipped with timestamp $t + 1$. The dequeuer simply reads all registers R_i and returns the value with the lowest timestamp that was not previously returned (ties broken arbitrarily, e.g., by picking the value enqueued by the enqueuer with the lowest id).

operation $propose(v)$:
if $(x = \perp)$ **then** $x := v$ **endif**;
return (x).

Figure 8.1: Consensus specification: sequential execution of $popose(v)$

Intuitively, the implementation is correct since we only need to break the ties for values that were concurrently enqueued and thus can be linearized either way. We encourage the reader to find a formal correctness argument.

[[PK exercise: prove that it is correct?]]

8.1.2 Two or more dequeuers

What about a general queue, shared by two or more processes, where every process is allowed to enqueue or dequeue elements? We show below that this

Schedules, configurations and values go here

Lemma 8 *Every queue implementation has a bivalent configuration.*

Lemma 9 *Every queue implementation has a critical configuration.*

Theorem 23 *There is no wait-free two-process queue implementation from atomic registers.*

Proof

$\Box_{Theorem}$ **??**

Since any n-process wait-free implementation ($n \geq 2$) implies a 2-process wait-free implementation, we have:

Corollary 2 *For any $n \geq 2$, there is no wait-free n-process queue implementation from atomic registers.*

8.2 Universal objects and consensus

An object type T is *universal* if, given any (total) type T, an object of type T' can be wait-free implemented from objects of type T, together with atomic registers. An algorithm providing such an implementation is called a *universal construction.*

In this chapter, we introduce *consensus* as an example of a universal object type. We present two consensus-based universal constructions. The first is wait-free, the second one is *bounded* wait-free. (Recall that an implementation is bounded wait-free if there is a bound on the number of base-object steps an operation must perform to terminate.)

The *consensus* object type exports an operation $propose()$ that takes one input parameter v in a *value set* V ($|V| \geq 2$) and returns a value in V. Let $\perp$ denote a default value that cannot be proposed by a process ($\perp \notin V$). Then $V \cup \{v\}$is the set of states a consensus object can take, $\perp$ is its initial state, and its sequential specification is defined in Figure 8.1. A consensus objects can thus be seen as a "write-once" register that keeps forever the value proposed by the first $propose()$ operation. Then, any subsequent $propose()$ operation returns the first written value.

Given a *linearizable* implementation of the consensus object type, we say that a process *proposes* v if it invokes $propose(v)$ (we then say that it is a *participant* in consensus). If the invocation of $propose(v)$ returns a value v', we say that the invoking process *decides* v', or v' is decided by the consensus object. We observe now that any execution of a *wait-free* linearizable implementation of the consensus object type satisfies three properties:

- *Agreement:* no two processes decide different values.
- *Validity:* every decided value was previously proposed.

  Indeed, otherwise, there would be no way to linearize the execution with respect to the sequential specification in Figure 8.1 which only allows to decide on the first proposed value.
- *Termination:* Every correct process eventually decides.

  This property is implied by wait-freedom: every process taking sufficiently many steps of the consensus implementation must decide.

8.3 A wait-free universal construction

In this section, we show that if, in a system of n processes, we can wait-free implement consensus, then we can implement *any* total object type.

Recall that a total object type can be represented as a tuple (Q, q_0, O, R, δ), where Q is a set of states, $q_0 \in Q$ is an initial state, O is a set of operations, R is a set of responses, and δ is a binary relation on $O \times Q \times R \times Q$, total on $O \times Q$: $(o, q, r, q') \in \delta$ if operation o is applied when the object's state is q, then the object *can* return r and change its state to q'. Note that for *non-deterministic* object types, there can be multiple such pairs (r, q') for given o and q.

The goal of our universal construction is, given an object type $\tau = (Q, O, R, \delta)$, to provide a wait-free linearizable implementation of τ using read-write registers and atomic consensus objects.

8.3.1 Deterministic objects

For deterministic object types, δ can be seen as a function $O \times Q \to R \times Q$ that associates each state an operation with a unique response and a unique resulting state. The state of a deterministic object is thus determined by a sequence of operations applied to the initial state of the object. The universal construction of an object of deterministic is presented in Figure 8.2.

Correctness.

Lemma 10 *At all times, for all processes p_i and p_j, $linearized_i$ and $linearized_j$ are related by containment.*

Proof We observe that $linearized_i$ is constructed by adding the batches of requests decided by consensus objects $C_1, C_2, \ldots$, in that order. The agreement property of consensus (applied to each of these consensus objects) implies that, for each j, either $linearized_i$ is a prefix of $linearized_i$ or vice versa. $\square_{Lemma\ 10}$

Lemma 11 *Every operation returns in a finite number of its steps.*

Shared objects:
R, store-collect object, initially $\perp$
$C_1, C_2, \ldots$, consensus objects

Local variables, for each process p_i:
integer seq_i, initially 0 { *the number of executed requests of* p_i }
integer k_i, initially 0 { *the number of batches of executed requests* }
sequence $linearized_i$, initially empty { *the sequence of executed requests* }

Code for operation op executed by p_i:
7 $seq_i := seq_i + 1$
8 $R.store(op, i, seq_i)$ { *publish the request* }
9 **repeat**
10 $V := R.collect()$ { *collect all current requests* }
11 $requests := V - \{linearized_i\}$ { *choose not yet linearized requests* }
12 $k_i := k_i + 1$
13 $decided := C[k].propose(requests)$
14 $linearized_i := linearized_i.decided$ { *append decided requests* }
15 **until** $(op, i, seq_i) \in linearized_i$
16 return the result of (op, i, seq_i) in $linearized_i$ using δ and q_0

Figure 8.2: Universal construction for deterministic objects

Proof Suppose, by contradiction, that a process p_i invokes an operation op and executes infinitely many steps without returning. By the algorithm, p_i forever blocks in the repeat-until clause in lines 20-15. Thus, p_i proposes batches of requests containing its request (op, i, seq_i) to an infinite sequence of consensus instances $C_1, \ldots$ but the decided batches never contain (op, i, seq_i). By validity of consensus, there exists a process $p_j \neq p_i$ that accesses infinitely many consensus objects. By the algorithm, before proposing a batch to a consensus object, p_j first collects the batches currently stored by other processes in a store-collect object R. Since p_i stores its request in R and never updates it since that, eventually, every such process p_j must collect the p_i's request and propose it to the next consensus object. Thus, every value returned by the consensus objects from some point on must contain the p_i's request—a contradiction. $\square_{Lemma\ 11}$

Theorem 24 *For each type $\tau = (Q, q_0, O, R, \delta)$, the algorithm in Figure 8.2 describes a wait-free linearizable implementation of τ using consensus objects and atomic registers.*

Proof Let H be the history an execution of the algotihm in Figure 8.2. By Lemma 10, local variables $linearized_i$ are prefixes of some sequence of requests $linearized$. Let L be the legal sequential history, where operations and are ordered by $linearized$ and responses are computed using q_0 and δ. We construct H', a completion of H, by adding responses to the incomplete operations in H that are present in L. By construction, L agrees with the local history of H' for each process.

