A Gap Theorem for Consensus Types
Extended Abstract

Gary L. Peterson[*]
Spelman College

Rida A. Bazzi[†‡]
Gil Neiger[†]
Georgia Institute of Technology

Abstract
This paper presents a strong characterization that precisely determines the ability of n–process deterministic types to solve n–process wait–free consensus. This characterization, called the High Gap Theorem, has several important corollaries including a proof that Jayanti's hierarchy that allows multiple copies and read/write shared memory is robust for deterministic types.

1. Introduction.
Herlihy [Herl91] defined a complexity measure for shared memory types based on their power to solve wait-free consensus. Jayanti formally defined several variations of Herlihy's hierarchy [Jay93] and introduced the notion of *robustness*. It concerns a basic question on the power of combined types. Robustness means that if objects of two or more types separately cannot solve a problem then they can not solve the problem together. Jayanti studied robustness using consensus as the problem to be used in classifying types. Depending on the assumptions made about the properties of types and whether read/write shared variables (i.e., registers) are permitted, Jayanti was able to show some consensus hierarchies are not robust.

There remained one hierarchy, h_m^r, defined by Jayanti whose robustness question was not resolved. For a type T, let n be the smallest value such that multiple copies of objects of type T, augmented with read/write shared memory, can solve n–process wait–free consensus. Then $h_m^r(T) = n$. The main result of this paper implies h_m^r is robust for deterministic shared memory types.

We are very careful when discussing a type to indicate the number of processes that can access the type. Note that a 3–valued test–and–set object that can be accessed by 3 processes can solve 3–process consensus but cannot solve 4–process consensus [LA87]. While it is common to consider types such as 3–valued test–and–set to be one type, it is more proper, and more useful, to consider it to be a *family* of types.

Our main result, called the High Gap Theorem, is a characterization of n–process types. If a type has a certain property, then its objects can be used to solve n–process consensus. If it doesn't have the property, then its objects can be simulated using objects of any type that can do $(n-1)$–process consensus. Therefore, there is a gap between n-process types with consensus numbers n and $n-1$. No comparable gaps exist at lower levels in general. (There are n-process types that cannot do $(n-1)$ process consensus but can solve problems that not all $(n-2)$-process consensus types can solve.) Basic corollaries to the High Gap Theorem follow easily:

Corollary. Jayanti's h_m^r hierarchy is robust for deterministic types.

Corollary. A deterministic 2–process type has consensus number 2 if and only if its objects cannot be simulated by read/write shared memory.

One of the other issues raised by Jayanti concerns the definition of a shared memory type. Different results can be obtained based on different notions of type. For example, one of Jayanti's results permitted types to be non-deterministic. We therefore state the following properties about types that are used in this paper (see also [BNP94]):

- An object of an n–process type is accessed via n *ports*. (One port may allow reading, another writing.) Which process is to use which port is set at compile time.
- The object is a restricted finite state automaton with

[*]Research supported in part by the W. F. Kellog Foundation under a grant to the Center for Scientific Applications of Mathematics at Spelman College. Author's address: Computer Science Department, PO Box 333, Spelman College, 350 Spelman Ln SW, Atlanta, GA 30314-0339.

[†]Research supported in part by the National Science Foundation under grants CCR–9106627 and CCR–9301454. Authors' address: College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332-0280.

[‡]Supported in part by a scholarship from the Hariri Foundation

transitions labelled by ports, the values of the inputs used in the call (including the names of functions), and the output values returned from the call. The sets of possible input and output values are finite.

- The transition function is deterministic and total.
- Any state of the type can be specified as the start state for the object.
- The object is accessed a bounded number of times.

The above assumptions are reasonable for the study of complexity of types based on consensus number. While Jayanti allowed non-deterministic types, it seems more reasonable to assume any shared memory object must have a fully deterministic specification. Allowing more than one transition for a given call to an object was shown by Jayanti to result in hierarchies that are not robust. Similarly, basic properties of types may be different if there is no transition defined for some of the calls to an object.

Note that the types are not required to be *oblivious* [KM93]. That is, processes may not be permitted to perform the same operations on the object since not all operations are necessarily permitted on all ports. Many shared memory types are non-oblivious, e.g., one-reader, one-writer shared variables. Many types used for consensus are oblivious [Herl91]. The results of this paper still hold for oblivious types since they are a subset of non-oblivious types.

The assumptions about types given above implies that all aspects of the use of shared objects are bounded. Note that the consensus problem is the basis for the study of the complexity of types as given here. A type's complexity is defined by its ability to solve a consensus problem using a wait-free algorithm. Therefore having bounded objects implies that all wait-free algorithms for consensus using such objects call the objects a bounded number of times. This differs from simulations involving read/write shared memory where unlimited calls to object are allowed.

Note that the same assumptions are used in [BNP94] to prove the useful result that all non-trivial deterministic shared memory types can simulate bounded-use read/write shared memory. This implies the existence of a gap at the lowest end of the hierarchies between local memory and general read/write shared memory. Also, we do not need to separately assume having read/write shared memory in our simulations as whatever type is being used can also be used to simulate read/write memory. [BNP94] also gives a proof using the High Gap Theorem that Jayanti's h_m hierarchy for deterministic types is robust.

2. Definitions

N, the set of process numbers, is $\{1, 2, \ldots, n\}$ with the ith process denoted by P_i. Schedules are elements of N^*. We assume we have a fixed object of the given type. For any string X, X_i denotes the ith symbol in X.

Definition. For any string S in N^*, let $Pos(S, i, k)$ denote the location in S of the kth occurrence of the number i. (It is undefined if i does not appear in S k times.) Let $Num(S, i)$ denote the number of occurrences in S of i. Let $Pref(S, s)$ denote the prefix of S of length s. E.g., $Pos(S, 3, 2)$ is 4, $Num(S, 1)$ is 3 and $Pref(S, 4)$ is 2133 for the string $S = 2133211$.

Definition. An n-process *type* T (with n ports) is (n, Q, I, R, δ), where Q is the set of states, I is a set of input invocations (composed of a function name and parameters), R is a set of output responses (functional return values, new values of referenced parameters) and $\delta : Q \times N \times I \rightarrow Q \times R$. As per Section 1, Q, I and R are finite. An n-process *object of type* T is (T, π) where π is a permutation over N that maps process numbers to ports. (See [BNP94] for more details.)

