Constructing a Reliable Test&Set Bit

Frank Stomp and Gadi Taubenfeld

Abstract—The problem of computing with faulty shared bits is addressed. The focus is on constructing a reliable test&set bit from a collection of test&set bits of which some may be faulty. Faults are modeled by allowing operations on the faulty bits to return a special distinguished value, signaling that the operation *may not* have taken place. Such faults are called *omission faults*. Some of the constructions are required to be *gracefully degrading* for omission. That is, if the bound on the number of component bits which fail is exceeded, the constructed bit may suffer faults, but only faults which are no more severe than those of the components; and the constructed bit behaves as intended if the number of component bits which fail does not exceed that bound. Several efficient constructions are presented, and bounds on the space required are given. Our constructions for omission faults also apply to other fault models.

Index Terms—Test&set bits, reliability, omission faults, gracefully degradation, wait-free algorithms.

1 Introduction

A concurrent system may be viewed as a collection of processes which communicate through shared objects. These shared objects allow different processes to exchange information. For example, an *atomic register* is a shared object which, in one atomic step, a process can either read or write (but not both). Other (stronger) shared objects are test&sets, semaphores, and data structures such as queues or stacks.

Much work has been carried out on enabling a concurrent system as a whole to continue to function despite the failure of a limited number of processes. Making such a system fault-tolerant depends also on constructing reliable shared objects. The fault of a shared object may be caused by various reasons: data contained in the object is corrupted, requests are lost due to switching failures, an algorithm enters an unintended infinite loop, or an algorithm mistakenly allows a process to affect memory not assigned to it.

Two recent papers [1], [6] have studied how to tolerate faults in shared objects. The first explores the possibility that faulty shared objects may return arbitrary values. The second, independent, paper explores a wider range of fault models. Among these models are the *arbitrary* fault model as in [1], and two others: the *crash* and *omission* fault models.

In the crash fault model, a faulty object behaves correctly until the object suffers a terminal, atomic crash event. Thereafter, all operations on that object return "$\perp$". In the omission fault model, which is of particular interest to this paper, operations on faulty objects may return a special value "?", which indicates only that the operation has terminated and that the operation has either affected the state of the object as intended or completely failed to affect it.

A constructed object is said to tolerate t faults of model $\mathcal{F}$ if the object does not exhibit any faulty behavior when at most t of the underlying primitive objects, from which it is constructed, fail by $\mathcal{F}$. If, in addition, more than t primitive objects fail by $\mathcal{F}$ and the constructed object exhibits faults of model $\mathcal{F}$ only, then the constructed object is called *gracefully degrading* for $\mathcal{F}$ [6]. This implies that if too many primitive objects fail, the constructed object is either correct or it does not exhibit more severe faults than those of the components in every execution.

Gracefully degrading constructions may be used as modules in larger constructions. The property that a construction can be carried out in a modular fashion is important because without this property no fault-tolerant interfaces or abstraction layers can be constructed. The same property ensures that correctness proofs of the system in case of faults can be carried out on the basis of its constituent components. Our results imply that, for omission faults, the requirement of graceful degradation increases the space complexity of the implementation.

1.1 Summary of Results

The subject of this paper is the construction of (reliable) *test&set* bits which can tolerate omission faults. The constructions of test&set bits which are presented apply also to the crash model introduced above and to the omission-crash and the eventual-crash models introduced in [1] and discussed later in this section.

The specification of a test&set bit can be described by means of a single bit, initially **0**, that can be accessed by processes through an operation called **test&set**: In one atomic step, the value **1** is assigned to the bit and the bit's old value is returned.

When it is assumed that test&set bits are accessed by at most two processes, we prove the following tight bounds:

- Two test&set bits are necessary and sufficient for constructing a test&set bit which can tolerate one omission fault.
- Three test&set bits are necessary and sufficient for constructing a test&set bit which can tolerate one

- F. Stomp is with the Department of Computer Science, Wayne State University, Detroit, MI 48202. E-mail: fstomp@cs.wayne.edu.
- G. Taubenfeld is with The Open University, 16 Klausner St., Tel-Aviv 61392, Israel. E-mail: gadi@cs.openu.ac.il.

Manuscript received 30 Oct. 1995.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100027.

omission fault and which is gracefully degrading for omission.

When it is assumed that test&set bits are accessed by n processes, we prove the following upper bounds:

- $n+2$ test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault.
- $2n+3$ test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault and which is gracefully degrading for omission.

When it is assumed that test&set bits and atomic bits are accessed by n processes, we prove:

- One atomic bit and two test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault.
- One atomic bit and three test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault and which is gracefully degrading for omission.

When it is assumed that test&set bits are accessed by n processes and that only test&set bits are used, we prove the following lower bound:

- n test&set bits are necessary for constructing a test&set bit which can tolerate one omission fault and which is gracefully degrading for omission.

In [1], [6], it is explained how to construct objects which tolerate many faults from objects which tolerate one fault. Using the ideas from [1], [6], all our constructions can be extended to tolerate multiple faults. In [1], a construction is given to convert a test&set bit without a reset operation (and which can tolerate a single *arbitrary* fault) into one which supports a reset. In essence, a similar construction applies to the bits constructed in the present paper. Although the above techniques ignore the issue of efficiency, this explains why we have decided to concentrate here on constructing test&set bits without reset which can tolerate one omission fault.

1.2 Related Work

As already pointed out, research on constructing reliable shared objects is reported in [1], [6]. In both these papers, a theory for combining algorithmic constructions in a modular way is developed, and various lower bounds and constructions of reliable objects from faulty components, including universality results, are presented. In particular, there are some constructions in [1] of reliable test&set bits for *arbitrary* faults. These constructions are not gracefully degrading for omission, however. The existence of reliable test&set bits which are gracefully degrading for omission follows from results in [6], but no such explicit constructions are given.

Two other fault models are introduced in [3]: *omission-crash* and *eventual-crash*. Both kinds of faults are more severe than crash faults, but no more severe than omission faults. In the omission-crash model, faulty objects may suffer a terminal crash event $\perp$ as in the crash model. However, operations *concurrent* with the crash may suffer omission faults, returning ?. Subsequent operations will return $\perp$. In the eventual-crash model, faulty objects may also suffer a terminal crash event. However, any operation prior to or concurrent with the crash event may suffer omission faults, returning ? before the terminal crash occurs. All subsequent operations will return $\perp$.

Some other research has explored memory faults which are restricted to occur during specific periods of time. For example, such constrained memory faults are studied in work on *self-stabilizing* systems [4]. Self-stabilizing systems are required to recover once the final memory fault has occurred and the system is in an arbitrary state. There is an immense body of work on self-stabilization as an approach for designing fault-tolerant systems. For a survey on this topic, see [12]. Initialization faults have been investigated in [5]. A shared register is subject to initialization fault if the shared register contains an arbitrary unknown value in the initial state.

2 Preliminaries

A concurrent system may be viewed as a collection of sequential processes communicating through *shared objects*. Each object $\mathcal{O}$ has a type $\langle D, Op\rangle$, where D is the set of all values which $\mathcal{O}$ can take and where Op is the set of all operations to access and manipulate $\mathcal{O}$.

Every system supports a number of *primitive objects* (or object types) which can be used to construct more complicated objects. Let $\mathcal{O}_1, \cdots, \mathcal{O}_n$ be a list of primitive objects. A *construction of object* $\mathcal{O}$ *from* $\mathcal{O}_1, \cdots, \mathcal{O}_n$ is a function whose image is $\mathcal{O}$ under $\mathcal{O}_1, \cdots, \mathcal{O}_n$. Hereafter, $\mathcal{O}$ will often be referred to as an object constructed or derived from $\mathcal{O}_1, \cdots, \mathcal{O}_n$. A *sequential specification* of an object is a description of its behavior when processes access the object sequentially.

We focus on an object called a *test&set* bit. Its sequential specification can be described by means of a single binary bit, initially **0**, which can be accessed by processes through an operation called **test&set**: In one atomic step value **1** is assigned to the bit and the bit's old value is returned. The bit is a "single_use" type of bit, because, as defined here, it does not support a reset operation.

