Structured Derivations of Consensus Algorithms for Failure Detectors

Jiong Yang* Gil Neiger† Eli Gafni‡

Abstract

In a seminal paper, Chandra and Toueg showed how unreliable failure detectors could allows processors to achieve consensus in asynchronous message passing systems. Since then, other researchers have developed consensus algorithms for other systems or based on different failure detectors. Each algorithm was developed and proven independently. This paper shows how a consensus algorithm for any of the standard models can be automatically converted to run in any other. These results show more clearly how the different system models and failure detectors can be related. In addition, they may permit the development of new results for new models also through transformations.

1 Introduction

The problem of achieving consensus among processors in a distributed system is fundamental in distributed computing. Unfortunately, consensus cannot be achieved in the presence of failures in completely asynchronous systems, either those with message passing [8,9] or those with shared memory [7,8,12]. This is true even for relatively benign stopping (or crash) failures. Intuitively, it is because, in such systems, it is impossible to distinguish a very slow processor from one that has failed.

Recognizing this, researchers have considered ways in which asynchronous systems might realistically be strengthened to allow consensus to be achieved. In a seminal paper [5], Chandra and Toueg studied the use of unreliable *failure detectors* for asynchronous message-passing systems. A failure detector is an oracle that gives processors some information about failures in the system; in the simplest cases, it gives a processor a list of other processors that it "suspects" have crashed. Chandra and Toueg considered a variety of failure detectors and showed that, even if the information they provide is imperfect, consensus can be achieved if the detector is sufficiently reliable.

This paper focuses on two of the failure detectors defined by Chandra and Toueg. They called the first S for "strong". It has the following properties:

- Strong Completeness. All faulty processors (those that crash) will eventually and forever be declared faulty by the failure detectors of all correct processors.

- Weak Accuracy. Some correct processor is never declared faulty by the failure detector of any processor.

The other failure detector is called $\diamond\mathsf{S}$ for "eventually strong". It eventually has the properties of S but might not initially; specifically, it supports Strong Completeness (which is already an eventual property) and the following:

- Eventual Weak Accuracy. Eventually, there is some correct processor that is thenceforth not declared faulty by the failure detector of any correct processor.

Chandra and Toueg used these failure detectors to implement the following algorithms:

- A consensus algorithm using S. It tolerates any number of failures. Its running time is $O(n)$, where n is the number of processors in the system.

- A consensus algorithm using $\diamond\mathsf{S}$. It can tolerate f failures so long as n is greater than $2f$. Although the

*Department of Computer Science, University of California at Los Angeles; jyang@cs.ucla.edu

†MicroComputer Research Labs, Intel Corporation, Hillsboro, Oregon; gil@cse.ogi.edu

‡Department of Computer Science, University of California at Los Angeles; eli@cs.ucla.edu

algorithm terminates in all runs, its running time cannot be bounded as it depends on how $\diamond\mathsf{S}$ behaves in a given execution (specifically, on how quickly Eventual Weak Accuracy becomes Weak Accuracy).

Lo and Hadzilacos [11] considered the use of these failure detectors in shared-memory systems. They obtained algorithms similar to those developed by Chandra and Toueg for message-passing systems. As before, the algorithm with S runs in time $O(n)$ while the running time of the algorithm with $\diamond\mathsf{S}$ is unbounded. They noted an important difference between their algorithm using $\diamond\mathsf{S}$ and that of Chandra and Toueg. While the message-passing algorithm requires $n > 2f$, the shared-memory algorithm is *wait-free*: like the two S-based algorithms, it can tolerate any number of failures.

More recently, Neiger [14] studied shared-memory systems augmented by more powerful objects. Specifically, he explored the failure detectors necessary for n processors to solve consensus if they have access to objects that can solve consensus for $i < n$ processors (for example, test-and-set bits can be used to solve consensus for 2 processors). He developed a hierarchy of failure detectors Ω_i (weaker than the others mentioned so far), one for each choice of i. He showed how Ω_i could be used to solve consensus if processors have access to objects that can solve consensus for i processors.

The algorithms presented in these three papers are similar in many ways. For example, each of the algorithms based on S has processors go through a bounded series of rounds, where a processor does not pass on to a new round until it has received communication (either by receiving a message or reading a memory location) from all processors that S does not report faulty. In contrast, the algorithms based on $\diamond\mathsf{S}$ (or on Ω_i) are all *coordinator-based*: processors go through rounds each of which is "coordinated" by a specific processor (or, for Ω_i, a set of i processors). The algorithms are such that every processor (or set of i processors) gets to coordinate infinitely often.

The correctness proofs of these algorithms are also similar. For example, the proofs of the algorithms using $\diamond\mathsf{S}$ (or Ω_i) use the following reasoning. The failure detector guarantees that eventually there some correct processor (or, for Ω_i, a set of i processors that contains a correct processor) that no processor suspects as being faulty. This processor (or set of processors) can then force termination and agreement the next time it coordinates a round after this point.

Despite the similarities between these algorithms and their proofs, they were each derived and proven independently. The goal of this paper is to unify these results by showing how a consensus algorithm developed for one model (i.e., system and failure-detector type) can be transformed automatically into one for a different model.

