Possibility and Impossibility Results in a Shared Memory Environment [*]

Gadi Taubenfeld
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Shlomo Moran[†]
Computer Science Department
Technion, Haifa 32000
Israel

Abstract

We focus on unreliable asynchronous shared memory model which support only atomic read and write operations. For such a model we provide a necessary condition for the solvability of problems in the presence of multiple undetectable crash failures. Also, by using game-theoretical notions, a necessary and sufficient condition is provided, for the solvability of problems in the presence of multiple undetectable initial failures (i.e., processes may fail only prior to the execution).

Our results imply that many problems such as consensus, choosing a leader, ranking, matching and sorting are unsolvable in the presence of a single crash failure, and that variants of these problems are solvable in the presence of $t-1$ crash failures but not in the presence of t crash failures.

We show that a shared memory model can simulate various message passing models, and hence our impossibility results hold also for those message passing models. Our results extend and generalize previously known impossibility results for various asynchronous models.

Key words: asynchronous protocols, impossibility, shared memory, atomic read and write operations, crash failures, initial failures, winning strategy.

[*]A preliminary version of this work appeared in the *Proceedings of the 3rd International workshop on distributed algorithms,* Nice, France, September 1989. In: *LNCS 392* (eds.:J.C. Bermond, M. Raynal), Springer Verlag 1989.

[†]Supported in part by Technion V.P.R. Funds - Wellner Research Fund, and by the Foundation for Research in Electronics, Computers and Communications, administrated by the Israel Academy of Sciences and Humanities.

1 Introduction

This paper investigates the possibility and impossibility of solving certain problems in an unreliable asynchronous shared memory system of $n \geq 2$ processes, which supports only atomic read and write operations. The faulty behaviours we consider are undetectable initial failures and undetectable crash failures. Initial failures are a very weak type of failures where it is assumed that processes may fail only prior to the execution and that no event can happen on a process after it fails. That is, once a process starts operating it is guaranteed that it will never fail. Initial failures are a special case of crash (fail stop) failures in which a process may become faulty at any time during an execution. Obviously, if a protocol cannot tolerate initial failures then it cannot tolerate crash failures but not necessarily vice versa.

In some of our examples we use the *consensus* problem, in which every process receives a binary input, and all non-faulty processes have to decide on the same *input* value. (In particular, if all input values are the same, then that value must be the decision value.) Define an input vector to be a vector $\vec{a} = (a_1, ..., a_n)$, where a_i is the input value of process p_i. A crucial assumption in most of the impossibility results for a single crash failure is that the set of input vectors is "large enough". To demonstrate this fact, consider the consensus problem where only two input vectors are possible: either all processes read as input the value "zero" or all processes read as input the value "one". It is easy to see that under this restriction, the problem can be solved assuming any number of process failures. (Each process outputs its input value.)

We concentrate, in this paper, on an asynchronous shared memory model and prove possibility and impossibility results within that model. For every $t < n$, where n is the number of processes, we define a class of problems that are unsolvable in such a system in the presence of t crash failures. This implies a (necessary) condition for solving a problem in such an unreliable system. Also, we provide a necessary and sufficient conditions for solving problems in an asynchronous shared memory model where only undetectable initial failures may occur. Similar condition for initial failures in a message passing model appears in [TKM89b]. However, unlike in [TKM89b] we do not need to assume that only up to half of the processes may fail. Our results extend and generalize previously known impossibility results for asynchronous systems.

It appears that the necessary and sufficient condition which we give here for initial failures assuming only deterministic protocols, is the same as the complete characterization which is given in [CM89] for crash failures assuming randomized protocols. An interesting result that follows from the similarities between these characterizations is that in a shared memory model which supports only atomic read and write operations, a problem can be solved by a *deterministic* protocol that can tolerate up to t *initial* failures if and only if the problem can be solved by a *randomized* protocol that can tolerate up to t *crash* failures.

We show that many problems such as consensus, choosing a leader, ranking, matching and sorting are unsolvable (in a nontrivial way) in the presence of a single crash failure, and that, for any t, there are variants of these problems that are solvable in the presence of $t - 1$ crash failures but not in the presence of t crash failures. An example is the consensus problem with the assumption that for each input vector, $\mid \#1 - \#0 \mid \geq t$. (i.e., the absolute difference between the number of ones and the number of zeros is at least t.) Following is a simple protocol that solves this problem assuming up to $t - 1$ crash failures (where $t > 0$).

Each process writes its input value into a shared register that the other processes can read and then repeatedly tries to read the input values of the other processes. Since no more than $t-1$ processes may fail, a non-faulty process eventually reads $n-t+1$ input values. A process decides 1 iff the sum of the $n+t-1$ inputs is more than $(n-t+1)/2$ otherwise it decides 0. The fact that $\mid \#1 - \#0 \mid \geq t$ guarantees that all the processes will decide the same.

Another example is the *shared-consensus* problem, defined for a parameter $t \geq 1$ [TKM89a], for which we can use our results to prove impossibility in the presence of up to t crash failures. The input of each process p_i is a real number x_i, such that $0 \leq x_i \leq 1$ and $|\frac{n}{2} - \sum_{i=1}^{n} x_i| \geq \frac{t}{2}$; Each process has to decide on an integer such that the sum of the integers decided upon is 0 if $\sum x_i < \frac{n}{2}$, and is n otherwise. A solution to this problem in the presence of up to $t-1$ crash failures is as in the previous example.

We show that a shared memory model can simulate several of the message passing models which are considered in [DDS87], and hence all our impossibility results hold also for those message passing models. In particular, the impossibility results for crash failures presented in this paper imply similar results, for an asynchronous message passing model, which appear in [TKM89a].

The proof of our result for the crash failures case is constructed as follows. We first identify a class of *protocols* that can not tolerate the (crash) failure of t processes, when operating in an asynchronous shared memory system. Then, we identify those *problems* which can be solved only by protocols which belong to the above class. Hence, these problems can not be solved in an asynchronous system where t processes may fail.

The class of protocols for which we prove the impossibility result (for crash failures) is characterized by two requirements on the possible input and decision (output) values of each member in the class. For the input, it is required that (for each protocol) there exists a group of at least $n-t$ processes and there exist input values such that after all the $n-t$ processes in the group read these input values, the eventual decision value of at least one of them is still not uniquely determined. The requirement for the decision values is that the decision value of any (single) process, say p_i, is uniquely determined by the input values of all the processes together with the decision values of all the processes except p_i.

In order to prove the above result for protocols, we use an axiomatic approach for proving properties of protocols (and problems) which is due to Chandy and Misra [CM85, CM86]. The idea is to capture the main features of the model and the features of the class of protocols for which one wants to prove the result by a set of axioms, and to show that the result follows from the axioms. We will present five axioms capturing the nature of asynchronous shared memory systems which support only atomic read and write operations, a single axiom expressing the fact that at most t processes may crash fail, and two axioms defining the class of protocols for which we want to prove the impossibility result (for crash failures). We then show that no protocol in the class can tolerate t faulty processes, by showing that the set of the eight axioms is inconsistent.

