# Benign Failure Models for Shared Memory*

Yehuda Afek[†] Michael Merritt[‡] Gadi Taubenfeld[†]

November 13, 1995

**Abstract**

This paper introduces two benign failure models for shared memory in distributed systems, *omission-crash* and *eventual-crash*. These models are of intermediate power between the *crash* model and *omission* models of [JCT92]. (They exhibit more faulty behaviors than *crash* and fewer than *omission*.) Unlike the *crash* model, *omission-crash* is shown to have universal, gracefully-degrading constructions. That is, for any integer $k$, any shared object may be constructed from shared registers and consensus objects, so that: (1) if no more than $k$ of the components suffer *omission-crash* failures, the constructed object exhibits no failures, and (2) if more than $k$ of the components suffer *omission-crash* failures, the constructed object exhibits *omission-crash* failures.

Simple constructions also demonstrate that registers and consensus objects in the (apparently) less benign *eventual-crash* model can be used to construct corresponding objects in the *omission-crash* model.

These results are cited as evidence that the *omission-crash* failure model may be an appropriate choice to consider in the formulation of a more extensive theory of fault-tolerant shared objects.

## 1 Benign failure models for shared memory

Shared memory is widely considered a useful programming abstraction for concurrent systems, masking the details of inter-procedural or inter-processor communication, while supporting natural proof-theoretic techniques [Owi75, OG76]. Many experimental and commercial processors provide direct support for this abstraction: indeed, Gordon Bell has predicted that "... the mainline, general-purpose computer is almost certain to be the shared memory multiprocessor after 1995" [Bel92]. Increasing attention is being paid to implementing shared memory systems either in hardware or in software [Bel92, CG89, LH89, TKB92].

This paper investigates fault-tolerance in shared memory systems, with an emphasis on benign, or constrained, fault models. Benign fault models are easier to program than are more malicious models. Just as shared memory is itself an abstraction of much more complex, timing-dependent implementations, benign fault models hide complex implementation details, and provide a simple programming abstraction.

*A preliminary version of this work appeared in the *Proceedings of the 7th International Workshop on Distributed Algorithms*, Lausanne, Switzerland, September 1993. In: *LNCS 725*, Springer Verlag 1993, pages 69–83.

[†]Computer Science Dept., Tel-Aviv U., Israel 69978, and AT&T Bell Labs.

[‡]AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974

Together with David Greenberg, we introduced the problem of tolerating failures in shared memory objects—previous work had considered the possibility of process failures, but assumed the shared memory was reliable [AGMT92]. Jayanti, *et al* independently posed this problem, and focused particular attention on whether fault models support modular, gracefully degrading constructions, in which failures of the abstract object are as benign as the failures of the components. They show that their *crash* model does not support such constructions, while their *omission* fault model does.

This paper further explores benign failure models that support gracefully degrading constructions. An appropriate model, serving as a contract between implementor and programmer, can greatly facilitate the development of fault-tolerant shared memory systems. This contract is ultimately a compromise between ease of implementation and ease of programming—motivating a search for "the most benign, implementable failure model".

We define two failure models that are more benign than *omission* faults, and stress the most benign, *omission-crash* faults. In this model, a shared memory object suffers a failure of type *omission-crash* if operations concurrent with the crash may respond with a special "?" value, and all operations after the crash always respond with $\perp$. Operations concurrent with the crash may respond normally to some requests because those requests may not access (or be through accessing) the crashed parts of the object. Our argument in favor of this choice of model will use the same framework, and several of the same techniques and constructions as are used by Jayanti, *et al* in support of the *omission* model. The principal contribution of this paper lies in the definition of (an inherently) more benign failure model, and observations that it nevertheless has the same positive attributes as the *omission* model.

## 1.1 Faulty objects

Two recent papers explore the possibility that shared objects might fail [AGMT92, JCT92]. The first explored the possibility that faulty shared objects might return arbitrary values, developed a compositional theory for combining algorithmic constructions in a modular way, and presented several lower bounds and constructions of fault-tolerant objects from faulty components, including a universality result. This showed that any reliable shared object can be constructed from potentially-faulty registers and consensus objects, or from read-modify-write objects, provided the number of faulty components was bounded.

The second, independent paper explored a wider range of fault models, including some "non-responsive" models that are particularly malicious, making the programming task essentially impossible.

Among the *responsive* fault models, which are guaranteed to return *some* response to an operation, is the same *arbitrary* fault model developed in [AGMT92], and two others of particular interest to this paper, the *omission* and *crash* models. A compositional theory is developed that depends upon constructions that *degrade gracefully*. That is, if the failure bound on the number of components that may fail is exceeded, the composite object may suffer failures, but only of the same type as the components. Such gracefully degrading constructions may be used as modules in larger constructions.

The possibility of building gracefully degrading constructions within a failure model is essential, if that model is to be useful in the modular design or verification of systems.

Without this property, fault-tolerance violates any attempt to build interfaces or abstraction layers, and the correctness of the system in the face of failures must be considered monolithically. As a general concept applicable to a variety of failure models, graceful degradation is a vital contribution to the theory of fault-tolerant systems.