This paper focuses on transformation of *algorithms* and not of the failure detectors. This contrasts with work of Chandra and Toueg. Their paper (which considered only message-passing systems) studied when one failure detector might be used to implement another. If failure detector A can implement B in a particular system, then any algorithm for that system that uses B can be transformed into one that uses A. Much of the current paper gives transformations where the system changes (as well as or instead of changing the failure detector). One exception is the result of Section 3, which converts converts shared-memory consensus algorithms for S to use $\diamond\mathsf{S}$. This result is *not* an implementation of S by $\diamond\mathsf{S}$; in fact, Chandra and Toueg proved that such an implementation is impossible.

The balance of the paper is organized as follows. Section 2 presents definitions and notation. Section 3 describes how to transform shared-memory consensus algorithms for S so that they require only $\diamond\mathsf{S}$ (actually, the result is for k-set agreement, which is a generalization of consensus). Section 4 describes how to transform shared-memory algorithms for $\diamond\mathsf{S}$ to run with message passing; Section 5 does the same for S. Section 6 shows how to transform a message-passing algorithm for $\diamond\mathsf{S}$ that requires $n > 2f$ into a shared-memory algorithm that is wait-free. Transformations between other pairs of models are discussed in Section 7. Section 8 presents results for systems with consensus objects. Section 9 explains how work on failure detectors can be systematically extended to consider k-set agreement. Section 10 presents conclusions.

2 Definitions

This section presents the definitions and background necessary to present and interpret the results of this paper.

A system consists of a set of n processors. In some systems, processors interact by sending messages; in others, they communicate through read/write memory. The systems considered here are fully *asynchronous*. There are no upper and lower bounds on the execution speeds of processors or on the time it takes a message to be delivered. In addition, any processor may fail by crashing: at a certain point in its execution, a faulty processor may simply stop. This paper makes a standard definition of failures: a processor is faulty if it takes only finitely many steps in an infinite execution.

The consensus problem specifies that the processors, each of which begins with some input value, agree on a output value that was one processor's input value. Specifically, all executions of a consensus algorithm have the following properties:

- Termination. Every correct processor eventually and irrevocably decides upon an output value.

- Validity. Any processor that decides chooses the input value of some processor.
- Agreement. All processors that decide choose the same value.

Sections 3 and 9 consider algorithms for k-set agreement [6]. This is a generalization of consensus in which Agreement is relaxed to the following:

- k-Agreement. At most k distinct values are chosen by the processes.

Note that Agreement is equivalent to 1-Agreement, so Consensus is the same as 1-set agreement.

An algorithm is *wait-free* if any participating processor can complete the algorithm in a finite number of steps regardless of the behavior (e.g., speed or failure) of the other processors. Wait-free algorithms can thus tolerate any number of failures. This paper also considers algorithms in which the number of failures is bounded.

As noted above, processors may fail by stopping (or crashing). Normally, other processes are not directly aware of these failures. *Failure detectors* allow processors to gain additional knowledge about failures. Chandra and Toueg [5] showed that, in order to solve consensus, this additional knowledge need not be perfect or even correct.

A failure detector is defined by specifying, for each failure pattern (i.e., which processes crash at what times), a set of allowable *failure-detector histories*. Each of these indicates, for each process and each time, the value returned to that process if it queries the failure detector at that time. For any particular failure pattern, a single failure detector may return different information to different processes at the same time, or even to the same process at the same time in different executions with the same failure pattern. The failures detectors S and $\Diamond\mathsf{S}$, defined informally in Section 1, can be formally defined in this model.

3 From S to $\Diamond\mathsf{S}$ for Shared Memory

This section shows how to convert shared-memory solutions to k-set agreement, designed for use with S, into ones that require only $\Diamond\mathsf{S}$. For the case of $k = 1$, this result thus applies to consensus algorithms.

Section 3.1 gives an overview of the transformation, which uses a subroutine called *k-Converge*. Section 3.2 gives an implementation of *k-Converge*.

3.1 The Transformation

The transformation does not apply to arbitrary algorithms for k-set agreement. It applies to any algorithm for S such that all executions of the algorithm using $\Diamond\mathsf{S}$ instead satisfy Termination and Validity (but not necessarily k-Agreement). Existing consensus algorithms that use S [5,11] meet this requirement (for either shared-memory or message-passing).[1]

The general idea behind the transformation is the following. Let A be a shared-memory k-set agreement algorithm for S. The algorithm for $\Diamond\mathsf{S}$, called A', runs A repeatedly, with a processor's output in one execution being used as its input for the next. After a certain point in time, $\Diamond\mathsf{S}$ will behave as S (that is, Weak Accuracy will hold from that point onward). Because A correctly solves k-set agreement when run with S, the next execution of A to run will guarantee not only Termination and Validity but also k-Agreement.

The problem with this naïve approach is that, while each invoked execution of A satisfies Termination, A' itself does not: so far, we have not specified when A' should stop running A. In fact, we have not specified how processors running A' choose their output values!

Termination and output in A' are controlled through use of a special subroutine called *k-Converge*. Processors call *k-Converge* as they would a consensus algorithm, passing it an input value. It returns a pair $\langle c, v\rangle$, where c is a boolean and v is a value. If a processor returns $\langle c, v\rangle$, it is said to *pick* value v. If $c = \textit{true}$, the processor *commits* to the value. Every execution of *k-Converge* must support the following properties regardless of how many processors participate:

- Convergent Termination. Every correct processor picks some value.
- Convergent Validity. If a processor picks v, then some processor invoked *k-Converge* with v.
- Convergent k-Agreement. If some processor commits to a value, then at most k values are picked.
- k-Convergence. If there are most k values v such that some processor calls *k-Converge* with input v, then every processor that picks a value commits to one of these k.

