A Study of Wait-Free Hierarchies in Concurrent Systems

D. Scott McCrickard*
Georgia Institute of Technology

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.

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 [6]. 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, atomic snapshot objects can be implemented from registers [1]. 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 [4]. 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 [7], 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 [7]. 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

*Author's address: College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280; `mccricks@cc.gatech.edu`.

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.

Some of the definitions given here are terse in the interest of space. Most definitions regarding wait-free implementations and hierarchies are given in more detail by Herlihy [4] and Jayanti [7] and in my technical report [9].

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 random (nondeterministic) responses from the object. In these cases, each process may need to order its operations such that an undefined operation can never happen. 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 user 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 [4]. 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 [8]. 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:	proc q:
if **enq**(0) $= \perp$	if **deq**() $= \perp$
decide q	decide q
else	else
decide p	decide p

Whichever process performs its operation first wins and decides on its own processor's value. The other process is returned $\perp$ and decides on the other's value (which would be written to a register earlier). A formal proof of correctness is included in my technical report [9].

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* [3]. 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 [5]. 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 [5] 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.

This 2-set consensus object can be used to achieve consensus for any number of processes as follows. 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 [8]. This result suggests that the N-bounded peek-queue may be the strongest object at level N. However, 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 bits 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 [6].

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 $\sum_{n=1}^{i} C_n > 2$ return $\perp$
    else if $Q1 = \perp$ return $\langle\rangle$
    else if $Q2 = \perp$ return $\langle Q1 \rangle$
    else return $\langle Q1, Q2 \rangle$

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

    increment C_i by one
    if $C_i = 1$
      write value in V_i
    if $T\&S(T1) = 0$
      write value in $Q1$
      return "completed"
    else
      examine all proc registers
      if exactly one $C_j > 0$ ($j \neq i$) and $C_j = 1$
        write V_j in $Q1$
      if $T\&S(T2) = 0$
        write value in $Q2$
      return "completed"
      else
        return $\perp$

A proof that the above implementation is linearizable is included in my technical report [9].

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^r_m 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 [5] that this object is at level 1 of h^r_m. 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^r_m is an open question.

Jayanti proved that Herlihy's hierarchy h^r_1 was not robust by showing a type T_{sp} at level 2 of h^r_1 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^r_1.

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^r_1 (and thus h^r_1 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 1: *If h^r_m is common, then types at level K of h^r_m can implement any object at level $J \leq K$ of h^r_m in systems with any number of processes.*

Theorem 2: *If h^r_m is common, then h^r_m is robust.*

The proofs for both of these theorems can be found in my technical report [9].

While the commonness of h^r_m implies robustness, the converse is not true. If h^r_m 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^r_m, 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^r_m that is not common remains open. Thus, the commonness of h^r_m 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^r_1 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] Soma Chaudhuri. Agreement is harder than consensus: Set consensus problems in totally asynchronous systems. *Information and Computation*, 103(1):132–158, July 1993.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] D. Scott McCrickard. A study of wait-free hierarchies in concurrent systems. Technical report, College of Computing, Georgia Institute of Technology, December 1993.