A Study of Wait-Free Hierarchies in Concurrent Systems

D. Scott McCrickard

GIT–CC–94/04

February 15, 1994

Abstract

An assignment of wait-free consensus numbers to object types results in a wait-free hierarchy. Herlihy was the first to propose such a hierarchy, but Jayanti noted that Herlihy's hierarchy was not robust. Jayanti posed as open questions the robustness of a hierarchy he defined and the existence of a robust, wait-free hierarchy. In this paper, we examine a number of object parameters that may impact on a hierarchy's robustness. In addition, we study a subset of Jayanti's hierarchy introduced by Afek, Weisberger, and Weisman called common2 and extend this subset to include the seemingly powerful 2-bounded peek-queue. Finally, we introduce a property called commonness and examine its applications to wait-free hierarchies and their robustness.

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280

1 Introduction

A *concurrent system* consists of a set of asynchronous processes that communicate through shared objects such as registers, queues, and test-and-set bits. Since no system could provide every type of object, implementations of objects using other objects may be required. In order for an object implementation to be correct, the method of implementation must be *linearizable*: operations on the object by concurrent processes must appear to occur in some legal sequence [14]. The traditional method, ensuring linearizability using critical sections, is ill-suited for fault-tolerant systems; a process could crash inside a critical section and prevent the other processes from accessing that object. Even in a non-faulty system, faster processes must wait to access an object when a slower process is in a critical section.

A *wait-free implementation* of a shared object guarantees that any process can complete any operation on the object in a finite number of its own steps, regardless of the execution speed of the remaining processes. Such an implementation is resilient to process crashes and variations in speed. A process that accesses an object implemented in a wait-free manner can complete all accesses regardless of the actions of other processes. Most recent work on wait-free implementations has demonstrated how an object can be implemented by a set of seemingly weaker objects. For example, complex registers can be implemented from simpler ones [6,9,17,18,19,21] and atomic snapshot objects from registers [1,3,4,5,8]. A number of object parameters such as determinism, initializability, and breakability must be considered when constructing a wait-free implementation. These parameters are discussed in Section 2 of this paper.

Herlihy discovered that the existence of a wait-free implementation using a given object is related to the object's ability to be used in achieving consensus [12]. A set of processes achieves *consensus* when all of the processes agree on a single value. In order to achieve consensus, the processes must communicate using shared objects. Every object type can be assigned a *consensus number* based on the maximum number of processes for which objects of the type can be used to achieve consensus. Herlihy's universality result states that an object of any type has a wait-free implementation in a system with N processes using N-process consensus objects and registers. The universality result led to the evaluation of types based on their ability to achieve consensus.

An assignment of consensus numbers to types results in a *wait-free hierarchy* in which each level N of the hierarchy contains object types of consensus number N. Since the consensus number of an object type depends on the number of objects and the number of registers that can be used in the implementation [15], different consensus numbers could be assigned to a single type resulting in different hierarchies. Herlihy's hierarchy, which Jayanti referred to as h_1^r, and Jayanti's hierarchy h_m^r differ in the number of objects that can be used to implement consensus. For any type T, $h_1^r(T)$ is the maximum number of processes for which consensus can be implemented using just a single object of type T and any number of registers, while $h_m^r(T)$ is the maximum number of processes for which consensus can be implemented using any number of objects of type T and any number of registers.

Jayanti asserted that a desirable hierarchy would be fully-refined and robust [15]. A hierarchy is *fully-refined* if there is some type at every level $\{1, 2, \ldots, \infty\}$ of the hierarchy. A hierarchy h is *robust* if, for all types T and sets of types S, if $h(T) = N$ and $h(T') < N$ for all types T' in S, then there exists no implementation of T from S in a system with N processes. Jayanti showed that a number of hierarchies, including Herlihy's hierarchy h_1^r, are fully-refined but not robust, and left as open questions the robustness his own hierarchy h_m^r and the existence of a fully-refined, robust, wait-free hierarchy.