Related Work

There has been extensive investigation about the nature of asynchronous message passing systems where undetectable crash failures may occur. The work in [FLP85] proves the nonexistence of a consensus protocols that can tolerate a single crash failure, for a completely asynchronous message passing system. Various extensions of this fundamental result, also for a single crash failure, prove the impossibility of other problems in the same model [MW87, Tau87, BMZ88]. Other works study the possibility of solving variety of problems in asynchronous systems with numerous crash failures, and in several message passing models [ABD$^+$87, BW87, DDS87, DLS88, TKM89a].

In [DDS87], Dolev, Dwork and Stockmeyer studied the consensus problem in partially synchronous message passing models. They showed that by changing the broadcast primitives it is possible to solve the consensus problem in the presence of $t-1$ crash failures but not in the presence of t crash failures. They also identify five critical parameters that may effect the possibility of achieving consensus. By varying these parameters they defined 32 models and found the maximum resiliency for each one of them.

In [LA88] an impossibility result for the binary consensus problem is shown for an asynchronous shared memory system, such as we consider here, where a single processes may (crash) fail. In [Abr88, CIL87, Her88] a weaker result than that of [LA88] proves the impossibility of the consensus problem in the presence of $n-1$ crash failures. This last impossibility result is used in [Her88] to derive a hierarchy of atomic operations (objects) such that no operation at one level has a wait-free (i.e., $(n-1)$-resilient) implementation using only operation from lower levels. Systems that support only atomic read and write operations are shown to be at the bottom of that hierarchy. In particular, it is impossible to implement using atomic read and write operations common data types such as sets, queues, stacks, priority queues, lists and most synchronization primitives.

Initial failures may occur in situations such as recovery from a breakdown of a network. Necessary and sufficient conditions are provided in [TKM89b], for solving problems in asynchronous message passing systems where up to half of the processes may fail prior to the execution, with and without a termination requirement. Several protocols were designed to properly operate in a message passing model where initial failures may occur. A protocol that solves the consensus problem which can tolerate initial failures of up to (not including) half of the processes was presented in [FLP85]. Protocols for leader election and spanning tree construction which can also tolerate initial failures of up to half of the processes were designed in [BKWZ87]. As for the shared memory model which supports atomic read and write operations, a leader election protocol that can tolerate up to $n-1$ initial failure is presented in [Tau89]. A complete combinatorial characterization, for the solvability of problems in asynchronous shared memory and message passing models where crash failures may occur using *randomized protocols* was given in [CM89].

2 Definitions and Basic Notations

Let I and D be sets of input values and decision (output) values, respectively. Let n be the number of processes, and let $\bar{I}$ and $\bar{D}$ be subsets of I^n and D^n, respectively. A *problem* T is a mapping $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ which maps each n-tuple in $\bar{I}$ to subsets of n-tuples in $\bar{D}$. We call the vectors $\vec{a} = (a_1, ..., a_n)$ where $\vec{a} \in \bar{I}$, and $\vec{d} = (d_1, ..., d_n)$ where $\vec{d} \in \bar{D}$, the input vector and decision vector respectively, and say that a_i (resp. d_i) is the input (resp. decision) value of process p_i.

Following are some examples of problems, which we will also refer to later in the paper (the input vectors for all problems are from I^n for an arbitrary set I): (1) The *permutation* problem, where each process $p_i(i = 1..n)$ decides on a value v_i from D, $D \equiv 1, ..., n$, and $i \neq j$ implies $v_i \neq v_j$; (2) The transaction commitment problem, where $I = D = \{0, 1\}$, and all processes are to decide on "1" if the input of each process is "1", otherwise all processes are to decide on "0"; (3) The *consensus* problem, where all processes are to decide on the same value from an arbitrary set D; (4) The (leader) *election* problem, where exactly one process is to decide on a distinguished value from an arbitrary set D; and (5) The *sorting* problem, where all processes have input values and each process p_i decides on the i^{th} smallest input value. In the permutation, consensus and election problems, the trivial solutions, in which only one vector of $\bar{D}$ is always chosen, is ruled out by the additional requirement that each process does not decide on the same value in all computations.

A *protocol* $P \equiv (N, R, C)$ consists of a set of process id's (abbv. processes) $N \equiv \{p_1, ..., p_n\}$, a (possibly infinite) set R of registers, and a nonempty set C of *computations*. A computation is a *finite* sequence of *events*. In protocols over shared memory read/write models there are four types of events. A *read* event, denoted $([read, r, v], p_i)$, represents reading a value v from register r by process p_i. A *write* event, denoted $([write, r, v], p_i)$, represents writing a value v into register r by process p_i. An *input* event, denoted $([input, a], p_i)$, represents reading an input value a by process p_i. A *decide* event, denoted $([decide, d], p_i)$, represents deciding on a value d by process p_i. (One may also consider an *internal* event in which a process executes some other local computation; however nowhere in this paper do we need to refer to such an event.)

We use the notation (e, p_i) to denote an arbitrary event, which may be an instance of any of the above types of events. For an event (e, p_i) we say that it occurred *on* process p_i. An event is *in* a computation iff it is one of the events in the sequence which comprises the computation. It should be emphasized that, given a set of processes N, a set of registers R, and a set of computations C, the triple (N, R, C) is a protocol in a given model only if the set C satisfies certain properties, which depend on N, R and the given model.

The *value* of a register at a finite computation is the last value that was written into that register, or its the special symbol $\perp$ if no process wrote into the register. We use $value(r, x)$ to denote the value of r at a finite computation x.

It is convenient to think of R as the set of shared memory registers, and to assume that each process may have in addition local variables that only it can read from and write to. In this work we do not need the notion of local variables.

In the rest of this paper Q denotes a set of processes where $Q \subseteq N$. The symbols x, y, z denote computations. An *extension* of a computation x is a computation of which x is a prefix. For an extension y of x, $(y - x)$ denotes the suffix of y obtained by removing x from y. For any x and p_i, let x_i be the subsequence of x containing all events in x which are on process p_i. Computation y *includes* computation x iff x_i is a prefix of y_i for all p_i.

Definition: Computations x and y are *equivalent w.r.t.* p_i, denoted by $x \overset{i}{\sim} y$, iff $x_i = y_i$.

Note that the relation $\overset{i}{\sim}$ is an equivalence relation. Also, for x a prefix of y, there is an event on p_i in $(y - x)$ iff $\neg(x \overset{i}{\sim} y)$.

Next, we define for a computation x and process p_i, the extensions of x which only have events on p_i.

Definition: $Extensions(x, i) \equiv \{y \mid y$ is an extension of x and $x \overset{j}{\sim} y$ for all $j \neq i\}$.
Process p_i *reads* input a in a computation x iff the input event $([input, a], p_i)$ is in x. Process p_i *decides* on d in a computation x iff the decision event $([decide, d], p_i)$ is in x. A computation x is *i-input* iff for some value a, p_i reads input a in x. We assume that a process may read an input value only once and decide only once.