In [JCT92], the *crash* model is shown to be unsuitable as a general failure model. Objects in this model suffer a terminal, atomic `crash` event, after which all operations return the special symbol "$\perp$". Though quite benign, and hence seemingly useful as a programming abstraction, the model is *so* benign that gracefully degrading implementations are impossible to design. The *omission* model, on the other hand, *does* support gracefully degrading implementations—even universal constructions, which can be used in fault-tolerant implementations of any shared object. In the *omission* model, operations on a faulty object may return a special return value, "?", signaling that the operation may not have taken place, and hiding any return value.

These results illustrate a trade-off in designing gracefully-degrading implementations: the more benign the fault model, the easier it is to find constructions that work correctly as long as the failure bound on the number of primitive objects is not exceeded, but the harder it is to make sure that the construction is gracefully degrading—that it exhibits the same type of benign failure when the failure bound on the number of primitive objects is exceeded.

Because of its powerful modular properties, the *omission* fault model is argued to be a superior model for the development of an algorithmic theory for shared memory under benign failures. Arbitrary failures are shown in both papers to support universal, gracefully degrading implementations, but are more malicious. In this paper, we argue that a still more benign model may be more appropriate in this context. We introduce an inherently more benign model, *omission-crash* faults, and show that it enjoys essentially the same positive properties as the *omission* and *arbitrary* models.

### 1.2 Four benign failure models

This section provides brief descriptions of the four failure models discussed in the paper—the *crash* and *omission* models defined in [JCT92], and the new *omission-crash* and *eventual-crash* models. All are defined in detail in the Appendix. They may be ordered from most to least benign: *crash*, *omission-crash*, *eventual-crash* and *omission*. (For example, a *omission-crash* failure is also a *eventual-crash* and *omission* failure, but may not be a *crash* failure.)

The *crash* faults were discussed in the previous section: operations on objects of this fault type behave correctly until the object suffers a terminal, atomic `crash` event. Thereafter, all operations return "$\perp$". As pointed out in [JCT92], this model is *too* benign—there are essentially no interesting objects that can be constructed which will fail in this way, even if the components fail in the correspondingly benign way. In brief, the problem is that the `crash` event in this model has very strong semantics. For example, if a read operation returns a good value, the reader may conclude that the corresponding write operation completes successfully, *even if it has not yet returned.* Such inferences are at the heart of the impossibility results in [JCT92].

The *omission* fault model avoids the strong semantics of the `crash` event in an extreme way. In this model, operations on faulty objects may return a special value, "?", which indicates only that the operation has terminated—and that it either effected the state of the object as intended, or completely failed to effect it.

The first new model defined in this paper, the *omission-crash* model, attempts to overcome the problems of the *crash* model, without introducing as much uncertainty as the *omission* model. In this model, as in the *crash* model, faulty objects again suffer a terminal, atomic `crash` event. However, operations that are *concurrent* with the crash event may suffer omission failures, returning "`?`". Subsequent operations return "$\perp$".

It is easily shown that such simple objects as read/write registers suffering only *omission-crash* failures, cannot be built from registers that suffer *omission* failures. (See Theorem 1, below.) Hence, the *omission-crash* model is truly more benign than omission failures. On the other hand, essentially the same constructions used for the *omission* case in [JCT92] can be used to show corresponding gracefully degrading compositional results for the *omission-crash* model.

The *eventual-crash* model is slightly less benign. In this model, faulty objects eventually suffer terminal `crash` events, but *any* operation prior to or concurrent with the `crash` may suffer an omission fault and return "`?`". Hence, in this model, multiple operations by a single process may suffer omission failures, returning "`?`", before the terminal crash event occurs, and all subsequent operations return `crash`. This contrasts with the *omission-crash* model, in which at most one operation by a single process will suffer an omission failure, returning "`?`". Every operation thereafter returns `crash`.

If an operation on a *eventual-crash* object returns "`?`", the calling process knows the object will eventually crash. If the object supports idempotent operations, the calling process can make repeated calls until the crash event occurs. Hence, for such objects, this fault model can be used to simulate the *omission-crash* model.

# 2 Results

## 2.1 Basic definitions

Following [LT87, Lyn88], we consider a shared object to be an automaton with a distinct interface to the calling processes. Following [Her91], an *implementation* of an object $Y$ from a list of one or more (shared) objects of type $X$ is a composition of automata from $X$, together with "front-end" automata running code local to each process, so that the processes only observe behavior from the composition that they could also observe from $Y$. The implementation is *wait-free* if each call to the implementation by any process is guaranteed to return, despite the behavior of any of the other processes.

Following [JCT92], such an implementation is *gracefully degrading* for a given failure type $F$, if the operations on $Y$ are either correct or suffer failures of type $F$, despite failures of type $F$ for any of the operations on the component objects. The following definitions are used through out the paper:

- Object type $Y$ has a (*gracefully degrading*) *f-reliable* wait-free implementation from object type $X$ for faults of type $F$, if there exists a (*gracefully degrading*) wait-free implementation of a $Y$ object from objects of type $X$, when at most $f$ of them suffer faults of type $F$.