An interesting subset of level 2 of Jayanti's hierarchy is common2. In this paper, *common2* is defined as the set of all types at level 2 of Jayanti's hierarchy whose objects can be implemented by registers and any other object at level 2 in systems with any number of processes. While Afek, Weisberger, and Weisman defined a specific set of objects as common2, they proved that our definitive property applies to all of its object types [2]. They conclude with a challenge to readers to extend common2 to include more or all of the objects at level 2 of Jayanti's hierarchy. Section 3 of this paper extends common2 to include a seemingly powerful object, the 2-bounded peek-queue.

Section 4 introduces a property evolved from the common2 class called commonness. An object type at level N of a hierarchy is *common* if its objects can be implemented by registers and objects of any other type at level N in systems with any number of processes, and a hierarchy is common if every type in the hierarchy is common. We examine the commonness of several types in different hierarchies, and we establish that, if a hierarchy is common, then it is robust. The existence of a common hierarchy remains as an open question.

2 Parameters of Wait-Free Implementations

This section shows that the solvability of consensus is sensitive to the determinism, breakability, and initializability of the type specification.

Determinism refers to the formal specification of the object; a object that nondeterministically returns one of a number of values on a given operation from a given state may be less predictable than a deterministically specified object that always returns the same value in the same condition. In a deterministically specified object, the process may gain additional information about the prior and current state of the machine from the value returned.

Breakability is related to determinism. Objects can break when an undefined operation is invoked; for example, a queue breaks when a dequeue is invoked on an empty queue. The breaking of an object could result in a system crash or random (nondeterministic) responses from the object. In these cases, each process must order its operations such that an undefined operation can never happen. However, deterministically specified objects must have a defined response for every operation. Processes could perform operations and break the object, then learn if other processes did (or did not) access the object previously. An example of this concept is given in Section 2.1.

Initializability refers to the ability of the implementor to start an object in any state in a wait-free implementation. For example, if a machine supports the stack data object, a stack could be initialized to contain certain values, and processes could gain information when they pop these initial values. The examples in Sections 2.1 and 2.2 both discuss initializability.

Since a real system would require objects to be deterministically specified, would allow initialization of objects, and would handle breakability in a deterministic manner, this paper will consider objects with respect to these parameters.

2.1 Initialization, Breakability, and Consensus for Queues

Herlihy proved that a queue initialized to contain 0 in the first slot and 1 in the second could achieve 2-process consensus [12]. Such a queue could be dequeued by each of two processes: the process that is returned the 0 is the winner and the process returned the 1 is the loser and knows the other process won. This is enough to implement consensus.

Jayanti and Toueg proved that an initially empty queue could not achieve 2-process consensus [16]. Their proof is based on the observation that each **enq** by a process could be

matched with a **deq** by the same process; thus, neither process would learn from the other and the processes would never reach consensus.

However, neither of the above directly addressed what happens if the queue is broken; namely, what happens if a **deq** is performed on an empty queue. If we assume that a **deq** on an empty queue returns $\perp$ (a special value that cannot be enqueued) and breaks the queue such that $\perp$ is returned on all subsequent operations by any process, we can achieve 2-process consensus as follows:

Two processes, p and q, both access an initially empty queue. Process p will try to **enq** the value 0 and process q will try to **deq**.

```
proc p:
    if enq(0) = ⊥
      decide q
    else
      decide p
proc q:
    if deq() = ⊥
      decide q
    else
      decide p
```

Proof of correctness: If process p enqueues before process q dequeues, then the queue will contain a single element 0 when process q dequeues. Therefore, process p will not receive $\perp$ since the queue is not broken, and process q will not receive $\perp$ because it will dequeue the value 0, and both will decide on process p. Conversely, if process q dequeues before process p enqueues, then the queue will break when the dequeue occurs. Therefore, process q will receive $\perp$ since it broke the queue, and process p will receive $\perp$ as well since the queue is broken when it tries to enqueue; thus, both processes will decide on process q. □

