On the Robustness of Herlihy's Hierarchy*

Prasad Jayanti

TR 93-1332
March 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG 2-593, grants from the IBM Endicott Programming Laboratory and Siemens Corp.

On the robustness of Herlihy's hierarchy*

Prasad Jayanti

Department of Computer Science, Cornell University
Ithaca, New York 14853, USA

prasad@cs.cornell.edu

Abstract

A *wait-free hierarchy* maps object types to levels in $Z^+ \cup \{\infty\}$, and has the following property: if a type T is at level N, and T' is an arbitrary type, then there is a wait-free implementation of an object of type T', for N processes, using only registers and objects of type T. The infinite hierarchy defined by Herlihy is an example of a wait-free hierarchy. A wait-free hierarchy is *robust* if it has the following property: if T is at level N, and $\mathcal{S}$ is a finite set of types belonging to levels $N-1$ or lower, then there is no wait-free implementation of an object of type T, for N processes, using any number and any combination of objects belonging to the types in $\mathcal{S}$. Robustness implies that there are no clever ways of combining weak shared objects to obtain stronger ones.

Contrary to what many researchers believe [AGTV92, AR92, Her91a], we prove that Herlihy's hierarchy is not robust. We then define some natural variants of Herlihy's hierarchy, which are also infinite wait-free hierarchies. With the exception of one, which is still open, these are not robust either. We conclude with the open question of whether non-trivial robust wait-free hierarchies exist.

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames grant NAG-2-593, grants from the IBM Endicott Programming Laboratory and Siemens Corp.

1 Introduction

A concurrent system consists of asynchronous processes communicating via typed shared objects such as registers, test&sets, and queues. Since any given system supports only a limited set of object types in its hardware, other useful types will need to be implemented in software. Thus, implementing an object of a given type using objects belonging to a given set of types is a fundamental problem. To be useful, implementations must guarantee *linearizability* [HW90]: concurrent accesses on an implemented object must appear to take effect in some sequential order. One way to ensure linearizability is to implement shared objects using critical sections [CHP71]. This approach however is not fault-tolerant: the crash of a process while in the critical section of an implemented object can permanently prevent the remaining processes from accessing the object. This lack of fault-tolerance led to the concept of *wait-free implementations* [Lam77]. An implementation is wait-free if every process can complete every operation on the implemented object in a finite number of its own steps, regardless of the execution speeds of the remaining processes. In particular, if object $\mathcal{O}$ is built using a wait-free implementation, then the crash of some processes cannot disable the remaining processes from completing their operations on $\mathcal{O}$.

How feasible are wait-free implementations? It is known that registers are too weak to implement[1] even a 2-process consensus object, *i.e.*, a consensus object that is accessed by at most two processes [LAA87, CIL87]. Test&sets and 1-bit read-modify-write objects can implement a 2-process consensus object, but not a 3-process consensus object [LAA87]. 3-valued read-modify-write, on the other hand, can implement an N-process consensus object, for all N. These results indicate that object types differ in their ability to support wait-free synchronization, and that there may be a way of ordering them accordingly. This issue was addressed in a seminal paper by Herlihy [Her88, Her91b]. Following are some important definitions and results in [Her91b].

1. For every object type T, an object of type T can be implemented for N processes using only registers and N-process consensus objects. This is the universality result of Herlihy.

2. For every $N \geq 1$, $(N+1)$-process consensus object cannot be implemented using just registers and N-process consensus objects.

3. The *consensus number* of a shared object $\mathcal{O}$ is the maximum number N such that an N-process consensus object can be implemented using just $\mathcal{O}$ and (any number of) registers. Define a hierarchy of shared objects such that $\mathcal{O}$ is at level N if and only if its consensus number is N. This will be referred to as Herlihy's hierarchy.