- Object type $Y$ has a (*gracefully degrading*) *reliable* wait-free implementation from object type $X$ for faults of type $F$, if for every positive integer $f$, $Y$ has a (*gracefully degrading*) *f-reliable* wait-free implementation from object type $X$ for faults of type $F$.

Finally, a *self-implementation* is an implementation where the base objects and the derived object are of the same type.

We focus on the *omission-crash* fault model in the next two subsections. While corresponding results can be proven directly for the *eventual-crash* model, in Section 2.4 we show that for many interesting objects, including `registers` and $n$-`consensus`, there is no real difference between these two models.

### 2.2 Basic results

In [Lam86], Lamport defined three general classes of shared read/write registers—safe, regular, and atomic—depending on their properties when several reads and/or writes are executed concurrently. The weakest possibility is `safe-register`, in which it is assumed only that a read not concurrent with any writes obtains the correct value. An `atomic register`, is a safe register in which the reads and writes behave as if they occur in some definite order. A formal definition of what it means for such registers to fail by *omission-crash* and *eventual-crash* is given in the Appendix.

**Theorem 1** *Given any collection of objects, all of which may experience omission failures, it is not possible to implement a* `safe-register` *object of failure type omission-crash.*

*Proof:* The key idea is to use omission faults to prevent the reader from knowing whether a write has occurred. Formally, assume to the contrary that there exists such an implementation, and let *reg* be the implemented register object. Consider first a run $Q$, in which process $q$ performs a single read operation, which must return the initial value of the register. Now consider a run $PQ$, in which $p$ first writes a (non-initial) value into *reg*, and then process $q$ reads the value of *reg*, but that all of the operations by $p$ on the component objects suffer omission failures, which fail to change the states of those objects. Then process $q$ will observe the same events in $PQ$ as in $Q$, and will return the initial value. But this cannot happen in a run in which *reg* suffers a *omission-crash* failure—none of the possible return values for $P$, normal, "?" or "$\perp$", are correct. The first violates the write-read semantics, and the others imply the crash occurred during operation $P$, and $Q$ should return "$\perp$". ∎

Theorem 1, can be shown to hold also for other objects such as `test-and-set` and `compare-and-swap`. In general the result holds for any object where it is possible to determine the order in which two operations should be serialized, based on the return values of the operations. (Such objects are called *order-sensitive* in [JCT92].)

This result should be contrasted with the following:

**Theorem 2** *Object type* `safe-register` *has a reliable gracefully degrading wait-free self-implementation for omission-crash.*

*Proof:* A single-reader/single-writer safe register can be implemented from $2f+1$ similar registers, at most $f$ of which fail by *omission-crash*. The construction is the obvious one:

**Reader:** The reader reads all $2f+1$ registers. If it observes more than $f$ failures (i.e., $f$ registers return either "$\perp$" or "?") it returns "$\perp$". Otherwise, if it sees a majority value ($f+1$ registers with the same value), it returns that value, otherwise it returns any value.

**Writer:** The writer writes all $2f+1$ registers and collects acknowledgments If it observes more than $f$ failures, it returns "$\perp$". Otherwise, it returns "success".

When less than $f$ registers fail by *omission-crash*, the faulty registers are are simply ignored by the writer and reader, and deciding according to the majority guarantees that the construction behaves as a correct safe registers. That is, a write returns "success", a read not concurrent with a write returns the new value, and a read concurrent with a write returns some value (in that case we do not care what the value is). This shows that the implementation is reliable – it can tolerate up to $f$ failures.

Next we need to prove that the implementation is gracefully degrading for *omission-crash*. That is, in the case of more than $f$ faults, the compound object fails by *omission-crash*. Now assume that more than $f$ registers fail by *omission-crash*. For a given execution, we call the earliest point in time when more than $f$ registers became faulty the crash-point of the compound object. To prove that the gracefully degrading property holds it is enough to show that an external observer can assume that the compound object failed at the crash-point by *omission-crash*.

We already showed that all read and write operations that ended before the crash-point return the correct value; next we show that all reads and writes that started after the crash-point return "$\perp$". But, any operation that starts after the crash-point will notice that at least $f+1$ registers are faulty (i.e.,at least $f+1$ will return "$\perp$") and hence will also return "$\perp$".

Finally, we should consider read and write operations that are concurrent with the crash point. As for a write operation, it is fine if it return either "success" or "$\perp$". A read that is not concurrent with a write will clearly return either the correct value (since it will read the same value from all non-faulty registers), or "$\perp$". If it is concurrent with the write than, by definition of safe register, we do not care what it returns ∎

An $n$-`consensus` object is an object that supports two operations, *propose 0* and *propose 1*. It may be invoked at most $n$ times and satisfies the following two properties: (1) Validity: it returns a response $v$, only if there was a prior invocation of propose $v$. (2) Agreement: all response values are the same, and are either 0 or 1. Thus, no two responses return conflicting values, and if all proposed values are the same then that value must be the return value.

**Theorem 3** *$n$-`consensus` has a reliable gracefully degrading, wait-free self-implementation for omission-crash.*

*Proof:* The proof uses a construction from the proof of Theorem 5.2 of [JCT92], which establishes a similar result for *omission* faults (using $2f+1$ $n$-`consensus` objects). The proof shows how the construction of [JCT92] can be augmented to get a construction of $n$-`consensus` which is reliable gracefully degrading, wait-free self-implementation for *omission-crash*.