Now we show that L respects the real-time order of H. Consider any two operations op and op' such that $op \rightarrow_H op'$ and suppose, by contradiction that $op' \rightarrow_L op$. Let (op, i, s_i) and (op', j, s_j) be the corresponding requests issued by the processes invoking op and op', respectively. Thus, in $linearized$, (op', j, s_j) appears before (op, i, s_i), i.e., before op terminates it witnesses (op', j, s_j) being decided by consensus objects $C_1, C_2, \ldots$ before (op', j, s_j). But, by our assumption, $op \rightarrow_H op'$ and, thus, (op', j, s_j) has been stored in the store-collect object R *after* op has returned. But the validity property of consensus

Shared objects:
  R, store-collect object, initially $\perp$ { *published requests* }
  $C_1, C_2, \ldots$, consensus objects
  S, store-collect object, initially $(1, \epsilon)$ { *the current consensus object and the last committed sequence of requests* }

Local variables, for each process p_i:
  integer seq_i, initially 0 { *the number of executed requests of* p_i }
  integer k_i, initially 0 { *the number of batches of executed requests* }
  sequence $linearized_i$, initially ϵ { *the sequence of executed requests* }

Code for operation op executed by p_i:
17 $seq_i := seq_i + 1$
18 $R.store(op, i, seq_i)$ { *publish the request* }
19 $(k_i, linearized_i) := \max(S.collect())$ { *get the current consensus object and the most recent state* }
20 **repeat**
21 $V := R.collect()$ { *collect all current requests* }
22 $requests := V - \{linearized_i\}$ { *choose not yet linearized requests* }
23 $k_i := k_i + 1$
24 $decided := C[k].propose(requests)$
25 $linearized_i := linearized_i.decided$ { *append decided requests* }
26 **until** $(op, i, seq_i) \in linearized_i$
27 $S.store((k_i + 1, linearized_i))$ { *publish the current consensus object and state* }
28 return the result of (op, i, seq_i) in $linearized_i$ using δ and q_0

Figure 8.3: Bounded wait-free universal construction for deterministic objects

does not allow to decide a value that has not yet been proposed—a contradiction. Thus, $op \rightarrow_L op'$, and we conclude that H is linearizable. $\square_{Theorem\ 24}$

8.3.2 Bounded wait-free universal construction

The implementation described in Figure 8.2 is wait-free but not *bounded* wait-free. A process may take arbitrarily many steps in the repeat-until clause in lines 20-15 to "catch up" with the current consensus object.

It is straightforward to turn this implementation into a bounded wait-free. Before returning an operation's response (line 16), a process posts in the shared memory the sequence of requests it has witnessed committed together with the id of the last consensus object it has accessed. On invoking an operation, a process reads the memory to get the "most recent" state on the implemented object and the "current" consensus id. Note that multiple processes concurrently invoking different operations might get the same estimate of the "current state" of the implementation. In this case only one of them may "win" in the current consensus instance and execute its request. But we argue that the requests of "lost" processes must be then committed by the next consensus object, which implies that every operation returns in a bounded number of its own steps. The bound here depends on the implementation of

The resulting implementation is presented in Figure 8.3.

To prove the following theorem, we assume that it takes $O(n)$ read-write steps to implement store-collect objects R and S (Chapter **??**).

Theorem 25 *For each type $\tau = (Q, q_0, O, R, \delta)$, the algorithm in Figure 8.3 describes a wait-free linearizable implementation of τ using consensus objects and atomic registers, where every operation returns in $O(n)$.*

Proof The proof of linearizability is similar to the one in the proof of Theorem 24.

To prove bounded wait-freedom, consider a request (op, i, ℓ) issued by a process p_i. By the algorithm, p_i first publishes its request and obtains the current state of the implemented object (line 19), denoted k and s, respectively. Then p_i proposes all requests it observes proposed but not yet committed to consensus object C_k. If (op, i, ℓ) is committed by C_k, then p_i returns after taking $O(n)$ read-write steps (we assume that both collect operations involve $O(n)$ read-write steps).

Suppose now that (op, i, ℓ) is not committed by C_k. Thus, another process p_j has previously proposed to C_k a set of requests that did not include (op, i, ℓ). Thus, p_j collected requests in line 21 before p_i published (op, i, ℓ) in line 18.

[[PK: to complete]]

$\Box_{Theorem\ 25}$

8.3.3 Non-deterministic objects

The universal construction in Figure 8.2 assumes the object type is deterministic, where for each state and each operation there exists exactly one resulting state and response pair. Thus, given a sequence of request, there is exactly one corresponding sequence of responses and state transitions.

A "dumb" way to use our universal construction is to consider any deterministic restriction of the given object type. But this may not be desirable if we expect the shared object to behave probabilistically (e.g., in randomized algorithms). A "fair" non-deterministic universal construction can be derived from the algorithm in Figure 8.3 as follows. Instead of only proposing a sequence of requests in line 24, process p_i (non-deterministically) picks a sequence of possible responses and state transitions assuming that the sequence of operations in $requests$ is applied to the last state in $linearized_i$.

[[PK: to complete]]

8.4 Bibliographic notes

Herlihy 1991

Attiya-Welch 1998

State machine replication: Lamport, Schneider

Chandra-Toueg total order broadcast from consensus

Guerraoui-Raynal 2004: tech report on FT atomic objects

Chapter 9

Consensus number and the consensus hierarchy

In the previous chapter, we introduced a notion of a *universal* object type. Using read-write registers and objects of a universal type and, one can wait-free implement an object of any total type. One example of a universal type is *consensus*. Therefore, the following question is fundamental:

Which object types allow for a wait-free implemention of consensus?

For example, do atomic registers can implement consensus on their own? If not, what about queues and registers? In this chapter, we address this question by introducing the notion of *consensus number* of an object type T, the largest number of processes for which T is universal. Consensus number is fundamental in capturing the relative power of object types, we show how to evaluate the consensus power of various object types.

9.1 Consensus number

The *consensus number* of an object type T, denoted by $CN(T)$, is the largest number n such that it is possible to wait-free implement a consensus object from atomic registers and objects of type T, in a system of n processes. If there is no such largest n, i.e, consensus can be implemented in a system of arbitrary number of processes, the consensus number of T is said to be infinite.

Note that if there exists a wait-free implementation in a system of n implies a wait-free implementation in a system of any $n' < n$ processes. Thus, that the notion of consensus number is well-defined. By the definition, if $CN(T) < CN(T')$, then there is no wait-free implementation of an object of type T' from objects of type T and registers in a system of $CN(T) + 1$ or more processes.

What if atomic registers are strong enough to wait-free implement consensus for any number of processes, i.e., $CN(regiter) = \infty$? Then all object types would have the same consensus number, and the very notion of consensus number would be useless. We show in this chapter that this is not the case. Moreover, we show that for each n, there exists object types T, such that $CN(T) = n$, i.e., the *consensus hierarchy* is populated for each level n.

9.2 Preliminary definitions

In this section, we introduce some machinery that facilitates computing consensus numbers of various object types. This includes the notions of a schedule, a configuration, and valence.

9.2.1 Schedule, configuration and valence

This section defines news notions (schedule, configuration and valence) that are central to prove the impossibility to wait-free implement a consensus object from some "base" object types. Before giving these definitions, it also reminds a few notions and results introduced in the first chapter, that are useful to better understand results presented in this chapter.