A *protocol* $P \equiv (N, R, C)$ *solves* a problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ iff (1) For every input vector $\vec{a} \in \bar{I}$, and for every decision vector $\vec{d} \in T(\vec{a})$, there exists a computation $z \in C$ such that in z processes p_1,...,p_n read input values a_1,...,a_n and decide on d_1,...,d_n; (2) For every computation $z \in C$ and $\vec{a} \in \bar{I}$ if in z processes p_1,...,p_n read input values a_1,...,a_n and decide on d_1,...,d_n, then $\vec{d} \in T(\vec{a})$; and (3) In any "sufficiently long" computation on input in $\bar{I}$ all processes decide (this last requirement is to be defined precisely later). It is also possible to define solvability so that (1) is replaced by the requirement that for each input vector $\vec{a} \in \bar{I}$, there exists a computation with $\vec{a}$ as input. In such a case we will say that a protocol P *minimally solves* a problem T. The difference between the two is that in the former case every possible decision vector is the result of some computation, while in the latter this is not so. It will be shown in section 6 that it is possible to prove the impossibility result for the former definition of solvability, and then to derive from it a result for the latter one.

We define when a set of input events is *consistent* w.r.t. a given task. Intuitively, this is the case when all the input events in the set can occur in the same computation. Formally, Let $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ be a given task, and let P be a protocol that solves T. Then a set of n input events $\{([input, a_1], p_1), \cdots, ([input, a_n], p_n)\}$ is consistent w.r.t. P only if $(a_1, \cdots, a_n) \in \bar{I}$, and any subset of a consistent set of input events is also consistent. In the sequel, we assume that for every protocol $P \equiv (N, R, C)$ that solves a problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$, the set of input events in any computation $x \in C$ is consistent.

3 Shared Memory Model

In this section we characterize an asynchronous shared memory model which supports atomic read and write operations. This is done by stating 5 axioms which define what are the ordering of events of a computation.

Definition: An *asynchronous read-write protocol* (abbv. asynchronous protocol) is a protocol whose computations satisfy the following properties,

P1: Every prefix of a computation is a computation.

P2: Let $\langle x; (e, p_i) \rangle$ be a computation where (e, p_i) is either a write event or a decision event, and let y be a computation such that $x \overset{i}{\sim} y$, then $\langle y; (e, p_i) \rangle$ is a computation.

P3: For any computation x, process p_i and input value a, if the set of all input events in x together with $([input, a], p_i)$ is consistent then there exists y in $Extensions(x, i)$, such that $([input, a], p_i)$ appears in y.

P4: For computations x and y and process p_i, if $\langle x; ([read, r, v], p_i) \rangle$ is a computation, and $x \overset{i}{\sim} y$ then $\langle y; ([read, r, value(r, y)], p_i) \rangle$ is a computation.

P5: For a computation x and an event $([read, r, v], p_i)$, the sequence $\langle x; ([read, r, v], p_i) \rangle$ is a computation only if $v = value(r, x)$.

Property $P2$ means that if some write event or decision event can happen on process p_i at some point in a computation, then this event can happen at a later point, provided that p_i has taken no steps between the two points. Property $P3$ means that a process which has not yet read an input value may read any of the input values not conflicting with those already read by other processes. For example, if we assume that the input values different processes may read in the same computation are distinct, then a process may read any value which has not already been read by other processes. Property $P4$ means that if a process is "ready to read" a value from some register then an event on some other process cannot prevent this process from reading some value from that register (although it may prevent this process from reading a specific value which it could read previously). Property $P5$ means that it is possible to read only the last value that is written into a register.

We will consider in this paper only deterministic protocols which means that at any point in a computation a process may perform at most one non-input action; in case the current action of a process is reading an input then the process may read one of several possible input values. I.e., if $\langle x; (e, p_i)\rangle$ and $\langle x; (e', p_i)\rangle$ are computations and both (e, p_i) and (e', p_i) are not input events then $(e, p_i) \equiv (e', p_i)$. This assumption does not restrict the generality of the results which will hold also for non-deterministic protocols.

We say that process p_i is *enabled* at computation x iff there exists an event (e, p_i) such that $\langle x; (e, p_i)\rangle$ is a computation. It follows from the above five properties that an enabled process (in some computation) cannot become disabled as a result of an event on some other process.

4 Classes of Protocols

In this section we identify two classes of protocols, called *dependent*(t) protocols, and *robust*(t) protocols. The important features of dependent(t) protocols are the requirements on the possible input and decision (output) values. For the input, it is required that there exists a group of at least $n-t$ processes and there exist input values such that after all the $n-t$ processes read these input values, the eventual decision value of at least one of them is still not uniquely determined. Compared with the usual requirement in other works where the above group should include all the processes (i.e., be of size n), this requirement is very weak. The requirement for the decision values is that the decision value of any (single) process p_i is uniquely determined by the input values of all the processes together with the decision values of all the processes except p_i.

Typical examples of dependent(t) protocols are the protocols that solve any of the problems described in the Introduction and Section 2, where various assumptions, depending on the value of t, are made about the set of input vectors for each of these problems. Having that class formally defined, we prove in the next section that for every $1 \leq t \leq n$, no protocol in the class of dependent(t) protocols can tolerate t process failures.

The following definition generalizes the notion of *valency* of a computation from [FLP85]. Let d be a possible decision value and let U, W be sets of values.

Definition: A computation x is (i, W)*-valent* iff (1) for every $d \in W$, there is an extension of x in which p_i decides on d, and (2) for every $d \notin W$, there is no extension of x in which p_i decides on d.

A computation is *i-univalent* iff it is $(i,\{d\})$*-valent* for some (single) value d. It is *i-multivalent* otherwise. It will follow from the sequel that no computation in a protocol studied here is $(i,\emptyset)$*-valent*. A computation may become *i-univalent* (i.e., its ultimate decision value can be uniquely determined) as a consequence of some other process' action. That is, it is possible to have two computations x and y such that $x \overset{i}{\sim} y$, yet x is *i-univalent* while y is *i-multivalent*. Also, for any computation x and any process p_i, if p_i has decided on some value then x is *i-univalent* but not vice versa. Note that for any computation x and process p_i there exists a single set W such that x is (i,W)*-valent*.

Definition: Let y and y' be (i,W)*-valent* and (i,W')*-valent*, respectively. Then y and y' are *i-compatible* iff $W \cap W' \neq \emptyset$. They are *compatible* iff they are *i-compatible* for all $i = 1..n$.

Using the above notions we can now characterize dependent(t) protocols formally. Two requirements are given, and a *protocol* is defined to be a dependent(t) protocol if it satisfies these requirements.

Definition: A *dependent*(t) *protocol* is a protocol that satisfies the requirements:

D1(t): There exists a computation x, set of processes Q and process $p_i \in Q$, such that $|Q| \geq n-t$, for every $p_j \in Q$ x is *j-input*, and x is *i-multivalent*. (non-triviality.)

D2: For any two computations x and y which are both *i-univalent*, if each process read the same input value in both x and y, and if each processes $p_j \neq p_i$ decide on the same value in both x and y then x and y are *i-compatible*. (dependency.)