Definition. An *input history* is $\mathcal{I} \in I^*$ denoting a sequence of input invocation values for a sequence of calls to the object. Similarly an *output history* is $\mathcal{R} \in R^*$ denoting a sequence of output response values for a sequence of calls to the object.

Definition. An *execution sequence* S_e is a 3-tuple: $(S, \mathcal{I}, \mathcal{R})$ where S is a schedule, $\mathcal{I}$ is an input history, and $\mathcal{R}$ is an output history, all of the same length. $|S_e|$ is $|S|$, and Pref(S_e, i) is (Pref(S, i),Pref$(\mathcal{I}, i)$,Pref$(\mathcal{R}, i)$). The execution sequence is *consistent* from a given start state q_0, if $\exists$ states $q_1, q_2, \ldots, q_s$ such that $\forall i$ $\delta(q_{i-1}, \pi(S_i), \mathcal{I}_i) = (q_i, \mathcal{R}_i)$ where $s = |S|$.

An execution sequence plus start state can be thought of as the complete record of a series of accesses to an object starting from the start state: the order of calls, the input values used in calls, and the return values of the calls. It is consistent if the given order and inputs to calls to the object produces the given responses.

Definition. For a schedule S of length s and process P_i, P_i's *observable operations at the end of* S, $V(S, i)$, is a pair of matrices ($I[1 \ldots n]$,$O[1 \ldots n,\ 1 \ldots s,\ 1 \ldots n]$) called the *I/O* matrix and the *Order* matrix, resp. For all j, $I[j]$ is $Num(S, j)$ but $I[i]$ is $Num(S, i) + 1$. For all j, k, s_j, $O[j, s_j, k]$ is $Num(Pref(S,Pos(S, j, s_j)), k) - 1$. but $O[i,Num(S, i) + 1, k]$ is $Num(S, k) - 1$.

Note that the word is "observable" and not "observed." In our algorithms, processes between calls to the object will read and write information using shared memory. The information processes can be guaranteed to easily obtain include: outputs from calls that are certain to have been completed, inputs to calls that are certain to have started, and any ordering information that processes could be sure to obtain between calls.

$I[j] = s_j$ means that is possible for P_i to learn via shared memory before its next call to the object the output of P_j's $s_j - 1$st and earlier calls and the input to P_j's s_jth and earlier calls to the object. For $i = j$, P_i will know the input to its own $s_i + 1$st call and the output of its s_ith call as well.

$O[j, s_j, k] = s_k$ means that P_i before its next call to the object will be able to learn that P_j's s_jth call to the object occurred after P_k's s_kth call to the object. (While in fact P_j's s_jth call occurred after P_k's $s_k + 1$st call, there is no easy way for P_i to learn this.) For $i = j$, P_i will also be able to know before its next call that P_k has completed its s_kth call.

It will be shown how processes can actually obtain the observable information in the next section. The *I/O* information is quite simple using read/write memory. The *Order* information can be obtained easily with a simple object that for two processes can be done with read/write memory and for three or more processes can be done with objects whose type is weaker than 2–process consensus types.

Definition. Given a start state q_0, a process number i and an execution sequence $S_e = (S, \mathcal{I}, \mathcal{R})$ of length s that is consistent from q_0, a *visible history instance for* P_i $h(q_0, S_e, i)$ is a triple of three matrices $(\mathcal{I}_h[1 \ldots n, 1 \ldots s], \mathcal{R}_h[1 \ldots n, 1 \ldots s], O)$. O is the order matrix in P_i's visible history $V(S, i) = (I, O)$. If $s \leq I[j]$ and $s' = Pos(S, j, s)$, $\mathcal{I}_h[j, s']$ is $\mathcal{I}_{s'}$ and is $\perp$ otherwise. If $s < I[j]$ and $s' = Pos(S, j, s)$, $\mathcal{R}_h[j, s']$ is $\mathcal{R}'_s$ and is $\perp$ otherwise.

A visible history instance for P_i records the actual information that is observable to P_i at a point in time: inputs values for all calls to the object that are observable, output values for all calls that are observable, and the order information. A visible history is a sequence of visible history instances, each one representing what is observable to P_i just before each of its calls to the object.

Definition. A *visible history for* P_i $H(q_0, S_e, i)$ is $(h_1, h_2, \ldots, h_{s'})$ where $Num(S, i) = s'$ $(S_e = (S, \mathcal{I}, \mathcal{R}))$ and $\forall m$, $h_m = h(q_0, Pref(S_e, m), i)$.

During a simulation of an object, each process needs to know the inputs to calls to the object by other processes. Since a process has choices of input values, these choices may make a simulation possible or impossible. Our simulation algorithms work under the assumption that the processes are "playing fair" in their choice of input values. It is reasonable to expect that input parameters might depend on things such as outputs from previous calls to the object, data obtained from other processes between calls to the object, etc. However, a process cannot be permitted to choose an input value that is dependent on the internal state of an object that is not visible to the process. This is formally stated using the definitions from Peterson and Reif's Multiple–Person Alternation automata (MPA's) [PR79].

Definition. A *strategy* is $\gamma : N^* \times N \rightarrow I$. (From a given point in a schedule and who is to call the object next, it gives the input invocation for that call.) An execution sequence $S_e = (S, \mathcal{I}, \mathcal{R})$ is *produced by* γ *from* q_0 if $\forall i$ $\gamma(Pref(S, i-1), S_i) = \mathcal{I}_i$ and S_e is consistent from q_0.

That is, a strategy is a complete list of what parameters to the object each process will use in a given algorithm. If P_i's visible history is the same for two different schedules, P_i must call the object with the same input parameters.

Definition. A strategy γ is a *legal strategy from* q_0 if for all execution sequences $S_e = (S, \mathcal{I}, \mathcal{R})$ and $S'_e = (S, \mathcal{I}, \mathcal{R})$ produced by γ from q_0 and all process numbers i, $H(q_0, S_e, i) = H(q_0, S'_e, i)$ implies $\gamma(S, i) = \gamma(S', i)$.