An implementation of *k-Converge* is given in Section 3.2.

The transformed algorithm A' is given in Figure 1 and consists of two coroutines that run in parallel. In the first coroutine, A is executed repeatedly. After the ith execution of A, a processor calls *k-Converge*$[i]$, using as input the output of the previous run of A. (The bracketed "i" emphasizes that each round's execution of *k-Converge* is

[1] Existing algorithms using S pass through a fixed number of rounds and then halt. Given the non-eventual nature of Weak Accuracy, it seems plausible that future k-set agreement algorithms for S will also likely halt in a fixed number of rounds, guaranteeing Termination even when run with $\Diamond\mathsf{S}$. (It seems likely that only contrived algorithms, designed specifically not to halt if it is detected that S is not being used, would not support Termination.) Validity is easy to satisfy if processors fail only by crashing.

```
algorithm A′(v)          // p solves k-set agreement with input v
  cobegin
    i = 0                                         // first coroutine
    repeat
      // invoke original k-set agreement algorithm
      v = A(v)
      // attempt to converge
      ⟨h, v⟩ = k-Converge[i](p, v)
      i = i + 1
    until h
    // each choice[j] initialized to ⊥
    choice[p] = v
    decide v and halt
  ||
    j = 0                                         // second coroutine
    repeat
      j = 1 + (j mod n)
      u = choice[j]
    until u ≠ ⊥
    choice[p] = u
    decide u and halt
  coend
```

Figure 1: k-Set Agreement Algorithm A' for $\diamond\mathsf{S}$

```
function k-Converge(p, in)
  // uses shared-memory arrays a[1...n] and b[1...n]
  // each element is initialized to ⊥
  a[p] = in
  for q = 1 to n                                  // read a array
    r[q] = a[q]
  if |{r[q] | r[q] ≠ ⊥}| ≤ k then
    b[p] = true
  else
    b[p] = false
  for q = 1 to n                                  // read b array
    s[q] = b[q]
  if (∀q)(s[q] ≠ false) then
    return ⟨true, in⟩
  else if (∃q)(s[q] = true) then
    return ⟨false, a[q]⟩ such that s[q] = true
  else
    return ⟨false, in⟩
```

Figure 2: Function k-*Converge*

implemented separately.) If it commits to a value, then the processor marks this fact in an associated *choice* variable; it then decides on the committed value and terminates A'. If it does not commit to a value, it uses the picked value as input to the next execution of A.

In the second coroutine, processors periodically check the *choice* variables, deciding and halting if they find another processor has already done so. (This coroutine is employed to compensate for the fact that a correct processor that decides will halt and not participate in a later execution of A run by others.)

We now argue for the correctness of A'. Validity is clear because A supports Validity (even when $\diamond\mathsf{S}$ is used) and because k-*Converge* supports Convergent Validity. To prove k-Agreement, it suffices to consider only processors that set *choice* in the first coroutine, i.e., after committing to a value in some k-*Converge*$[i]$. Let p be a processor that does so with the least value of i and suppose that it decides v. By Convergent k-Agreement, at most k values are picked in k-*Converge*$[i]$; let K be the set of these values. Thus, all processors that run A for the $(i+1)$st time use values from K as their inputs. If that execution does not terminate (perhaps because earlier deciding processors do not participate), then processors will eventually find a value from K in some processor's *choice* variable and will then decide on that value. If the last execution of A does terminate, it outputs values from K for all participants (because of Validity, which holds even when $\diamond\mathsf{S}$ is used). They will thus all use values from K as input to k-*Converge*$[i+1]$. Since there are at most k of these, k-Convergence ensures that all processors commit to one of these values and decide on it in A'. (Recall that k-*Converge* is correct regardless of how many processors participate, so k-*Converge*$[i+1]$ will support k-Convergence, even if some processors "drop out" after k-*Converge*$[i]$.)

The Termination of A' is proven by contradiction. Suppose that some correct processor never terminates A'. This implies that no processor ever sets its *choice* variable and thus no processor ever commits in an execution of k-*Converge*. This means that processors execute A an infinite number of times (Convergent Termination ensures that no processor blocks when executing k-*Converge*). Because $\diamond\mathsf{S}$ eventually behaves like S, there is some execution of A that effectively runs with S. Because A solves k-set agreement when run with S, all correct processors will end that execution with one of at most k values. By k-Convergence, they will all commit to one of those values in the next execution of k-*Converge* and A' will terminate, a contradiction.

It should be noted that this transformation does not itself use failure detectors. The detectors are used only when called by the algorithm being transformed.

3.2 An Implementation of k-*Converge*

The implementation of k-*Converge* is given in Figure 2. It uses two arrays of shared-memory cells (a and b), each with one element for each processor.[2] Each processor first writes its value to the a array and then reads the entire array. It then writes a bit to the b array that is true if and only if the processor read at most k non-$\perp$ values from the a array. Finally, it reads the entire b array. If the only non-$\perp$ values there are *true*, it commits to its original value. If there are a mix of *true* and *false*, then it picks the original value (from the a array) of some process that wrote *true*. If *false* is the only non-$\perp$ value in the b array, then it picks its original value.