2.2 The 2-Set Consensus Object

Recently, there has been interest in a weaker version of the consensus problem called the *k-set consensus problem* [10,11]. In this problem, each process begins with a value (as in consensus) and decides on a single value such that there are at most k decided values. It has been shown that there is no wait-free solution to k-set consensus using only registers in systems with more than k processes [7,13,20]. Consider types that support such wait-free solutions. At what level must they be in a hierarchy such as h_m^r? Interestingly, Herlihy and Shavit [13] showed that knowing type T can achieve k-set consensus is not sufficient to show that $h_m^r(T) > 1$. That is, the ability to achieve k-set consensus (for any number of processes) does not guarantee the ability to achieve even 2-process consensus!

To address this apparently anomalous situation, it seems reasonable to give a specification of an object type that performs k-set consensus. The following considers a deterministic specification of such an object and shows that, if the object can be initialized to any state, then it can be used to achieve consensus in systems with any number of processes.

Consider the case of 2-set consensus. Two values are stored by the object, VAL1 and VAL2, the first and second values submitted. It has one operation; **submit**(n), which returns either VAL1 or VAL2. Since the object is deterministic, there must be some fixed order in which the two values are returned.

Theorem 1: *The 2-set consensus object can be used to achieve consensus for any number of processes and thus is at level* ∞ *of* h_m^r.

Proof: If either VAL1 or VAL2 is returned a finite number of times, then there exists some k such that all **submit** after the kth **submit** will always return VAL1 or always return VAL2. Each process P_i (proposing value n_i) could perform $k+1$ **submit**(n_i) and decide on the result of the $k+1$st **submit**.

If no such k exists, then initialize the 2-set consensus object with a **submit(0)**. Next, each process P_i ($i > 0$) writes its value to a table in position i and performs **submit(i)** until a positive integer is returned. The integer returned is the winning process number. Each process looks up the value corresponding to the integer, decides on the value, and quits submitting values. Since VAL2 is returned regularly (that is, there is no k for which all **submit** after the kth are returned VAL1), each process will eventually decide, and all of the processes will decide on the value in VAL2's table slot. Since the 2-set consensus object can be used to achieve consensus for any number of processes, it is at level ∞ of h_m^r. □

3 The 2-Bounded Peek-Queue

When Herlihy defined his hierarchy, he was unable to prove that that there is an object at every level. Jayanti and Toueg showed that every level N contains the N-bounded peek-queue, a queue of size N that, instead of a dequeue operation, supports a peek operation which returns the values currently in the queue [16]. This result suggests that the N-bounded peek-queue may be the strongest object at level N. If the 2-bounded peek-queue is in common2, then it is no stronger than any other object. Below is an implementation of the 2-bounded peek-queue using test-and-set objects and registers for a system with any number of processes, proof that the 2-bounded peek-queue indeed is in common2.

3.1 Definition of the 2-Bounded Peek-Queue

Jayanti and Toueg defined a 2-bounded peek-queue as follows:

1. When **enq(value)** is invoked, if the queue has fewer than 2 items in it, then value is written to the end of the queue and "completed" is returned; otherwise, the queue enters a faulty state and returns $\perp$.

2. A queue in a faulty state remains faulty forever and returns $\perp$ to every subsequent operation.

3. **peek** returns the state of the queue. If the queue is faulty then $\perp$ is returned; otherwise, a list of 0–2 enqueued values is returned.

The above conditions characterize any sequential execution on a 2-bounded peek queue. Any implementation must be *linearizable* in the following sense: for any execution of the implementation, one can give a linear order of its operations such that the order meets the above conditions and such that for any two operations o_1 and o_2 in the execution, if o_1 ends before o_2 begins, then o_1 precedes o_2 in the linear order. That is, the real-time ordering of non-overlapping operations must be preserved.