Requirement $D1(t)$ generalizes a requirement which appears in [FLP85],(i.e., Lemma 2), which says that a non trivial consensus protocol must has a bivalent initial configuration. As we explain latter in section 7, any problem that can be solved by a protocol that does not satisfy $D1(t)$, has also the following trivial solution. Each process sends its input value to all other processes, then it waits until it receives $n-t$ values; assuming it does not satisfy $D1(t)$, it has now enough information to decide. Notice that $D1(t)$ implies $D1(t+1)$. It is not difficult to see why any protocol that solves the variant of the consensus problem, with the assumption that for each input vector $|\#1 - \#0| \geq t$, or the *shared-consensus* problem which is mentioned in the introduction, must satisfy $D1(t)$. The proof of that fact is similar to the proof of Lemma 2 in [FLP85].

Requirement $D2$ means that an external observer who knows all the input values and all decision values except one can always determine the missing decision value. All protocols which solve the problems mentioned in the Introduction and in Section 2 satisfy $D2$.

Next we identify the class of protocols which can tolerate t crash failures ($0 \leq t \leq n$). A crash failure of a process means that no subsequent event can happen on this process. Note that if an impossibility result holds for crash failures it also holds for any stronger type of failure. Informally, a protocol is robust(t) if, in spite of a failure of any group of t processes at any point in the computation, each of the remaining processes eventually decides on some value.

In order to define robust(t) protocols formally, we need the concept of a *Q-fair sequence*. Let Q be a set of processes, a *Q-fair sequence* w.r.t. a given protocol is a (possibly infinite) sequence of events, where: (1) Each finite prefix of the sequence is a computation; (2) For an enabled process $p_i \in Q$ at some prefix x, there exists another prefix y that extends x

such that there is an event (e, p_i) in $(y - x)$. It follows from $P5$ and requirement (1) that the sequence $\langle x; ([read, r, v], p_k)\rangle$ is a prefix of a $Q-$fair sequence only if $v = value(r, x)$.

A Q-fair sequence captures the intuition of an execution where all enabled processes which belong to Q can proceed. Notice that a Q-fair sequence may be infinite and in such a case it is not a computation. It follows from $P1 - P5$ that, in asynchronous protocols, for every set of processes Q, any computation is a prefix of a Q-fair sequence.

Definition: A *robust*(t) *protocol* $(0 \leq t \leq n)$ is a protocol that satisfies the requirement:

R(t): For every set Q of processes where $|Q| \geq n - t$, every *Q-fair sequence* has a finite prefix in which any $p_i \in Q$ decides on some value.

Note that the class of robust$(t+1)$ protocols is included in the class of robust(t) protocols. Furthermore, the inclusion is strict since there are protocols which are robust(t) but not robust$(t+1)$. Requirement $R(0)$ means that in any "long enough" execution of a protocol, if no process fails then each process (eventually) decides on a value. In fact, $R(0)$ formally expresses requirement (3) from the definition of *solves* given in Section 2. Thus, any protocol that solves a problem should satisfy $R(0)$. From $R(0)$ and from the fact that every computation is a prefix of some N-fair sequence it follows that (in asynchronous robust(0) protocols) no computation is $(i, \emptyset)$*-valent.*

In order to define robust(t) protocols we did not have to define the notion of a faulty process. We concentrated on the role of the correct processes in order to capture the nature of robustness. By using the notion of a Q-fair sequence we have described an execution in which all processes in Q are correct, and only for those processes we required that they eventually decide. We may say that process $p_i \notin Q$ is faulty in some Q-fair sequence if that sequence is not a $(Q \cup \{p_i\})$-fair sequence. There is a way to define a fault tolerant protocol by first defining the notion of a faulty process as done in [Had87]. This involves the introduction of an additional type of event which signals the fact that a process is faulty. Our approach seems to be more suitable for the model under consideration, since it captures the fact that in systems where a failure of a process is not detectable, a faulty process cannot be distinguished from a process that operates very slowly. It also simplifies the presentation and the proofs.

Lemma 1: *In any asynchronous robust(1) protocol, for any two computations x and y and for any process p_i, if $x \overset{j}{\sim} y$ for every $j \neq i$, and $value(r, x) = value(r, y)$ for every $r \in R$, then x and y are j-compatible for every $j \neq i$.*

Proof: It follows from $P1 - P5$ that the computation x is a prefix of some $(N - \{p_i\})$-fair sequence, and there are no events on p_i in that sequence after x. Apply requirement $R(1)$ to the above sequence to conclude that there exists an extension z of x such that $x \overset{i}{\sim} z$ and any $p_j \neq p_i$ has decided in z. From $P1 - P4$, it follows that $w \equiv \langle y; (z - x)\rangle$ is also a computation. Clearly, for any $j \neq i$, z and w are *j-compatible.* Hence also, for any $j \neq i$, x and y are *j-compatible.* $\square$

We postpone the formal definition of initial failures to Section 7. In the next two sections we consider only crash failures.

5 Impossibility Results for Protocols

In the previous sections we have defined several classes of protocols in the shared memory model which supports only atomic read and write operations. In this section we investigate a class which is the intersection of all the previous classes. This class is defined by the entire eight axioms and is called the class of RObust(t) Asynchronous Dependent(t) Protocols (abbv. ROAD(t) P's), where $1 \leq t \leq n$. We prove in this section that the class of ROAD(t) P's is empty. Put another way, we show that there does not exist any ROAD(t) P.

The following lemma shows that for any ROAD(t) P, every two computations which differ only by the events on a single process p_i and in which the values of all registers are the same are compatible.

Lemma 2: *In any ROAD(t) P, for any two computations x and y and any process p_i, if, (1) p_i did not read different input values in x and y, (2) $x \overset{j}{\sim} y$ for any $j \neq i$, and (3) $value(r, x) = value(r, y)$ for any $r \in R$, then x and y are compatible.*

Proof: It follows from $P1 - P5$ that the computation x is a prefix of some $(N - \{p_i\})$-fair sequence, and there are no events on p_i in that sequence after x. Apply requirement $R(1)$ to the above sequence to conclude that there exists an extension z' of x such that $x \overset{i}{\sim} z'$ and for any $p_j \neq p_i$, p_j has decided in z'. By $P3$ there is an extension z of z' in which all processes except maybe p_i read their input and $z \overset{i}{\sim} z'$. From $P1 - P4$, it follows that $w \equiv < y; (z - x) >$ is also a computation. By $P1 - P5$ and $R(0)$, there are two *i-univalent* extensions $\hat{z}$ and $\hat{w}$ of z and w respectively, in which p_i reads the same input value. From $D2$, $\hat{z}$ and $\hat{w}$ are *compatible* and hence also x and y are *compatible*. $\square$

Theorem 1: *In any ROAD(t) P, for any process p_i and any j-multivalent computation x, if x is i-input and p_i is enabled at x then there exists a j-multivalent extension $\hat{x}$ of x such that $\neg(x \overset{i}{\sim} \hat{x})$.*

Proof: To prove the theorem we first assume to the contrary: for some process p_i and some *j-multivalent* computation x where x is *i-input* and p_i is enabled at x, there is no *j-multivalent* extension $\hat{x}$ of x such that $\neg(x \overset{i}{\sim} \hat{x})$. Then we show that this leads to a contradiction. It follows from the assumption that for any extension m of x such that p_i is enabled at m, the unique extension of m by a single event on p_i is *j-univalent*. Let us denote that *j-univalent* extension of m by $\Phi(m)$.