As an obvious consequence of the universality result, Herlihy's hierarchy has the following important property: if an object $\mathcal{O}$ of type T is at level N, then for every object type T', an object of type T' can be implemented for N processes using just registers and objects of type T. We will call any hierarchy with this property a *wait-free hierarchy*. Thus, in a

[1] Hereafter "implementation" stands for "wait-free implementation".

wait-free hierarchy such as Herlihy's, if an object $\mathcal{O}$ of type T is at level N, we can immediately infer that arbitrary wait-free synchronization among N processes is feasible using just registers and objects of type T. Notice that this definition allows $\mathcal{O}$ to be at level N even if arbitrary wait-free synchronization among more than N processes is feasible using registers and objects of the type of $\mathcal{O}$. Thus, the level of an object in a wait-free hierarchy does not reflect the object's full potential; it is only a lower bound on the extent to which the object can support arbitrary wait-free synchronization. To understand the exact potential of objects, we define a *tight* wait-free hierarchy. In such a hierarchy, an object $\mathcal{O}$ is at level N if N is the maximum number of processes for which arbitrary wait-free synchronization is feasible using registers and objects of the type of $\mathcal{O}$.

What other properties are important in a hierarchy? We argue below that robustness is one. A hierarchy is *robust* if for every object $\mathcal{O}$, the following holds: if $\mathcal{O}$ is at level N, then it is impossible to implement $\mathcal{O}$ for N processes using any number and any combination of objects at levels $N-1$ or lower. Robustness guarantees that there are no clever ways of putting weak objects together to implement a strong one. We now present an example to illustrate the significance of robustness in analyzing the power of shared primitives. Consider two systems $\mathcal{S}_1$ and $\mathcal{S}_2$. Suppose that $\mathcal{S}_1$ supports only **registers** and **test&sets**, and $\mathcal{S}_2$ supports only **registers** with 3-register assignment. Herlihy showed that arbitrary wait-free synchronization is impossible for 3 or more processes in $\mathcal{S}_1$, and for 5 or more processes in $\mathcal{S}_2$. What implications do these results have on a third system $\mathcal{S}_3$ which supports both **test&sets**, and **registers** with 3-register assignment? In particular, can we conclude, based on just the above results, that arbitrary wait-free synchronization among 5 processes is still impossible? We can, provided that Herlihy's hierarchy is robust. Otherwise we cannot. More generally, if Herlihy's hierarchy is robust, the consensus number of a set of objects, belonging (possibly) to different types, is just the maximum of the consensus numbers of the individual objects in the set. Thus, robustness reduces the difficult problem of analyzing the power of a combination of shared objects to the simpler problem of analyzing the power of the individual objects. On the other hand, if robust wait-free hierarchies do not exist, then there is a possibility of combining weak objects to implement strong ones. In particular, it opens up the possibility of implementing universal objects from non-universal objects! Thus, from a pragmatic point of view, it would also be interesting to prove that robust wait-free hierarchies do not exist.

Is Herlihy's hierarchy robust? A study of this question with respect to common object types, such as **register**, **test&set**, **fetch&add**, **queue**, **compare&swap**, and **sticky-bit**, does not present any evidence to the contrary. In fact, many prominent researchers have attributed robustness to Herlihy's hierarchy [AGTV92, AR92, Her91a][2] We prove that it

[2][AGTV92] states "An object has a consensus number k if k is the maximum number of processes for which the object can be used to solve the consensus problem. Thus objects with higher consensus number cannot be deterministically implemented by employing objects with lower consensus numbers."

[AR92] states "In fact, Herlihy [Her88] describes a full hierarchy of atomicity assumptions, and proves that atoms of a higher class cannot be implemented by those of a lower class, in a wait-free fashion in the deterministic setting."