Such a simple specification does not suffice in case of a concurrent system in which several processes may access a bit concurrently. One concept used most often to characterize such concurrent behavior is that of *linearizability* [9]. Informally, linearizability means that every concurrent behavior is *equivalent*, in a sense discussed below, to a behavior in which processes access the bit sequentially.

An *execution* of a concurrent system is a finite sequence of operation *invocations* and *responses*. An invocation of operation *op* by process p on object *obj* is represented by a triple $\langle obj, op, p\rangle$. A response of object *obj* to process p's invocation of operation *op* is represented by a triple $\langle obj, op, res, p\rangle$, where res is the result returned. A response *matches* an invocation if their object, operation, and process components are the same. A response is called *correct* if its result is allowed by the object's type. An invocation is *pending* in an execution if there is no matching response following that invocation in the execution. An execution is called *sequential* if every invocation, except possibly the last one, is immediately followed by a (correct) matching response and every response is immediately preceded by an invocation. An execution is *complete* if there are no pending invocations. A specification is sequential if it only constrains sequential executions; otherwise, it is concurrent. Every execution α is assumed to be *well-formed* [9], that is, for every process p, the subsequence of all events in α whose process component is p is sequential. Notice that well-formedness does not imply sequentiality, and that sequentiality implies well-formedness.

Every execution α defines a partial order $\preceq_\alpha$ on (the occurrences of) the invocations and responses in the execution: $a \preceq_\alpha b$ holds iff 1) either a and b are invocations by or responses to the same process and a occurs before b in α, or 2) a is a response which precedes invocation b in α. A (concurrent) execution α of a system is called linearizable if the following is true: There exists a complete execution α' of the system extending α by appending responses for incomplete invocations in α and there exists a sequential execution β of that system such that $\preceq_{\alpha'}$ and $\preceq_\beta$ are the same.

Faulty objects can be modeled by allowing incorrect responses to invocations to that object. In the present paper we concentrate on *omission faults*, as was first introduced in [6]. An object *fails by omission in an execution α* if it may return a distinguished response ? in α, and

1. Every response from the object is either ? or one allowed by its type;
2. For every response ? in α, replacing ? by one allowed by the object's type or omitting both the response and the corresponding invocation results in a linearizable execution.

An object may *fail by omission* if, in every execution, the object may fail only by omission.

A construction can tolerate t omission faults if it does not exhibit any faulty behavior when at most t of the underlying primitive objects fail by omission. If, in addition, more than t primitive objects fail by omission and the constructed object may fail only by omission, then it is called *gracefully degrading* for omission [6]. This means that, if too many primitive objects fail by omission, the constructed object is either correct or it does not exhibit more severe faults than omission faults in every execution. An object is called *strongly wait-free* [1] if every invocation on the object by any process will result in an eventual response, irrespectively of the status of other processes and objects. In this paper, all primitive objects are assumed to be strongly wait-free and all constructed objects are required to be strongly wait-free.

We have the following simple, but useful, property whose proof is immediate.

Lemma 1. *An object is a test&set bit if, for every nonempty execution of that object without pending requests,*

1. *every process which participates returns either **0** or **1**,*
2. *exactly one process returns **0**,*
3. *if a process returns **1**, then every process which starts strictly thereafter returns **1**.*

We end this section with the following observations:

- A test&set bit which fails by omission may respond by ? and then respond correctly thereafter. Thus, after response ?, the responses are not necessarily all ?. A test&set bit which has responded by ? (at least once) is considered to be faulty.
- If a test&set bit first responds by ?, then the second response may be either **0** or **1**.
- For a constructed test&set bit which is gracefully degrading for omission, the above observations are also true.

3 Tight Bounds for Two Processes

In this section, we construct test&set bits which are accessed by at most two processes and assume that only test&set bits are allowed to be used in the constructions. The two main results proven are:

- Two test&set bits are necessary and sufficient for constructing a test&set bit which can tolerate one omission fault.
- Three test&set bits are necessary and sufficient for constructing a test&set bit which can tolerate one omission fault and which is gracefully degrading for omission.

These results imply that, for omission faults, the requirement of graceful degradation increases the space complexity of the implementation.

3.1 An Optimal Nongracefully Degrading Construction

We have the following obvious lower bound:

Theorem 1. *Two test&set bits are necessary for constructing a test&set bit which can tolerate one omission fault.*

Next, we show that the above lower bound is tight. Notice that, in the theorem below, the construction is not required to be gracefully degrading.

Theorem 2. *Two test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault.*

Proof. Consider the construction in Fig. 1, whose code is given in Fig. 2. It consists of two shared primitive test&set bits A, B. In Fig. 1, gray nodes labeled $s0$ and $s1$ denote states. Invocations of test&set operations are implicit. Edges out of state $s0$ are labeled by responses to the test&set operation on A; edges out of the state $s1$ are labeled by responses to test&set on B; white nodes with some value in it represent the value returned by a process.

If a process accesses bit A and A responds with **1**, then the process immediately returns **1**. (The other process has already accessed A.) If A responds with **0** or ?, then the process cannot determine whether it is the first to access A. It then accesses bit B. The process returns **1** if B responds with **1**. Otherwise, the process returns **0**. The proof of the theorem follows from Lemma 2 given next, and the observation that the construction in Fig. 1 is strongly wait-free. □

Lemma 2. *The bit in Fig. 1, whose code is given in Fig. 2, is a test&set bit which can tolerate one omission fault.*

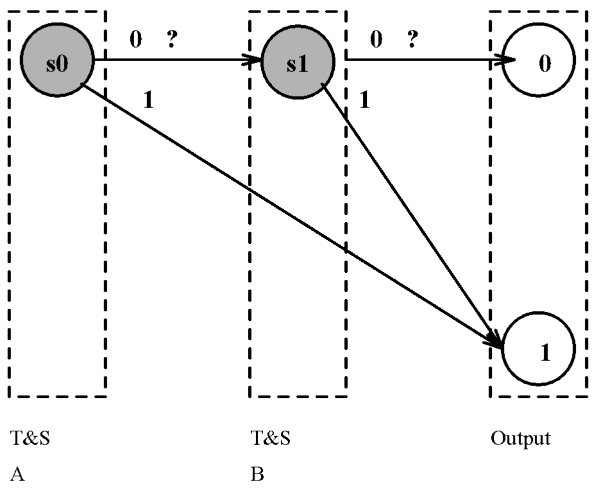

Fig. 1. Schematic of a simple construction of a test&set bit for two processes which can tolerate one omission fault. The construction is not gracefully degrading for omission.

Proof. Assume that at most one primitive bit of bit $\mathcal{T}$ constructed in Fig. 1 is faulty. Consider an arbitrary nonempty, complete execution α of $\mathcal{T}$. The lemma follows from conditions 1, 2, 3 below, and Lemma 1.

1. In α, every process returns either **0** or **1**.

   This is obvious.
2. In α, exactly one process returns **0**.

   Recall that in α, every process returns **0** or **1**. We first show that at least one process returns **0** in α.

   At least one of the processes receives **0** or ? from bit A. Consequently, at least one of the processes accesses B. The first process to do so receives either **0** or ? from bit B. This process returns **0**.

   To complete the proof of condition 2, it suffices to show the following: If both processes access bit $\mathcal{T}$, then at least one process returns **1**. Suppose that this is not true. Then, both processes return **0**. It follows that each of these processes accesses both bits A and B and that each of these processes receives **0** or ? as responses from A and B. This implies that either both bits A and B are faulty or that one of the bits A, B responds twice with **0**. The first possibility contradicts the assumption that at most one primitive bit is faulty; the second possibility contradicts the specification of a test&set bit (it can respond at most once with **0**).
3. In α, if a process returns **1**, then every process which starts strictly thereafter returns **1**.

   Assume, to obtain a contradiction, that one process P has returned **1** and that, strictly thereafter, another process Q starts which also returns **1**. Since only two processes access bit $\mathcal{T}$, we obtain that none of these processes returns **0**, contradicting condition 2 above. □

3.2 An Optimal Gracefully Degrading Construction

The previous construction is not gracefully degrading. For example, if one process receives responses **0** from A and B, and the other process receives responses ? from A and B, then both processes will return **0**. In this case, both primitive bits exhibit a fault and there does not exist an equivalent sequential execution, because of the two **0**s returned.