Our proofs hinge on the fact that if an object can solve n–process consensus it can do so with each process calling the object a bounded number of times. On the other hand, it will be proven that if the object cannot solve n–process consensus it is due to some processes not being able to determine the ordering in the schedule no matter how many times they call the object. The following formalizes these bounds where one process P_i is unable to decide within a bounded number of steps.

Definition. A *schedule bound* σ is $(s_1, s_2, \ldots, s_n)$ and $|\sigma| = \sum s_i$. Given an schedule S, a process number i and a schedule bound $\sigma = (s_1, s_2, \ldots, s_n)$, *$S$ is bounded by* (i, σ) if $\forall j \neq i$ $Num(S, j) \leq s_j$ and $Num(S, i) = s_i$. An execution sequence $S'_e = (S, \mathcal{I}, \mathcal{R})$ is bounded by (i, σ) if S is bounded by (i, σ).

Definition. Given three process numbers i, j, and k with $j \neq k$, a start state q_0, a schedule bound σ, and a legal strategy γ starting from q_0, *P_i cannot distinguish between j and k in schedules bounded by σ starting from q_0 using γ* if there exists a pair of execution sequences $S_e = (S, \mathcal{I}, \mathcal{R})$ and $S'_e = (S', \mathcal{I}', \mathcal{R}')$ produced by γ from q_0, where $S_1 = j$, $S'_1 = k$ and S_e and S'_e are bounded by (i, σ) such that P_i's visible histories $H(q_0, S_e, i)$ and $H(q_0, S'_e, i)$ are equivalent.

That is, P_i is unable to tell apart two execution sequences (one started by P_j and the other by P_k) using its history of visible information. The power of a type is directly related to the abilities of processes to determine the order of their operations.

Definition. For objects of an n–process type, a start state q_0, a schedule bound σ, and a legal strategy γ from q_0 define an *equivalence graph* $G(q_0, \sigma, \gamma) = (V, E)$ where $V = \{1, 2, \ldots, n\}$, $E = \{(j, k) | \exists i$ such that P_i cannot distinguish between j and k in schedules bounded by σ starting from q_0 using $\gamma\}$.

Note that equivalence graphs are undirected and there is an equivalence graph for each start state, schedule bound and strategy. An edge (j, k) in an equivalence graph captures the notion that at least one process is not be able to tell whether at least one pair of execution sequences starts with j or k.

High Gap Theorem. If, for objects of an n–process deterministic type T, there is a state q_0, a schedule bound σ, and a legal strategy γ starting from q_0 such that the Equivalence Graph $G(q_0, \sigma, \gamma)$ is not connected then objects of type T can solve n–process consensus, otherwise objects of type T can be simulated by any object that solves $n-1$–process consensus.

3. Basic Types and Algorithms

We define some basic types and algorithms needed later. The proofs and some constructions are given in the full version of the paper. For simplicity we equate the consensus problem with the election problem. That is, processes elect one of the competing processes and return its process number. Clearly an election algorithm can be converted into a consensus algorithm and vice versa.

3.1 Basic consensus types.

Definition. Let objects of type T_n, accessed by n processes, with default start state $\perp$ be the *standard consensus type*. That is, it returns the index of the first process that calls it when started from the start state.

Definition. Given two disjoint non–empty sets of processes $\mathcal{P}_1$ and $\mathcal{P}_2$ (of size n and n'), let $T_{s.n,n'}$ denote the *splitting type*. Processes calling the type when started in $\perp$ state return the set number that the first process to call the type belongs to.

It is easy to prove that objects of type $T_{s:n,n'}$ can do $n+n'$–process consensus using a variation of Loui and Abu–Amara's tournament algorithm for three–valued test–and–set [LA87].

3.2 The overlooking type.

For some of our simulations we will need a type that allows processes to know the result of one process reading another process's variable (without having to wait for the reader to write the value it saw). Let q be a one–writer, one–reader, write–once, read–once Boolean variable. q has initial value `false`, and P_2 writes `true` to it. P_1 will return `false` or `true` depending on whether it read before or after P_2 wrote. We want to allow the $n-2$ *overlooking* processes to be able to know whether P_1 will return `true` or `false` even though P_1 may not have written its return value to shared memory. A special value $\perp$ is returned to overlooking processes if neither P_1 nor P_2 has executed its operation (because their order is not yet known). For n processes this type, called the *overlooking* type, is denoted by $T_{o:n}$. A type $T_{o:n}$ object can be simulated using type T_2 objects (and with just read/write memory for $n=2$).

Definition. Define objects of type $T_{o:n}$, called the *overlooking* type, by the atomic operations given in Figure 3.1. Let $\perp$ denote a special non-Boolean value. The default initial value of the object's shared variables (q1,q2) is ($\perp$,`false`). If either P_1 or P_2's operation has occurred then the overlookers will return the same value that the reader returned (or will return).

Theorem. An object of type $T_{o:n}$ can be simulated using objects of type T_2 and by read/write shared memory for $n=2$.

```
shared variables
  boolean+ q1,q2
For $P_1$, the reader:
integer function Read
q1=q2
return q1
For $P_2$, the writer, writing true:
function Write
q2=true
For $P_i$, an overlooker:
integer function Overlook
if q1 $\neq \perp$ then
  return q1 /* $P_1$ done, return its value */
elseif q2 then
  return true /* $P_2$ done, $P_1$ will return true */
else return $\perp$ /* Neither $P_1$ or $P_2$ done */
```

Figure 3.1 The overlooking type.

3.3 The process order algorithm.

We next outline the *process order algorithm*. The process order algorithm is a system which allows processes to determine an approximate ordering of a series of operations on an object using only objects of type $T_{o:n}$ (which has consensus number at most 2). An exact ordering cannot be obtained since that is equivalent to n–process consensus. (An exact ordering would indicate which process went first). A process will not have an exact view of the order of operations but the process order algorithm allows processes to rule out schedules that are very dissimilar to the actual schedule. The system consists of three components: a read and two write steps. The read step, called *ReadOrder*, returns to the process the most recent information on the order and actions of the processes, i.e., a visible history instance. The write steps, called *WriteInput* and *WriteOutput*, are used to notify other processes of the input values and result values to a call to an object. The algorithms are called in the order: *ReadOrder*, *WriteInput*, call (or simulate a call to) the object, *WriteOutput*. The process order algorithms can be implemented using read/write shared variables and the overlooking type. The constuction is given in the full version of the paper.