Since x is *j-multivalent*, there exists an extension z of x ($z \neq x$) such that z and $\Phi(x)$ are not *j-compatible*. Let z' be the longest prefix of z such that $x \overset{i}{\sim} z'$. From the assumption it follows that $\Phi(x)$ and $\Phi(z')$ are not *j-compatible*. Consider the extensions of x which are also prefixes of z'. Since $\Phi(x)$ and $\Phi(z')$ are not *j-compatible*, there must exist extensions y and y' (of x) such that $\Phi(y')$ and $\Phi(y)$ are not *j-compatible*, and y is a one event extension of y'. Therefore, $y =< y'; (e, p_h) >$ for some event (e, p_h) where $p_i \neq p_h$. For later reference we denote $w \equiv < \Phi(y'); (e, p_h) >$. We do not claim at this point that w is a computation. (See Figure 1.)

There are four possible cases.

Case 1: (e, p_h) is not a write event. By $P2$ and $P4$, $(\Phi(y') - y') = (\Phi(y) - y)$ and hence for any $p_k \neq p_h$, $\Phi(y') \overset{k}{\sim} \Phi(y)$. Also, the values of all registers are the same in both $\Phi(y)$ and $\Phi(y')$, and obviously p_h does not read different input values in $\Phi(y')$ and $\Phi(y)$.

By Lemma 2, $\Phi(y')$ and $\Phi(y)$ should be *compatible.* Hence, we reach a contradiction.
At this point we know that (e, p_h) is a write event and (from $P2$) that w is a computation.

Case 2: $(\Phi(y') - y')$ is not a write event. For every $p_k \neq p_i$, $w \overset{k}{\sim} \Phi(y)$. Also, the values of all registers are the same in both w and $\Phi(y)$. Since y is *i-input* obviously p_i reads the same input values in w and $\Phi(y)$. By Lemma 2, w and $\Phi(y)$ are *compatible.* Since, w is an extension of $\Phi(y')$ then $\Phi(y')$ and $\Phi(y)$ should be *compatible.* Hence again we reach a contradiction.
Now we know that for some registers r_1 and r_2, and values v_1 and v_2, $(\Phi(y') - y') = ([write, r_1, v_1], p_i)$, and $(y - y') = ([write, r_2, v_2], p_h)$.

Case 3: $r_1 \neq r_2$. Since the two write events on p_i and p_h are independent, the values of all registers are the same in w and $\Phi(y)$. Also, for every process p_k, $w \overset{k}{\sim} \Phi(y)$. This leads to a contradiction as in the second case.

Case 4: $r_1 = r_2$. Clearly, $value(\Phi(y'), r_1) = value(\Phi(y'), r_2) = value(\Phi(y), r_1) = value(\Phi(y), r_2) = v_1$. Hence, the values of all registers are the same in $\Phi(y')$ and $\Phi(y)$. Also, for any $p_k \neq p_h$, $\Phi(y') \overset{k}{\sim} \Phi(y)$. By Lemma 2, $\Phi(y')$ and $\Phi(y)$ are *compatible.* Hence, again we reach a contradiction.
This completes the proof. $\Box$

Theorem 2: *There is no ROAD(t) P.*

Proof: By $D1(t)$, there exists a computation x, process p_i and set of processes Q, such that $|Q| \geq n - t$, for every $p_j \in Q$ x is *j-input*, $p_i \in Q$, and x is *i-multivalent.* Using Theorem 1, we can construct inductively starting from the computation x a Q-fair sequence such that all the finite prefixes of that sequence are *i-multivalent.* This contradicts requirement $R(t)$. $\Box$

Consider the eight requirements mentioned so far. Apart from requirement $D2$, all the requirements capture very natural concepts: $P1 - P5$ and $R(t)$ express the well known notions of asynchronous and robust protocols respectively, while $D1(t)$ requires that a given solution is not trivial. This motivates the question of what can be said about protocols that satisfy all the above requirements except $D2$. For later reference we call these protocols Decision(t) Asynchronous Robust(t) Protocols (abbv. DEAR(t) P's). A simple example for a DEAR($n-1$) protocol, is a protocol where there is only one shared register, each process first writes its input value into the shared register, then it reads the value of the shared register and decides on that value.

It follows immediately from the impossibility result of Theorem 2 that DEAR(t) P's cannot satisfy requirement $D2$. Also, if we inspect the proof of Theorem 2 we see that requirement $D2$ is only used in the proof of Lemma 2. Hence, we conclude that DEAR(t) P's have to satisfy the negation of Lemma 2. These observations leads to the following theorem.

Theorem 3: *In any DEAR(t) P, there exist two computations x and y, and there exists process p_i, such that: (1) p_i did not read different input values in x and y, (2) $x \overset{j}{\sim} y$ for any $j \neq i$, (3) $value(r, x) = value(r, y)$ for any $r \in R$, and yet x and y are not i-compatible.*

Proof: Immediate from Lemma 1 and the negation of Lemma 2. $\square$

Theorem 3 gives the intuition for the nonexistence result for ROAD(t) P's as stated in Theorem 2. This result follows from a conflict between two requirements. One is requirement $D2$ which means that at any time a process may be forced by the group of all other processes to a situation where it has only one possible decision left. As opposed to that requirement there is the necessary condition given in Theorem 3 which means that there exist two computations such that the sets of values some process may still decide on in each one of this computations are disjoint and those computation are indistinguishable from the point of view of the group of all other processes.

6 Impossibility Results for Problems

In this section we identify the problems that cannot be solved in an unreliable asynchronous shared memory environment which support only atomic read and write operations. We do this by identifying those problems which are solved only by ROAD(t) protocols. Hence, the impossibility of solving these problems will follow from Theorem 2. Results for completely asynchronous message passing systems which are similar to those presented in the sequel appear also in [TKM89a]. As we shall see in Section 8, the results presented in this section imply those in [TKM89a].

We say that a problem can be solved in an environment where t processes may fail, if there exists a robust(t) protocol that solves it. Since we assume an asynchronous shared memory environment where t processes may fail, any protocol that solves a problem should satisfy properties $P1-P5$, and requirement $R(t)$. Thus, we are now left with the obligation of identifying those problems which force any protocol that solves them also to satisfy requirements $D1(t)$ and $D2$. Let Q denote a set of processes, and $\vec{v}$ and $\vec{v'}$ be vectors. We say that $\vec{v}$ and $\vec{v'}$ are *Q-equivalent*, if they agree on all the values which correspond to the indices (of the processes) in Q. A set of vectors H is *Q-equivalent* if any two vectors which belong to H are *Q-equivalent.* Also, we define: $T(H) \equiv \bigcup_{\vec{a} \in H} T(\vec{a})$.