[Her91a] states "Elsewhere [17, 15], we have shown that any object X can be assigned a *consensus number*, which is the largest number of processes (possibly infinite) that can achieve consensus asynchronously [13] by

is not robust. More specifically, we present an object type $\mathbf{T_{sp}}$ with the property that k objects of this type, together with registers, can implement a $(k+1)$-process consensus object, but not a $(k+2)$-process consensus object. In particular, one $\mathbf{T_{sp}}$ object, with registers, can implement a 2-process consensus object, but not a 3-process consensus object. Thus, by definition, a $\mathbf{T_{sp}}$ object has a consensus number of 2, and is consequently at level 2 in Herlihy's hierarchy. However, since multiple $\mathbf{T_{sp}}$ objects, with registers, can implement a consensus object for arbitrarily large number of processes, it follows from Herlihy's universality result that for all types T and all N, an object of type T can be implemented for N processes using just registers and $\mathbf{T_{sp}}$ objects. Together with the fact that a $\mathbf{T_{sp}}$ object is at level 2, this implies that Herlihy's wait-free hierarchy is not robust.

Does there exist a robust wait-free hierarchy? We do not know the answer yet. However, we define three natural variants of Herlihy's hierarchy, which are also infinite wait-free hierarchies. We prove that two of these are not robust.[3] The third hierarchy, whose robustness is still open, has the following property: if it is not robust, then there is no robust wait-free hierarchy. We believe that resolving the robustness of this hierarchy is an important open problem in wait-free synchronization.

This paper is the first to formalize and study robustness. The technical arguments involved in proving the impossibility result that k $\mathbf{T_{sp}}$ objects cannot implement a $(k+2)$-process consensus object are novel. Traditional bivalency arguments are inadequate to prove such lower bounds.

2 Informal model

A concurrent system consists of processes and shared objects. We write $(P_1, \ldots, P_n; O_1, \ldots, O_m)$ to denote a concurrent system consisting of processes $P_1, \ldots, P_n$ and shared objects $O_1, \ldots, O_m$. Besides a unique name, every object has two attributes: a type and a positive integer which denotes the maximum number of processes which may apply operations on that object. We say that $\mathcal{O}$ is an N-process object if N is the maximum number of processes which may apply operations on $\mathcal{O}$. The type specifies the behavior of the object when operations are applied sequentially, without overlap. More precisely, an *object type* T is a tuple (OP, RES, G), where OP and RES are sets of operations and responses respectively, and G is a directed finite or infinite multi-graph in which each edge has a label of the form (op, res) where $op \in OP$ and $res \in RES$. We refer to G as the *sequential specification* of T, and the vertices of G as the *states* of T. Intuitively, if there is an edge, labeled (op, res), from state σ to state σ', it means that applying the operation op to an object in state σ may change the state to σ' and return the response res.

---

applying operations to a shared X. It is impossible to construct a non-blocking implementation of any object with consensus number n from objects with lower consensus numbers in a system of n or more processes, although any object with consensus number n is universal (it supports a wait-free implementation of any other object) in a system of n or fewer processes."

[3] In proving this, we show the following result which is interesting in its own right. There exist two types such that (i) Even 2-process consensus cannot be solved using objects of either type, and (ii) N-process consensus (for all N) can be solved using the two types of objects together.

A sequence $S = (op_1, res_1), (op_2, res_2), \ldots, (op_l, res_l)$ is *legal from state σ of T* if there is a path labeled S in G from the state σ. T is *deterministic* if for every state σ of T and every operation $op \in OP$, there is at most one edge from σ labeled (op, res) (for some $res \in RES$). T is *non-deterministic* otherwise. T is *total* if for every state σ of T and every operation $op \in OP$, there is at least one edge from σ labeled (op, res) (for some $res \in RES$). In this paper, we restrict our attention to total types.

An N-process object $\mathcal{O}$ of type T supports the set of procedures $\texttt{Apply}(P_i, op, \mathcal{O})$, for all $1 \leq i \leq N$ and $op \in OP(T)$. A process P *invokes* operation op on object $\mathcal{O}$ by calling $\texttt{Apply}(P, op, \mathcal{O})$, and *executes* the operation by executing this procedure. The operation *completes* when the procedure terminates. The *response* for an operation is the value returned by the procedure. We denote the event of P invoking operation op on $\mathcal{O}$ by $inv(P, op, \mathcal{O})$, and the event of $\mathcal{O}$ returning a response v to P by $resp(P, v, \mathcal{O})$.