3.4 The partial edge choice problem

The general structure of the proof that if a type's equivalence graph is connected, then objects of that type can be simulated using T_{n-1} type objects requires two subalgorithms. The first is used to have processes select an edge in a connected equivalence graph. The second is used to ensure that processes simulations either stay in lockstep or all processes choose the same simulation. The lockstep algorithm is discussed in the next subsection.

Each edge in an equivalence graph represents a pair of schedules whose first symbols are the end nodes of the edge and at least one process cannot tell the execution sequences apart. Our construction requires that processes select one such edge and carry out simulations of both execution sequences. Using objects of type T_{n-1}, we cannot ensure that all processes can choose the same edge, but we can come close enough.

Definition. Given an n–node connected graph with each node associated with 1 of n processes, an algorithm solves the *partial edge choice problem* if:
1. Every process's procedure is wait–free.
2. Process P_i returns with a non–empty set $\mathcal{E}_i$ of edges.
3. For at least $n-1$ processes P_i, $|\mathcal{E}_i| = 1$.
4. For all P_i with $\mathcal{E}_i = \{(j,k)\}$, for all i', $(j,k) \in \mathcal{E}_{i'}$.
5. For any edge (j,k) in any $\mathcal{E}_i$, at least one of P_j or P_k has started its procedure.
6. For any process P_i with $\mathcal{E}_i = \{(j,k)\}$, there is a shared variable `chosen` that is set to (j,k) before P_i completes its procedure.
7. If $|\mathcal{E}_i| > 1$, then $\mathcal{E}_i \subseteq \{(i,j)|(i,j)$ is an edge in the graph.$\}$

That is, at least $n-1$ processes will agree on just one edge and at least one process on that edge is active. There is at most one process that may not know which edge the others have chosen but it does know that the edge is incident to its node. It will also be able to know the particular edge by reading `chosen` once any other process completes its algorithm. Solvability of the partial edge choice problem hinges on the graph being connected.

Outlined in Figure 3.2 is the Partial Edge Choice algorithm that solves the partial edge choice problem using only type T_{n-1} objects. Select any node in the graph and make that the root. Assume that the nodes/processes are numbered in breadth first search order. Each non–root node will use its parent link as its initial choice of an edge. The root's initial choice is any child edge. (All initial choices are fixed and known to all processes.) The algorithm has the processes doing competitions at various nodes starting at the root going down towards active processes. Each process is initially trying to direct the competition towards itself. At some point the competition will terminate at some node and that node's initial edge choice is selected. $\mathtt{T}_{n-1}\mathtt{(j,i)}$ denotes a call by P_i to the jth of n copies of type T_{n-1} objects.

Theorem. The partial edge choice algorithm solves the partial edge choice problem.

3.5 The lockstep algorithm.

The `lockstep` algorithm given in Figure 3.3 is used to force processes to either execute their operations in a specified order or agree on one value. It uses only objects of type T_{n-1} and read/write shared memory. The algorithm uses n objects of type T_{n-1} with a call by P_i of the jth copy denoted by $\mathtt{T}_{n-1}\mathtt{(j,i)}$. Many copies of this algorithm will be needed in some simulations.

Theorem. The `lockstep` algorithm has the following *lockstep property*: Either all processes return the same choice or for all i and j, the calls to `lockstep` by P_i and P_j overlap.

```
Assumes G is a connected n–node graph
shared variables
  boolean active[1..n] initially false
    /* who's active */
  integer Winner[1..n] initially 0
    /* winner at node i */
  integer favorite[1..n]
    /* P_i's favorite to win */
local variables
  integer j,contender
set_of_edges function EdgeChoice(
    graph G, integer i)
/* let others know you're active */
active[i] = true
Contender = 1 /*root of G, first possible winner */
favorite[i] = i /* initially you're trying to win */
while true do
  if i is not a descendant of Contender
    then favorite[i] = any j, active[j] and
          (j is a descendant of Contender)
  if Contender ≠ i then
    /* not competing at your node */
    /* compete, look at winner's favorite */
    j = favorite[T_{n-1}(Contender,i)]
    if j <= Contender then
      /* competition ended at j */
      chosen = j's initial edge
      return chosen
    /* indicate winner */
    Winner[Contender] = j;
    Contender = j
  else /* P_i not allowed to compete at note i */
    for each j = child of i do
      if T_{n-1}(j,i)≠ i then exit loop
    if Winner[i] = 0 then return
      all edges incident on i;
    else Contender = Winner[i]
```

Figure 3.2 The partial edge choice algorithm.

```
Called by P_i with initial choice inchoice:
shared variables
  integer winner[1..n] initially 0
  integer choice[1..n]
integer function lockstep(integer i,inchoice)
choice[i] = inchoice /* display choice */
for j = 1 to n do
  if j ≠ i then /* on n – 1 of n iterations */
    k = T_{n-1}(j,i) /* first on jth iteration */
    choice[i]= choice[k] /* winner's choice */
    winner[j] = k /* signal who won to P_j */
   else /* on one of n – 1 iterations */
    if winner[j] ≠ 0 then
      /* winner of ith iteration known */
      choice[i] = winner[j]
    /* else keep choice */
return choice[i]
```

Figure 3.3 The lockstep algorithm.

4. High Gap Theorem: "then" case.

High Gap Theorem ("then case".) If, for objects of an n–process deterministic type T, there is a state q_0, a schedule bound σ, and a legal strategy γ starting from q_0 such that the Equivalence Graph $G(q_0, \sigma, \gamma)$ is not connected then objects of type T can solve n–process consensus.