Theorem 3. *Three test&set bits are necessary for constructing a test&set bit which can tolerate one omission fault and which is gracefully degrading for omission.*

Proof. Suppose, to obtain a contradiction, that a construction exists with only two primitive bits A and B. (In view of Theorem 1, a construction consists of at least two primitive bits.) Assume that A always responds with ? and does not undergo a state-change when accessed by process P and assume that B always responds correctly and undergoes an appropriate state-change when accessed by process P. Analogously, assume that A always responds correctly and undergoes an appropriate state-change when accessed by process Q and that B always responds ? and does not undergo a state-change when accessed by process Q. Then, consider scenario S1 in which only P accesses the constructed bit. Upon completion, P returns **0**. And consider scenario S2 in

```
shared $A$, $B$: primitive test&set bits, initially 0
local $w[1..2]$: local on {0,1,?}

function 2-process-test&set
1 $w[1]$:= test&set($A$);
2 if $w[1]=1$ then return 1 fi;
3 $w[2]$:= test&set($B$);
4 if $w[2]=1$ then return 1 else return 0 fi
end_function
```

Fig. 2. A construction of a test&set bit for two processes which can tolerate one omission fault. The construction is not gracefully degrading for omission.

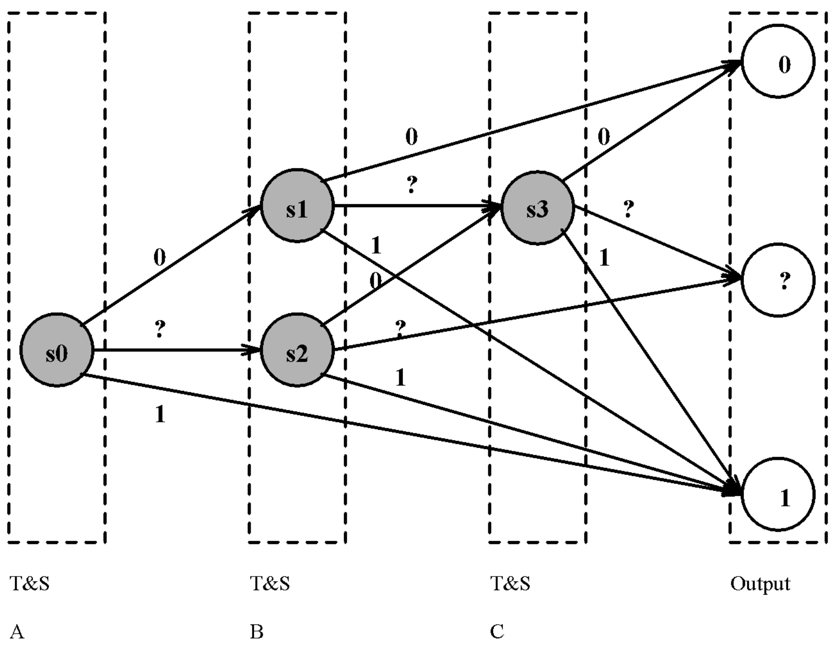

Fig. 3. Schematic of a gracefully degrading construction of a test&set bit for two processes which can tolerate one omission fault.

which only Q accesses the constructed bit. Upon completion, Q returns **0**. Define S3 as the scenario in which first P and thereafter Q accesses the constructed bit. Process P cannot distinguish between the scenarios S1 and S3. Thus, in S3, P returns **0**. Similarly, in S3, Q returns **0**, a contradiction. □

As shown, the construction in the previous section is not gracefully degrading. Now, if a process has received two responses ?, then it is safe that this process returns ?. We modify the construction in the previous subsection by adding a third node $s2$ which represents the state after bit A responds with ? to a process (instead of node $s1$. The response received from bit B in that state is then the value returned by that process. This ensures that if a process receives two ?s, it will return ?, and that a process returns the same values as in the previous construction otherwise. Another problem arises in the construction in the previous subsection when one process receives ? from one bit and the other process receives ? from the other bit and both processes return **0**. This occurs if each of the processes has received **0** and ? as responses.

We proceed as follows: Rather than immediately returning a value when a process has received the responses **0** and ?, we make the process access a third test&set bit C. The response from C is the value returned by that process. Since at most one process can receive **0** as response from C, it follows that at most one process will return **0** in this case. The same is also true for response **1**. Notice that if a process receives ? from C, then that process has "seen" two faulty bits and it is safe for the process to return ?. The gracefully degrading construction is given in Fig. 3 and its code is given in Fig. 4. It is obtained from Fig. 1 by first adding gray node $s2$ to represent the state after bit A has responded with ? and, then, replacing the white node with value 0 in it by state $s3$. In state $s2$, a process can access bit B and, in state $s3$, a process can access bit C. In Fig. 3, edges out of state $s2$ are labeled by responses to the test&set operation on bit B and edges out of state $s3$ are labeled by responses to the test&set operation on bit C.

Theorem 4. *Three test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault and which is gracefully degrading for omission.*

Proof. Consider the construction in Fig. 3. Suppose that, in an execution, no process reaches state $s3$. If a process has received two ?s, then it will return ?. Otherwise, that is, if the process has received less than two ?s, we can simulate this execution on the construction in Fig. 1, which shows that the processes return admissible values.

Suppose that, in an execution, exactly one process reaches state $s3$. Then, we can simulate the part of the execution up to the process reaching state $s3$ on the construction in Fig. 1. That process would return **0** there. The discussion before the formulation of this theorem implies that another process would then return a value different from **0** in Fig. 3. The proof then follows from the observation that if exactly one process reaches state $s3$ in Fig. 3, it will return either **0** or ?.

Now, suppose that, in some execution, two processes reach state $s3$. Then, both bits A and B are faulty. Bit C will never return **0** to both processes, nor will it return **1** to both processes. This is so because if C is faulty, then it fails by omission. This completes the proof of this case.□

4 A Nongracefully Degrading Construction for n Processes

We now focus on constructing a test&set bit for n processes ($n \geq 1$), which can tolerate one omission fault. As before, it is assumed that only test&set bits are used. (In Section 7, the

```
shared $A$, $B$, $C$: primitive test&set bits, initially 0
local $w[1..2]$: local on {0,1,?}

function gracefully-degrading-2-process-test&set
1 $w[1]$:= test&set($A$);
2 if $w[1]=1$ then return 1 fi;
3 $w[2]$:= test&set($B$);
4 if $w[1]=?$ then if $w[2] \neq 0$ then return $w[2]$ else return test&set($C$) fi
5                else if $w[2] \neq ?$ then return $w[2]$ else return test&set($C$) fi
6 fi
end_function
```

Fig. 4. A gracefully degrading construction for omission of a test&set bit for two processes which can tolerate one omission fault.

case that also *atomic* bits are used is considered.) We prove the following upper bound:

- $n+3$ test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault.

In Section 8, we show how this bound can be improved by one test&set bit.

Let us reconsider the constructions in Section 3 and analyze their behavior in case more than two processes access those constructions. The scenario below shows that the property "If one process returns **1**, then every process which starts strictly thereafter returns **1**" is not satisfied for the construction of Fig. 2. Consider the case that only bit A is faulty. Take three processes P_1, P_2, and P_3. Consider the scenario described by:

1. Process P_1 accesses bit A which responds with **0**.
2. Process P_2 accesses bit A which responds with **1**. Hence, process P_2 returns a **1**.
3. Process P_3 accesses bit A which responds with ?. Thereafter, P_3 accesses bit B which responds with **0**. Hence, process P_3 returns a **0**.

A similar scenario as the one described above can be used to show that the bit constructed in Fig. 4 in Section 3.2 also does not satisfy the above mentioned property. The two other properties formulated in Lemma 1 do hold. This observation is used to obtain gracefully degrading constructions from the earlier constructions. Our result for the case of n processes is:

Theorem 5. *$n+3$ test&set bits are sufficient for constructing a test&set bit for n processes which can tolerate one omission fault.*

To prove the theorem, we construct a test&set bit which consists of $n+3$ test&set bits and which can tolerate one omission fault. The construction uses a so-called *gate*.