Reminder Let us consider an execution made up of sequential processes that invoke operations on atomic objects of types $T_1, \ldots, T_x$. These objects are called "base objects" (equivalently, the types $T_1, \ldots, T_x$ are called "base types"). We have seen in the first chapter (theorem 1), that, as each base object is atomic, that execution can be modeled, at the operation level, by an atomic history $\widehat{S}$ on the operations issued by the processes. This means that $\widehat{S}$ is a sequential history that (1) includes all the operations issued by the processes (except possibly the last operation of a process if that process crashes), (2) is legal, and (3) respects the real time occurrence order on the operations. As we have seen, such a history $\widehat{S}$ is also called a *linearization*.

Schedules and configurations A *schedule* is a sequence of operations issued by processes. Sometimes an operation is represented in a schedule only by the name of the process that issues that operation.

A *configuration* C is a global state of the system execution at a given point in time. It includes the value of each base object plus the local state of each process. The configuration $p(C)$ denotes the configuration obtained from C by applying an operation issued by the process p. More generally, given a schedule S and a configuration C, $S(C)$ denotes the configuration obtained by applying to C the sequence of operations defining S.

Valence The *valence* notion is a fundamental concept to prove consensus impossibility results. Let us consider a consensus object such that only the values 0 and 1 can be proposed by the processes. Such an object is called a *binary consensus* object. Let us assume that there is an algorithm A implementing such a consensus object from base type objects. Let C be a configuration attained during an execution of the algorithm A.

The configuration C is *v-valent*, if from C, no matter the schedule it applies to C, the algorithm always leads to v as the decided value; v is the valence of that configuration. If $v = 0$ (resp., $v = 1$) C is said to be 0-valent (resp., 1-valent) and 0 (resp., 1). A 0-valent or 1-valent configuration is said to be *monovalent*. A configuration that is not monovalent is said to be *bivalent*.

While a monovalent configuration states that the decided value is determined (be processes aware of it or not), the decided value is not yet determined in a bivalent configuration.

9.2.2 Bivalent initial configuration

The next theorem shows that, for any wait-free consensus algorithm A, there is at least one initial bivalent configuration, i.e., a configuration in which the decided value is not predetermined: any one from several

proposed value can still be decided (for each of these values v, there is a schedule generated by the algorithm A that, starting from that configuration, decides v).

This means that, while the decided value is only determined from the inputs when the initial configuration is univalent, this is not always true for all configurations, as there is at least one initial bivalent configuration. The value decided by a wait-free consensus algorithm cannot always be deterministically determined from the inputs. It can also depend on the execution of the algorithm A itself.

Theorem 26 *Let us assume that there is an algorithm A that wait-free implements a consensus object in a system of n processes. There is then a bivalent initial configuration.*

Proof Let C_0 be the initial configuration in which all the processes propose 0 to the consensus object, and C_i, $1 \leq i \leq n$, the initial configuration in which the processes from p_1 to p_i propose the value 1, while all the other processes propose 1. So, all the processes propose 1 in C_n. These configurations constitute a sequence in which any two adjacent configurations C_{i-1} and C_i, $1 \leq i \leq n$, differ only in the value proposed by the process p_i: it proposes the value 0 in C_{i-1} and the value 1 in C_i. Moreover, it follows from the validity property of the consensus algorithm A, that C_0 is 0-valent, while C_n is 1-valent.

Let us assume that all the previous configurations are univalent. It follows that, in the previous sequence, there is (at least) one pair of consecutive configurations, say C_{i-1} and C_i, such that C_{i-1} is 0-valent and C_i is 1-valent. We show a contradiction.

Assuming that no process crashes, let us consider an execution history $\widehat{H}$ of the algorithm A that starts from the configuration C_{i-1}, in which the process p_i executes no operation for an arbitrarily long period (the end of that period is defined below). As the algorithm is wait-free, all the processes decide after a finite number of their operations. The sequence of operations that starts at the very beginning of the history and ends when all the processes have decided (but p_i, which has not yet executed an operation), defines the schedule S. (See the upper part of Figure 9.1. Within the vector of the values proposed by the processes, the value proposed by p_i has been placed inside a box.) Then, after S terminates, p_i starts executing and eventually decides. As C_{i-1} is 0-valent, $S(C_{i-1})$ is also 0-valent.

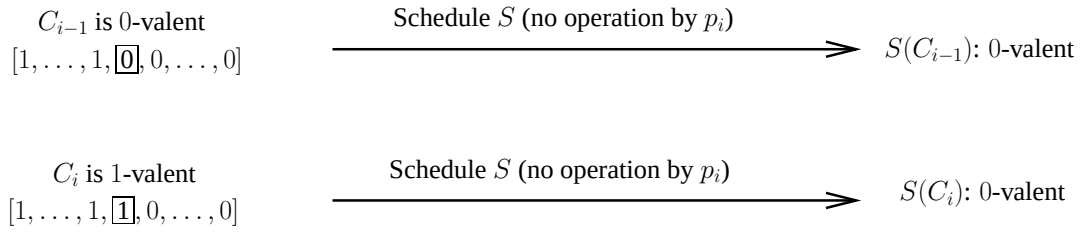

Figure 9.1: There is a bivalent initial configuration

Let us observe (lower part of Figure 9.1) that the same schedule S can be produced by the algorithm A from the configuration C_i. This is because (1) the configurations C_{i-1} and C_i differ only in the value proposed by p_i, and, (2) as p_i executes no operation in S, that schedule cannot depend on the value proposed by p_i. It follows that, as $S(C_{i-1})$ is 0-valent, the configuration $S(C_i)$ is also 0-valent. But as, on another side, C_i is 1-valent, we conclude that $S(C_i)$ is 1-valent, a contradiction. $\square_{Theorem\ 26}$

Crash vs asynchrony The previous proof is based on (1) the assumption stating that the consensus algorithm A is wait-free (intuitively, the progress of a process does not depend on the "speed" of the other

processes), and (2) asynchrony (a process progresses at its "own speed"). This allows the proof to play with process speed, and consider a schedule (part of an execution history) in which a process p_i does not execute operations. We could have instead considered that p_i has initially crashed (i.e., p_i crashes before executing any operation). During the schedule S, the wait-free consensus algorithm A (the existence of which is a theorem assumption) has no way to know in which case the system really is (has p_i initially crashed or is it only very slow?). This shows that, for some problems, asynchrony and process crashes are two facets of the same "uncertainty" wait-free algorithms have to cope with.

9.3 The weak wait-free power of atomic registers

We have seen in the second part of this book that atomic registers allows wait-free implementing atomic counters and atomic snapshot objects. As atomic registers are very basic objects, an important question from a computability point of view, is then: can atomic registers wait-free implement objects such as a queue or a stack shared by concurrent processes. This section shows that the answer to this question is "no".

More precisely, this section shows that MWMR atomic registers are not powerful enough to wait-free implement a consensus object in a system of two processes. This means that the consensus number of the type "atomic register" is 1, which means that atomic registers allow wait-free implementing consensus in a system made up of a single process! Stated another way, atomic registers have the "poorest" power when one is interested in wait-free implementations of atomic objects in systems of asynchronous processes prone to process crashes.