Definition: A problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ is a *dependent(t) problem* iff it satisfies the requirements:

T1(t): There exists a set of processes Q where $|Q| \geq n-t$, and there exists a *Q-equivalent* set $H \subseteq \bar{I}$ such that $T(H)$ is not a *Q-equivalent* set.

T2: For every $\vec{a} \in \bar{I}$, every set of processes Q where $|Q| = n-1$, and every two different decision vectors $\vec{d}$ and $\vec{d'}$, if both $\vec{d}$ and $\vec{d'}$ belong to $T(\vec{a})$ then they are not *Q-equivalent.*

Requirement $T1(t)$ means that $n-t$ input values (in an input vector) do not determine the corresponding $n-t$ decision values (in the decision vectors). Any problem that does not satisfy requirement $T1(t)$ can easily be solved in a completely asynchronous environment where t processes may fail. (Each process sends its input value to all other processes, then it waits until it receives $n-t$ values; assuming it does not satisfies $T1(t)$, it has now enough information to decide.) Note that $T1(t)$ implies $T1(t+1)$. Requirement $T2$ means that a single input vector cannot be mapped into two decision vectors that differ only by a single value.

Theorem 4: *A dependent(t) problem cannot be solved in an asynchronous shared memory system which supports only atomic read and write operations and where t failures may occur.*

Proof: As already explained any protocol that solves a dependent(t) problem, in an asynchronous shared variable model where t processes may fail, should satisfy $P1-P5$ and $R(t)$. It follows from $T1(t)$ that such a protocol must satisfy $D1(t)$. Also, it follows from $T2$ that the protocol satisfies $D2$. Hence, such a protocol is necessarily a ROAD(t) P. Applying Theorem 2 the result is proven. $\Box$

Clearly, the shared-consensus problem is a dependent(t) problem and hence can not be solved in the presence of up to t crash failures. For the two corollaries of Theorem 4, we use the following definitions and observations. A problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ *includes* a problem $T : \bar{I}' \rightarrow 2^{\bar{D}'} - \{\emptyset\}$ iff (1) $\bar{I}' = \bar{I}$, and (2) for every $\vec{a'} \in \bar{I}'$: $T'(\vec{a'}) \subseteq T(\vec{a'})$. It is easy to see that a protocol P *minimally solves* a problem T iff there exists a problem T' which is included in T such that P *solves* T'. A problem $T' : \bar{I} \rightarrow 2^{\bar{D}'} - \{\emptyset\}$ is a *sub-problem* of a problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ iff (1) $\bar{I}' \subseteq \bar{I}$, and (2) for every $\vec{a'} \in \bar{I}'$: $T(\vec{a'}) = T'(\vec{a'})$. It is easy to see that if a protocol P *solves* (*minimally solves*) a problem T then P *solves* (*minimally solves*) any sub-problem T' of T.

Corollary 4.1: *If some sub-problem of T includes only dependent(t) problems then T cannot be minimally solved in a completely asynchronous environment where t processes may fail.*

Corollary 4.2: *If a problem T has a dependent(t) sub-problem then T cannot be solved in a completely asynchronous environment where t processes may fail.*

Example: Consider the following variant of the consensus problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ where all processes are to decide on the same value from the set D; $\bar{I}$ is the set of all vectors $\vec{a}$ such that $\vec{a} \in (0+1)^n$ and $|\#1 - \#0| \geq t$, and there exist two input vectors $\vec{a}$ and $\vec{a'}$ such that $T(\vec{a}) \cap T(\vec{a'}) = \emptyset$. It is not difficult to see that T is a dependent(t) problem and furthermore that T includes only dependent(t) problems. From Corollary 4.1 we conclude that T cannot be minimally solved in a completely asynchronous environment where t processes may fail.

Nowhere up to now, have we assumed anything about the process ids, hence the results we proved hold even if all processes have distinct id's which are mutually known.

7 Initial Failures

In this section we give a complete characterization of the problems that can be solved in an asynchronous shared memory environment where t processes may initially fail. We use the intuitive appeal of a game-theoretical characterization by reducing the question of solvability in the model under consideration to whether there is a winning strategy to a certain game which we describe below. The exposition here is influenced by the "Ehrenfeucht Game" [EFT84], which is used in mathematical logic to determine if two structures are elementarily equivalent (that is, if they satisfy the same first-order sentences.) Similar results for message passing model appears in Section 4 of [TKM89b]. However, unlike in [TKM89b] we do not need to assume here that only up to half of the processes may fail. Also, similar characterization, for the solvability of problems in an asynchronous shared memory model where crash failures may occur using *randomized protocols* is given in [CM89].

Informally, a protocol can tolerate up to t initial failures if in spite of a failure of any group of up to t processes at the beginning of the computation, each of the remaining processes eventually decides on some value. We now characterize such protocols formally.

Definition: A protocol *can tolerate up to t initial failures* iff for every set Q of processes where $|Q| \geq n-t$, every $Q-$fair sequence which consists only of events on processes which belong to Q, has a finite prefix in which any $p_i \in Q$ has decided.

Note that the class of protocols that can tolerate up to t initial failures strictly includes the class of protocols that can tolerate only up to $t-1$ initial failures. To see that the inclusion is strict consider the *rotating(t)* problem, where each process p_i has to decide on a decision value from the set of input values of processes $p_{i(mod\ n)+1},...,p_{i+t-1(mod\ n)+1}$. In any protocol that solves this problem, process p_i will never be able to decide if all processes $p_{i(mod\ n)+1},...,p_{i+t-1(mod\ n)+1}$ fail. We say that a problem can be solved in an environment where t initial failures may occur, if there exists a protocol which can tolerate up to t initial failures that solves the problem.

The game $G(T,t)$, corresponding to a problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ and a number t ($0 < t \leq n-1$), is played by two players A (Adversary) and B, according to the following rules. Each play of the game begins with a move of player A and in the subsequent moves both players move alternately. The game is played on a board which has n empty circles, which are numbered from 1 to n. At the first move player A chooses $n-t$ input values from (the set of input values) I and "places" them on arbitrary $n-t$ empty circles. Then player B chooses $n-t$ decision values from (the set of decision values) D and uses them to *cover* all the $n-t$ input values placed by player A in the previous move. The other subsequent moves consist of player A choosing a *single* value from I, in each move, and placing it on an empty circle, and then player B choosing a single value from D and covering the previous value placed by player A. The play is completed when all the n circles are covered with decision values from D. We emphasize that at any time each player knows all the previous moves.

We denote by $a_i \in I$ and $d_i \in D$ the values players A and B placed on the $i'th$ circle in the course of the play, respectively. For simplicity we assume that the final vector $(a_1,...,a_n)$ belongs to $\bar{I}$. Player B has *won* the play iff $\vec{d} \in T(\vec{a})$. Player B has a *winning strategy* in the game $G(T,t)$, denoted B *wins* $G(T,t)$, if it can always win each play.

For simplicity we assume that the processes have distinct identities which are mutually known. We will remove these assumptions later.