3.2 Implementation of the 2-Bounded Peek-Queue

The 2-bounded peek-queue can be implemented in a system with k processes using 2 test-and-set bits and $2k+2$ registers. Each process P_i has an associated "count" register C_i, initially 0, that indicates how many times the process has invoked **enq**; and a "value" register V_i, initially $\perp$, that contains the first value the process tries to **enq**. Two "queue" registers $Q1$, $Q2$, initially $\perp$, will hold the values of the processes that enqueue successfully. Two test-and-set objects $T1$, $T2$, initially 0, will determine which process's value gets enqueued to each queue register.

The 2-bounded peek-queue functions are implemented as follows:

- On **peek**,

```
if ∑_{n=1}^{i} C_n > 2 return ⊥        // queue is broken
else if Q1 = ⊥ return ⟨⟩                // queue is empty
else if Q2 = ⊥ return ⟨Q1⟩
else return ⟨Q1, Q2⟩
```

- On **enq(value)** invoked by P_i,

```
increment C_i by one
if C_i = 1                             // this is the 1st time P_i has enqueued
  write value in V_i
if T&S(T1) = 0                         // P_i is the first to enqueue
  write value in Q1
  return "completed"
else
  examine all proc registers           // try to find winner of T1
  if exactly one C_j = 1 (j ≠ i) and ∑_{n=1}^{i} C_n = 2
    write V_j in Q1                    // j was the first to enqueue, write its value in Q1
  if T&S(T2) = 0
    write value in Q2
  return "completed"
  else                                 // process loses and queue broken
    return ⊥
```

3.3 Proof of Correctness

To prove that the above implementation is linearizable, a method must be described by which the operations in any execution can be put in a linear order that meets the conditions in Section 3.1 and that preserves the real-time ordering of non-overlapping operations. The ordering chosen is such that the following hold:

1. All **peek**s that return $\langle\rangle$ appear before all other operations.

2. The **enq** whose value is written to $Q1$ appears after the operations in 1 but before any others.

3. All **peek**s that return $\langle Q1\rangle$ appear after the operations in 1–2 but before any others.

4. The **enq** whose value is written to $Q2$ appears after the operations in 1–3 but before any others.

5. All **peek**s that return $\langle Q1, Q2\rangle$ appear after the operations in 1–4 but before any others.

6. One **enq** that returns $\perp$ appears after the operations in 1–5 but before any others.

7. Any other **enq**s and all **peek**s that return $\perp$ appear after the operations in 1–6.

The ordering of operations within items 1, 3, and 5 can be done in any way that preserves the real-time ordering of non-overlapping operations. The choice of the **enq** for item 6 and the ordering of operations in item 7 is chosen similarly. To prove that this ordering is correct, we need to see that it satisfies the three conditions from Section 4.1 and that it preserves the real-time ordering of overlapping operations.

To prove that Jayanti and Toueg's condition 1 is satisfied, we will use the following lemmas:

Lemma 2: *For any* **enq**(*value*)*,*

- *If "completed" is returned, then value was written in $Q1$ or $Q2$.*
- *If $\perp$ is returned, then queue is in a faulty state (i.e.: will always return $\perp$).*

Proof: There are exactly two cases in which "completed" is returned; after $Q1$ is written by the winner of $T1$ and after $Q2$ is written. There is only one case in which $\perp$ is returned; when a process loses both $T1$ and $T2$. In that case, the count registers C_i of the winning processes sum to at least 2, and the current enqueue adds an additional 1 to its C_i, so the sum of all the registers becomes greater than 2 and **peek** will always return $\perp$. Furthermore, both $T1$ and $T2$ will always return 1, so all subsequent enqueues will return $\perp$. Thus, q is in a faulty state. □

Lemma 3: *All* **enq**s *but 2 return* $\perp$*.*

Proof: An **enq** returns $\perp$ if and only if it loses both $T1$ and $T2$. Since each test-and-set can be won only once, and each **enq** results in at most one winner of a test-and-set, only one **enq** can produce a winner of $T1$, and only one **enq** can produce a winner of $T2$, so all other **enq**s but 2 return $\perp$. □

Lemma 2 proves that "completed" is returned only if the process wrote its value, and $\perp$ is returned only if the queue is in a faulty state. Lemma 3 proves that at most two **enq**s will return a non-$\perp$ value. Thus, condition 1 is satisfied.