9.3.1 The consensus number of atomic registers is 1

To show that there is no algorithm that wait-free implements a consensus object in a system of two processes p and q, the proof assumes such an algorithm and derives a contradiction. The concept central in that proof is the notion of valence previously introduced.

Theorem 27 *There is no an algorithm A that wait-free implements a consensus object from atomic registers in a set of two processes (i.e., the consensus number of atomic registers is* 1.*)*

Proof Let us assume (by contradiction) that there is an atomic register-based algorithm A that wait-free implements a consensus object in a set of two processes. Due to theorem 26, there is an initial bivalent configuration. The proof of the theorem consists in showing that, starting from a bivalent configuration C, there is always an arbitrarily long schedule S produced by A that leads from C to another bivalent configuration $S(C)$. It follows that A has a run in which no process ever decides, which proves the theorem.

Given a configuration D, let us remind that $p(D)$ is the configuration obtained by applying the next operation of the process p -as defined by the algorithm A- to the configuration D. Let us also remind that the operations p or q can issue are reading or writing a base atomic register.

Let us assume that, starting the algorithm from the bivalent configuration C, there is a maximal schedule S such that $D = S(C)$ is bivalent. "Maximal" means that both the configuration $p(D)$ and the configuration $q(D)$ are monovalent, and have different valence (otherwise, D would not be bivalent). Without loss of generality, let us consider that $p(D)$ is 0-valent, while $q(D)$ is 1-valent.

The operation that leads from D to $p(D)$ is a read or a write by p of a base register $R1$. Similarly, the operation that leads from D to $q(D)$ is a read or a write by q of a base register $R2$. The proof consists in a case analysis.

1. $R1$ and $R2$ are distinct registers (Figure 9.2).
   In that case, whatever are the (read or write) operations OP1() and OP2() issued by p and q on the base registers $R1$ and $R2$, as the processes access different registers, the configurations $p(q(D))$ and $q(p(D))$ are the same configuration, i.e., $p(q(D)) \equiv q(p(D))$.

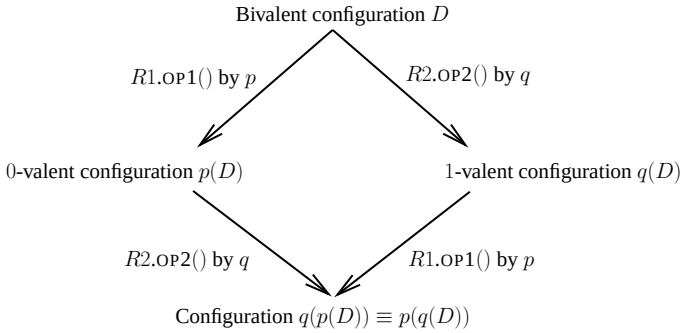

Figure 9.2: Operations issued on distinct registers

   As $q(D)$ is 1-valent, it follows that $p(q(D))$ is also 1-valent. Similarly, as $p(D)$ is 0-valent, it follows that $q(p(D))$ is also 0-valent. A contradiction, as the configuration $p(q(D)) \equiv q(p(D))$ cannot be both 0-valent and 1-valent.

2. $R1$ and $R2$ are the same register R.

   - Both p and q read R.
     As a read operation on an atomic register does modify its value, this case is the same as the previous one where p and q access distinct registers.
   - p reads R, while q writes R (Figure 9.3).
     (Let us notice that the case where q reads R, while p writes R is similar.) Let $Read_p$ be the read operation issued by p on R, and $Write_q$ be the write operation issued by q on R. As $Read_p(D)$ is 0-valent, so is $Write_q(Read_p(D))$. Moreover, $Write_q(D)$ is 1-valent.

     The configurations D and $Read_p(D)$ differ only in the local state of p (it has read R in $Read_p(D)$, while it has not in D). These two configurations cannot be distinguished by q. Let us consider the following two executions:

     – After the configuration D has been attained by the algorithm A, p stops executing for an arbitrarily long period, and during that period only q executes operations. As by assumption the algorithm A is wait-free, there is a finite sequence of operations issued by q at the end of which q decides. Let S' be the schedule made up of these operations. As $Write_q(D)$ is 1-valent, it decides 1. (Thereafter, p wakes up and executes operations as specified in the the algorithm A. Alternatively, p could crash after the configuration D has been attained.)
     – Similarly, after the configuration $Read_p(D)$ has been attained by the algorithm A, p stops executing for an arbitrarily long period The same schedule S' (defined in the previous item) can be issued by q after the configuration $Read_p(D)$. This is because, as p issues no operation, q cannot distinguish D from $Read_p(D)$. It follows that, q decides at the end of that schedule, and, as $Write_q(Read_p(D))$ is 0-valent, q decides 0.

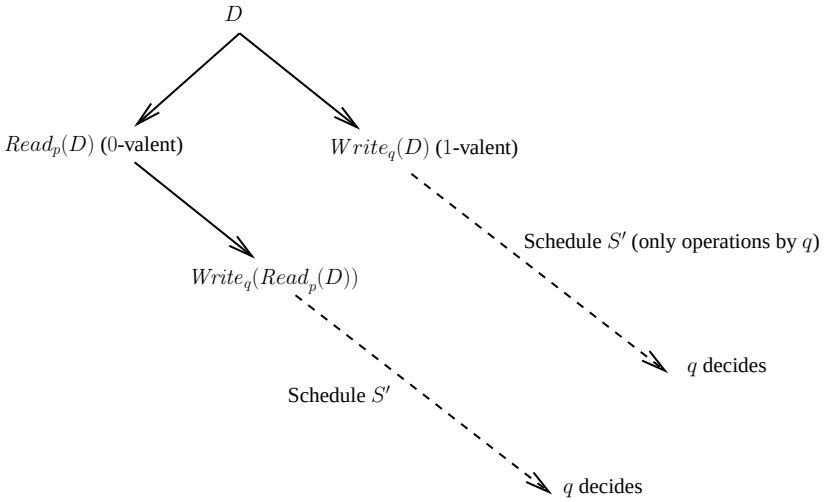

Figure 9.3: Read and write issued on the same register

But, while executing the schedule S', q cannot know which (D or $Read_p(D)$) was the configuration when it started executing S' (this is because, these configurations differ only in a read of R by p). As the schedule S' is deterministic (it is composed only of read and write operations issued by q on base atomic registers), q must decide the same value, whatever the configuration at the beginning of S'. A contradiction as it decides 0 in the first case and 1 in the second case.

- Both p and q write the same register R.
  Let $Write_p$ and $Write_q$ be the write operations issued by p and q on R, respectively. By assumption the configurations $Write_p(D)$ and $Write_q(D)$ are is 0-valent and 1-valent, respectively.
  The configurations $Write_q(Write_p(D))$ and $Write_q(D)$ cannot be distinguished by q: the write of R by p in the configuration D that produces the configuration $Write_p(D)$ is overwritten by q when it produces the configuration $Write_q(Write_p(D))$.
  The reasoning is then the same as in the previous item. It follows that, if q executes alone from D until it decides, it decides 1 after executing a schedule S''. The same schedule from the configuration $Write_p(D)$ leads to decide 0. But, as q cannot distinguish D from $Write_p(D)$, and S'' is deterministic, it follows that it has to decide the same value in both executions, a contradiction as it decides 0 in the first case and 1 in the second case. (Let us observe that Figure 9.3 is still valid. We have only to replace $Read_p(D)$ and S' by $Write_p(D)$ and S'', respectively.)