Intuitively, a process which accesses a gate can either pass the gate, indicating that the process returns **0** or it does not pass the gate, indicating that it returns **1**. A gate satisfies the following two conditions, provided that at most one of its primitive bits is faulty by omission:

1. Every process which accesses the gate either passes the gate or does not pass the gate. And, if at least one process accesses the gate, then at least one process will pass the gate.
2. If some process has returned a value (i.e., it has passed the gate or not passed the gate), then every process which starts strictly thereafter does not pass the gate.

Notice that, for the gate, unlike for the test&set bit, more than one process may return **0**. For the above specification, the construction in Fig. 5 is a test&set bit for n processes which can tolerate one omission fault. If a process passes the gate, it accesses the bit presented in Fig. 1; if a process does not pass the gate, it returns **1**.

Lemma 3. *Consider bit $\mathcal{T}$ in Fig. 5. Assume that the gate can tolerate one fault. Then, $\mathcal{T}$ is a test&set bit for n processes which can tolerate one omission fault.*

Proof. Assume that bit $\mathcal{T}$ in Fig. 5 may be accessed by n processes. Also assume that the gate can tolerate one fault. Let α be an arbitrary nonempty, complete execution of $\mathcal{T}$ in which at most one primitive bit exhibits a fault. The lemma follows from condition 1, 2, 3 below, and Lemma 1.

1. In α, every process returns either **0** or **1**.

   If a process does not pass the gate, then it returns **1**. (The gate can tolerate one fault.) If, on the other hand, a process passes the gate, then it accesses the bit in Fig. 1. Such a process cannot return a ? because, by assumption, at most one bit is faulty. It follows that every process which accesses $\mathcal{T}$ returns **0** or **1**.
2. In α, exactly one process returns **0**.

   First, we show that at most one process returns **0**. A process P_i, $1 \leq i \leq n$, returns **0** when it has passed the gate, and it executes line 4 of the code in Fig. 2. It is clear that at most one process returns **0** if bit B is not faulty. Let us therefore assume that bit B is faulty. Since B is faulty, the assumption of the lemma implies that A is not faulty. Consequently, the second process to access A receives **1** as response. Then, that process will return **1**. It follows that at most one process will return **0**.

```
function multiple-process-test&set
if passed-gate(gate)=0 then call 2-process-test&set else return 1 fi
end_function
```

Fig. 5. A construction of a test&set bit for n processes which can tolerate one omission fault. Fuction **passed-gate** (see Fig. 6) returns **0** if the process passes the gate and **1** if it does not pass the gate. Function **2-process-test&set** is from Fig. 2. The construction is not gracefully degrading for omission.

Next, we show that at least one process will return **0**. To do so, observe that at least one of the processes which accesses $\mathcal{T}$ passes the gate. We distinguish two cases:

a. Bit A is not faulty.

   In this case, exactly one process will access bit B. This process will return **0**.

b. Bit A is faulty.

   In this case, bit A will respond with **0** or **?** to at least one process. At least one of the processes will therefore access B. The first one to do so will return **0**. (Notice that, by assumption, bit B is not faulty.)

3. In α, if a process returns **1**, then every process which starts strictly thereafter returns **1**.

   Assume, to obtain a contradiction, that this is not true. Thus, some process P_i returns **1** and strictly thereafter another process P_j starts which returns **0**. This implies that process P_j has passed the gate strictly after process P_i has returned **1**. We consider two cases:

a. Process P_i has not passed the gate.

   We immediately obtain a contradiction from the gate's specification. (If a process does not pass the gate, then no process which has started strictly thereafter can pass the gate.)

b. Process P_i has passed the gate.

   It follows that strictly after P_i has passed the gate, process P_j has passed the gate. Again, this contradicts the gate's specification. □

To complete the proof of the Theorem 5, it remains to find a construction for the gate which can tolerate one fault and which consists of $n+1$ primitive test&set bits. Such a construction is given in Fig. 6. To access the gate, a process accesses all the gate's primitive bits. The process returns **1** if at least n of the bits have responded with **1** to that process; otherwise, the process returns **0**.

Our construction of the gate satisfies its specification. This is the subject of the following lemma, formulated in terms of the code of the gate:

Lemma 4. *Object $\mathcal{O}$, with code as in Fig. 6, is a gate which can tolerate one omission fault.*

Proof. Assume that at most one of the primitive test&set bits of the constructed object $\mathcal{O}$ is faulty by omission. Let α be an arbitrary nonempty, complete execution of $\mathcal{O}$. The lemma follows from conditions 1 and 2 below and the specification of the gate.

1. Every process which accesses the gate either passes the gate or does not pass the gate; and if at least one process accesses the gate, then at least one process will pass the gate.

   The first part follows from the code. We therefore concentrate on the second part. At most n processes will access object $\mathcal{O}$ and at least one process will access $\mathcal{O}$ (because α is nonempty). Since there exist $n+1$ primitive bits accessed by all these processes, at least one of these processes obtains non-**1** responses from (at least) two bits. Thus, at least one of these processes will return **0**.

2. If some process has passed the gate or not passed the gate, then every process which starts strictly thereafter does not pass the gate.

   Assume that one process P_i has passed the gate or that it has not passed the gate. Assume also that strictly thereafter another process P_j starts its execution. Since P_i has accessed all the primitive bits, and since at most one primitive bit is faulty, it follows that process P_j obtains at least n **1**s as responses from the primitive bits. Thus, process P_j will return **1**. □

Notice that our construction of the test&set bit in Fig. 5 can tolerate one omission fault in the gate and one omission fault in the other part of the implementation. The construction is not gracefully degrading because if all the gate's constituent bits respond with **?**, then every process which accesses the gate will return **0**.

5 A Gracefully Degrading Construction for n processes

In the gracefully degrading case, our result for n processes is:

Theorem 6. $2n+4$ *test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault, and which is gracefully degrading for omission.*

In Section 8, we show how this bound can be improved by one test&set bit.

To extend the construction of Section 3.2 to a gracefully degrading one, we again use a gate; however, this time we construct a gate which is gracefully degrading. The gate will consist of $2n+1$ test&set bits and it can tolerate one omission fault.

In order for the gate to be gracefully degrading, we ensure, in addition to conditions 1 and 2 formulated in Section 4, that the following conditions are also satisfied when at least two of its primitive bits are faulty by omission:

```
shared $A[1..n+1]$: array of primitive test&set bits, all entries initially 0
local $count$: local on $\{0, 1, \cdots, n+1\}$, initially 0

function passed-gate
1 for $i$:= 1 to $n$+1 do if test&set($A[i]$)=1 then $count$:= $count$ +1 fi od;
2 if $count \geq n$ then return 1 else return 0 fi
end_function
```

Fig. 6. A construction of a gate which can tolerate one omission fault from $n+1$ test&set bits.

3. Every process which accesses the gate either passes the gate, does not pass the gate, or returns ?; and if at least one process accesses the gate, then at least one process passes the gate or at least one process returns ?.
4. If some process has accessed the gate and returned the value **0** or **1**, then every process which starts strictly thereafter returns **1** or ?.

Condition 3 guarantees that if one process has passed the gate, we may pretend that all other processes which have returned ? do not pass the gate and that if no process has passed the gate, we may pretend that one process which has returned ? does pass the gate. Condition 4 guarantees that if some process has returned **0** or **1**, then every process which starts strictly thereafter may be pretended to return **1** (not to pass the gate). Note that if some process returns ?, then another process which starts strictly thereafter may return **0**.

In Fig. 7, we have combined such a gate with the construction of Section 3.2. This combination is a test&set bit for n processes which can tolerate one omission fault and which is gracefully degrading.

Lemma 5. *Consider bit $\mathcal{T}$ in Fig. 7. Assume that the gate can tolerate one fault and is gracefully degrading. Then, $\mathcal{T}$ is a test&set bit for n processes which can tolerate one omission fault and which is gracefully degrading for omission.*

Proof. Take an arbitrary nonempty, complete execution α of bit $\mathcal{T}$. In case at most one primitive bit of $\mathcal{T}$ exhibits a fault in α, the proof is similar to the one of Lemma 3. Assume, therefore, that at least two primitive bits exhibit a fault in α.