We now prove that the implementation has the properties specified in Section 3. Convergent Termination is obvious. Convergent Validity easily follows from the fact that every processor picks either its own initial value or the initial value some processor wrote to the a array. k-Convergence is also straightforward: if there are at most k initial values, then each correct processor will write *true* to the b array and will thus commit to its initial value.

Define value v to be *proposed* if there is some processor p such that p writes $a[p] = v$ and $b[p] = \mathit{true}$. It should be clear that only first k (distinct) values written to the a array can be proposed. Obviously, processors commit only to proposed values. To show that Convergent k-Agreement holds, assume that processor p commits to v and that processor q picks (but does not commit to) v'. This means that p saw only *true* values in the b array but that q saw at least one *false* value. This implies that q saw $b[p] = \mathit{true}$ and thus picks a proposed value. Since all picked values are proposed values, there are at most k picked values.

4 From Shared Memory to Message Passing for $\Diamond\mathsf{S}$

This section shows how to convert shared-memory algorithms designed for use with $\Diamond\mathsf{S}$ into message-passing algorithms that also use $\Diamond\mathsf{S}$.

We begin by making an important observation. Chandra and Toueg proved that there can be no message-passing consensus algorithm with $\Diamond\mathsf{S}$ if $n \leq 2f$ [5,Theorem 6.3.1]. Thus, no transformation that yields such an algorithm can exist if $n \leq 2f$. Our transformation thus requires $n > 2f$.

The requirement $n > 2f$ is exactly what is required to simulate shared memory in a message-passing system. Such a simulation was developed by Attiya, Bar-Noy, and Dolev [2] and later improved by Attiya [1]. All that is necessary to transform a shared-memory algorithm for $\Diamond\mathsf{S}$ into a message-passing algorithm for $\Diamond\mathsf{S}$ is to use either simulation.

Notice that this transformation applies to any shared-memory algorithm that uses $\Diamond\mathsf{S}$ so long as the resulting algorithm need not be correct with $n \leq 2f$. This contrasts with transformation of Section 3, which applies only to algorithms for k-set agreement.

5 From Shared Memory to Message Passing for S

This section shows how to convert shared-memory consensus algorithms designed for use with S into message-passing algorithms that also use S.

Like the transformation of Section 4, this transformation simulates shared-memory cells in a message-passing system. In fact, if $n > 2f$, the implementation of that section (due to Attiya [1]) suffices here as well. However, unlike the situation with failure detector $\Diamond\mathsf{S}$, message-passing consensus algorithms do exist for S even if $n \leq 2f$ (for example, Chandra and Toueg [5] presented such an algorithm that can tolerate any number of failures). This section presents a transformation to message-passing systems with S with the following property: any algorithm produced can tolerate as many failures as the one from which it was transformed. The transformation thus applies to wait-free shared-memory algorithms, such as the S-based shared-memory consensus algorithm of Lo and Hadzilacos [11].

As noted in Section 4, Attiya's implementation of shared-memory requires $n > 2f$. An examination of the implementation explains the requirement and points to an alternative implementation (using S) without the requirement. The following is a highly simplified description of Attiya's implementation. To simulate a write to a location, a processor sends the value to be written (with a timestamp) to all n processors. It then waits for acknowledgments from a majority before proceeding. In order to read from a location, a processor sends a request to all processors and waits for responses from a majority. It then chooses the value with the highest timestamp.

The correctness proof exploits two important facts. First, the requirement $n > 2f$ ensures that a processor will not block forever while waiting for responses from a majority of processors. Second, the fact that every response set is a majority ensures all pairs of such sets intersect. This intersection property ensures that the shared memory is maintained consistently. Our modified implementation uses S to prevent blocking and to ensure the intersection property without requiring $n > 2f$.

Attiya's implementation is modified as follows. Rather than waiting for a majority of processors to respond to a message, a processor waits until it gets responses from all processors not reported faulty by S. While Attiya's implementation used the requirement $n > 2f$ to ensure that this waiting terminated, the modification uses the Strong Completeness of S. Any faulty processor is eventually reported by S and any correct processor eventually responds. Thus, the waiting always terminates. While Attiya's im-

[2] In Figure 1, processor invoke *k-Converge* once for each iteration in the first coroutine. The successive invocations of *k-Converge* are implemented independently, each using its own two shared-memory arrays.

plementation used the fact that all majorities intersect to ensure the intersection property, the modification uses the Weak Accuracy of S. This guarantees that there is some correct processor p that is never reported as faulty by S (at any processor). This means that every pair of response sets intersect because they all contain p. This is sufficient for the implementation of shared memory to be correct.

Note certain distinctions between this transformation and those given earlier. Unlike the one given in Section 3, this transformation applies to all shared-memory algorithms that use S and not just to algorithms for k-set agreement. The transformations in Sections 3 and 4 did not themselves use a failure detector; the detector was used only by the transformed algorithm. In contrast, the failure detector S is used here in the modified implementation of shared memory in addition to any use by the transformed algorithm.

6 Transformations That Increase Fault Tolerance

All the transformations presented so far preserve the fault tolerance of the algorithms transformed. That is, if the original algorithm is correct for a particular choice of n and f, then so is the resulting algorithm. (The fact that some transformations apply only if $n > 2f$ does not change this fact.)

This section considers transformations that actually increase the fault tolerance of a consensus algorithm by generating one that tolerates more failures. Such transformations are of most interest if they convert algorithms for message-passing systems with $\diamond$S because these systems admit consensus algorithms only if $n > 2f$ (other systems admit wait-free algorithms).