$\square_{Theorem\ 27}$

9.3.2 The wait-free limit of atomic registers

Theorem 28 *It is impossible to wait-free implement any object with consensus number greater than 1 from atomic registers.*

Proof The proof is an immediate consequence of theorem 27 (the consensus number of atomic registers is 1), and theorem **??** (if $CN(X) < CN(Y)$, X does allow wait-free implementing Y in a system or more than $|CN(X)|$ processes). $\square_{Theorem\ 28}$

9.3.3 Another limit of atomic registers

Naming the anonymous

9.4 Objects whose consensus number is 2

As atomic registers are too weak to wait-free implement a consensus object for two processes, the question posed at the beginning of the chapter becomes: are they objects that allow wait-free implementing a consensus object for two or more processes. This section first considers three base objects (test&set objects, queue, and swap objects) and show that they can wait-free implement consensus in a set of two processes denoted p_0 and p_1 (considering the process indices 0 and 1 makes the presentation simpler). It then shows that they cannot wait-free implement consensus in a set of three or more processes.

9.4.1 Consensus from a test&set objects

Test&set objects A test&set object is an atomic object that provides the processes with a single operation (called *test&set*, hence the name of the object). Such an object can be seen as maintaining an internal state variable x that can contain the value 0 or 1. It is initialized to 0 and can be accessed by the operation *test&set*(). Assuming only one operation at a time is executed, its sequential specification is defined as follows:

operation $test\&set\ ()$:
  $prev\_val \leftarrow x$;
  if $(prev\_val = 0)$ **then** $x \leftarrow 1$ **endif**;
  return $(prev\_val)$.

From test&set objects to consensus The algorithm described in Figure 9.4 constructs a consensus object for two processes from a test&set object TS. It uses two additional 1W1R atomic registers $REG[0]$ and $REG[1]$ (a process p_i can always keep a local copy of the atomic register it writes, so we do not count it as one of its readers). The construction is made up of two parts:

- When the process p_i invokes $propose(v)$ on the consensus object, it deposits the value it proposes into $REG[i]$ (line 1). This part consists for p_i in making public the value it proposes.

- Then p_i executes a control part to know which value has to be decided. To that aim, it uses the test&set object (line 2). If it obtains the initial value of the test&set object (0), it decides the value it has proposed (line 3); otherwise it decides the value proposed by the other process p_{1-i}(line 4).

In the following, we call *winner* the process that is the first to execute line 2. More precisely, as the test&set object is atomic, the winner is the process whose $TS.test\&set()$ operation is the first to appear in the linearization order associated with the object TS. The proof shows that the value decided by the consensus object is the value deposited by the winner p_j in its register $REG[j]$.

Theorem 29 *The algorithm described in Figure 9.4 is a wait-free construction of a consensus object from a test&set object, in a system of two processes.*

```
operation propose(v) issued by p_i:
(1)  REG[i] ← v;
(2)  aux ← TS.test&set ();
(3)  case (aux = 0) then return (REG[i])
(4)       (aux = 1) then return (REG[1 − i])
(5)  endcase
```

Figure 9.4: From test&set to consensus

Proof The algorithm is clearly wait-free. Let p_j be the winner. Let us observe that it deposits the value v it proposes in $REG[j]$ before invoking $TS.test\&set()$ (this follows from the fact that, as both $REG[j]$ and TS are atomic, an execution that involves both of them is also atomic, and consequently the linearization order -with which we reason- respects process order). When the winner p_j executes line 2, the test&set object TS changes its value from 0 to 1, and then, as any other invocation finds $TS = 1$, the test&set object keeps forever the value 1. As p_j is the only process that obtains the value 0 from the object TS, it decides the value v it has just deposited in $REG[j]$ (line 3). Moreover, as the other process obtains the value 1 from TS, that process does not decide the value it proposes but the other proposed value, namely, the value deposited $REG[j]$ by the winner p_j (line 4). It follows that a single value is decided, and that value has been proposed by a process. Consequently, the algorithm described in Figure 9.4 is wait-free wait-free implementation of a consensus object in a system of two processes. $\square_{Theorem\ 29}$

9.4.2 Consensus from queue objects

Queue objects These objects have been already used in several chapters. A queue is defined by two total operations with a sequential specification. The enqueue operation adds an item at the end of the queue. The dequeue operation removes the item at the head of the queue and returns it to the calling process; if the queue is empty, the default value $\perp$ is returned.

From queue objects to consensus An wait-free algorithm that constructs a consensus object from a queue, in a system of two processes, is described in Figure 9.5. This algorithm is based on the same principles as the previous one, and its code it nearly the same. The only difference is in line 2 where a queue Q is used instead of a test&set object. The queue is initialized to the sequence of items $< w, \ell >$. The process that dequeues w (the value at the head of the queue) is the winner. The process that dequeues ℓ is the loser. The value decided by the consensus object is the value proposed by the winner.

```
operation propose(v) issued by p_i:
(1)  REG[i] ← v;
(2)  aux ← Q.dequeue ();
(3)  case (aux = w) then return (REG[i])
(4)       (aux = ℓ) then return (REG[1 − i])
(5)  endcase
```

Figure 9.5: From queue to consensus

Theorem 30 *The algorithm described in Figure 9.5 is a wait-free construction of a consensus object from queue object, in a system of two processes.*

Proof The proof is the same as the proof of theorem 29. The only difference is the way the winner process is selected. Here, the winner is the process that dequeues the value w that is initially at the head of the queue. As suggested by the text of the algorithm, the proof is then verbatim the same. $\square_{Theorem\ 30}$

9.4.3 Consensus from swap objects

Swap objects A swap object R is an atomic read/write register that has an additional operation denoted $swap()$. That operation has an input parameter, the name of a local variable ($local\_var$) of the process that invokes it. It atomically exchanges the content of the register R with the content of the local variable. The swap operation can be described by the following statements:

operation $swap\ (local\_var)$:
$aux \leftarrow R$;
$R \leftarrow local\_var$;
$local\_var \leftarrow aux$.

From swap objects to consensus An algorithm that wait-free implements a consensus object from a swap object in a system of two processes is described in Figure9.5. That algorithm uses a swap object R, initialized to $\perp$. Its design principles are the same as in the previous algorithms. The winner is the process that succeeds in depositing its index in R while obtaining the value $\perp$ from R. The proof of the algorithm is the same as the proof of the previous algorithms.

operation $propose(v)$ **issued by** p_i:
(1) $REG[i] \leftarrow v$;
(2) $aux \leftarrow i$;
(3) $R.swap(aux)$;
(4) **case** $(aux = \perp)$ **then** *return* $(REG[i])$
(5) $(aux \neq \perp)$ **then** *return* $(REG[1-i])$
(6) **endcase**

Figure 9.6: From swap to consensus

9.4.4 Other objects for consensus in a system of two processes

It is possible to build a wait-free implementation of a consensus object, in a system of two processes, from other objects such as a stack, a set, a list, a priority queue. When they do not provide total operations, the usual definition of of these objects has to be extended in order all the operations be total. As an example, a pop on an empty stack has to be extended to the case where the stack is empty. This can easily be done, by specifying that $pop()$ returns a default value $\perp$ when the stack is empty.