At first, we make use `safe-bits` which we will later replace by consensus objects. By Theorem 2, we know how to construct reliable safe-bits from unreliable such bits, and hence in the following we assume that the bits we are using are reliable.

The original construction of [JCT92] is augmented with a pair of (reliable) safe registers between every pair of processes, and a boolean flag local to each process, all initialized to 0. (For any pair of processes $i$ and $j$ there is a single-writer/single-reader safe bit that $i$ can write and $j$ can read, and a single-writer/single-reader safe bit that $j$ can write and $i$ can read.) Before proposing a value, each process reads all the "crash" bits written by other processes. If any are set to 1 the process writes 1 to all of them, and returns "$\perp$".

Otherwise, the construction of [JCT92] is run, and returns normally in if the response is not "?". But if the object returning "?", the (invoking) process immediately writes 1 to the crash bits, and returns "?".

The `crash` event can be serialized after the first point in the run in which every process has an "incoming" crash bit that has been set to 1. This ensures that the `crash` event is serialized after every operation that returns normally, before every operation that returns "$\perp$", and within the duration of every operation that returns "?".

Now, observe that each of the single-writer/single-reader safe bits, is read at most once and is written at most once. Hence such a (reliable) bit can be replaced (implemented) by a `2-consensus` object, where the read is implemented as *propose 0* operation and the write (of the value 1) is implemented as *propose 1* operation. ∎

In addition to showing that a result corresponding to Theorem 3 holds for *omission* failures, Jayanti, *et al* prove that a similar result does *not* hold for *crash* faults. That is, $n$-`consensus` has *no* reliable gracefully degrading, wait-free self-implementation for *crash* (Theorem 7.2 [JCT92]). Together, the results of this section demonstrate that *omission-crash* is distinct from *crash* and *omission*.

### 2.3 Universality results

Herlihy introduced the notion of *universal* collections of shared objects: object types powerful enough so that they can be used to construct wait-free implementations of any shared object. Both [AGMT92] and [JCT92] explore the relationship between fault models and this result. In particular, Jayanti, *et al* observe that any universal collection of objects can be used not only in wait-free implementations of arbitrary objects, but also in implementations that are reliable and gracefully degrading for *omission* faults. In this section, we prove corresponding results for *omission-crash* faults.

First, given safe registers, we show that *any* wait-free implementation can be made gracefully degrading for *omission-crash* failures. The safe registers can be used to signal between processes that a fault has occurred, thus implementing the `crash` event.

**Theorem 4** *If there is a wait-free implementation of an object of type $X$ from a list $Y$ of object types, then there is a wait-free implementation of $X$ from $Y$ and* `safe-register` *that is gracefully degrading for omission-crash faults.*

*Proof:* Note first that neither construction is fault-tolerant. What distinguishes them is that *omission-crash* failures of components in first construction can result in arbitrary faulty behavior at the higher level. The proof shows that this construction can be augmented so that *omission-crash* failures of components are experienced as *omission-crash* failures at the higher level.

The original construction is augmented with a pair of safe registers between every pair of processes, and a boolean flag local to each process, all initialized to 0. (For any pair of processes $i$ and $j$ there is a single-writer/single-reader safe bit that $i$ can write and $j$ can read, and a single-writer/single-reader safe bit that $j$ can write and $i$ can read.) Before performing an operation on the compound object, each process reads it's local "crash" bit and those written by other processes. If any are set to 1 or if any reads return "?" or "$\perp$", the process writes 1 to all of them, and returns "$\perp$".

Otherwise, the construction is run, and returns normally in the absence of faults in the component operations. But if any of the basic components suffers faults, returning "?" or "$\perp$", the process immediately writes 1 to the crash bits, and returns "?".

The remainder of the proof argues that the `crash` event can be serialized after every operation that returns normally, before every operation that returns "$\perp$", and within the duration of every operation that returns "?". Intuitively, this is the first point in the run in which every process has an "incoming" crash bit that has either crashed or is set to 1. ∎

It is interesting to note that the same result holds if all the components in $Y$ (but not the safe registers) suffer the more malicious *omission* faults. Hence, the inability to implement *omission-crash* safe registers (Theorem 1) characterizes the differences between the models.

The following definitions are needed to state the universality theorem.

- A list $X$ of object types is *$n$-universal* if every $n$-process object has a wait-free implementation from $X$. (The wait-free implementation is not required to be gracefully degrading or reliable.)
- A list $X$ of object types is *$n$-GD-universal for fault type $F$* if every $n$-process object has a wait-free implementation from $X$ that is **G**racefully **D**egrading for $F$.
- A list $X$ of object types is *$n$-RGD-universal for fault type $F$* if every $n$-process object has a **R**eliable wait-free implementation from $X$ that is **G**racefully **D**egrading for $F$.

(These implementations may contain any number of objects of each type in $X$.)