Lo and Hadzilacos [11] developed a shared-memory algorithm for $\diamond$S that tolerates any number of failures. They considered why this is possible here while it is not with message-passing systems. In doing so, they reasoned about the nature of their algorithm and of the message-passing algorithm of Chandra and Toueg [5], which requires $n > 2f$. More fundamentally, the distinction relates not to specifics of the algorithms, but to the nature of the two systems and when a simulation is possible between them (see Sections 4 and 5 above). If $n > 2f$ or if S is available, then shared memory can be simulated by message passing. If $n \leq 2f$ and only $\diamond$S is available, this is not possible.

We now describe a transformation from message passing and $\diamond$S to shared memory and $\diamond$S.[3] The transformation increases the fault tolerance of the algorithms to which it is applied. Specifically, it takes a message-passing algorithm for $\diamond$S that requires $n > 2f$ and produces a shared-memory algorithm for $\diamond$S that tolerates any number of failures. (We assume that the input algorithm is parameterized by n and f. Assume that the algorithm has some fault-tolerance function p such that it is correct so long as $n > p(f)$. Clearly, $p(f) \geq 2f$ for all f.) The transformation is based on a modification of the simulation technique of Borowsky and Gafni [4,13].

At a high level the transformation operates as follows. There are n processors (simulators—$s_1, \ldots, s_n$) that seek to simulate a message-passing algorithm for m processors (targets—$t_1, \ldots, t_m$). The simulators collaborate to simulate the targets' execution of a desired algorithm. Each simulator cycles through the targets, trying to simulate a step of one and then going on to the next.[4] For the simulators to consistently simulate the targets, the execution of each step by a simulator has two parts: a wait-free part followed by a waiting part. It is the waiting part in which the simulators agree on how to simulate this step. If a simulator crashes while executing the wait-free part of a step, then other simulators may block in the waiting part of that step (the step's target will appear to crash in the simulation). To ensure progress, a simulator in the waiting part of a step may choose to simulate a step of another target (it cannot do so while executing the wait-free part of a step). In this way, the failure of f simulators blocks the simulation of at most f targets.

Our transformation works as follows. Suppose that we wish to construct a wait-free shared-memory algorithm with $\diamond$S for n processors. The n processors will function as simulators of a message-passing algorithm A with $\diamond$S with $m = p(2n-2)$ targets; recall that A tolerates $2n-2$ failures. The simulation is different from that developed by Borowsky and Gafni in the following way. Each target t_i, $1 \leq i \leq n$, is simulated only by simulator s_i; these are called *dedicated targets*. (Note that $p(2n-2) > 2(2n-2)$ and $n > 1$ imply $m = p(2n-2) > n$.) The targets t_i, $i > n$, are simulated by all simulators as before; these are called *shared targets*. A simulator alternates between simulating its dedicated target and the shared targets. A dedicated target crashes in the simulation only if its simulator does.

The Borowsky-Gafni simulation was not designed for systems with failure detectors. Thus, our transformation must describe what a simulator does when the target it is simulating consults a failure detector. To do this, we augment the original simulation in the following way. Each simulator has a shared variable *running* that

[3] The transformation given here also applies to algorithms with S and can transform shared-memory algorithms as well as message-passing algorithms. We concentrate on message-passing algorithms with $\diamond$S because no such wait-free algorithms can exist. There is more need to convert such algorithms into wait-free ones for systems that allow such solutions.

[4] Borowsky and Gafni carefully define what is meant here by "step". In their case, the step refers to operations on shared memory. Because we are simulating a message-passing algorithm that uses a failure detector, a "step" will be slightly different, but not in any significant way.

indicates which of the the shared targets it is simulating. Specifically, it sets its *running* to a target's name before it executes the wait-free part of a step of that target. It sets *running* to $\perp$ when it finishes that part and before it executes the waiting part of the step.

When a simulator finds that a target wants to consult failure detector $\Diamond\mathsf{S}$, it begins by consulting its own $\Diamond\mathsf{S}$. This detector gives a list of simulators (not targets) that are suspected as being faulty. The simulator then generates a list of suspected targets as follows. For each simulator in the original list, its dedicated target is included in the new list. Moreover, if a simulator is suspected and its *running* variable points to a shared target, then that target is also added to the list. The resulting list of targets is then used for $\Diamond\mathsf{S}$ in the simulation.[5]

We now see that this simulated failure detector has the properties of $\Diamond\mathsf{S}$. Consider first Strong Completeness—all targets faulty in the simulation must eventually be permanently suspected. A dedicated target crashes in the simulation only if its simulator crashes. By Strong Completeness of the actual detector, this simulator is eventually and permanently detected. By the simulation, the dedicated target is detected in the simulation. A shared target crashes in the simulation only if if some simulator crashes in the wait-free part of one of that target's steps (this is a property of the Borowsky-Gafni simulation). Again, the failure of this simulator is eventually and permanently detected. The simulator's *running* variable will "point" to the blocked target, which will then be detected in the simulation.

For Eventual Weak Accuracy, there must be some target that is correct and whose failure is never suspected (in the simulation). By the Eventual Weak Accuracy of the actual detector, there is some correct simulator whose failure is eventually never suspected. The dedicated target of that simulator behaves correctly and is eventually never suspected in the simulation. Thus, the simulated detector behaves as $\Diamond\mathsf{S}$.