The type of an object, by itself, is not sufficient to characterize the behavior of the object in the presence of concurrent operations. To characterize such behavior, we use the concept of *linearizability* [HW90]. Roughly speaking, linearizability requires every operation execution to appear to take effect instantaneously at some point in time between its invocation and response. We make it more precise below.

Consider a concurrent system $\mathcal{S} = (P_1, P_2, \ldots, P_n; O_1, O_2, \ldots, O_m)$. A *configuration* of $\mathcal{S}$ is a tuple consisting of the states of the processes $P_1, \ldots, P_n$ and the states of the objects $O_1, \ldots, O_m$. An *execution* E of $\mathcal{S}$ is a sequence $C_0, e_0, C_1, e_1, C_2, e_2, \ldots$, where C_i's are configurations of $\mathcal{S}$, C_0 is the initial configuration, e_i's are events, and C_{i+1} is the configuration that results when event e_i occurs in configuration C_i. The *history* in E is the subsequence of events in E. The *history of object $\mathcal{O}$* in E is the subsequence of events of $\mathcal{O}$ in E. If e and e' are two events in a history H, we write $e <_H e'$ if e is before e' in H. A *complete operation* in H is a pair of events in H — an invocation and its matching response. An *incomplete operation* in H is an invocation that has no matching response. H is *complete* if it has no incomplete operations. If op and op' are two operations in H, we write $op <_H op'$ if the response of op is before the invocation of op' in H. Two operations op and op' are *concurrent* if neither $op <_H op'$ nor $op' <_H op$. H is *sequential* if it has no concurrent operations.

Let H be a history of object $\mathcal{O}$. A *linearization* of H is a complete sequential history S with the following properties:

1. S includes every complete operation in H.
2. Let $inv(P_i, op, \mathcal{O})$ be an invocation in H with no matching response (and is thus an incomplete operation). Then, either S does not include this incomplete operation or S includes a complete operation $(inv(P_i, op, \mathcal{O}), resp(P_i, v, \mathcal{O}))$ for some v.

   Intuitively, this captures the notion that some incomplete operations in H had a "visible" effect, while the others did not.
3. S includes no operations other than the ones mentioned in 1 or 2.
4. For all operations op, op' in S, if $op <_H op'$ then $op <_S op'$.

Thus, the order of non-overlapping operations in H is preserved in S.

Notice that a given history may have several linearizations. A *history H of object $\mathcal{O}$ is linearizable with respect to type T, initialized to state σ,* if H has a linearization which is legal from state σ of T.

Processes are *asynchronous*: there are no bounds on the relative speeds of processes. Furthermore, a process may *crash*: a process may stop at an arbitrary point in an execution and never take any steps thereafter. A process is *correct* in an execution E if it does not crash in E. We assume that every correct process has an infinite number of events in an infinite execution. An *object $\mathcal{O}$ is wait-free in an execution E* if either (i) E is finite, or (ii) every invocation on $\mathcal{O}$ from a process that does not crash in E has a matching response.

Let T be an object type and $\mathcal{L} = (T_1, T_2, \ldots)$ be a (possibly infinite) list of (not necessarily distinct) object types. Let $\Sigma = (\sigma_1, \sigma_2, \ldots)$ be a list where σ_i is a state of type T_i. An *implementation of T, initialized to state σ, from $(\mathcal{L}, \Sigma)$ for N processes* is a function $\mathcal{I}(O_1, O_2, \ldots)$ such that if $O_1, O_2, \ldots$ are N-process objects of type $T_1, T_2, \ldots$, initialized to states $\sigma_1, \sigma_2, \ldots$, respectively, then $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots)$ is an N-process object of type T, initialized to σ. Intuitively, $\mathcal{I}(O_1, O_2, \ldots)$ returns a set of procedures $\texttt{Apply}(P_i, op, \mathcal{O})$, for $1 \leq i \leq N$ and $op \in OP(T)$. $\texttt{Apply}(P_i, op, \mathcal{O})$ specifies how process P_i should "simulate" the operation op on $\mathcal{O}$ in terms of operations on $O_1, O_2, \ldots$. We say $\mathcal{O}$ is a *derived object* of the implementation $\mathcal{I}$, and $O_1, O_2, \ldots, O_n$ are the *base objects* of $\mathcal{O}$.