We can now state the following result:

**Theorem 5**

- *Any list $X$ of object types that is $n$-universal and includes (or implements)* `safe-registers` *that may only fail by omission-crash, is also $n$-RGD-universal for omission-crash faults.*
- *Any list $X$ of object types that is $n$-GD-universal for omission-crash, is also $n$-RGD-universal for omission-crash faults.*

The proof of this theorem requires some additional results. First, Herlihy's universal construction [Her91], together with constructions of atomic registers from safe registers [Pet83, Lam86, BP87, PB87, Blo87, SAG87, LTV89, Tro89], implies the following:

**Proposition 1** {$n$-`consensus, safe-register`} *is $n$-universal.*

Next, we need results for composing fault-tolerant and fault-intolerant constructions, observed for a malicious failure type in [AGMT92], and for general failure classes in [JCT92].

**Proposition 2**

- *If $X$ has a reliable gracefully-degrading wait-free construction from $Y$, and $Y$ has a gracefully-degrading wait-free construction from $Z$, then $X$ has a reliable gracefully-degrading wait-free construction from $Z$.*
- *If $X$ has a gracefully-degrading wait-free construction from $Y$, and $Y$ has a reliable gracefully-degrading wait-free construction from $Z$, then $X$ has a reliable gracefully-degrading wait-free construction from $Z$.*

*Proof of Theorem 5:* Both parts of the theorem depend on the availability of `safe-registers` that fail by *omission-crash*. Using these and Theorem 4, the objects in $X$ can be used to implement $n$-`consensus` objects, in wait-free implementations that are gracefully degrading for *omission-crash* faults. Now Theorems 2, 3, and (the first part of) Proposition 2 are used to make these implementations reliable, as well: `n-consensus` and `safe-register` have wait-free implementations from $X$ that are reliable and gracefully degrading for *omission-crash*.

From Proposition 1 and Theorem 4, any object $Y$ has a wait-free implementation from {$n$-`consensus, safe-register`} that is gracefully degrading for *omission-crash* faults. Applying (the second part of) Proposition 2, the theorem follows.∎

**Corollary 1** *Every object has a wait-free implementation from* `n-consensus` *and* `safe-register` *that is reliable and gracefully degrading for omission-crash faults.*

## 2.4 Converting *eventual-crash* faults to *omission-crash* faults

Recall that objects which suffer *eventual-crash* faults may return the omission response "?" any number of times to calling processes, but must eventually `crash`. Thus, they constitute an intermediate model between *omission-crash* faults, which return at most one "?" to any single process, and *omission* faults, which can suffer any number of omissions and never `crash`.

However, consider objects such as $n$-`consensus`, `single-writer register` or `safe-register`, which satisfy the following idempotent, or *stuttering property*: every process may invoke some operation repeatedly, without changing the behavior observed by other processes. The operation may depend on the process's history. For example, a write to a single-writer register may be repeated, so long as the value written is the same. Similarly, since only the first operation on an $n$-consensus object affects it's state, processes may repeatedly access it.

**Theorem 6** *Let $X$ be an object type satisfying the stuttering property. Then $X$ has a wait-free self-implementation from one instance of $X$, with the following property: if the embedded component suffers* eventual-crash *failures, the constructed object suffers* omission-crash *failures.*

*Proof:* Each process makes calls to the *eventual-crash* embedded object, passing return values transparently, until the first omission failure. The calling process then repeats the appropriate stuttering operation until the object crashes and returns "$\perp$". This high-level operation returns "?", and all subsequent high-level operations by the same process return "$\perp$". (If a process observes a crash without seeing an omission failure, it returns "$\perp$".)∎

This result indicates that *eventual-crash* faults are not significantly different from *omission-crash*, and provides further evidence that *omission-crash* is an implementable failure model.

The details of this construction are also of some interest. Since there is no bound on the number of omission faults that may precede a crash, no upper bound can be placed on the number of embedded calls that a high-level operation may make before returning. Hence, this construction satisfies the "eventual" property of wait-freedom, but not more restrictive "bounded wait-free" constraints that require an upper bound on the number of embedded operations in a high-level operation.

# 3 Discussion

We close with a few further remarks on the importance of benign fault models. Our work follows a large body of work in making a strong asynchrony assumption, that processes and shared memory operations may run at arbitrarily different rates. This relieves the implementation from any timing constraints, and provides a particularly simple programming model. We also adopt a common practice of assuming processes fail by simply taking no more steps, but that any number of them may fail—robust algorithms in this model are called wait-free, as each process is guaranteed to make progress, despite actions of other processes (cf. [Lam86]).

Unfortunately, if the shared memory supports only read and write operations, too many synchronization tasks are impossible in this simple model [DDS87, FLP85, CIL87, Her91, LA87, TM89]. Instead of introducing timing assumptions explicitly, the programmer's job can be made easier (possible) by abstracting specific synchronization tasks as operations supported directly by the shared memory. Shared abstractions (objects) that are more powerful than read/write memory, such as semaphores or test&sets, have long been used in actual shared memory systems. Invigorated especially by the work of [Her91], a rich theory of such primitives is emerging. It is in this context that we investigate the additional question of failures of these primitives, and consistent with the search for simple, implementable models, for the most benign, implementable fault model.