Finally, note that each faulty simulator can cause at most two targets to crash in the simulation: its dedicated target and possibly one shared target (the latter is a property of the Borowsky-Gafni simulation). Since at most $n-1$ of the n simulators can fail, at most $2n-2$ targets can fail in the simulation. Since the original message-passing algorithm (for $p(2n-2)$ targets) tolerates $2n-2$ failures, the transformed algorithm (which simulates it) will correctly solve consensus.

[5]For any given step of a shared target, the simulators must agree on the value returned by $\Diamond\mathsf{S}$. However, the waiting part of a step of the Borowsky-Gafni simulation is designed to provide this kind of agreement.

Table 1: Summary of Transformations

To	From A	From B	From C	From D
A	-	triv	M $\Rightarrow$ S	M $\Rightarrow$ S
B	§3*	-	M $\Rightarrow$ S + §3*	M $\Rightarrow$ S
C	§5	§5	-	triv
D	§3* + §4†	§4†	M $\Rightarrow$ S + §3* + §4†	-

A: Shared memory with S

B: Shared memory with $\Diamond\mathsf{S}$

C: Message passing with S

D: Message passing with $\Diamond\mathsf{S}$

* k-set agreement and consensus only

† requires $n > 2f$

7 Trivial and Composed Transformations

The previous sections considered transformations between some of the models considered in this papers. For some pairs of models, there are trivial or straightforward transformations. For others, there are appropriate transformations that are compositions of those presented here. All these transformations are summarized in Table 1 and discussed below.

The trivial or straightforward transformations are the following:

1. From $\Diamond\mathsf{S}$ to S (with either message passing or shared memory). Any algorithm that is correct when run with $\Diamond\mathsf{S}$ must behave correctly when run with S. This is because the behaviors of S are a subset of the behaviors of $\Diamond\mathsf{S}$. Thus, there is a trivial transformation here. This transformation in marked "triv" in Table 1.

2. From message passing to shared memory (either with S or with $\Diamond\mathsf{S}$). Such a simulation is straightforward. A dedicated shared variable can be used to implement each directed interprocessor link. This transformation is marked "M $\Rightarrow$ S" in Table 1.

Given these and the transformations described earlier, there are transformations among all the models (either directly or through composition). For composed transformations, Table 1 lists the constituent transformations in the order in which they are applied.

Notice that, of all the transformations presented in the previous sections, only the one in Section 4 has limited applicability in that it applies only if $n > 2f$. The composed transformations given in Table 1 use this transformation only when the final target system is message passing with $\Diamond\mathsf{S}$ (system D in the table). Thus, the composed

Table 2: Transformations that Increase Fault Tolerance

To	From A	From B	From C	From D
A	§6			
B	§6 + §3*	§6	§6 + §3*	§6
C	§6 + §5			
D	impossible			

A: Shared memory with S

B: Shared memory with $\diamond$S

C: Message passing with S

D: Message passing with $\diamond$S

* k-set agreement and consensus only

transformations have limited applicability only when this is unavoidable.

Table 1 considers only transformations that retain the fault tolerance of the algorithms being transformed. Section 6 showed how algorithms could be converted to increase the fault tolerance of an algorithm that requires $n > 2f$ so that it is wait-free. Table 2 summarized transformations that do this, including those that result from the composition of transformations. This table is quite simple because the transformation given in Section 6 applies to a broad variety of systems (see footnote 3 above).

8 Transformations with Consensus Objects

As noted in Section 1, Neiger [14] developed consensus algorithms for systems with objects that can be used to implement consensus for a fixed number i of processors (henceforth, *i-consensus objects*). Neiger's algorithms implement consensus for any number of processors. They use failure detectors weaker than those required for shared memory or for message passing.

While Neiger's algorithm resembles the $\diamond$S-based algorithm of Lo and Hadzilacos [11], it is distinct and required its own correctness proof. This section presents a transformation that can be used to convert shared-memory algorithms for $\diamond$S into algorithms using consensus objects and one of Neiger's failure detectors. The transformation is general and does not apply only to consensus algorithms.

Neiger defined a class of failure detectors Ω_i. The output of Ω_i is always a set of size at most i. In contrast to S and $\diamond$S, this is a set of *trusted*, not suspected, processors. In each execution, there is a set G of at most i processors, containing at least one correct processor, such that the failure detectors of all correct processors eventually always output G. (Ω_1 can easily implement $\diamond$S.)

We now present the transformation. Suppose that A is wait-free consensus algorithm for shared-memory that uses $\diamond$S. We convert it into A', a wait-free algorithm using shared memory, i-consensus objects, and Ω_i. Suppose that A' is to run for n processors; these processors will be simulators for $\binom{n}{i}$ targets. These targets are neither dedicated nor completely shared as in the transformation of Section 6. Rather, each set of i simulators is responsible for one of the targets. In contrast to the Borowsky-Gafni technique (Section 6), the i simulators of a target can use i-consensus objects to agree on its simulation. Thus, a target crashes in the simulation only if *all* of its simulators crash. As in Section 6, a simulator takes turns simulating the different targets for which it is responsible.

Suppose that simulator s is simulating target t and t wants to consult $\diamond$S. To simulate this, s consults its own Ω_i, obtaining a set T of trusted simulators. Recall that each target is associated with a set of i simulators. Simulator s will consider a target faulty if T is a not a subset of the target's simulators. (Actually, t's simulators—including s—use an i-consensus object to select a single $\diamond$S output from those proposed by the simulators.)