Proof: Given $G(q_0,\sigma,\gamma)$, let $(\mathcal{P}_1,\mathcal{P}_2)$ be a partition of nodes into two non-empty sets with no edges between them. Note that $(\mathcal{P}_1,\mathcal{P}_2)$ is also a partition of the processes. Let $\sigma = (s_1,\ldots,s_n)$. For each process P_i consider the algorithm in Figure 4.1.

```
HistoryInstance $h$[1..$s_i$] initially null
ObjectInputType $I$
ObjectOutputType $R$
integer $m$ initially 1 /* No. calls to the object */
while true
  $h$[$m$]=$ReadOrder(i,m)$
  /* Determine sequences that could
     correspond to visible information */
  $\mathcal{S}$ = $\{S|\exists S_e = (S,\mathcal{I},\mathcal{R})$ produced by $\gamma$ from $q_0$
        and bounded by $\sigma$ such that
        $H(q_0,S_e,i) = h\}$
  /* All start with nodes in same set? */
  if $\{j|j = S_1$ and $S \in \mathcal{S}\} \subseteq \mathcal{P}_t$ return $t$
  $S$ = any element in $\mathcal{S}$
  $I$ = $\gamma(S,i)$ /* Use strategy for inputs */
  $WriteInput(i,m,I)$
  $R$ = $CallObject(i,I)$ /* Call the object */
  $WriteOutput(i,m,R)$
  $m$++
```

Figure 4.1 Algorithm for P_i, "then" case.

Note that the above simulation uses only the order algorithms to communicate and therefore requires only type T_2 objects (and read/write memory if n is 2). While P_i can choose from many S's to use in calling γ, all of them must give the same I. Otherwise, there would be two execution sequences whose visible histories are the same to P_i but γ returns different values. That is, γ is not legal.

Assume that P_j was the first process to call the object and $P_j \in \mathcal{P}_t$. (Therefore no other process completed its algorithm before P_j started.) Eventually, P_i's $\mathcal{S}$ will not contain any schedule starting with k for all k not in $\mathcal{P}_t$. Otherwise, after s_i steps, there would be two execution sequences whose schedules start with j and k resp., and P_i cannot distinguish them. But then there must be an edge between j and k in the equivalence graph and the node sets $\mathcal{P}_1$ and $\mathcal{P}_2$ are not unconnected, a contradiction. Hence all processes are guaranteed to stop and all will return the same t. Therefore this algorithm using $T_{o.n}$ type objects can implement the split type $T_{|\mathcal{P}_1|,|\mathcal{P}_2|}$ which in turn can solve n-process consensus. Note that for two process, $T_{o:2}$ requires only read/write memory and therefore objects of type T alone can implement T_2. For n three or more, first T_2 is implemented using the two-process version and from that the $T_{o:n}$ type is implemented from T. Therefore objects of type T alone can then be used to solve n-process consensus. □

5. High Gap Theorem: "else" case.

High Gap Theorem: "else" case. For an n-process deterministic type T, if for all states q_0, schedule bounds σ, and legal strategies γ starting from q_0 the Equivalence Graph $G(q_0,\sigma,\gamma)$ is connected then objects of type T can be simulated by any object that solves $n-1$-process consensus.

The main features of the proof of the simulation include:

• The proof is by induction on $|\sigma|$. The base case of $|\sigma| = 1$ is trivial. Only one process calls the object once, hence there is no parallelism. The simulation assumes that for all σ', $|\sigma'| < |\sigma|$, that a simulation of the type using type T_{n-1} objects can be done. The simulation will call itself recursively but with a smaller σ.

• The simulation basically forces the adversary scheduler to do one of two things, both of which are to our advantage. Either it does the simulation in a specified order (in which case only $n-1$ processes need to make a decision) or it does the simulation out of order (which due to the use of the lockstep algorithm allows all n processes to make the same decision).

• The strategy γ being used when accessing the object is not known in advance. Hence multiple simulations are started, one for each possible γ that is unique up to $|\sigma|$ steps. As the simulation progresses, some strategies may be discarded as not being consistent with the actual calls to the object. At least one strategy simulation will continue to execute.

• The core of the simulation is for the processes to simulate a pair of execution sequences that correspond to an edge in the equivalence graph. The processes use the partial edge choice algorithm to choose the edge. However, one process might get back more than one edge from the algorithm. In the beginning it will have to simulate pairs of execution sequences for all edges in its set. If no other process joins the simulation, all simulations will return the same result. If another process does join the simulation, the first process can now determine which edge the others have chosen and only simulate that edge from then on.

• Because of the very large number of simulations that processes may be participating in simultaneously, we must ensure that processes doing the same simulation use the same shared variables and when doing different simulations use different shared variables. For each simulation we have a simulation number `w`. *All* global and shared memory variables given in the algorithms are components of an array of records called **`sharedmem[1..max_num_sims]`**. At the beginning of every function we have a "**`with sharedmem[w] do`**" statement so that the process is accessing only the variables it needs to during that simulation step. When a process initiates a new subsimulation, it must give the subsimulation a simulation number. We assume a function `SimNum(`q_0`,w,(j,k),`γ`)` that maps a start state, the current simulation number, the edge in the graph and strategy that is to be simulated into a unique simulation number. At the top level, the simulation will start with a simulation number of 1.

The basic task of the simulation is to try as best as possible to determine a linearization of the calls to the simulated object and return the appropriate values for that linearization. In particular, knowing which process is considered first is crucial. But as [Her94,Plo89] have shown, this is in fact n–process consensus. Given only objects of type T_{n-1}, n processes cannot agree on just one value. But they can agree on at most two values (2–set consensus [Cha93]). That is, all processes can at least agree on two processes, say P_j and P_k, that are *candidates* for going first.

T_{n-1} is sufficiently strong that we can limit in our favor which two values can be chosen. Ideally, we want to choose two processes whose nodes are adjacent in an equivalence graph. By the definition of connection in an equivalence graph, that means there is a pair of execution sequences from the start state, S_e and S'_e (whose schedules start with j and k resp.), such that there is at least one process P_i that cannot distinguish between S_e and S'_e. That is, P_i will always see the same return values, order of execution information, etc. in both sequences. Hence P_i does not have to choose between j and k, it will return the same result no matter which of j and k the others choose as the first. That leaves $n-1$ other processes that might need to choose, and T_{n-1} can be used to do that.

The above discussion sounds naive in that there is no requirement that the adversary scheduler actually cooperate and schedule operations so that the simulated sequences are linearizations of the chosen sequences. This is where the lockstep algorithm is used. All processes do their simulation steps in a particular order combined with several calls to the lockstep algorithm. If the schedule corresponds to the desired simulations, then there are at most $n-1$ processes that need to decide. On the other hand, if the scheduler does not order the steps as planned, then there are two calls to the same lockstep algorithm that do not overlap and all processes will end up making the same choice. In order to construct the sequence of simulated steps and locksteps, we first need some more definitions.