1. In α, every process returns **0**, **1**, or ?.
   This is obvious.
2. In α, at least one process returns **0** or ?.
   Since the gate is gracefully degrading, at least one process passes the gate or returns ?. In the latter case, condition 2 holds. Assume, therefore, that at least one of the processes has passed the gate. A simple argument, similar to the one applied in Lemma 3, shows that of all the processes which have passed the gate, at least one will return **0** or ?.
3. In α, if some process has accessed $\mathcal{T}$ and returned a value, then every process which starts strictly thereafter returns **1** or ?.

Let P_i be a process which has accessed $\mathcal{T}$ and returned a value; and let P_j, $i \neq j$, be a process which starts strictly thereafter. We consider two cases:

a. Process P_i did not pass the gate.
   Because of the specification of the gate, every process which starts thereafter either returns **1** or ?.
b. Process P_i did pass the gate.
   The specification of the gate again implies that no process started strictly thereafter can pass the gate. Consequently, P_j returns **1** or ?. □

To establish Theorem 6, it remains to construct a gate consisting of $2n+1$ test&set bits which can tolerate one omission fault and which is gracefully degrading for omission. The code of such a construction is given in Fig. 8. A process accessing the gate performs a test&set-operation on all the $2n+1$ shared test&set bits. It returns ? if at least two bits have responded with ?. It returns **1** if at most one bit has responded with **0** to it. In all other cases, i.e., if the process has received at most one ? and at least two **0**s from all the bits, it returns **0**.

Lemma 6. *Object $\mathcal{O}$ whose code is given in Fig. 8 is a gate which can tolerate one omission fault and which is gracefully degrading for omission.*

Proof. Let α be a nonempty, complete execution of object $\mathcal{O}$. The proof follows from conditions 1, 2, 3, and 4 below.

1. Assume that in α at most one primitive bit fails by omission. Then, every process which accesses the gate either passes the gate or does not pass the gate. And at least one process will pass the gate.
   One process will receive at least two **0**s when accessing the shared bits. (This is so because there is at most one faulty bit and $2n+1$ bits and at most n processes.) This process will return **0**. It is clear that every process which accesses $\mathcal{O}$ either passes the gate or does not pass the gate because no process can return ?. (By assumption, at most one bit exhibits a fault in α.)
2. Assume that in α at most one primitive bit fails by omission. Then, the following holds: If some process has passed the gate or not passed the gate, then every process which starts strictly thereafter does not pass the gate.

```
local $w$: local on $\{0, 1, ?\}$

function multiple-process-gracefully-degrading-test&set
$w$:= passed-gracefully-degrading-gate(gate);
if $w$=0 then call gracefully-degrading-2-process-test&set
        else if $w$=1 then return 1 else return ? fi fi
end_function
```

Fig. 7. A construction of a test&set bit for n processes which can tolerate one omission fault and which is gracefully degrading. Function **passed-gracefully-degrading gate** (see Fig. 8) returns **0** if the process passes the gate, **1** if it does not pass the gate, and ? otherwise.

Assume that process P_i has returned **0** or **1** and that another process P_j starts strictly thereafter. Process P_i has accessed all the $2n + 1$ shared bits. Since there is at most one fault, it follows that at least $2n$ of the bits will return a **1** to process P_j. Thus, process P_j can receive at most one **0** when it accesses the $2n + 1$ bits. We conclude that P_j will return **1**.

3. Assume that in α at least two bits fail by omission. Then, the following holds: Every process which accesses the gate either passes the gate or does not pass the gate, or returns ?; and if at least one process accesses the gate, then at least one process enters the gate or at least one process returns ?.

   The first part of condition 3 should be clear. We concentrate on the second part of the condition. If one process receives at least two ?s as responses from the bits, then we are done: This process will return a ?. Consequently, assume that no process will receive more than one ?s from the bits. Then, because there are $2n + 1$ bits and n processes, there exists at least one process which will receive at least two **0**s. This process will then return **0**.
4. Assume that in α at least two bits fail by omission. Then, the following holds: If some process has accessed the gate and returned a value, then every process which starts strictly thereafter returns **1** or ?.

   Assume that process P_i has returned **0** or **1** and that another process P_j starts strictly thereafter. We have that process P_i has received at most one ? when it accessed the bits. Thus, the value of at most one bit can be **0** after P_i has returned a value. Therefore, when P_j accesses the shared bits it can receive at most one **0**. It follows that process P_j will either return ? or **1**. □

6 A Lower Bound for n Processes

We now prove a lower bound on the number of test&set bits which are necessary to construct a *reliable* test&set bit for n processes (when only test&set bits can be used in a construction). We assume throughout this section that the processes are *symmetric*, i.e., the code executed by different processes is the same except for the names of local variables.

Theorem 7. *For n symmetric processes at least n test&set bits are necessary for constructing a test&set bit which can tolerate one omission fault and which is gracefully degrading for omission.*

The proof is by contradiction. Assume that there exists some test&set bit $\mathcal{T}$ for n processes which can tolerate one omission fault, is gracefully degrading for omission, and uses fewer than n primitive test&set bits. We construct a scenario in which a set of the processes which access bit $\mathcal{T}$ concurrently return **1** and in which no process has returned a ? or a **0** previously. This contradicts the correctness requirement of a test&set bit and, hence, the existence of $\mathcal{T}$. In the proof below, denote by $\langle P, b, v\rangle$ that process P has accessed primitive bit b and that b has responded with value v to P.

We define scenario **S** in which the processes $P_1, \cdots, P_n$ (in that order) access $\mathcal{T}$ until one of these processes receives only **1**s from the primitive bits and this process returns a value. In scenario **S** the primitive bits constituting $\mathcal{T}$ return either ? or **1** (and not a **0**) to the processes. Whenever one of the bits has returned a ?, its value is changed to **1**. (Thus, every test&set-operation on a primitive bit in this scenario is successful, although the responses may indicate faults.) Every primitive bit's first response is always ? and

```
shared $A[1..2n+1]$: array of primitive test&set bits, all entries initially 0
local $w$: local on {0,1,?}
local $count_1, count_2$: local on $\{0, 1, \cdots, 2n+1\}$, initially 0

function passed-gracefully-degrading-gate
1 for $i$:= 1 to $2n$+1 do $w$:= test&set($A[i]$);
2                          if $w$=0 then $count_1$:= $count_1$+1 fi;
3                          if $w$=? then $count_2$:= $count_2$+1 fi od;
4 if $count_2 \geq 2$ then return ? fi;
5 if $count_1 < 2$ then return 1 fi;
6 return 0
end_function
```

Fig. 8. A gracefully degrading construction of the gate which can tolerate one omission fault from $2n + 1$ test&set bits.

subsequent responses are always **1** in **S**. In scenario **S**, we first make every process P_i accessing $\mathcal{T}$ run until one of the following occurs:

1. P_i accesses a bit b which has not been accessed by any other process before. As a consequence of how the primitive bits behave in scenario **S**, it follows that bit b returns ? to process P_i. All bits different from b accessed by P_i return **1** to P_i.
2. P_i accesses only bits which have been accessed before and P_i returns some value (and terminates its test&set-operation). Again because of how the primitive bits behave in scenario **S**, each of the bits accessed by P_i returns **1**.