We say that *$\mathcal{I}$ is an implementation of T, initialized to state σ, from a set $\mathcal{S}$ of types for N processes* if there is a list $\mathcal{L} = (T_1, T_2, \ldots)$ of types and a list $\Sigma = (\sigma_1, \sigma_2, \ldots)$ of states such that $T_i \in \mathcal{S}$, σ_i is a state of T_i, and $\mathcal{I}$ is an implementation of T, initialized to σ, from $(\mathcal{L}, \Sigma)$ for N processes. We say that *a type T has an implementation from a set $\mathcal{S}$ of types for N processes* if for every state σ of T, there is an implementation of T, initialized to σ, from $\mathcal{S}$ for N processes.

An *implementation is wait-free* if it has the following property: if all base objects are wait-free in an execution E, then the derived object is wait-free in E. Hereafter when we write "implementation", it stands for "wait-free implementation".

We now define **consensus** and **register** — two object types that appear frequently in this paper. Type **consensus** supports two operations: **propose**(0) and **propose**(1). The sequential specification of **consensus** is in Figure 1. From the specification, it is clear that a consensus object $\mathcal{O}$ has the following properties: (i) If $\mathcal{O}$ returns a response v, then there is an invocation of **propose**(v) preceding this response, and (ii) $\mathcal{O}$ returns the same response to all operations. These are known as the *validity* and *agreement* properties, respectively, of a consensus object. Sometimes we refer to the *consensus problem* for processes $P_1, P_2, \ldots P_n$. This problem is stated as follows. Each process P_i is initially given a binary input v_i. Each correct process P_i must eventually decide a value d_i such that (i) $d_i \in \{v_1, v_2, \ldots, v_n\}$, and (ii) $\forall 1 \leq i, j \leq n : d_i = d_j$. These two conditions are commonly referred to as the validity and agreement requirements of the consensus problem.

Type **register** supports the operations $\{\texttt{read}\} \cup \{\texttt{write}(v) | v \geq 0\}$, and has the sequential specification given in Figure 2.

$OP = \{\texttt{propose}(v) | v \in \{0,1\}\}$
Object State:
$X \in \{\perp, 0, 1\}$

```
propose(v)
    if X = ⊥ then
        X := v
    return(X)
```

Figure 1: Sequential specification of **consensus**

$OP = \{\texttt{read}\} \cup \{\texttt{write}(v) | v \geq 0\}$
Object State:
$X \in \{0, 1, 2, \ldots\}$

```
read()
    return(X)

write(v)
    X := v
    return(ack)
```

Figure 2: Sequential specification of **register**

3 Hierarchy Preliminaries

A *hierarchy of shared types* is a function that maps object types to levels in $\{1, 2, 3, \ldots\} \cup \{\infty\}$. An object type T is at level l in hierarchy h if $h(T) = l$. A hierarchy is *non-trivial* if it has at least two non-empty levels. An object type T is *universal for N processes* if for every type T', there is an implementation of T' from $\{T, \texttt{register}\}$ for N processes. T *is universal (for ∞ processes)* if for all N, T is universal for N processes. A hierarchy h is a *wait-free hierarchy* if for all T, $h(T) = N$ implies that T is universal for N processes. Thus, in a wait-free hierarchy, the level of T is a lower bound on the number of processes for which T (together with registers) can support arbitrary wait-free synchronization. The following proposition is immediate from the definition.