Definition. *Overlap sequences* are elements of N^*.

An overlap sequence denotes a form of schedules that allows for operations to take a non–zero amount of time, i.e. there are non–atomic. The digits in an overlap sequence indicate the order of the *ends* of the respective processes' operations. (One can safely assume that, except for the first operation, the beginning of one process's operation occurs immediately after the end of its previous operation.)

Definition. A schedule S is a *linearization of an overlap sequence* $\mathcal{S}$ if S is a permutation of $\mathcal{S}$ and for all i, j, k, k', with i and j appearing at least k and $k'+1$ times in $\mathcal{S}$, $Pos(\mathcal{S},i,k) < Pos(\mathcal{S},j,k')$ implies $Pos(S,i,k) < Pos(S,j,k'+1)$.

That is, if P_i's kth operation ends before P_j's k'th operation ends then P_i's kth operation must occur before P_j's $k'+1$st operation in any linearization. Note that the order of the last operations of the processes (in a linearization) is not necessarily constrained.

Definition. Two schedules S_1 and S_2 *share an overlap sequence* if there exists an overlap sequence $\mathcal{S}$ such that S_1 and S_2 are each linearizations of $\mathcal{S}$.

It will be shown that if two schedules do not share an overlap sequence, it is relatively easy for the processes to distinguish between them during a simulation. For example, in the case of two processes, each performing two operations, the only schedules that do not share an overlap sequence are 1122 and 2211. It is quite simple to have the processes use shared variables between calls in order to rule out one of these as a possible linearization. In fact, the process order algorithm (which uses the overlook type) does this and more.

We first state two basic properties of schedules that do not share an overlap sequence.

Lemma. Schedules S_i and S_j containing operations by P_i and P_j do not share an overlap sequence iff $\exists k, k'$ such that $Pos(S_i,i,k) > Pos(S_i,j,k'+1)$ and $Pos(S_j,i,k+1) < Pos(S_j,j,k')$.

Lemma. If $S_e = (S,\mathcal{I},\mathcal{R})$ and $S'_e = (S',\mathcal{I},\mathcal{R})$ are execution sequences associated with an edge in an equivalence graph then S and S' share an overlap sequence.

An example. Given two process P_i and P_j, with $S_1 = jjijiij$ and $S_2 = ijjijji$, S_1 and S_2 are linearizations of the overlap sequence $\mathcal{S} = jjiijij$. The sequences for simulating the operations for S_1 and S_2 are given below. The processes do simulated steps in S_1 (denoted by 1) and simulated steps in S_2 (denoted by 2) intermixed with calls to lockstep algorithms (denoted by L's). Brackets denote the beginning and end of a simulated call. (L) denotes calls to a lockstep that processes will perform only if a choice has to be made. The calls to matching locksteps have been lined up. The scheduler is not required to order the events as given below, but if it doesn't then *all* processes agree on the same choice.

```
Pi: [2L    L1   L]  [2L1] [L1   L2] (L) (L)
Pj: [1L2] [1L2] [L1   L2   L] [2L1   L] (L)
```

E.g., P_i and its third call to the object will: do the third lockstep, simulate a step in S_1, do the fourth lockstep, simulate a step in S_2 and do the fifth lockstep. The basic properties of simulation orderings is given below. The construction of such an ordering from the three sequences is easy using these properties and is given in the simulation algorithm.

• During a call to the simulation, each process will do exactly one simulation step in each sequence.

• The number of locksteps needed is $|S|+1$ (one extra for the final decision).
• Between any pair of lockstep calls, there is exactly one simulation step by any process in either sequence.
• Taking who does a simulated step in S_1 as given above in order does in fact give S_1. The same with S_2.
• if the tth occurrence of i in S_1 is at m, then the tth call by P_i is done between the $m-1$st and mth lockstep.
• Let m be the position of the tth i in S and m' be the last lockstep P_i did in its most recent call to the object. P_i on its tth call to the simulation will do locksteps $m'+1$ to m.
• If the scheduler were to arrange it so that two operations in the same sequence were performed out of order, then the intervening locksteps would not all overlap and all processes would choose the same simulation sequence. (Which of course would no longer correspond to either of the originals.)

The simulation algorithm is given in Figures 5.1, 5.2, 5.3a, and 5.3b. Note that only read/write shared memory types and the types needed by the partial edge choice and lockstep algorithms are used (which in turn use no types more powerful than T_{n-1}).

The toplevel algorithm `SimObject`, for simulating the object from a start state q_0, for at most σ steps, for simulation number `w`, for the `t`th call having input `I` is given in Figure 5.1. On the first call, it creates a set of possible strategies and will simulate all of them. The process collects visible history information using the process order algorithm. The visible history information and input values can be used to eliminate possible strategies.

The midlevel algorithm `SimStrategy`, for simulating the object using a particular strategy γ, etc. is given in Figure 5.2. On the first call, it calls the partial edge choice algorithm to try and select an edge (and therefore two execution sequences) to simulate. Since more than one edge may be returned all are simulated. But once the choice of the others is known, the extras are dropped.

The lowest level algorithm `SimEdge`, for simulating an object using a given edge, etc. is given in Figure 5.3. On its first call it determines for the two possible candidates for first process, P_j and P_k, what their next step would be from q_0, etc. On each call it looks at the two schedules and their overlap sequence to determine the order in which to perform its own simulated steps and locksteps. Note that it calls the toplevel simulation `SimObject` but with a smaller schedule bound.