We show Jayanti and Toueg's condition 2 is satisfied as follows:

> Condition 1 implies that a queue enters a faulty state when **enq** is invoked on a queue with 2 items in it. If a queue has 2 items in it, then the sum of the registers is greater than or equal to 2 and both test-and-sets have been won. When **enq** is invoked, a register is incremented, raising the sum over 2. All subsequent **peek**s will see that the sum of the registers is greater than 2 and will return $\perp$. All **enq**s will lose both test-and-sets and will return $\perp$. At no point are the registers decremented. At no later point can a test-and-set be won. Thus, the queue will return $\perp$ to every operation and is "broken", and condition 2 is satisfied.

Jayanti and Toueg's condition 3 is satisfied as follows:

> A **peek** will return one of four possible values: $\langle\rangle$; $\langle Q1\rangle$; $\langle Q1, Q2\rangle$; $\perp$; where $Q1$ contains the value of the first process to **enq** and $Q2$ contains the value of the second process to **enq**. If the sum of the count registers C_i is greater than 2 when examined by the **peek**, then $\perp$ is returned. $\langle\rangle$ is returned if $Q1 = \perp$. If $Q1$ is returned, then the process that won $T1$ wrote its value in $Q1$ or another process that lost $T1$ wrote the winning process's value in $Q1$. If $Q2$ is also returned, then the process that won $T2$ wrote its value in $Q2$. $Q2$ cannot be returned unless $Q1$ has a non-$\perp$ value written in it. $Q2$ can only be written by the winner of $T2$ and can only contain the winner's process value. $Q1$ can be written by either the winner of $T1$ or by a process that was able to determine the winner of $T1$, but will only contain the winning process's value. Thus, condition 3 is satisfied.

To show that our ordering of operations preserves real-time ordering, we will use the following lemmas:

Lemma 4: *If process A completes an enqueue before process B starts an enqueue, then*

1. *process B does not have its value written to* $Q1$.
2. *if process* C $(A \neq C)$ *completed an enqueue before process B started, then process B did not write its value to* $Q2$.
3. *if process B wrote to* $Q2$, *then process A's value was written to* $Q1$.
4. *if process A returned* $\perp$, *then process B returned* $\perp$.

Proof:

1. Since A completes an enqueue before B starts, it executes $T1$ before B. Since only the first process to execute $T1$ will win and write its own value to $Q1$, B can not win $T1$. Furthermore, the only other way in which B could have its value written to $Q1$ is for another process to read that B is the only other process to increment its counter. Since A completes an enqueue before B, A increments its counter before B, so no process could see only B's counter incremented. Therefore, B cannot have its value written to $Q1$.

2. Similar to (1). Either A or C (or both) lost $T1$, so at least one of the two executes $T2$ before B does. Since only the first process to execute $T2$ will win and write its value to $Q2$, and no other value is ever written to $Q2$, B's value is never written to $Q2$.

3. If process B wrote to $Q2$, then process B must have won $T2$. Since A completes before B starts, A must not have executed $T2$, but the only process that does not execute $T2$ must have returned before its execution point, and the only process to do that is the process that wins $T1$ and writes its value to $Q1$. Thus, if B writes to $Q2$, then A writes to $Q1$.