This paper has focused on gracefully degrading implementations, building objects with benign failures from components that suffer benign failures. It is also important to ground this work in more detailed computational models that make explicit reference to time. For example, the construction of *omission-crash* objects from *eventual-crash* is strongly suggestive of practical fault-tolerant implementations of *omission-crash* objects, that would likely use waiting and explicit notification (messages) to "crash" shared objects. Similarly, the *omission* model is strongly suggestive of a simple implementation in an unreliable message-passing system that uses timeouts. An omission response of "?" would indicate that a timeout occurred, and either the request or the reply message was lost. An exploration of these implementation issues in an appropriate real-time model is an important open problem.

## References


[AGMT92] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared memory. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, pages 47–58, 1992.

[Bel92] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):27–47, August 1992.

[Blo87] B. Bloom. Constructing two-writer atomic registers. In *Proc. of the Sixth ACM Symp. on Principles of Distributed Computing*, pages 249–259, 1987.

[BP87] J. E. Burns and G. L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proc. of the Sixth ACM Symp. on Principles of Distributed Computing*, pages 222–231, 1987.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CIL87] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 86–97, 1987.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, January 1991.

[JCT92] P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. In *33rd Annual Symp. on the Foundations of Computer Science*. IEEE Computer Society Press, October 1992.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Programming Languages and Systems*, 7(4):321–359, 1989.

[LA87] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research, JAI Press*, 4:163–183, 1987.

[Lam86] L. Lamport. On interprocess communication, parts I and II. *Distributed Computing*, 1:77–101, 1986.

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. of 6th ACM Symp. on Principles of Distributed Computation*, pages 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, April 1987.

[LTV89] M. Li, J. Tromp, and P. M. B. Vitányi. How to construct concurrent wait-free variables. Technical Report CS-8916, CWI, Amsterdam, April 1989. See also: pp. 488-505 in: *Proc. International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 372, Springer Verlag, 1989.

[Lyn88] N. A. Lynch. I/O automata: A model for discrete event systems. In *22nd Annual Conf. on Information Science and Systems*. Princeton University, March 1988. Also MIT technical report number MIT/LCS/TM-351.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(1):319–340, 1976.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, August 1975.

[PB87] G. L. Peterson and J. E. Burns. Concurrent reading while writing II : The multi-writer case. In *Proc. of the 28th IEEE Annual Symp. on Foundations of Computer Science*, pages 383–392, October 1987.

[Pet83] G. L. Peterson. Concurrent reading while writing. *ACM Trans. on Programming Languages and Systems*, 5(1):46–55, 1983.

[SAG87] A. K. Singh, J. H. Anderson, and M. G. Gouda. The elusive atomic register revisited. In *Proc. of the Sixth ACM Symp. on Principles of Distributed Computing*, pages 206–221, 1987.

[TKB92] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Balvrije. Parallel programming using shared objects. *IEEE Computer*, pages 10–19, August 1992.

[Tro89] J. Tromp. How to construct an atomic variable. In J.C. Bermond and M. Raynal, editors, *Proc. of the 3rd International Workshop on Distributed Algorithms*, pages 292–302. Springer-Verlag LNCS 392, September 1989.

[TM89] G. Taubenfeld and S. Moran. Possibility and impossibility results in a shared memory environment. In *3rd International Workshop on Distributed Algorithms*, 1989. Lecture Notes in Computer Science, vol. 392 (eds.: J.C. Bermond and M. Raynal), Springer-Verlag 1989, pages 254–267.

# 4 Appendix: Specifications of faulty objects

We briefly describe how atomic shared objects can be specified as I/O automata, then precisely define the failure models used in this paper. Finally, we extend these models to the non-atomic `safe-register`.

## 4.1 Operations

Let $S$ be an arbitrary set of *states*. An *operation*, *op*, on $S$ is a 3-tuple $(arg(op), ret(op), spec(op))$, where $arg(op)$ and $ret(op)$ are non-empty sets of *arguments* and *return values*, and $spec(op) \subseteq arg(op) \times S \times ret(op) \times S$.

If $(x, s_1, y, s_2)$ is in $spec(op)$, this means that applying operation *op* with arguments $x$ to the object in state $s_1$ can result in return values $y$, leaving the object in state $s_2$. An operation *op* is *total* if for every $x \in arg(op)$ and every $s_1 \in S$, there exist $y \in ret(op)$ and $s_2 \in S$ such that $(x, s_1, y, s_2)$ is in $spec(op)$.

For example, a write operation $write_1$ on a binary atomic register could be defined with state space $S = \{0, 1\}$, $arg(write_1) = \{0, 1\}$, $ret(write_1) = \{\mathtt{ok}\}$, and $spec(write_1) = \{(0, 0, \mathtt{ok}, 0), (0, 1, \mathtt{ok}, 0), (1, 0, \mathtt{ok}, 1), (1, 1, \mathtt{ok}, 1)\}$. Similarly, a read operation $read_2$ on a binary atomic register would be defined with state space $S = \{0, 1\}$, $arg(read_2) = \{null\}$, $ret(read_2) = \{0, 1\}$, and $spec(read_2) = \{(null, 0, 0, 0), (null, 1, 1, 1)\}$.