Other objects such as fetch&add objects allow wait-free implementing a consensus object in a system of two processes. Such an object is an atomic object that can be seen as encapsulating an integer state variable x and that can be accessed by the atomic operation $fetch\&add()$. That operation has an input parameter, an integer denoted $incr$. Its behavior can be defined as follows:

operation $fetch\&add\ (incr)$:
$prev\_val \leftarrow x$;
$x \leftarrow x + incr$;

return $(prev\_val)$.

9.4.5 Power and limit of the previous objects

As we have shown, all the objects described previously allow wait-free implementing a consensus object in a system of two processes. Do they allow implementing a consensus object in a system of three or more processes? Surprisingly, The answer to that question is "no". This section gives the proof that the queue objects have consensus number 2. The corresponding proofs for the other objects presented in this section are similar.

Theorem 31 *Atomic wait-free queues have consensus number 2.*

Proof The proof has the same structure as the proof of Theorem 27. Considering binary consensus, it assumes that there is an algorithm based on queues and atomic registers that wait-free implements a consensus object in a system of three processes (denoted p, q and r). As in Theorem 27, we show that starting from an initial bivalent configuration C (due to theorem 26, such a configuration does exist), there is an arbitrarily long schedule S produced by A that leads from C to another bivalent configuration $S(C)$. This shows that A has a run in which no process ever decides, which proves the theorem by contradiction.

Starting the algorithm A in a bivalent configuration C, let S be a maximal schedule produced by A such that the configuration $D = S(C)$ is bivalent. As we have seen in the proof of theorem 27, "maximal" means that the configurations $p(D)$, $q(D)$ and $r(D)$ are monovalent. Moreover, as D is bivalent, two of these configurations have not the same valence. Without loss of generality let us say that $p(D)$ is 0-valent and $q(D)$ is 1-valent; $r(D)$ is either 0-valent or 1-valent (the important point here is that $r(D)$ is not bivalent).

Let OP_p the operation issued by p that leads from D to $p(D)$, OP_q the operation issued by q that leads from D to $q(D)$, and OP_r the operation issued by r that leads from D to $r(D)$. Each of OP_p, OP_q and OP_r, is a read or an atomic register, a write of an atomic register, an enqueue on an atomic queue, or a dequeue on an atomic queue.

Let us consider p and q (the processes that produce configurations with different valences), and let us consider that, from D, r does not execute operations for an arbitrarily long period.

- If both OP_p and OP_q are operations on atomic registers the proof of theorem 27 still applies.
- If one of OP_p and OP_q is an operation on an atomic register, while the other is an operation on an atomic queue, the reasoning used in the item 1 of the proof of theorem 27 applies. This reasoning, based on the argument depicted in Figure 9.2, allows concluding that $p(q(D)) \equiv q(p(D))$, while one is 0-valent and the other is 1-valent.

It follows that the only case that remains to be investigated in when both OP_p and OP_q are operations on the same atomic queue Q. We proceed by a case analysis. There are three cases.

1. Both OP_p and OP_p are $Q.dequeue()$.
   As $p(D)$ and $q(D)$ are 0-valent and 1-valent, respectively, the configuration $\text{OP}_q(\text{OP}_p(D))$ is 0-valent, while $\text{OP}_p(\text{OP}_q(D))$ is 1-valent. But these configurations cannot be distinguished by the process r: in both configurations, r has the same local state and each base object -atomic register or atomic queue- has the same value. So, starting from any of these configurations, let us consider a schedule S' in which only r issues operations (as defined by the algorithm A) until it decides (the fact that neither p nor q executes an operation in this schedule is made possible by asynchrony). We have

then (1) $S'(\text{OP}_q(\text{OP}_p(D))$ is 0-valent, and (2) $S'(\text{OP}_p(\text{OP}_q(D))$ is 1-valent. But, as $\text{OP}_q(\text{OP}_p(D))$ and $\text{OP}_q(\text{OP}_p(D))$ cannot be distinguished by r, it has to decide the same value in both $S'(\text{OP}_q(\text{OP}_p(D))$ and $S'(\text{OP}_p(\text{OP}_q(D))$. A contradiction.

2. OP_p is $Q.dequeue()$, while OP_p is $Q.enqueue(a)$.
(The case where OP_p is $Q.enqueue(a)$ while OP_p is $Q.dequeue()$ is the same.) There are two subcases according to the current state of the wait-free atomic queue Q.

   - Q is not empty. In that case, the configurations $\text{OP}_q(\text{OP}_p(D))$ and $\text{OP}_p(\text{OP}_q(D))$ are identical: in both each object has the same state, and each process is in the same local state. A contradiction because $\text{OP}_q(\text{OP}_p(D))$ is 0-valent, and $\text{OP}_p(\text{OP}_q(D))$ is 1-valent.
   - Q is empty. In that case, r cannot distinguish the configuration $\text{OP}_p(D)$ and $\text{OP}_p(\text{OP}_q(D))$. The same reasoning as the one in item 1 above shows a contradiction (the same schedule S' starting from any of these configurations and involving only operations by r has to decide both 0 and 1).

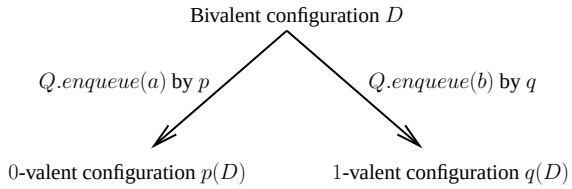

Figure 9.7: $enqueue()$ operations by p and q

3. OP_p is $Q.enqueue(a)$ and OP_p is $Q.enqueue(b)$. (This case is described in Figure 9.7.)
Let k be the number of items in the queue Q in the configuration D. This means that $p(D)$ contains $k+1$ items, and $q(p(D))$ (or $p(q(D))$) contains $k+2$ items (see Figure 9.8).

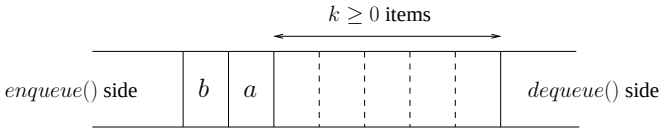

Figure 9.8: State of the queue object Q in configuration $q(p(D))$

As the algorithm A is wait-free and $p(D)$ is 0-valent, there is a schedule S_p, starting from the configuration $q(p(D))$ and involving only operations[1] issued by p, that ends with p deciding 0. We claim that the schedule S_p contains an operation (by p) that dequeues the $k+1$ element of Q. Assume by contradiction that p issues at most k dequeue operations in S_p (and so never dequeues the value a it has enqueued). In that case, if we apply the schedule $p\ S_p$ to the configuration $q(D)$, we obtain the configuration $S_p(p(q(D)))$ in which p decides 0 (p decides 0 because as it dequeues at most k items from Q, it cannot distinguish $p(q(D))$ and $q(p(D))$ (these two configurations differ only in the state of Q: its two last items in $q(p(D))$ are a followed by b, while they are b followed by a in $p(q(D))$), and as we have just seen, p decides 0 in $S_p(q(p(D)))$). But this contradicts the fact that, as $q(D)$ is 1-valent, p should decide 1, which proves the claim.