4. The contrapositive of (4) follows from (1) and (3).

□

Lemma 5: *If* **peek** *A ends before* **peek** *B begins, then*

1. *if* **peek** *A returned* $\perp$*, then* **peek** *B returned* $\perp$*.*
2. *if* **peek** *B returned* n *queue values, then* **peek** *A returned* $m \leq n$ *values.*
3. *if* **peek** *A returned* m *queue values, then* **peek** *B returned* $n \geq m$ *values or* $\perp$*.*

Proof:

1. **peek** A would return $\perp$ only if the sum of the count registers C_i is greater than 2. Since **peek** B began after **peek** A completed and the count registers are never decremented, their sum was greater than 2 when **peek** B summed them, so **peek** B must have returned $\perp$ as well.
2. If **peek** B returned $\langle\rangle$, then the sum of the count registers C_i must have been less than or equal to 2 (otherwise, $\perp$ would have been returned) and $Q1$ must have contained $\perp$ when **peek** B examined it (otherwise, $\langle Q1\rangle$ or $\langle Q1, Q2\rangle$ would have been returned). Since a queue register is never overwritten with $\perp$ and the count registers are never decremented, the sum of the count registers C_i must have been less than or equal to 2 and $Q1$ must have contained $\perp$ when **peek** A examined it; thus **peek** A must have also returned $\langle\rangle$. Similarly, if **peek** B returned one queue element, then the sum of the count registers must have been less than or equal to 2 and $Q2$ must have contained $\perp$ when **peek** B examined them. As before, **peek** A could not have returned $\perp$ or $\langle Q1, Q2\rangle$ since the sum of the count registers was less than or equal to two and $Q2$ was empty. Similarly, if **peek** B returned two queue values, the sum of the count registers must not have been greater than or equal to 2 when **peek** B examined them, so **peek** A could not have returned $\perp$.
3. If **peek** A returned m queue values, since neither $Q1$ nor $Q2$ can be written with $\perp$, either the sum of the count registers was greater than 2 when **peek** B read it (and $\perp$ was returned) or **peek** B returned any values in $Q1$ and $Q2$ that **peek** A returned (and possibly other values that had been written after **peek** A).

□

Lemma 6: *Consider any execution of the implementation.*

1. *If a* **peek** *completes before any* **enq** *begins, then* $\langle\rangle$ *will be returned.*
2. *If a* **peek** *completes before all but one* **enq** *begins, then* $\langle\rangle$ *or* $\langle Q1\rangle$ *will be returned.*
3. *If a* **peek** *completes before all but two* **enq**s *begin, then* $\langle\rangle$ *or* $\langle Q1\rangle$ *or* $\langle Q1, Q2\rangle$ *will be returned.*

Proof:

1. Since no **enq** has begun when **peek** examines the count registers C_i, the sum of the registers will be 0. Also, no value will have been written in $Q1$ since **enq** is the only operation that can alter this register. Thus, **peek** will read the initial value $\perp$ from $Q1$ and return $\langle\rangle$.

2. Similar to 1. The sum of the count registers C_i is at most 1, and **peek** will return $\langle Q1\rangle$ if the first **enq** has won $T1$ and written its value in $Q1$ and $\langle\rangle$ otherwise.

3. Similar to the previous two. The sum of the count registers C_i is at most 2, and $\langle\rangle$ will be returned if no process has written to $Q1$; $\langle Q1\rangle$ will be returned if no process has written to $Q2$; and $\langle Q1, Q2\rangle$ returned otherwise.

□

Lemma 7: *Consider any execution of the implementation.*

1. *If a* **peek** *starts after three* **enq** *completes, then* $\perp$ *will be returned.*

2. *If a* **peek** *starts after two* **enq** *completes, then* $\langle Q1, Q2\rangle$ *or* $\perp$ *will be returned.*

3. *If a* **peek** *starts after one* **enq** *completes, then* $\langle Q1\rangle$ *or* $\langle Q1, Q2\rangle$ *or* $\perp$ *will be returned.*

Proof:

1. If three **enq**s have completed, then each will have incremented a process register, so the sum of the process registers will be greater than 2; thus $\perp$ is returned.