Theorem 5: *A problem T can be minimally solved in an asynchronous shared memory environment where t processes may initially fail ($0 < t \leq n-1$), iff player B wins $G(T,t)$.*

Proof: This proof is similar to Theorem 1 in [TKM89b]. We first prove the *if* direction. The proof is based on the fact that in the model under consideration (which supports atomic read and write operations), it is possible to elect a leader in the presence of up to $n-1$ initial failures [Tau89]. Suppose player B has a winning strategy in the game $G(T,t)$. We describe a protocol that minimally solves T in the presence of t initial failures. First each process writes its input into a shared register, and then one of the processes is elected as a leader. Then the leader try to read the input value of all the processes. Since at most t processes might be faulty, the leader is guaranteed to read $k \geq n-t$ input values (including its own). The leader then uses the winning strategy of player B to determine the corresponding k decision values, and transfer (by writing to a shared register) the relevant decision value to each process from which it read an input value. Afterwards the leader repeatedly tries to read the input value of the other processes. Upon reading an additional input value, it uses again the winning strategy of player B, to produce an appropriate decision value, and transfer it to the process from which it read the input value. Each process that gets a decision value from the leader decides on that value. The fact that B has a winning strategy implies that for every input vector $\vec{a}$, and for every possible run on input $\vec{a}$, the output vector belongs to $T(\vec{a})$.

We now prove the *only if* direction. Let P be an asynchronous protocol that minimally solves T in the presence of t initial failures. We describe a winning strategy for player B in $G(T,t)$. Let $\vec{a} \in \bar{I}$ be an arbitrary input vector, and let Q be a set of process where $|Q| = n-t$ such that player A chooses in his first move the set of values $\{a_i | p_i \in Q\}$ and places each value a_i on the circle numbered with i. Since P can tolerate up to t initial failures, there exists a computation $x \in C$ such that x consists only of events on processes which belong to Q, and any process $p_i \in Q$ reads the input value a_i and decides in x. Let d_i denotes the value on which process $p_i \in Q$ decided in x. By using P, player B can simulate the computation x, output the $n-t$ decision values, and cover each input value a_i by the corresponding decision value d_i. Now assume that player A chooses next some value a_j where $p_j \notin Q$ and places it on the $j'th$ circle. Since P can tolerate up to t initial failures, there is an extension y of x in which process p_j reads the input value a_j and decides on some value d_j, x consists only of events on processes which belong to $Q \cup \{p_j\}$. Thus again, by using P, player B can continue the simulation of x in order to simulate computation y and choose d_j. A similar construction holds also for any further input values that A chooses. Finally, since P minimally solves T, $\vec{d} \in T(\vec{a})$ and hence player B wins the game. $\Box$

Examples of problems that can be shown to be unsolvable using the above theorem (in the model under consideration), are transaction commitment, sorting and rotating(t). To show the impossibility for transaction commitment we demonstrate that B has no winning strategy. The adversary can choose at its first move $n-t$ "1" values; B then must also use $n-t$ "1" values since player A may later choose only "1" values. Then A can add the value "0" and B loses. The above theorem also points out how to construct a solution (i.e., a protocol) to any solvable problem T. First find a winning strategy for player B in the game $G(T,t)$ and then plug it into the (schematic) protocol presented in the "if" part of the proof of Theorem 5.

In the proof of Theorem 5, the assumption that the processes have distinct identities which are mutually known is used at the point where the leader (in the "if" part) has to consult with the winning strategy of player B. We now remove this assumption and only require that the input values are distinct. (This, of course, also covers the case where the processes have distinct identities which are not mutually known). Next, we modify Theorem 5 so that it holds under this weaker requirement.

Recall that $\vec{a}$ and $\vec{d}$ are the vectors players A and B placed in the course of the play, respectively. Let $\pi \equiv (\pi_1,...,\pi_n)$ be a permutation of $1,...,n$, and let $\pi(\vec{a})$ denote the vector $(a_{\pi_1},..., a_{\pi_n})$. We say that player B *strongly won* the play iff for every permutation π of $1,...,n$ where $\pi(\vec{a}) \in \bar{I}$, and for every vector $\vec{d} \in T(\vec{a})$, it is the case that $\pi(\vec{d}) \in T(\pi(\vec{a}))$. Player B has a *strong winning strategy* in the game $G(T,t)$ and write "*B strongly wins $G(T,t)$*" if it is possible for him to strongly win each play. If we now substitute in Theorem 5 the term "strongly wins" for "wins" then the modified theorem will hold under the requirement that the input values are distinct, and with no need to assume anything about the process identities. The proof of this theorem involves some technical modification of the previous proof, and is based on the fact that a leader can still be elected. Another way of resolving this problem is the following. We say that a problem $T : \bar{I} \rightarrow 2^{\bar{D}} - \{\emptyset\}$ is *symmetric* iff for every vector $\vec{a} \in \bar{I}$, for every permutation π of $1,...,n$, and for every vector $\vec{d} \in T(\vec{a})$, it is the case that $\pi(\vec{a}) \in \bar{I}$ and $\pi(\vec{d}) \in T(\pi(\vec{a}))$. For symmetric problems the notion of *strongly wins* and *wins* coincide, and hence for such problems the original formulation of Theorem 5 still holds (without the assumption that the processes have distinct identities).

8 Simulations of Various Message Passing Models by a Shared Memory Model

We show how our results can be used to decide whether the shared memory model, as defined in Section 3, can simulate various message passing models. To prove our results about the possible simulations we need to use several results for message passing systems which have been proven in [DDS87]. Thus, we first informally review some of the results presented in [DDS87].

The authors identify five critical parameters in message passing systems that may effect the possibility of achieving consensus. The digits 0 and 1 below refer to situations that are unfavorable or favorable for solving a problem, respectively. The notion of atomic step is used for an undivided sequence of events on some process. A process which executes an atomic step cannot fail before completing that step. The five parameters are:

Processes

- **0.** *Asynchronous* - Any finite numbers of events can take place between any two consecutive events on a process. That is, there is no assumption on the relative speed of the processes.

- **1.** *Synchronous* - There is a constant $\Psi \geq 1$ such that for any computation $\langle x; y\rangle$, if there are $\Psi + 1$ events on some process in y then there is an event on any nonfaulty process in y.

Communication

0. *Asynchronous* - Any finite numbers of events can take place between the sending and receiving of a certain message. That is, no assumption is made about the time it takes for a message to arrive to its destination.

1. *Synchronous* - There is a constant $\Delta \geq 1$ such that, every message that is sent is delivered within Δ attempts made to accept it. That is, there is an apriori bound on time delivery.

Messages

0. *Unordered* - Messages can be delivered out of order.

1. *Ordered* - If m_1 is sent before the message m_2 (w.r.t. real time), and both message are sent to the same process, then m_1 must be received before m_2. That is, messages must be delivered in the order they are sent.

Transmission Mechanism

0. *Point to point* - In an atomic step a process can send to at most one process.

1. *Broadcast* - In an atomic step a process can send to all processes.

Receive/Send

0. *Separate* - In an atomic step a processes cannot both receive and send.

1. *Atomic* - In an atomic step a processes can receive and send.