Now, there exists a process P_k participating in scenario **S** which receives only **1**s as responses from the primitive bits it accesses. Process P_k exists indeed because of the pigeonhole principle: By assumption, there are n processes and fewer than n bits constitute bit $\mathcal{T}$. Now, $k = 1$ immediately contradicts the existence of $\mathcal{T}$. (If P_1 returns a value without accessing any primitive bit, then each of the processes will return the same value as P_1 when they access bit $\mathcal{T}$. This is so because the processes are symmetric. A similar argument immediately leads to a contradiction if P_1 returns a value after receiving its first ?.) Thus, hereafter we assume that $k > 1$ is true. Scenario **S** looks like

$$\langle P_1, b_1, ?\rangle \circ \sigma_2 \circ \langle \mathrm{P}_2, \mathrm{b}_2, ?\rangle \circ \sigma_3 \cdots \sigma_{\mathrm{k}-1} \circ \langle \mathrm{P}_{\mathrm{k}-1}, \mathrm{b}_{\mathrm{k}-1}, ?\rangle \circ \sigma_{\mathrm{k}}.$$

Here, $\circ$ denotes concatenation as usual; b_i the primitive bit which returns ? to P_i; and σ_{i+1} the sequence of the form $\langle P_{i+1}, b^1, \mathbf{1}\rangle \circ \cdots \circ \langle P_{i+1}, b^\ell, \mathbf{1}\rangle$ with $\{b^1, \cdots, b^\ell\} \subseteq \{b_1, \cdots, b^i\}$ $(i = 1, \cdots, k-1)$.

The next lemma states that process P_k returns value **1**.

Lemma 7. *Assume that bit $\mathcal{T}$, scenario **S**, and process P_k are as defined above. Then, P_k returns **1**.*

Proof. Consider scenario $\mathbf{S}'$ defined by

$$\langle P_1, b_1, \mathbf{0}\rangle \circ \sigma_2 \circ \langle P_2, b_2, \mathbf{0}\rangle \circ \sigma_3 \cdots \sigma_{k-1} \circ \langle P_{k-1}, b_{k-1}, \mathbf{0}\rangle \circ \tau_1 \cdots \tau_{k-1} \circ \sigma_k.$$

Scenario $\mathbf{S}'$ differs from scenario **S** in that in $\mathbf{S}'$ the primitive bit b_i returns **0** (instead of ?) to process P_i; and each of the processes P_i runs to completion, indicated by τ_i, before P_k accesses bit $\mathcal{T}$ $(i = 1, \cdots, k-1)$. In scenario $\mathbf{S}'$, no primitive bit responds with ?. Notice that $\mathbf{S}'$ defines an execution of bit $\mathcal{T}$ indeed. Also notice that in $\mathbf{S}'$ process P_k returns the same value as in scenario **S**. (Process P_k cannot distinguish **S** from $\mathbf{S}'$.) In scenario $\mathbf{S}'$, none of bit $\mathcal{T}$'s primitive bits exhibits a fault and the processes $P_1, \cdots, P_{k-1}$ have all returned some value strictly before P_k has started. It follows that in scenario $\mathbf{S}'$, hence, in **S**, process P_k returns **1**, because $\mathcal{T}$ satisfies the correctness requirement of a test&set bit. (A process cannot return **0** if it has started strictly after another process has already returned a value.) □

Thus, process P_k returns value **1** in scenario **S**. We next extend scenario **S** to one in which all the processes $P_1, \cdots, P_k$ return **1**. To do so, let us call a process *active* in some scenario if it has accessed bit $\mathcal{T}$ in that scenario, but it has not (yet) returned any value. Notice that, by construction, all of the processes $P_1, \cdots, P_{k-1}$ are active in scenario **S**.

Consider the following scenario $\mathbf{S}'$ extending **S**: In $\mathbf{S}'$, the processes and the bits behave first as they do in scenario **S**. Choose an active process P_j. (Thus, P_j is accessing bit $\mathcal{T}$ and has not returned any value yet.) We then make process P_j continue its execution until it is about to access a bit which has not been accessed by any other process before or until it returns a value otherwise. As before, a bit returns value **1** to a process if it has been accessed before by any process. We next distinguish two cases:

1. Process P_j returns a value without accessing a bit which has not been accessed before.
2. Process P_j is about to access a bit which has not been accessed before.

Lemma 8. *Assume that bit $\mathcal{T}$, scenario $\mathbf{S}'$ and process P_k are as defined above. Consider process P_j as in case 1 above. Then, P_j returns **1** in scenario $\mathbf{S}'$.*

Proof. Scenario $\mathbf{S}'$ looks like

$$\langle P_1, b_1, ?\rangle \circ \sigma_2 \circ \langle \mathrm{P}_2, \mathrm{b}_2, ?\rangle \circ \sigma_3 \cdots \sigma_{\mathrm{j}} \circ \langle \mathrm{P}_{\mathrm{j}}, \mathrm{b}_{\mathrm{j}}, ?\rangle \circ \sigma_{\mathrm{j}+1} \cdots \sigma_{\mathrm{k}-1} \circ \langle \mathrm{P}_{\mathrm{k}-1}, \mathrm{b}_{\mathrm{k}-1}, ?\rangle \circ \sigma_{\mathrm{k}} \circ \tau_{\mathrm{j}}.$$

Here, τ_j denotes that part of the execution where process P_j runs to completion. Define scenario $\mathbf{S}''$ by

$$\langle P_1, b_1, \mathbf{0}\rangle \circ \sigma_2 \circ \langle P_2, b_2, \mathbf{0}\rangle \circ \sigma_3 \cdots \sigma_j \circ \langle P_j, b_j, ?\rangle \circ \sigma_{\mathrm{j}+1} \cdots \sigma_{\mathrm{k}-1} \circ \langle \mathrm{P}_{\mathrm{k}-1}, \mathrm{b}_{\mathrm{k}-1}, \mathbf{0}\rangle \circ \tau_{\mathrm{j}}.$$

Scenario $\mathbf{S}''$ is the same as $\mathbf{S}'$ except that in $\mathbf{S}''$ we do not activate process P_k; and the bits accessed by processes different from P_j return **0** (instead of ?). It is easy to see that $\mathbf{S}''$ defines a scenario of bit $\mathcal{T}$ indeed. Process P_j cannot distinguish scenario $\mathbf{S}'$ from scenario $\mathbf{S}''$. Notice that in scenario $\mathbf{S}''$ only one primitive bit exhibits a fault, namely the bit which returns a ? to process P_j. It follows that in $\mathbf{S}''$ and, hence, in **S**, process P_j returns a **0** or a **1** (and not a ?). However, P_j cannot return **0** in scenario $\mathbf{S}''$. This is true because of the following: Assume that process P_j returns **0** in scenario $\mathbf{S}''$. Then, we activate process P_k strictly thereafter. Let the responses from the primitive bits to P_k be the same as the responses to process P_j. (This is possible, since these responses are either **1** or ?.) It follows that P_k also returns **0** in scenario $\mathbf{S}''$ (because of symmetry). Consequently, there exists a scenario in which two processes return **0**, contradicting the correctness requirement of bit $\mathcal{T}$.

We conclude that in scenario $\mathbf{S}'$, P_j returns a **1**, hence, that there exists an extension of **S** in which P_j returns a **1**. □

In order to show that this extension also exists when process P_j is about to access a bit which has not been accessed by any process before (case 2 above), let us assume that process P_j is about to access such a bit. Observe that there exists a process P_ℓ which has not accessed $\mathcal{T}$ before. We now activate P_ℓ and let the responses from bits to it be the same as those to process P_j until P_ℓ reaches bit b, which P_j is about to access. Then, P_ℓ accesses b, which returns **0**.

Thereafter, we make P_j access bit b which then returns a **1**, and make it run until it is about to access a bit which has not been accessed before or until it returns a value otherwise. If scenario $\mathbf{S}'$ is of the form

$$\langle P_1, b_1, ?\rangle \circ \sigma_2 \circ \langle \mathrm{P}_2, \mathrm{b}_2, ?\rangle \circ \sigma_3 \cdots \sigma_j \circ \langle \mathrm{P_j}, \mathrm{b_j}, ?\rangle \circ \sigma_{j+1} \cdots \sigma_{k-1} \circ \langle \mathrm{P}_{k-1}, \mathrm{b}_{k-1}, ?\rangle \circ \sigma_k \circ \tau_j,$$

then scenario $\mathbf{S}_1$ we have constructed is of the following form:

$$\langle P_1, b_1, ?\rangle \circ \sigma_2 \circ \langle \mathrm{P}_2, \mathrm{b}_2, ?\rangle \circ \sigma_3 \cdots \sigma_j \circ \langle \mathrm{P_j}, \mathrm{b_j}, ?\rangle \circ \sigma_{j+1} \cdots \sigma_{k-1} \circ \langle \mathrm{P}_{k-1}, \mathrm{b}_{k-1}, ?\rangle \circ \sigma_k \circ \tau_j \circ \tau_\ell.$$

Here, τ_ℓ is $\sigma'_\ell \circ \langle P_\ell, b_j, ?\rangle \circ \tau'_\ell \circ \langle \mathrm{P}_\ell, \mathrm{b}, \mathbf{0}\rangle \circ \sigma'_j$, where σ'_ℓ and τ'_ℓ are obtained from σ_j and τ_j, respectively, by replacing the first component, P_j, of every element in the sequence by P_ℓ, and where σ'_j is that part of the P_j's execution after P_ℓ has accessed bit b until P_j returns a value or it accesses a bit which has not been accessed before.