2. Similar to above. If two **enq**s have completed, then the sum of the count registers C_i is at least 2. If a third **enq** has begun and incremented its register before the **peek** reads it, then the sum of the count registers C_i will be greater than 2 and $\perp$ is returned. If a third **enq** has not incremented its register, then the winner of $T1$ wrote its value in $Q1$, and the winner of $T2$ wrote its value in $Q2$ since both completed, so **peek** will return $\langle Q1, Q2\rangle$.

3. Similar to the previous two.

□

Lemma 4 shows that the **enq**s occur in the order specified at the beginning of Section 3.3 relative to other **enq**s, and lemma 5 shows that the **peek**s occur in the specified order relative to other **peek**s. Lemmas 6 and 7 show that peeks and queues occur in the specified order relative to each other. Thus, the method specified in Section 3.2 not only meets Jayanti and Toueg's three conditions, but also preserves the real-time ordering of non-overlapping operations. Thus, the implementation is linearizable.

A 2-bounded peek-queue initialized to a non-empty state can be implemented similarly with appropriately initialized test-and-sets and registers. Thus, the seemingly powerful 2-bounded peek-queue type is yet another object type in common2. Since no type at level 2 of h_m^r has been proven to not be in common2, one may surmise that all types at level 2 are in common2. The next section discusses this possibility and introduces an extension of the common2 class to a property "commonness".

4 An Extension of the Common2 Class

Consider a property evolved from the common2 class called commonness. A type at level N of a hierarchy is *common* if its objects can be implemented with registers and objects of any type at level N of h_m^r in systems with any number of processes. Thus, all types in common2 are common since objects of any type at level 2 can implement common2 objects in systems with any number of processes.

If objects are not initializable and are nondeterministically specified, a very simple and general specification of a 2-set consensus object can be given. It follows from the results of Herlihy and Shavit [13] that this object is at level 1 of h_m^r. However, their results also show that it cannot be implemented by registers in systems with 3 processes. Thus, this object is not common. If we restrict our objects to be initializable and deterministic, then the existence of a non-common object in h_m^r is an open question.

Jayanti proved that Herlihy's hierarchy h_1^r was not robust by showing a type T_{sp} at level 2 of h_1^r such that N T_{sp} objects could implement consensus in systems with $N+1$ processes (for any $N \geq 1$). If test-and-set could implement the T_{sp} object, then this implementation could be used to implement consensus with just test-and-sets in systems with N processes. Since no number of test-and-set objects can implement consensus in systems with more than 2 processes, test-and-set must not be able to implement T_{sp} in systems with more than 2 processes. Thus, T_{sp} is not common with respect to Herlihy's hierarchy h_1^r.

Since commonness is defined in relation to the levels of a hierarchy, it is interesting to consider the commonness of an entire hierarchy. We define a hierarchy to be common if every object in the hierarchy is common. Previously, we showed that T_{sp} was not common with respect to h_1^r (and thus h_1^r is not common) by using the same type T_{sp} that Jayanti used in his non-robustness proof. That leads us to believe that robustness and commonness may be related. The following two theorems address this relationship.

Theorem 8: *If h_m^r is common, then types at level K of h_m^r can implement any object at level $J \leq K$ of h_m^r in systems with any number of processes.*

Proof: By definition of common, objects at level K will be able to implement other objects at level $J = K$. So consider any object O_K at level K and any object O_J at level $J < K$. Since the K-bounded peek-queue is at level K, it can be implemented by object O_K, assuming h_m^r is common. Clearly the K-bounded peek-queue can implement the J-bounded peek-queue for any $J < K$ in systems with any number of processes. Finally, the J-bounded peek-queue can implement object O_J (again assuming commonness). Thus, objects O_K at level K can implement any object O_J at level $J \leq K$ in systems with any number of processes. □