Once a process completes all of its simulation steps and locksteps during a call to `SimEdge`, it has two possible results to choose from. If there are both equal it does not matter which to return. If they are not equal then it must choose. It does this by finishing any remaining locksteps and returning the value from the simulation chosen at the end. There may be further calls to the object, therefore on later calls it need only run the simulation for the previously chosen one.

```
ObjectResultType function SimObject(
      State $q_0$, /* Initial state of simulation */
      ScheduleBound $\sigma$, /* Max. num. steps */
      integer w, /* Simulation No., init. 1 */
      integer t, /* Count of $P_i$'s calls to object */
      ObjectInputType I) /* Input values to call */
static variables
   set of Strategy $\Gamma$ /* Current legal strategies */
   History h[1..$|\sigma_i|$] /* Visible history */
local variables
   Schedule S
   set of Schedule $\Psi$ /* Possible schedules */
   ObjectResultType R /* Result from a sim. */
/* Use the right shared variables */
with SharedMem[w] do
if t=1 then /* determine possible strategies */
   $\Gamma$ = $\{\gamma|$ $\gamma$ is a legal strategy from
         $q_0$ for schedules bounded by $\sigma\}$
h[t]=ReadOrder(i,t) /* Get order info. */
/* Determine possible schedules */
$\Psi$ = { S $|\exists S_e = (S, \mathcal{I}, \mathcal{R})$ produced by some
         $\gamma \in \Gamma$ from $q_0$ and bounded by $\sigma$
         such that $H(q_0, S_e, i)$ =h}
forall $\gamma \in \Gamma$ /* Remove invalid strategies */
   if forall S$\in \Psi$, $\gamma$(S,i) $\neq I$ then
      $\Gamma$ = $\Gamma$ - $\{\gamma\}$
WriteInput(i,t,I) /* Display input values */
forall $\gamma \in \Gamma$ /* Simulate a step in each strategy */
   R = SimStrategy($q_0$,w,$G(q_0,\sigma,\gamma)$,t,I,$\sigma$,$\gamma$)
WriteOutput(i,t,R) /* Display result values */
t++ /* completed one sim. call */
return R /* return the value from a simulation */
```

Figure 5.1 Top-level simulation algorithm for P_i.

Proof outline, High Gap Theorem, else case: Assume that the simulation algorithm fails for some object starting from some start state q_0, using a strategy γ, and bounded by σ calls. Since the equivalence graph $G(q_0, \sigma, \gamma)$ is connected, a simulation is initiated for some edge in the equivalence graph, using γ from q_0 (and perhaps others). Recall that this is a proof by induction on $|\sigma|$. What are the possible ways for the entire simulation to fail?

The simulation fails for the selected edge. The first possibility is that incorrect results are returned from the subsimulation. But by the induction hypothesis, the subsimulations are correct regardless of the actions of the scheduler.

The second possibility is that the scheduler choose a schedule that did not correspond to the sequences for the edge. But then the locksteps do not overlap and all processes choose the same subsimulation.

Another consideration is that a process P_j may get different results from the subsimulations (which were done in order) and have to choose but still has remaining operations in the simulated sequences. But that means that the process P_i that cannot distinguish between the two schedules will in fact be able to note the difference due to P_j writing the result of that call before starting

```
ObjectResultType function SimStrategy(
      State $q_0$, /* Initial state of simulation */
      integer w, /* Sim. number */
      Graph G, /* Connected Eq. Graph */
      integer t, /* Count of calls */
      ObjectInputType I, /* Input values to call */
      ScheduleBound $\sigma$, /* Max. num. steps */
      Strategy $\gamma$) /* The strategy to use */
shared variables
   Edge chosen /* From EdgeChoice */
   Boolean active[1..n] /* From EdgeChoice */
   ObjectInputType I[1..n], /* Inputs to calls */
static variables
   integer mychoice[1..n,1..n]
      /* current choice of winners */
   ObjectResultType result[1..n,1..n]
      /* Results of edge simulations */
   integer w[1..n,1..n]
      /* Edge simulation numbers */
   set of Edge $\mathcal{E}$ /* Set of edges to simulate */
local variables
   integer j,k /* Process numbers for an edge */
/* Use the right shared variables */
with SharedMem[w] do
if t = 1 then /* First call choose an edge */
   I[i] = I /* Tell others input to call */
   $\mathcal{E}$ = EdgeChoice(G,i,chosen,active)
   forall (j,k)$\in\mathcal{E}$ do
      if active[j] then
         mychoice[j,k] = j /* Initial choice */
      else mychoice[j,k] = k
      /* Assign Sim. Nos. */
      w[j,k] = SimNum($q_0$,$w$,(j,k),$\gamma$)
forall (j,k)$\in\mathcal{E}$ do /* Sim. each edge 1 step */
   result[j,k] = SimEdge($q_0$,w[j,k],G,j,k,$\sigma$,t,
      mychoice[j,k],I,I[j],I[k])
if chosen $\neq$ null then /* Only 1 edge to sim. */
   E ={chosen}
return result[j,k] for any (j,k)$\in\mathcal{E}$
```

Figure 5.2 Strategy–level simulation algorithm for P_i.

the next one. I.e., the edge does not in fact exist in the equivalence graph. Hence, all processes will always return a correct value from SimEdge.

For SimStrategy to return an incorrect value would require that different edge simulations return different results and the "wrong" one is returned. But different results from the edge simulations would mean some other process has also started its edge simulations. That means the first process can now find out which edge the others choose and drop the others. Hence, it will only have the one result from SimEdge to use and it is correct for the given simulation.

For SimObject to return an incorrect result would require that the SimStrategy calls return different values and the wrong result is chosen. But different results from the calls means that some strategies can be eliminated. Hence the results from all remaining calls are the same. Otherwise, there would be two strategies that are identical up to the known ordering information but return different values. I.e., at least one of them is in fact not a legal strategy and would not still be in Γ. □

```
ObjectResultType function SimEdge(
      State $q_0$, /* Initial state of simulation */
      integer w, /* Sim. number */
      Graph G, /* Connected Eq. Graph */
      integer j,k, /* Processes on edge */
      ScheduleBound $\sigma$, /* Max. num. steps */
      integer t, /* Count of calls */
      integer mychoice /* j,k choice */
      ObjectInputType I,Ij,Ik)
         /* Input values to calls */
shared variables
   integer choice[1..$|\sigma|+1$][1..n]
      /* for Locksteps */
   integer winner[1..$|\sigma|+1$][1..n] initially 0
static variables
   integer m /* index in schedules */
   Boolean decided initially false
      /* chosen a sim.? */
   State q[1..2] /* states after $q_0$ for j and k */
   ScheduleBound $\sigma$[1..2] /* subsim. bounds */
   Schedule S[1..2], $\mathcal{S}$ /* sequences for edge */
   ObjectResultType result[1..2]
      /* results of simulations */
   ObjectResultType R[1..2]
      /* Results from 1 step of $\delta$ */
```

Figure 5.3a Edge simulation algorithm for P_i, data.