The properties of Ω_i can be used to show that the simulated failure detector has the properties of $\diamond$S. For Strong Completeness, consider a target t that crashes in the simulation. Because there is no blocking in the simulation, this can happen only if all i of t's simulators crash. By the properties of Ω_i, all correct simulators eventually trust a set of i simulators, at least one of which is correct. Because that processor does not crash, it is not one of t's simulators. By the simulation, $\diamond$S always reports t faulty after this point. Thus, Strong Completeness is satisfied.

For Eventual Weak Accuracy, consider the set G that Ω_i eventually always trusts. Since $|G| \leq i$, there is at least one target t such that all simulators in G simulate t. Because G contains at least one correct processor, t will be correct in the simulation. Once all simulators trust G, target t will never be suspected in the simulation. Thus, Eventual Weak Accuracy is simulated.

A is thus properly simulated by A' because $\diamond$S is properly simulated. Since A achieves consensus, A' does also.

Neiger's algorithm using Ω_i is similar to that of Lo and Hadzilacos for $\diamond$S. However, the result of applying our transformation to the latter does not yield the former. The processors in the algorithm we produce cooperate more closely: for example, the simulators of target agree, with each step, on the output of $\diamond$S. Neiger's algorithm does not do this.

9 New Failure Detectors for k-Set Agreement

Section 3 gave a shared-memory transformation from S to $\diamond$S that applied both the consensus algorithms and to solutions to k-set agreement. However, there has been little research on using failure detectors to solve k-set agree-

```
algorithm KSMP(v)            // p solves k-set agreement with v
    for i = 1 to n do
        if p = p_i then
            send v to all
        else
            wait until received m from p_i or p_i ∈ S_k
            if p_i ∉ S_k then
                v = m
    decide v
```

Figure 3: k-Set Agreement Algorithm for S_k

ment. (Gafni [10] shows how k-set agreement can be solved in systems defined by certain *round-by-round fault detectors*.) One reason for this is that the failure detectors traditionally studied (e.g., S and $\diamond$) can be used to solve consensus, which is a hard problem. This section defines a set of weaker failure detectors that generalize S and $\diamond\mathsf{S}$, just as k-set agreement generalizes consensus.

The failure detector S was defined in Section 1 to satisfy Strong Completeness and Weak Accuracy. Suppose now that Weak Accuracy is weakened as follows:

- Weak k-Accuracy. There is some correct processor p such that the failure detectors of at most $k-1$ processor ever declare p as faulty.

Define S_k to be a failure detector satisfying Strong Completeness and Weak k-Accuracy. Note that Weak Accuracy (from Section 1) is Weak 1-Accuracy and thus S is S_1.

Similarly, we can define an eventual property:

- Eventual Weak k-Accuracy. Eventually, there is some correct processor p such that the failure detectors of at most $k-1$ processor ever declare p as faulty.

Define $\diamond\mathsf{S}_k$ to be a failure detector satisfying Strong Completeness and Eventual Weak k-Accuracy. Again, Eventual Weak Accuracy is Eventual Weak 1-Accuracy and $\diamond\mathsf{S}$ is $\diamond\mathsf{S}_1$.

Figure 3 gives a message-passing algorithm *KSMP* for k-set agreement that uses S_k and tolerates any number of failures. It is not hard to see that this algorithms solves k-set agreement. Termination follows because there are a bounded number of rounds and each round must terminate because of the Strong Completeness of S_k. Validity is obvious as the set of values never grows from one round to another. To prove k-Agreement, let p_i be the correct processor that at most $k-1$ processors ever suspect. The processors that never suspect p_i all end round i with the same value. There are at most $k-1$ other processors and thus at most $k-1$ other values. Since the set of values never grows, there are at most k different values when the algorithm ends. It should be clear that there is an equally simple shared-memory algorithm *KSSM*.

Notice that the algorithm in Figure 3 has the properties required by the transformation given in Section 3 (it satisfies Termination and Validity even when run with $\diamond\mathsf{S}_k$). Because $\diamond\mathsf{S}_k$ eventually becomes S_k, it is not hard to see that that transformation is correct: if applied to *KSSM*, it yields a shared-memory k-set agreement algorithm for $\diamond\mathsf{S}_k$. Similarly, the composed message-passing transformation from S to $\diamond\mathsf{S}$ (see Table 1) can convert *KSMP* to run with $\diamond\mathsf{S}_k$ if $n > 2f$ (it uses the transformation of Section 4).

The limitation of $n > 2f$ was acceptable when considering message-passing solutions to consensus with $\diamond\mathsf{S}$ because Chandra and Toueg showed that solutions are impossible if $n \leq 2f$ [5]. However, when similar arguments are generalized to k-set agreement, the following result is obtained:

Theorem 1 *There is no asynchronous message-passing solution to k-set agreement with $\diamond\mathsf{S}_k$ if $kn \leq (k+1)f$.*[6]

Proof Sketch: Assume $kn \leq (k+1)f$; note that this implies $n \geq (k+1)(n-f)$. Assume also that algorithm A solves k-set agreement. Divide the n processes into $k+1$ groups of size at least $n-f$ and suppose that there are no initial values in common between any two groups. Allow A to run in an environment in which $\diamond\mathsf{S}_k$ declines to suspect any processes until after A terminates. (The Termination of A and the eventual nature of $\diamond\mathsf{S}_k$ allows this.) Then each group will choose at least one of its initial values. Since there are initial values in common between the groups, at least $k+1$ values are chosen. This contradicts the assumption that A solves k-set agreement. □

This raises the following question: can $\diamond\mathsf{S}_k$ be used to solve k-set agreement in a message-passing system with $n \leq 2f$ and $kn > (k+1)f$ (e.g., 2-set agreement with $n = 5$ and $f = 3$)?