By varying the above five parameters the authors of [DDS87] defined 32 models and found the maximum resiliency for each one of them. These results are presented in the table in Figure 2. In an entry of the table the letters $0, 1, n$ describe the maximum resilience for the relevant model as proved in [DDS87]. (Recall that n is the number of processes, thus when n appears in an entry it means that it is possible to tolerate any number of faulty processes.)

We have examined all the 32 message passing models considered in [DDS87]. For each of 30 out of those models we can either prove that it can be simulated by an asynchronous shared memory model which support only atomic read and write operations (abbv. shared memory model), or can prove that it cannot. By saying that model A can *simulate* model A', we mean that the existence of a protocol which solves some problem in the presence of t failures in model A', implies the existence of a protocol which solves the same problem in the presence of t failures in model A. Evidently, all the impossibility results that we proved so far hold for any model that can be simulated by a shared memory model.

Our results are also presented in the table in Figure 2. The words "**Yes** " and "**No** " state whether the particular model can be simulated by a shared variable model, while

"**?**" declares that we do not know the answer. To prove this results we need to use the results from [DDS87] together with the result that it is not possible to solve the consensus problem in an asynchronous shared memory model which support only atomic read and write operations and where a single processes may fail.

As can be seen from the results of [DDS87] there are 19 models that can tolerate a single process failure. Clearly this 19 models cannot be simulated by a shared memory model, because this would imply that it is possible to solve consensus in an asynchronous shared memory model where a single processes may fail, a claim that we know is incorrect.

As for the other 7 models for which we claim that they cannot be simulate by a shared memory model, the proof of that follows easily from the following observation. Let A and A' be two models that are the same in all parameters except that in A communication is asynchronous and in A' communication is synchronous. If a shared memory model can simulate A then it also can simulate A' (and vice versa). Put another way, if A' cannot be simulated by a shared memory model then so do A. The correctness of this observation follows from fact that, assuming that no write override a previous write, communication (by reading and writing) is always instantaneous (i.e., synchronous) in a shared memory model and hence any simulation for model A will work also for A'.

We show now how a shared memory can simulate message passing model where communication is synchronous, transmission mechanism is broadcast and all the other three parameters are set to 0.

With each process we associate an unbounded array of shared register which all processes can read from but it only can write into. (Instead of an unbounded array we can use one unbounded size register.) To simulate a broadcast of a message a process writes to the next unused register in its associate array. When it has to read, it reads from each process all the new broadcast messages.

Exactly the same simulation is used to show that a shared memory can simulate the other three message passing models (at the upper left corner) where (1) communication is asynchronous and transmission mechanism is broadcast, (2) communication is synchronous and transmission mechanism is point-to-point, a (3) communication is asynchronous and transmission mechanism is point-to-point. We notice that in this simulations we strongly used the fact that the initial value of each shared register is $\perp$ (or it is set to some other value which is mutually known to all the processes).

Also, the simulation shows that in all of the above four models (where the parameter of message order is 0) the fact that they can be simulated by a shared memory model holds, even under the assumption that messages sent from one process to another are received in the order they were sent.

9 Discussion

We used an axiomatic approach to show that there is a class of problems which cannot be solved in a completely asynchronous shared memory system which supports only atomic read and write operations and where multiple undetectable crash failures may occur.

We introduced a simple game and reduced the question of whether a certain problem can be solved in asynchronous shared memory model where a number of processes may fail

mb pc	00	01	11	10	00	01	11	10
00	0 Yes	0 Yes	n No	0 ?	0 No	0 No	n No	0 No
01	0 Yes	0 Yes	n No	0 ?	1 No	n No	n No	1 No
11	n No	n No	n No	n No	n No	n No	n No	n No
10	0 No	0 No	n No	n No	0 No	0 No	n No	n No
	s=0				s=1			

Figure 2: *Each entry in the table is defined by different setting of the five system parameters, processes (p), communication (c), messages (m), transmission mechanism (b), and receive/send (s).*

prior to the execution to the question of whether there is a winning strategy for this game.

As mentioned in the Introduction, it follows from the results in [CM89] together with our results in section 7, that in a shared memory model which support atomic read and write operations, a problem can be solved by a *deterministic* protocol that can tolerate up to t *initial* failures if and only if the problem can be solved by a *randomized* protocol that can tolerate up to t *crash* failures. This result can also be shown to hold for asynchronous message passing model (assuming termination). It would be nice to show that this relationship holds also for other models.

It follows from our results that for both initial failures and crash failures, there exists a resiliency hierarchy. That is, for each $0 \leq t < n-1$ there are problems that can be solved in the presence of $t-1$ failures but can not be solved in the presence of t failures. These results extend and generalize previously known impossibility results for various asynchronous systems.

One conclusion that follows from our results is that for solving certain problems it is necessary to use stronger synchronization primitives than atomic read and write such as the well known test-and-set primitive, or alternatively to use randomized protocols.

Acknowledgements: We thank Michael J. Fischer and Shmuel Katz for helpful discussions.

References

[ABD$^+$87] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk. Achievable cases in an asynchronous environment. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 337–346, 1987.

[Abr88] K. Abrahamson. On achieving consensus using shared memory. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 291–302, 1988.

[BKWZ87] R. Bar-Yehuda, S. Kutten, Y. Wolfstahl, and S. Zaks. Making distributed spanning tree algorithms fault-resilient. In *4th Annual Symposium on Theoretical Aspects of Computer Science; Lecture Notes in Computer Science 247*, pages 222–231, 1987.

[BMZ88] O. Biran, S. Moran, and S. Zaks. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 263–275, 1988.

[BW87] M. Bridgland and R. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 52–63, 1987.

[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.

[CM85] M. Chandy and J. Misra. On the nonexistence of robust commit protocols. Manuscript, November 1985.

[CM86] M. Chandy and J. Misra. How processes learn. *Journal of Distributed Computing*, 1:40–52, 1986.

[CM89] B. Chor and L. Moscovici. Solvability in asynchronous environments. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 422–427, 1989.

[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.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[EFT84] H. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic.* Springer-Verlag, 1984.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Had87] V. Hadzilacos. A knowledge theoretic analysis of atomic commitment protocols. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pages 129–134, 1987.

[Her88] P. M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 276–290, 1988.

[LA88] C. M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1988.

[MW87] S. Moran and Y. Wolfsthal. An extended impossibility result for asynchronous complete networks. *Information Processing Letters*, 26:141–151, 1987.

[Tau87] G. Taubenfeld. Impossibility results for decision protocols. Technical Report 445, Technion, January 1987. Revised version appeared as Technion TR-#506, April 1988.

[Tau89] G. Taubenfeld. Leader election in the presence of $n-1$ initial failures. *Information Processing Letters*, 33:25–28, 1989.

[TKM89a] G. Taubenfeld, S. Katz, and S. Moran. Impossibility results in the presence of multiple faulty processes. *Information and Computation*, 113(2):173-198, 1994. Also in: *LNCS 405* (eds.:C.E. Veni Madhavan), Springer Verlag 1989, pages 109-120.

[TKM89b] G. Taubenfeld, S. Katz, and S. Moran. Initial failures in distributed computations. *International Journal of Parallel Programming*, 18:255–276, 1989.