## 4.2 Atomic object shared by $n$ processors, $\{1, \ldots, n\}$

Input/output automata can [LT87, Lyn88] be used to capture the notion of a *shared* object, on which operations can be performed concurrently by different users of the object. (We will call these users *processors*.) *Atomicity* captures the intuition that despite concurrency, operations should appear to perform serially. This is captured by breaking a processor's execution of an operation into three steps: the request for the operation issued by the processor, a step internal to the shared object in which the operation is performed, and a reply from the object to the processor containing any return values.

The request and reply actions only do the bookkeeping for the pending operations– semantics of the operation are determined completely by the internal event, when the operation is performed on the object state. But the separation of this internal action from the request and reply is crucial–since neither this action, nor the state itself is directly visible to the processor, alternative implementations may substitute many internal actions, on very different states, but must preserve the same sequences of requests and replies, which are observed by the processors.

An atomic object shared by $n$ processors, $\{1, \ldots, n\}$, is specified via a state space $S$, and a set of *operations*, $OP_i = \{op_i^1, ..., op_i^k\}$, associated with each processor $i$. The operations in $OP_i$ are those which may be invoked by processor $i$. We also assume two special symbols not in any $ret(Op_i)$, ? and $\perp$, and also that $\perp \notin S$. For example, a binary atomic register with a single writer and a single reader could be defined with state space $S = \{0, 1\}$, $OP_1 = \{write_1\}$, and $OP_2 = \{read_2\}$.

The state space and operations are used to specify an I/O automaton, $\mathbf{O}$, as follows:

- Actions: for each $i$ in $\{1, \ldots, n\}$, each operation $op_i^j \in OP_i$

- input action $\mathbf{Req}(op_i^j, x)$, for each $x \in arg(op_i^j)$,
- internal action $\mathbf{Perform}(op_i^j(x))$, for each $x \in arg(op_i^j)$,
- output action $\mathbf{Ret}(op_i^j, y)$, for each $y \in ret(op_i^j)$.

For each $i$ in $\{1, \ldots, n\}$, define $input_i(\mathbf{O})$ to be input actions of $\mathbf{O}$ indexed by $i$. Similarly, define $internal_i(\mathbf{O})$ and $output_i(\mathbf{O})$ to be the internal and output actions of $\mathbf{O}$ indexed by $i$, respectively.

- Fairness classes: there are $n$, $\{F_1, ..., F_n\}$, one for each $i$ in $\{1, \ldots, n\}$, with $F_i = internal_i(\mathbf{O}) \cup output_i(\mathbf{O})$. Thus, fairness set $F_i$ contains the internal and output actions indexed by $i$.
- States: cartesian product of $n+1$ variables: for each $i$ in $\{1, \ldots, n\}$, $Status_i$ in $\{nil\} \cup input_i(\mathbf{O}) \cup output_i(\mathbf{O})$, and $state$ in $S$.
- Initial states:
  - $S_0 \subseteq S$
  - $Status_i = nil$, for each $i$ in $\{1, \ldots, n\}$.
- Transition relation below:

| | |
|---|---|
| $\mathbf{Req}(op_i^j, x)$<br>Effect:<br>$Status_i := \mathbf{Req}(op_i^j, x)$<br><br>$\mathbf{Perform}(\mathbf{op}_i^j(x))$<br>Precondition:<br>$Status_i = \mathbf{Req}(op_i^j, x)$<br>$state = s_1$<br>$(x, s_1, y, s_2) \in spec(op_i^j)$<br>Effect:<br>$Status_i := \mathbf{Ret}(op_i^j, y)$<br>$state := s_2$ | $\mathbf{Ret}(op_i^j, y)$<br>Precondition:<br>$Status_i = \mathbf{Ret}(op_i^j, y)$<br>Effect:<br>$Status_i := nil$ |

### 4.3 A *omission-crash* version of the same object

This object captures the notion of a faulty version of the same object, which suffers an instantaneous crash. Unlike the *crash* objects of [JCT92], this crash introduces uncertainty into the behavior of operations that are concurrent with the crash. Such operations *may* suffer omission faults. But all operations which entirely follow the crash return crash, and have no impact on the object state (or on other pending operations).

Add $\perp$ to the state component $S$, and the following actions, for each $i$ in $\{1, \ldots, n\}$, each operation $op_i^j \in OP_i$:

- internal action **CRASH**,
- internal action $\mathbf{Perform}(op_i^j(\texttt{crash}))$,

- output action $\mathbf{Ret}(op_i^j, \perp)$, and
- output action $\mathbf{Ret}(op_i^j, ?)$.

The additional actions $\mathbf{Ret}(op_i^j, \perp)$ and $\mathbf{Ret}(op_i^j, ?)$ are also added as possible values of $Status_i$. The new actions indexed by $i$ are added to the fairness set $F_i$. An additional fairness set contains just **CRASH**.