Proposition 3.1 *If h is a wait-free hierarchy, and h' is a hierarchy such that $\forall T : h'(T) \leq h(T)$, then h' is a wait-free hierarchy.*

Proposition 3.2 *If h is a wait-free hierarchy, then $h(\texttt{register}) = 1$. Thus, level 1 of any wait-free hierarchy is non-empty.*

Proof There exist object types (for example, `queue`) which have no implementation from `register` for two or more processes [Her91b]. Thus, `register` must be at level 1 in any wait-free hierarchy. □

From Proposition 3.1, it is clear that there can be "slack" in a wait-free hierarchy. This motivates us to define tightness. A wait-free hierarchy h is *tight* if for every wait-free hierarchy h' and every type T, $h(T) \geq h'(T)$. A wait-free hierarchy is *fully-refined* if for all levels $k \in \{1, 2, 3, \ldots\} \cup \{\infty\}$, there is some type in level k. A wait-free hierarchy h is *robust* if for every type T and every finite set $\mathcal{S}$ of types, if $h(T) = N$ and $\forall T' \in \mathcal{S} : h(T') < N$, then there is no implementation of T from $\mathcal{S}$ for N processes. The reader should note the difference between tightness and robustness. The trivial wait-free hierarchy which maps every object type to level 1 is obviously robust, but not tight. The wait-free hierarchy $\mathbf{h}_\blacksquare^r$ (to be defined soon) is tight, but it is not known whether it is robust.

In the remainder of this section, we define some natural wait-free hierarchies, and highlight some simple properties of these hierarchies. In the following definitions, the subscript indicates whether the definition allows just 1 or many objects of the argument type. The superscript r indicates that the definition allows the use of registers.

1. $\mathbf{h}_1(T)$ = maximum number of processes for which a consensus object can be implemented using just a single object of type T. If there is no such maximum, then $\mathbf{h}_1(T) = \infty$.

2. $\mathbf{h}_1^r(T)$ = maximum number of processes for which a consensus object can be implemented using just a single object of type T and any number of registers. If there is no such maximum, then $\mathbf{h}_1^r(T) = \infty$.

   Notice that this is Herlihy's hierarchy.

3. $\mathbf{h}_\blacksquare(T)$ = maximum number of processes for which a consensus object can be implemented using any number of objects of type T. If there is no such maximum, then $\mathbf{h}_\blacksquare(T) = \infty$.

4. $\mathbf{h}_\blacksquare^r(T)$ = maximum number of processes for which a consensus object can be implemented using any number of objects of type T and any number of registers. If there is no such maximum, then $\mathbf{h}_\blacksquare^r(T) = \infty$.

Proposition 3.3 *Each of $\mathbf{h}_1, \mathbf{h}_1^r, \mathbf{h}_\blacksquare, \mathbf{h}_\blacksquare^r$ is a fully-refined wait-free hierarchy.*

Proof Herlihy's universality result trivially implies that these are wait-free hierarchies. That these are fully-refined follows from the easy observation that $\forall h \in \{\mathbf{h}_1, \mathbf{h}_1^r, \mathbf{h}_\blacksquare, \mathbf{h}_\blacksquare^r\}$ and

$OP = \{\textbf{propose}(v) | v \in \{0,1\}\}$
Object State:
  $X \in \{\perp, 0, 1\}$
  $N \in \{0, 1, 2, \ldots\}$

propose(v)
  $N := N + 1$
  if $X = \perp$ then
    $X := v$
  if $N \leq k$ **then**
    return(X)
  else return($\perp$)

Figure 3: Sequential specification of k-**cons**

$k \in \{1, 2, 3, \ldots\} \cup \{\infty\}$, $h(k\text{-}\textbf{cons}) = k$. (See Figure 3 for the definition of the type k-**cons**.)
□

Proposition 3.4 $\mathbf{h}^{\mathbf{r}}_{\blacksquare}(T) = N < \infty$ *if and only if* T *is universal for* N *processes, but not for* $N + 1$ *processes.* $\mathbf{h}^{\mathbf{r}}_{\blacksquare}(T) = \infty$ *if and only if* T *is universal.*