We conjecture that the answer is yes. Rather than developing an algorithm from scratch, we seek to use the methods developed in this paper. In particular, we hope to use techniques similar to those given in Section 3. It should be clear that the algorithm in Figure 1 can easily be modified to run with message passing so long as an implementation of *k-Converge* is available. The shared-memory implementation of *k-Converge* given in Figure 2 can be adapted to message passing if $n > 2f$. Proving our conjecture requires an implementation that is correct

[6] Actually, the result holds for any failure detector all of whose properties hold only eventually, including the eventually perfect failure detector of Chandra and Toeug.

with $kn > (k+1)f$. Based on earlier work of Bazzi and Neiger [3], we conjecture that this can be done with $2k$ asynchronous rounds of message exchange.

10 Conclusions and Future Work

This paper exhibited a series of transformations that convert consensus algorithms from one model to another. The models may vary by system type (shared-memory versus message-passing) or by failure detector. To see that these transformations are meaningful, it is important to see that they are not performing consensus themselves. If they were, they could effectively "throw away" their input algorithm, replacing it with "canned" consensus implementation.

The transformations of Section 3, 4, 6, and 8 do not themselves use failure detectors. Because consensus is impossible in asynchronous systems without failure detectors, these transformations cannot be concealing a consensus algorithm. The transformation of Section 5 is simply an adaptation of Attiya's simulation of shared memory and clearly does not conceal a consensus algorithm any more than her original simulation does. We elaborate upon this in the full version of the paper.

Section 9 defines hierarchies of failure detectors S_k and $\diamond\mathsf{S}_k$. It gives shared-memory and message-passing solutions to k-set agreement for S_k. It shows how these can be converted into $\diamond\mathsf{S}_k$ solutions for shared memory and, if $n > 2f$, for message passing. It shows that there can be no such message-passing solutions if $kn \leq (k+1)f$. It leaves open the possibility of message-passing $\diamond\mathsf{S}_k$-solutions if $kn > (k+1)f$ and $n \leq 2f$.

Acknowledgments

Vassos Hadzilacos was our pillar of fire through the land-mined desert of failure detectors. His gracious guidance was invaluable.

References

[1] ATTIYA, H. Efficient and robust sharing of memory in message-passing systems. In *Proceedings of the Tenth International Workshop on Distributed Algorithms*, O. Babaoğlu and K. Marzullo, Eds., no. 1151 in Lecture Notes on Computer Science. Springer-Verlag, Oct. 1996, pp. 56–70.

[2] ATTIYA, H., BAR-NOY, A., AND DOLEV, D. Sharing memory robustly in message-passing systems. *J. ACM 42*, 1 (Jan. 1995), 124–142.

[3] BAZZI, R. A., AND NEIGER, G. Simulating crash failures with many faulty processors. In *Proceedings of the Sixth International Workshop on Distributed Algorithms*, A. Segall and S. Zaks, Eds., no. 647 in Lecture Notes on Computer Science. Springer-Verlag, Nov. 1992, pp. 166–184.

[4] BOROWSKY, E., AND GAFNI, E. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth ACM Symposium on Theory of Computing* (May 1993), ACM Press, pp. 91–100.

[5] CHANDRA, T. D., AND TOUEG, S. Unreliable failure detectors for asynchronous systems. *J. ACM 43*, 2 (Mar. 1996), 225–267.

[6] CHAUDHURI, S. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. *Information and Computation 103*, 1 (July 1993), 132–158.

[7] CHOR, B., ISRAELI, A., AND LI, M. Wait-free consensus using asynchronous hardware. *SIAM J. Comput. 23*, 4 (Aug. 1994), 701–712.

[8] DOLEV, D., DWORK, C., AND STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *J. ACM 34*, 1 (Jan. 1987), 77–97.

[9] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr. 1985), 374–382.

[10] GAFNI, E. Round-by-round fault detectors: Unifying synchrony and asynchrony. Unpublished manuscript, Mar. 1998.

[11] LO, W.-K., AND HADZILACOS, V. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the Eighth International Workshop on Distributed Algorithms*, G. Tel and P. Vitányi, Eds., no. 857 in Lecture Notes on Computer Science. Springer-Verlag, Sept. 1994, pp. 280–295.

[12] LOUI, M. C., AND ABU-AMARA, H. H. Memory requirements for agreement among unreliable asynchronous processors. In *Advances in Computing Research*, F. P. Preparata, Ed., vol. 4. JAI Press, 1987, pp. 163–183.

[13] LYNCH, N., AND RAJSBAUM, S. On the Borowsky-Gafni simulation algorithm. In *Proceedings of the Fourth Israel Symposium on Theory of Computing and Systems* (May 1996), pp. 4–15.

[14] NEIGER, G. Failure detectors and the wait-free hierarchy. In *Proceedings of the Fourteenth ACM Symposium on Principles of Distributed Computing* (Aug. 1995), ACM Press, pp. 100–109.