Note that if P_j is about to access a bit b' which has not been accessed before, then there exists a process which has not accessed $\mathcal{T}$ at all. Consequently, we can choose one such process, have it run as P_j until it reaches b'. Thereafter, that process accesses b' which responds with **0**. Then, P_j continues its execution and the whole procedure is repeated. Eventually, P_j will not access any "new" bits any more. It follows that eventually the procedure described above terminates and P_j returns some value.

Lemma 9. *P_j as considered above will eventually return a* **1**.

Proof. Consider scenario $\mathbf{S}_1$ of the form

$$\langle P_1, b_1, ?\rangle \circ \sigma_2 \circ \langle \mathrm{P}_2, \mathrm{b}_2, ?\rangle \circ \sigma_3 \cdots \sigma_j \circ \langle \mathrm{P_j}, \mathrm{b_j}, ?\rangle \circ \sigma_{j+1} \cdots \sigma_{k-1} \circ \langle \mathrm{P}_{k-1}, \mathrm{b}_{k-1}, ?\rangle \circ \sigma_k \circ \tau_j \circ \tau_\ell$$

as above and assume that P_j returns a value. (The proof for the general case is similar.) When P_j returns a value, it has received one ? and **1**s otherwise. Consider scenario $\mathbf{S}_2$ defined by

$$\langle P_1, b_1, \mathbf{0}\rangle \circ \sigma_2 \circ \langle P_2, b_2, \mathbf{0}\rangle \circ \sigma_3 \cdots \sigma_j \circ \langle P_j, b_j, ?\rangle \circ \sigma_{j+1} \cdots \sigma_{k-1} \circ \langle \mathrm{P}_{k-1}, \mathrm{b}_{k-1}, \mathbf{0}\rangle \circ \tau_j \circ \tau_\ell.$$

It is obtained from $\mathbf{S}_1$ by not activating process P_k and by replacing responses ? of bits to processes in $P_1, \cdots, P_{k-1}$ different from P_j by **0** (instead of ?). Notice that $\mathbf{S}_2$ defines an execution of bit $\mathcal{T}$ indeed. Also notice that process P_j cannot distinguish scenario $\mathbf{S}_1$ from the scenario $\mathbf{S}_2$. It follows that P_j cannot distinguish the scenario it is in from one in which only one of the primitive bits of bit $\mathcal{T}$ exhibits a fault and in which process P_k has not accessed $\mathcal{T}$ at all. We conclude that P_j will return a **0** or a **1** (and not a ?). A **0** is not possible, however. This is so because, otherwise, we can take a process, e.g., P_k, which has not accessed $\mathcal{T}$ which, when it accesses $\mathcal{T}$, behaves exactly as P_j. That process would also return **0** (because of symmetry). This implies that a process returns **0** after another process which has started strictly thereafter also returns **0**. This contradicts the correctness requirement of bit $\mathcal{T}$. We conclude that process P_j will return **1**. □

Proof of Theorem 7. We repeat the above described procedure for all processes P_j with $1 \le j \le k-1$. From the lemmas and the discussion above, it follows that each of these processes will eventually return **1**. Thus, there exists a scenario in which processes which access bit $\mathcal{T}$ concurrently all return **1**, and in which no process has returned a **?** or **0** previously. This contradicts the correctness requirement of a test&set bit, hence, the existence of bit $\mathcal{T}$. Note that it is irrelevant for obtaining a contradiction which values the processes P_j with $j > k$ return when they access bit $\mathcal{T}$. This is so because all these processes have started strictly after process P_k has returned value **1**. □

7 Using Atomic Bits

By using atomic bits, the constructions of a gate as defined in Sections 4 and 5 can be much simplified. An atomic bit can only take the values **0** and **1**. It supports two operations: A read operation which returns the value of the bit and a write operation which modifies the value of the bit to the one given as an argument to the write operation. We assume that an atomic bit can be read and written by all the processes. The results of this section, which are rather simple, are:

- Two atomic bits are sufficient for constructing a gate which can tolerate one omission fault.
- Three atomic bits are sufficient for constructing a gate which can tolerate one omission fault and which is gracefully degrading for omission.

These results and the constructions of Section 4 and Section 5 imply that, for n processes,

- Two atomic bits and two test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault.
- Three atomic bits and three test&set bits are sufficient for constructing a test&set bit which can tolerate one omission fault and which is gracefully degrading for omission.

These bounds are improved in the next section.

We start with the trivial observation that one atomic bit is sufficient for constructing a gate which *cannot* tolerate even one omission fault. The trivial construction of a gate is given in Fig. 9.

Theorem 8. *One atomic bit is sufficient for constructing a gate.*

The next step is to modify the construction so that it can tolerate one omission fault. This is done by replacing the single bit in the previous construction by two bits. Intuitively, a process which accesses the gate simply reads both bits. If the process sees at least one **1**, then it returns **1**; otherwise, it returns **0**. This simple construction is given in Fig. 10 and provides a proof of the following:

Theorem 9. *Two atomic bits are sufficient for constructing a gate which can tolerate one omission fault.*

```
shared $A$: atomic bit on {0,1}, initially 0

function gate-1
1 if $A = 1$ then return 1 fi;
2 $A := 1$; return 0
end_function
```

Fig. 9. A trivial construction of a gate which cannot tolerate even one omission fault.

Observe that the construction in Fig. 10 is not gracefully degrading. In Fig. 11, we give a construction of a gate which is also gracefully degrading. Here, we use three bits. Intuitively, a process which accesses the gate reads all three bits. If the process receives two ?s from the bits, then it returns ?; otherwise, if it receives at least one **1**, then it returns **1**; in all other cases, it returns **0**. This construction can be easily shown to be gracefully degrading. Thus, we have the following theorem:

Theorem 10. *Three atomic bits are sufficient for constructing a gate which can tolerate one omission fault and which is gracefully degrading for omission.*

8 Improvements

In the constructions for n processes (in Sections 4, 5, 7), we have emphasized modularity. Constructing a gate first and then combining it with a solution for two processes resulted in a solution that satisfies all the requirements for n processes. Modularity eases the presentation and the correctness proof of the solution. Now that we have these solutions, we point out how to modify them to obtain solutions with slightly better space complexity.

8.1 A Nongracefully Degrading Construction

We next describe how to construct a test&set bit which can tolerate one omission fault for any number of processes, from one (instead of two) atomic bit and two test&set bits.

Consider the construction in Fig. 12. It consists of one atomic bit R and two test&set bits A and B. The gray nodes labeled $s0$, $s1$, and $s2$ denote states. Edges out of state $s0$ are labeled by responses to read operation on R; edges out of the states $s1$ and $s2$ are labeled by responses to test&set on A and B, respectively; white nodes with some value in it represent the value returned by a process.

When a process accesses bit R, if R responds with **1**, then the process immediately returns **1**. (Some other process has already accessed R.) Otherwise, the process cannot determine whether it is the first to access R. It then accesses A. If A responds with **1**, then the process sets R to **1** (closing the gate) and returns **1**. If A's response is not **1**, then the process also accesses B and returns **1** if B responds with **1** and returns **0** otherwise.

Using the above construction, it is also possible to improve the construction presented in Section 4 by one test&set bit. Replace the atomic bit R in Fig. 12 by n test&set bits $U_1, ..., U_n$. To read bit R, process P_i does a test&set on bit U_i and uses the result as the value of R; and to write **1**, process P_i does a test&set on all the n bits.