Proposition 3.5 *If* h *is a tight wait-free hierarchy, then* $h = \mathbf{h}^{\mathbf{r}}_{\blacksquare}$. *In other words,* $\mathbf{h}^{\mathbf{r}}_{\blacksquare}$ *is the unique wait-free hierarchy which is tight.*

The hierarchy $\mathbf{h}^{\mathbf{r}}_{\blacksquare}$ is uniquely important in the study of robust wait-free hierarchies. To formally state this, we need a definition. Let $\sigma = (l_1, l_2, \ldots)$ be a finite/infinite sequence such that $1 = l_1 < l_2 < l_3 \ldots$ and $l_i \in \{1, 2, 3, \ldots\} \cup \{\infty\}$. We say g is a *coarsening of hierarchy* h *with respect to* σ if, for all object types T, we have:

1. If $l_i \leq h(T) < l_{i+1}$, then $g(T) = l_i$.
2. If $l_i \leq h(T)$ and l_i is the last element of σ, then $g(T) = l_i$.
3. If $h(T) = \infty$ and σ is infinite, then $g(T) = \infty$.

Intuitively, levels $l_i \ldots (l_{i+1} - 1)$ in h are lumped into level l_i of g, causing levels $(l_i + 1) \ldots (l_{i+1} - 1)$ to be empty in g. We say g is a *coarsening* of a hierarchy h if there is a σ of the form $1 = l_1 < l_2 < l_3 \ldots$ such that g is a coarsening of h with respect to σ. It is obvious that if h is a wait-free hierarchy, so is every coarsening of h.

Theorem 3.1 *If* h *is a robust wait-free hierarchy, then* h *is a coarsening of* $\mathbf{h}^{\mathbf{r}}_{\blacksquare}$.

Proof Assume that h is a robust wait-free hierarchy, and is not a coarsening of $\mathbf{h}_\blacksquare^r$. Let $\sigma = (l_1, l_2, \ldots)$, where $1 = l_1 < l_2 < l_3 \ldots$ are all the non-empty levels of h. Define g to be the coarsening of $\mathbf{h}_\blacksquare^r$ with respect to σ. From our assumption that h is not a coarsening of $\mathbf{h}_\blacksquare^r$, it follows that $h \neq g$. Thus, there is a type T such that $h(T) \neq g(T)$. Let $m = h(T)$ and $n = g(T)$. By definition of g, a level k of g is non-empty if and only if level k of h is non-empty. Together with $m \neq n$, this implies that there exist types T' and T'', each different from T, such that $g(T') = m$ and $h(T'') = n$. Since $m \neq n$, we are left with two cases to consider.

1. $m < n$.

   Since $g(T) = n$, it follows that $\mathbf{h}_\blacksquare^r(T) \geq n$. Thus, by Proposition 3.4, T is universal for n processes. In particular, there is an implementation of T'' from $\{T, \texttt{register}\}$ for n processes. Since $h(T) = m < n = h(T'')$, h is not robust. This is a contradiction.

2. $m > n$.

   From the above, $g(T') = m$. Thus, level m of g is not empty. This, together with $m > n$, implies that $n \leq \mathbf{h}_\blacksquare^r(T) < m$. This implies, by Proposition 3.4, that T is not universal for m processes. Since $h(T) = m$, it follows that h is not a wait-free hierarchy. This is a contradiction.

This completes the proof of the theorem. □

What can we say about the robustness of $\mathbf{h}_1, \mathbf{h}_1^r$, and $\mathbf{h}_\blacksquare$? This question is addressed by the following proposition.