Theorem 9: *If h_m^r is common, then h_m^r is robust.*

Proof: By contrapositive. Assume h_m^r is not robust. Then there exists a type T and a set of types S for which $h(T) = N$, $h(T') < N$ for all $T' \in S$, and there exists an implementation of T from S in a system with N processes. Consider the type $T_{max} \in S$ with the maximum consensus number. By Theorem 8 all other objects in S can be implemented by objects of type T in systems with any number of processes. Thus, we can consider a new implementation of T from just T_{max} in which all other objects in S are replaced with objects of type T_{max}. Since type T_{max} can implement type T in a system with N processes and T can achieve

consensus in a system with N processes, T_{max} can also achieve consensus in a system with N processes and must be at level N of h_m^r. However, we assumed that all types in S were at levels less than N, a contradiction! Thus, if h_m^r is not robust, h_m^r is not common; and if h_m^r is common, then h_m^r is robust. □

While the commonness of h_m^r implies robustness, the converse is not true. If h_m^r is robust, then there could still exist two types at the same level for which one type cannot implement the other in a system with any number of processes. In order to understand the true power of the object types in h_m^r, it is important to study the commonness of the hierarchy.

5 Conclusions and Open Problems

The commonness of a hierarchy is important: it guarantees that objects of one type can perform the same functions as objects of any type at the same level in a system with any number of processes. Commonness also implies robustness, an important property of wait-free hierarchies. With the addition of the 2-bounded peek-queue to common2, the existence of a type in h_m^r that is not common remains open Thus, the commonness of h_m^r and the existence of a common, fully-refined, wait-free hierarchy are also open.

Another important problem is the complexity issues within each level of a hierarchy; namely, for any given type, how many objects of the type are required to implement consensus? In proving that h_1^r is not robust, Jayanti showed that $N-1$ T_{sp} objects are required to implement consensus in systems with N processes. Thus, certain types require more than one object to implement consensus. It might be possible to find a bound on the number of objects required, perhaps related to the number of processes in the system.

The model discussed in this paper required that objects be deterministic and initializable. While these qualities exist in a real system, it may be worthwhile from a theoretical point of view to consider the results when one or more of these parameters is changed.

Acknowledgements

I would like to thank Gil Neiger for all of his comments and suggestions and everyone in the CoC wait-free seminar for priming my interest in the subject.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[2] 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.

[3] James H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, April 1993.

[4] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Efficient atomic snapshots using lattice agreement. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth International*

Workshop on Distributed Algorithms, number 647 in Lecture Notes on Computer Science, pages 35–53. Springer-Verlag, November 1992.

[5] Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 29–40. ACM Press, August 1993.

[6] Bard Bloom. Constructing two-writer atomic registers. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 249–259. ACM Press, August 1987.

[7] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth ACM Symposium on Theory of Computing*, pages 91–100. ACM Press, May 1993.

[8] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 41–52. ACM Press, August 1993.

[9] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 222–231. ACM Press, August 1987.

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

[11] Soma Chaudhuri, Maurice Herlihy, Nancy Lynch, and Mark R. Tuttle. A tight lower bound for k-set agreement. In *Proceedings of the Thirty-Fourth Symposium on Foundations of Computer Science*, pages 206–215. IEEE Computer Society Press, November 1993.

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

[13] Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the Twenty-Fifth ACM Symposium on Theory of Computing*, pages 111–120. ACM Press, May 1993.

[14] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[15] 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.

[16] Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality, and decidability of consensus. In A. Segall and S. Zaks, editors, *Proceedings of the Sixth International Workshop on Distributed Algorithms*, number 647 in Lecture Notes on Computer Science, pages 69–84. Springer-Verlag, November 1992.

[17] Richard Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 232–248. ACM Press, August 1987.

[18] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[19] Gary L. Peterson and James E. Burns. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the Twenty-Eighth Symposium on Foundations of Computer Science*, pages 383–392. IEEE Computer Society Press, October 1987.

[20] Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the Twenty-Fifth ACM Symposium on Theory of Computing*, pages 101–110. ACM Press, May 1993.

[21] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 206–221. ACM Press, August 1987.