$\mathbf{Req}(op_i^j, x)$
  Effect:
    $Status_i := \mathbf{Req}(op_i^j, x)$

$\mathbf{Perform}(op_i^j(x))$
  Precondition:
    $Status_i = \mathbf{Req}(op_i^j, x)$
    $state = s_1$
    $(x, s_1, y, s_2) \in spec(op_i^j)$
  Effect:
    $Status_i := \mathbf{Ret}(op_i^j, y)$
    $state := s_2$

$\mathbf{Ret}(op_i^j, y)$
  Precondition:
    $Status_i = \mathbf{Ret}(op_i^j, y)$
  Effect:
    $Status_i := nil$

**CRASH**
  Precondition:
    $state \neq \perp$
    $omit \subseteq \{1, \ldots, n\}$
  Effect:
    $state := \perp$
    $\forall k \in omit$:
      **if** $(Status_k = \mathbf{Ret}(op_i^j, y)$
      **or** $Status_k = \mathbf{Req}(op_i^j, x))$
          **then** $Status_k := \mathbf{Ret}(op_i^j, ?)$

$\mathbf{Perform}\,op_i^j(\texttt{crash})$
  Precondition:
    $Status_i = \mathbf{Req}(op_i^j, x)$
    $state = \perp$
  Effect:
    $Status_i := \mathbf{Ret}(op_i^j, \perp)$

### 4.4 A *eventual-crash* version of the same object

This faulty object is superficially less benign than the *omission-crash* object. Specifically, it suffers any number of omission failures before a final crash step. Once it has crashed, however, it behaves like the *omission-crash* object. Hence, where the *omission-crash* object suffers a single, instantaneous crash, the *eventual-crash* object suffers any finite number of omission faults before finally crashing.

As with the *omission-crash* object, $\perp$ is added to the state component $S$. The following actions, for each $i$ in $\{1, \ldots, n\}$ and each operation $op_i^j \in OP_i$, are added:

- internal action **OMISSION**,
- internal action **CRASH**,
- internal action $\mathbf{Perform}(op_i^j(\texttt{crash}))$,
- output action $\mathbf{Ret}(op_i^j, \perp)$, and
- output action $\mathbf{Ret}(op_i^j, ?)$.

The additional actions $\mathbf{Ret}(op_i^j, \perp)$ and $\mathbf{Ret}(op_i^j, ?)$ are also added as possible values of $Status_i$. The new actions indexed by $i$ are added to the fairness set $F_i$. Two additional fairness sets contain just **CRASH** and **OMISSION**, respectively.

**Req**$(op_i^j, x)$
Effect:
$Status_i := \mathbf{Req}(op_i^j, x)$

**Perform**$(op_i^j(x))$
Precondition:
$Status_i = \mathbf{Req}(op_i^j, x)$
$state = s_1$
$(x, s_1, y, s_2) \in spec(op_i^j)$
Effect:
$Status_i := \mathbf{Ret}(op_i^j, y)$
$state := s_2$

**Ret**$(op_i^j, y)$
Precondition:
$Status_i = \mathbf{Ret}(op_i^j, y)$
Effect:
$Status_i := nil$

**OMISSION**
Precondition:
$state \neq \perp$
$omit \subseteq \{1, \ldots, n\}$
Effect:
$\forall k \in omit$:
**if** $(Status_k = \mathbf{Ret}(op_i^j, y)$
**or** $Status_k = \mathbf{Req}(op_i^j, x))$
**then** $Status_k := \mathbf{Ret}(op_i^j, ?)$

**CRASH**
Precondition:
$state \neq \perp$
Effect:
$state := \perp$

**Perform**$(op_i^j(\texttt{crash}))$
Precondition:
$Status_i = \mathbf{Req}(op_i^j, x)$
$state = \perp$
Effect:
$Status_i := \mathbf{Ret}(op_i^j, \perp)$

## 4.5 Safe registers

We specify safe registers [Lam86], and define faulty safe registers. However, since these objects are not atomic, the simple paradigm for atomic objects will not work. Below is the specification of single-writer/single-reader safe registers, with process 1 the writer and 2 the reader. The only thing that distinguishes this from the atomic specifications above is the **Perform**$(\mathbf{read}_2)$ action, which allows any read operation that is concurrent with a write to return arbitrary values. Safe registers which fail by *omission-crash* and *eventual-crash* can be defined by transforming this automaton exactly as with the atomic case.

**Req**$(write_1, x)$
Effect:
$Status_1 := \mathbf{Req}(write_1, x)$

**Perform**$(\mathbf{write}_1(x))$
Precondition:
$Status_1 = \mathbf{Req}(write_1, x)$
Effect:
$Status_1 := \mathbf{Ret}(read_1, x)$
$state := x$

**Ret**$(write_1, x)$
Precondition:
$Status_1 = \mathbf{Ret}(write_1, x)$
Effect:
$Status_1 := nil$

**Req**$(read_2)$
Effect:
$Status_2 := \mathbf{Req}(read_2)$

**Perform**$(\mathbf{read}_2())$
Precondition:
$Status_2 = \mathbf{Req}(read_2)$
$(state = y)$ **or** $Status_1 \neq nil$
Effect:
$Status_2 := \mathbf{Ret}(read_2, y)$

**Ret**$(read_1, y)$
Precondition:
$Status_2 = \mathbf{Ret}(read_2, y)$
Effect:
$Status_2 := nil$