8.2 A Gracefully Degrading Construction

An idea similar to the one presented in the previous subsection can be used to slightly improve the space complexity of the gracefully degrading solutions for n processes. Following is an informal description of a gracefully degrading construction of a test&set bit which can tolerate one omission fault for any number of processes from one (instead of three) atomic bits and three test&set bits.

In the construction for n processes in Section 5, we replace the gate by one atomic bit R. When a process accesses bit R, if R responds with **1**, then the process immediately returns **1**. (Some other process has already accessed R.) Otherwise, the process cannot determine whether it is the first to access R. It then accesses the construction for two processes (as described in Figs. 3 and 4) and executes it with only two changes: 1) If a process ever sees two ?s, then it immediately returns ?; 2) Whenever a process has to return **1**, it first sets R to 1 and only then returns **1** and terminates.

Using the above construction, it is also possible to improve the construction presented in Section 5 by one test&set bit. Replace the atomic bit R in the above solution by $2n$ test&set bits $U_1, ..., U_{2n}$. Instead of reading the atomic bit R, process P_i does a test&set on the bits U_{2i-1} and U_{2i} and uses the result as the value of R: If one of the two values is ?, then the value of R is ?; otherwise, if one of the two values is **0**, then the value of R is **0**; and, in all other cases (if both values are **1**), the value of R is **1**. To write 1, process P_i does a test&set on all the $2n$ bits.

```
shared $A[0..1]$ : array of atomic bits, all entries initially 0
local $count$ : integer, initially 0

function gate-2
1 for $i$:= 0 to 1 do if $A[i] = 0$ then $count$:= $count$+1 fi; $A[i] := 1$ od;
2 if $count = 0$ then return 0 else return 1 fi
end_function
```

Fig. 10. A trivial construction of a gate which can tolerate one omission fault.

```
shared $A[0..2]$ : array of atomic bits, all entries initially 0
local $count[0..1]$ : array of integers, all entries initially 0

function gate-3
1 for $i := 0$ to 2 do
2    if $A[i] = 0$ then $count[0] := count[0] + 1$
3    else if $A[i] = 1$ then $count[1] := count[1] + 1$ fi fi; $A[i] := 1$ od;
4 if $count[0] + count[1] \leq 1$ then return ? fi;
5 if $count[1] \geq 1$ then return 1 fi;
6 return 0
end_function
```

Fig. 11. A gracefully degrading construction of a gate which can tolerate one omission fault.

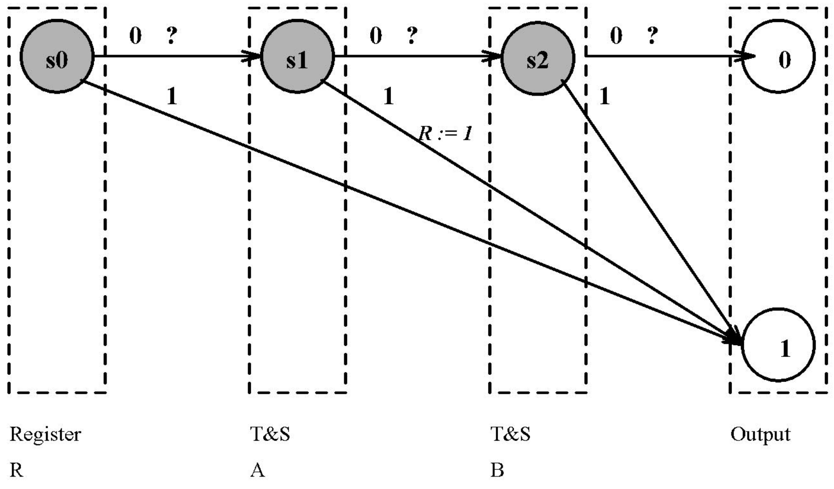

Fig. 12. Schematic of a construction of a test&set bit for any number of processes which can tolerate one omission fault. The construction is not gracefully degrading for omission.

9 Discussion

We have constructed reliable test&set bits from faulty bits. The fault model considered in this paper is that of omission. That is, operations on faulty bits either return a "correct" value or a special value ?, indicating that the operation may have not taken place. Some of the constructions are gracefully degrading, meaning that when too many component bits fail by omission, the constructed bit is either correct or also fails only by omission.

None of the constructions in the present paper can tolerate even one fault when the arbitrary fault model is assumed. (Here, responses ? are irrelevant because every bit responds with **0** or **1**.) Recall that faults in the crash fault model, the omission-crash fault model, and the eventual-crash fault model are no more severe than omission faults. Therefore, all our constructions of a test&set bit which can tolerate one omission fault can also tolerate one fault in the above mentioned three models. (In these models, ? is interpreted as a terminal crash event or as an omission fault consistent with the model under consideration.) Also, those constructions of the test&set bit which are not gracefully degrading for omission are not gracefully degrading for crash, omission-crash, or eventual-crash. Our constructions which are gracefully degrading for omission can be shown to be also gracefully degrading for crash, omission-crash, and eventual-crash.

Acknowledgments

We thank Michael Merritt for remarks concerning the presentation. Part of this work was performed while the authors were with AT&T Bell Labs.

References

[1] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld, "Computing with Faulty Shared Memory," *Proc. 11th Ann. ACM Symp. Principles of Distributed Computing,* pp. 47-58, Aug. 1992.

[2] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld, "Computing with Faulty Shared Memory," *J. ACM,* vol. 42, no. 6, pp. 1,231-1,274, Nov. 1995.

[3] Y. Afek, M. Merritt, and G. Taubenfeld, "Benign Failure Models for Shared Memory (Preliminary Version)," *Proc. Seventh Int'l Workshop Distributed Algorithms,* pp. 69-83, Sept. 1993.

[4] E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Comm. ACM,* vol. 17, pp. 643-644, Nov. 1974.

[5] M.J. Fischer, S. Moran, S. Rudich, and G. Taubenfeld, "The Wakeup Problem," *SIAM J. Computing,* vol. 25, no. 6, pp. 1,332-1,357, Dec. 1996. Also in *Proc. 31st IEEE Ann. Symp. Foundations of Computer Science,* pp. 106-116, Oct. 1990.

[6] P. Jayanti, T. Chandra, and S. Toueg, "Fault-Tolerant Wait-Free Shared Objects," *Proc. 33rd IEEE Ann. Symp. Foundation of Computer Science,* Oct. 1992. (Older version of [7].)

[7] P. Jayanti, T. Chandra, and S. Toueg, "Fault-Tolerant Wait-Free Shared Objects," *J. ACM,* vol. 45, no. 3, pp. 451-500, May 1998.
[8] M. Herlihy, "Wait-Free Synchronization," *ACM Trans. Programming Languages and Systems,* vol 11, no. 1, pp. 124-149, Jan. 1991.
[9] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Programming Languages and Systems,* vol. 12, no. 3, pp. 463-492, July 1990.
[10] L. Lamport, "On Interprocess Communication, Parts I and II," *Distributed Computing,* vol. 1, pp. 77-101, 1986.
[11] M.C. Loui and H.H. Abu-Amara, "Memory Requirements for Agreement Among Unreliable Asynchronous Processes," *Advances in Computing Research,* vol. 4, pp. 163-183, 1987.
[12] M. Schneider, "Self-Stabilization," *ACM Computing Surveys,* vol. 25, no. 1, pp. 45-67, Mar. 1993.

Frank Stomp received his BSc and MSc degrees in mathematics from the University of Utrecht, The Netherlands, and his PhD degree in computer science from Eindhoven University of Technology, The Netherlands. He is an assistant professor in the Department of Computer Science at Wayne State University, Detroit, Michigan. His research interests include distributed algorithms and formal verification.

Gadi Taubenfeld received the BS, MSc, and PhD degrees in computer science from the Technion (Israel Institute of Technology) in 1982, 1984, and 1988, respectively. From 1988 to 1990, he was a research scientist at Yale University and, from 1991 to 1995, he was a member of the technical staff at AT&T Bell Laboratories. Since 1995, he has been a faculty member at the Open University of Israel. His primary research interests are in concurrent and distributed computing.