[1] These operations by p are on the atomic registers and the atomic queues.

It follows from this discussion that S_p contains at least $k+1$ dequeue operations on Q (issued by p). Let S'_p be the longest prefix of S_p that does not contain the $(k+1)$th dequeue operation on Q by p.

As the algorithm A is wait-free and $p(D)$ is 0-valent, there is a schedule S_q, starting from the configuration $S'_p(q(p(D)))$ and involving only operations from q, that ends with q deciding 0. Similarly to the above discussion, we claim that the schedule S_q contains an operation (by q) that dequeues an item from Q. To show it, assume by contradiction, that q never dequeues from Q in S_q. In that case, if we apply the schedule S_q to the configuration $S'_p(p(q(D)))$, q decides 0 (as the only difference between $S'_p(p(q(D)))$ and $S'_p(q(p(D)))$ is in the state of Q), which contradicts the fact that $q(D)$ is 1-valent.

It follows from this discussion that S_q contains at least one operation that dequeues from Q. Let S'_q be the longest prefix of S_q that does not contain that dequeue operation. We now define two schedules that start from D, lead to different decisions, and cannot be distinguished by the third process r (Figure 9.9).

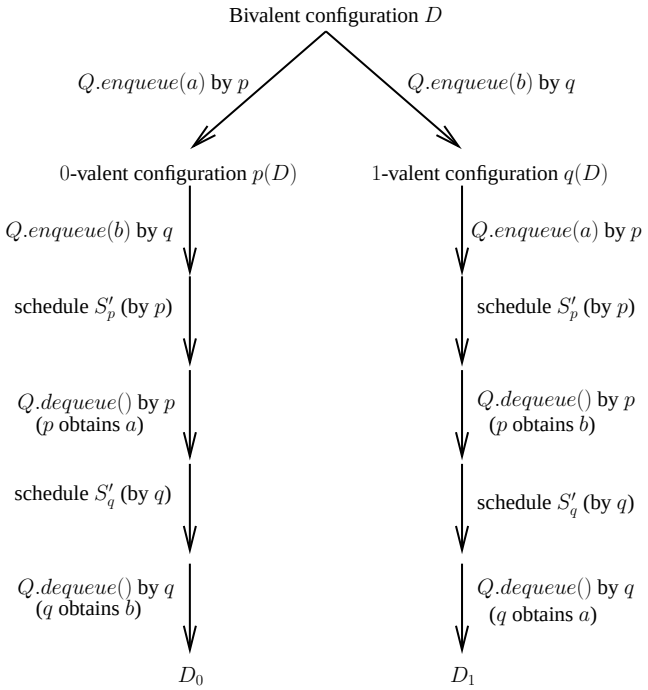

Figure 9.9: From the configuration D to D_0 or D_1

- The first schedule is defined by the following sequence of operations:
  - p executes $Q.enqueue(a)$ (producing $p(D)$),
  - q executes $Q.enqueue(b)$ (producing $q(p(D))$),
  - p executes the operations in S'_p, then executes $Q.dequeue()$ and obtains a,
  - q executes the operations in S'_q, then executes $Q.dequeue()$ and obtains b.

  That schedule leads from the configuration D to the configuration denoted D_0. As D_0 is reachable from $p(D)$, it is 0-valent.
- The second schedule is defined by the following sequence of operations:
  - q executes $Q.enqueue(b)$ (producing $q(D)$),

- p executes $Q.enqueue(a)$ (producing $p(q(D))$),
- p executes the operations in S'_p, then executes $Q.dequeue()$ and obtains b,
- q executes the operations in S'_q, then executes $Q.dequeue()$ and obtains a.

That schedule leads from the configuration D to the configuration denoted D_1. As D_1 is reachable from $q(D)$, it is 1-valent.

Let us now consider the third process r (that has not executed operations since configuration D).

All the objects have the same state in the configurations D_0 and D_1. Moreover, r has also the same state in both configurations. It follows that D_0 and D_1 cannot be distinguished by r (the third process that has executed no operation since the configuration D). Consequently, as the algorithm A is wait-free and D_0 is 0-valent, there is a schedule S_r, starting from the configuration D_0 and involving only operations issued by r in which r decides 0. As r cannot distinguish D_0 and D_1, the very same schedule can be applied from D_1 at the end of which r decides 0. A contradiction, as D_1 is 1-valent.

$\square_{Theorem\ 31}$

The following corollary is an immediate consequence of the theorems 28 and 31.

Corollary 3 *Wait-free atomic objects such as queues, stacks, sets, lists, priority queues, test&set objects, swap objects, fetch&add objects cannot be wait-free implemented from atomic registers.*

9.5 Objects whose consensus number is $+\infty$

This section shows that some atomic objects have an infinite consensus number. They can wait-free implements a consensus object whatever the number n of processes, and consequently can be used to wait-free implement any object defined by a sequential specification on total operations in a system made up of an arbitrary number of precesses. Three such objects are presented here: compare&swap objects, memory to memory swap objects, and augmented queues.

9.5.1 Consensus from compare&swap objects

A compare&swap object CS is an atomic object that can be accessed by a single operation that is denoted $compare\&swap()$. That operation, that returns a value, has two input parameters (two values called old and new). Such an object can be seen as maintaining an internal state variable x. The effect of a $compare\&swap()$ operation can be described by the following specification:

operation $compare\&swap(old, new)$:
  $prev \leftarrow x$;
  if $(x = old)$ **then** $x \leftarrow new$ **endif**;
  return $(prev)$.

From compare&swap objects to consensus The algorithm described in Figure 9.10 is a wait-free construction of a consensus object from a compare&swap object in a system of n processes, for any value of n. The base compare&swap object CS is initialized to $\perp$, a default value that cannot be proposed by the processes to the consensus object. When a process proposes a value v to the consensus object, it first invokes $CS.compare\&swap\ (\perp, v)$ (line 1). If it obtains $\perp$ it decides the value irt proposes (line 2). Otherwise, it decides the value returned from the compare&swap object (line 3).

```
operation propose(v) issued by p_i:
(1)  aux ← CS.compare&swap (⊥, v);
(2)  case aux = ⊥ then return (v)
(3)       aux ≠ ⊥ then return (aux)
(4)  endcase
```

Figure 9.10: From compare&swap to consensus

Theorem 32 *The compare&swap objects have infinite consensus number.*

Proof The algorithm described in Figure 9.10 is clearly wait-free. As the base compare&swap object CS is atomic, there is a first process that executes $CS.compare\&swap\ ()$ (as previously, "first" is defined according to the linearization order of all the invocations $CS.compare\&swap\ ()$). Let that process be the winner. According to the specification of the $compare\&swap\ ()$ operation, the winner has deposited v (the value it proposes to the consensus object) in CS. As the input parameter old of any invocation of the $compare\&swap\ ()$ operation is $\perp$, it follows that all the future $compare\&swap\ ()$ invocations returns the first value deposited in CS, namely the value v deposited by the winner. It follows that all the processes that propose a value and do not crash decide the value of the winner. The algorithm is trivially independent of the number of processes that invoke $CS.compare\&swap\ ()$. It follows that the algorithm wait-free implements a consensus object for any number of processes. $\square_{Theorem\ 32}$

9.5.2 Consensus from mem-to-mem-swap objects

Mem-to-mem-swap objects A mem-to-mem-swap object is an atomic register that provides the processes with three operations. The classical read and write operations plus a a binary *mem-to-mem-swap*() operation that is on two registers, $R1$ and $R2$. *mem-to-mem-swap*$(R1, R2)$ atomically exchanges the content of $R1$ and the content of $R2$. (This operation has not to be confused with the swap operation described in Section 9.4.3. The latter involves two base atomic registers, the former involves a single base atomic register and a local variable.)