Since the equivalence graphs for a type are either all connected or one of them is not, an n–process type's power cannot be in between T_n and T_{n-1}. This gives the following simplification:

Corollary: For all n–process deterministic types T, either T has consensus number n or objects of type T can be simulated by objects of any type with consensus number $n-1$.

The High Gap Theorem is more than enough to answer Jayanti's robustness question for these types:

Corollary. For two deterministic n–process types T and T', if $h_m^r(T)$ and $h_m^r(T')$ are less than n then objects of type T and T' combined cannot solve n–process consensus.

Proof: Since objects of type T and T' cannot do n–process consensus then each can be simulated by type T_{n-1} objects, which in turn cannot do n–process consensus.□

For two processes, the High Gap Theorem implies that there are no types strictly between 2–process consensus and read/writes. In [BNP94] it is shown that objects of any non-trivial deterministic type can be used to simulate read/write memory (a low gap theorem). Hence we have:

Corollary. All 2–process deterministic types can placed in exactly one of the following categories: local variables, read/write shared variables, or 2–process consensus types.

The class *common2* was introduced in [AWW3] as a set of standard types that have two main properties: they all can do 2–process consensus and each can simulate any other type in *common2*. As a result, all types in *common2* can be simulated by any 2–process type that

```
/* Use the right shared variables */
with SharedMem[w] do
if t=1 then /* first call? */
   /* If j goes first... */
   (q[1],R[1])=$\delta(q_0,\pi(j),Ij)$
   /* If k goes first... */
   (q[1],R[2])=$\delta(q_0,\pi(k),Ik)$
   /* What seqs. to simulate */
   (S[1],S[2],$\mathcal{S}$)= sequences and overlap seq.
        for edge (j,k) in G
   /* smaller schedule bounds for subsimulations */
   $\sigma$[1] = $(\sigma_1,\ldots,\sigma_j-1,\ldots)$
   $\sigma$[2] = $(\sigma_1,\ldots,\sigma_k-1,\ldots)$
if not decided then /* still simulate both */
   while true do /* Cycle thru schedules */
      if $Pos$(S[1],i,t) = m then
         /* 1 step for i in S[1] */
         if t=1 then result[1] = R[1] /* i=j */
         else result[1]=
            SimObject(q[1],$\sigma$[1],w,t,I)
      if $Pos$(S[2],i,t) = m then
         /* 1 step for i in S[2] */
         if t=1 then result[2] = R[2] /* i=k */
         else result[2]=
            SimObject(q[2],$\sigma$[2],w,t,I)
      /* Done all steps? */
      if $Pos$($\mathcal{S}$,i,t) < m then exit loop;
      /* Doalockstep */
      mychoice =
      lockstep(i,mychoice,winner[m],choice[m]);
      m++;
   if result[1] = result[2] then
      return result[1] /* no need to choose */
   else /* must choose */
      for mfinal = m to $|\sigma|$+1 do
         /* force others to finish */
         mychoice =
            lockstep(i,mychoice,lock[mfinal]);
      decided = true;
      return result[mychoice]; /* made choice */
else
   result[mychoice] = SimObject(q[mychoice],
         $\sigma$[mychoice],w,t,I)
   return result[mychoice]
```

Figure 5.3b Edge simulation algorithm for P_i, code.

can do 2–process consensus. This latter point, when combined with the corollary above, gives:

Corollary. Let T be any n–process type that can be simulated by some 2–process type. Either T can be simulated by read/write shared variables or T can be added to *common2*.

6. Conclusions and open problems.

Note that the High Gap Theorem gives a gap between n–process types with consensus number n and all types that have consensus number less than n. It's easily shown that no such gap exists at even the next lower level for n–process types. Hence robustness for n–process types below n–process consensus is an open question. However, if a "universal" set of simple types were found such that all types could be classified as being equivalent to exactly one universal type, then a technique similar to the one used in this paper might be able to prove this more general robustness.

While a useful characterization for the high end of the n–process type hierarchy has now been found, at the opposite end there is still a need for a comparable characterization. Note that for 2–processes there is a significant gap between read/write and non–read/write memory (i.e., 2–process consensus types). What is the nature of the gap between read/write and non-read/write memory for more than two processes? Note that Herlihy and Shavit [HS94] have recently given a characterization of a very restricted form of wait–free computations that can be solved with read/write memory. However, their methods are extremely complex and their results do not directly apply to our more general definition of types. Much simpler characterizations, such as our High Gap Theorem, are needed if they are to be used to study the power of types.

Bibliography

[AWW93] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. "A completeness theorem for a class of synchronization objects." In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 159–170. ACM Press, August 1993.

[BNP94] Rida A. Bazzi, Gil Neiger, and Gary L. Peterson. "On the use of registers in wait–free consensus." To appear in *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, 1994.

[Cha93] Soma Chaudhuri. "Agreement is harder than consensus: Set consensus problems in totally asynchronous systems." *Information and Computation*, 103(1):132–158, July 1993.

[Her91] Maurice Herlihy. "Wait–free synchronization." *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HS94] Maurice Herlihy and Nir Shavit. "A simple constructive computability theorem for wait–free computation." In *Proceedings of the 26th ACM Symposium on Theory of Computing*, 1994.

[Jay93] Prasad Jayanti. "On the robustness of Herlihy's hierarchy." In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 145–158. ACM Press, August 1993.

[KM93] Jon Kleinberg and Sendhil Mullainathan. "Resource bounds and combinations of consensus objects." In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 133–144. ACM Press, August 1993.

[LA87] Michael C. Loui and Hosame H. Abu-Amara. "Memory requirements for agreement among unreliable asynchronous processors." In Franco P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, 1987.

[Plo89] Serge Plotkin. "Sticky bits and the universality of consensus." In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, 1989, 159–175.

[PR79] John Reif and Gary L. Peterson. "Multiple–person alternation." In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, 1979, 348–363.