From mem-to-mem-swap objects to consensus The algorithm described in Figure 9.11 is a wait-free construction of a consensus object from base atomic registers and mem-to-mem-swap objects, for any number n of processes.

A base 1WMR atomic register $REG[i]$ is associated with each process p_i. That register is used to make public the value it proposes (line 1). A process p_j can read it at line 4 if it has to decide the value proposed by p_i.

There are $n + 1$ mem-to-mem-swap objects. The array $A[1 : n]$ is initialized to $[0, \ldots, 0]$, while the object R is initialized to 1. The object $A[i]$ is written only by p_i, and this write is due to a mem-to-mem-swap operation: p_i exchange the content of $A[i]$ with the content of R (line 2). As we can see, differently from $A[i]$, the mem-to-mem-swap object R can be written by any process. As described in lines 2-4, these objects are used to determine the decided value. After it has exchanged $A[i]$ and R, a process looks for the first entry j of the array A such that $A[j] \neq 0$, and decides the value deposited by the process p_j it has h=just determined the index name.

Before proving the next theorem and to better understand how the algorithm works let us observe that

```
operation propose(v) issued by p_i:
(1)   REG[i] ← v;
(2)   mem-to-mem-swap(A[i], R);
(3)   for j from 1 to n do
(4)         if (A[j] = 1) then return (REG[j]) endif;
(5)   endfor
```

Figure 9.11: From mem-to-mem-swap to consensus

the following relation is invariant:

$$R + \sum_{i=1}^{i=n} A[i] = 1.$$

As initially, $R = 1$, and $A[i] = 0$ for each i, this relation is initially satisfied. Then, due to the fact that the operation *mem-to-mem-swap*$(A[i], R)$ issued at line 2 is executed atomically, it follows that the relation remains true forever.

Lemma 12 *The mem-to-mem-swap object type has consensus number n in a system of n processes.*

Proof The algorithm is trivially wait-free (the loop is bounded). As before, let the winner be the process p_i that sets $A[i]$ to 1 when it executes line 2. As any $A[j]$ is written at most once, we conclude from the previous invariant, that there is a single winner. Moreover, due to the atomicity of the mem-to-mem-swap objects, the winner is the first process that executes line 2. As, before becoming the winner, the winner process p_i has deposited in $REG[i]$ the value v it proposes to the consensus object, we have $REG[i] = v$ and $A[i] = 1$ before the other processes terminate the execution of line 2. It follows that all the processes that decide return the value proposed by the single winner process. $\square_{Lemma\ 12}$

An object type is universal in a system of n processes if it allows wait-free contructing a consensus object for n processes. The following corollary is an immediate consequence of the previous lemma.

Corollary 4 *Mem-to-mem-swap objects are universal in a system of n processes.*

Theorem 33 *The mem-to-mem-swap objects have infinite consensus number.*

Proof The proof follows from the fact that, whatever n, it is always possible to construct a consensus object in a system of n processes from mem-to-mem-swap objects. $\square_{Theorem\ 33}$

9.5.3 Consensus from augmented queue objects

This type of objects is very close to the previous queue we have studied. Interestingly, the augmented queues have infinite consensus umber. This shows that enriching an object with an additional operation can infinitely increase its power when one is interested in the wait-free implementation of consensus objects.

An augmented queue is a queue with an additional operation denoted $peek()$ that returns the first item of the queue without removing it. In some sense, that operation allows reading a part of a queue without modifying it.

The algorithm in Figure 9.12 provides a wait-free construction of a consensus object from an augmented queue. The construction is pretty simple. The augmented queue Q is initially empty. A process first

```
operation propose(v) issued by p_i:
    Q.enqueue(v);
    return (Q.peek())
```

Figure 9.12: From an augmented queue to consensus

enqueues the value v it proposes to the consensus object. Then, it invokes the $peek()$ operation to obtain the first value that has been enqueued. It is easy to see that the construction works for any number of processes, and we have the following theorem:

Theorem 34 *The augmented queue objects have infinite consensus number.*

9.5.4 Impossibility result

Corollary 5 *There is no wait-free implementation of an object of type compare&swap, mem-to-mem-swap or augmented queue from base objects of type stack, queue, set, priority queue, swap, fetch&add or test&set.*

Proof The proof follows directly from the combination of theorem 31 (the cited base objects have consensus number 2), theorems 32, 33 and 34 (compare&swap or mem-to-mem-swap objects have infinite consensus number) and theorem **??** (impossibility to wait-free implement Y from X when $CN(X) < CN(Y)$). $\square_{Corollary\ 5}$

9.6 Hierarchy of atomic objects

9.6.1 From consensus numbers to a hierarchy

Consensus numbers establish a hierarchy on the power of object types to wait-free implement a consensus object, i.e., to wait-free implement any object defined by a sequential specification on total operations. More generally:

- Consensus numbers allow ranking the power of classical synchronization primitives (provided by shared memory parallel machines) in presence of process crashes: compare&swap is stronger than tes&set that is in turn stronger than atomic read/write operations. Interestingly, they also show that classical objects encountered in sequential computing such as stacks and queues are as powerful as the test&set or fetch&add synchronization primitives when one is interested in providing upper layer application processes with wait-free objects.
- Fault masking can be impossible to achieve when the designer is not provided with powerful enough atomic synchronization operations. As an example, a first in/first out queue that has to tolerate the crash of a single process, can not be built from atomic registers. This follows from the fact that the consensus number of a queue is 2, while the he consensus number of atomic registers is 1.

9.6.2 Robustness of the hierarchy

Let us remind the definition of consensus number, stated at the beginning of this chapter: the *consensus number* associated with an object type T is the largest number n such that it is possible to wait-free implement, in a system of n processes, a consensus object from atomic registers and objects of type T.

The previous object hierarchy is *robust* in the following sense. Any set of object types with consensus numbers equal to or smaller than k cannot wait-free implement an object whose consensus number is at a higher level of the hierarchy, i.e., an object whose consensus number is greater than k. The hierarchy would no longer be robust if the definition of the consensus number notion prevented the use of base atomic registers.

Bibliographic notes

Herlihy 1991

FLP 85; Loui-Abu Amara, Anderson-Gouda, etc (voir dans H91)

Attiya-Welch 98, Lynch 96

Chandra-Jayanti-Toueg JACM 98

Part IV

Memory with Oracles

Bibliography

[1] Yehuda Afek, Eytan Weisberger and Hanan Weisman. A Completeness Theorem for a Class of Synchronization Objects (Extended Abstract). *PODC*, 1993, 159-170.

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

[3] 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.

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

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

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

[7] 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.

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

[9] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

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

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

[41] 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.

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

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

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

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

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

[47] 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.

[48] 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.

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

[50] 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.

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

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

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