Proposition 3.6 *Let $h \in \{\mathbf{h}_1, \mathbf{h}_1^r, \mathbf{h}_\blacksquare\}$. If $h \neq \mathbf{h}_\blacksquare^r$, then h is neither tight nor robust.*

Proof Proposition 3.5 implies that h is not tight. Theorem 3.1 and Proposition 3.3 imply that h is not robust. □

Does one of $\mathbf{h}_1, \mathbf{h}_1^r$, and $\mathbf{h}_\blacksquare$ define the same hierarchy as $\mathbf{h}_\blacksquare^r$? The answer is not easy. For instance, $\mathbf{h}_1^r$ differs from $\mathbf{h}_\blacksquare^r$ if and only if there is a type such that multiple objects of this type (together with registers) can solve consensus among a larger number of processes than a single object (together with registers) can. Does such a type exist? No common object type exhibits such a property and, hence, it is a non-trivial question. Similarly, $\mathbf{h}_\blacksquare$ differs from $\mathbf{h}_\blacksquare^r$ if and only if there is a type such that the use of registers increases the number of processes for which consensus can be solved using objects of this type. Again, common object types do not exhibit this property, making it difficult to answer whether such types exist.

In the rest of the paper, we prove that each of $\mathbf{h}_1, \mathbf{h}_1^r$, and $\mathbf{h}_\blacksquare$ differs from $\mathbf{h}_\blacksquare^r$. Thus, none of $\mathbf{h}_1, \mathbf{h}_1^r$, and $\mathbf{h}_\blacksquare$ is robust. In particular, $\mathbf{h}_1^r$, which is the same as Herlihy's wait-free hierarchy, is not robust. Unfortunately, we do not yet know whether $\mathbf{h}_\blacksquare^r$ or some coarsening of it is robust. This is an important open question. We hope that the ideas employed in this paper would provide useful insights.

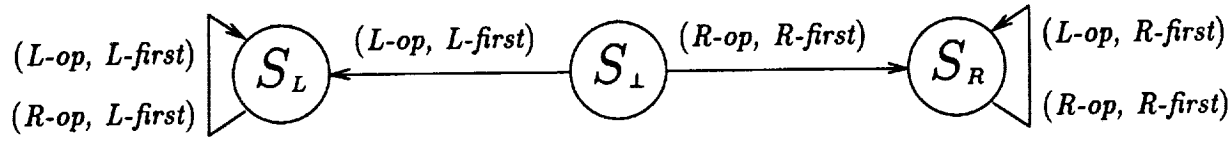

Figure 4: Object type $\mathtt{T_{sticky}}$

4 On the robustness of $\mathtt{h}_1^r$ (Herlihy's hierarchy)

The main result of this section is that $\mathtt{h}_1^r$ is not robust. We prove this result by presenting an object type $\mathtt{T_{sp}}$ with the following property: n $\mathtt{T_{sp}}$ objects, together with registers, can implement a consensus object for $n+1$ processes, but not for $n+2$ processes. This implies $\mathtt{h}_1^r(\mathtt{T_{sp}}) = 2$ and $\mathtt{h}_m^r(\mathtt{T_{sp}}) = \infty$. Thus, $\mathtt{h}_1^r \neq \mathtt{h}_m^r$, and by Proposition 3.6, $\mathtt{h}_1^r$ is not robust.

Consider the object type $\mathtt{T_{sticky}}$ in Figure 4. It supports two operations, *L-op* and *R-op*, and responds with either *L-first* or *R-first*. If *L-op* is applied on a $\mathtt{T_{sticky}}$ object $\mathcal{O}$, initialized to state $S_\perp$, $\mathcal{O}$ changes state to S_L and returns *L-first* as the response. Furthermore, $\mathcal{O}$ returns *L-first* to all subsequent operations, reflecting the fact that *L-op* was the first operation applied on $\mathcal{O}$. The behavior is symmetric if, instead of *L-op*, *R-op* was the first operation applied on $\mathcal{O}$. In essence, the first operation "sticks" to $\mathcal{O}$ and determines the response for all operations. Notice that $\mathtt{T_{sticky}}$ is similar to the **consensus** [Her91b] and **sticky-bit** [Plo89] object types.

Now consider the type $\mathtt{T_{sp}}$, a variant of $\mathtt{T_{sticky}}$, shown in Figure 5. $\mathtt{T_{sp}}$ lacks the symmetry of $\mathtt{T_{sticky}}$: If *R-op* is applied to a $\mathtt{T_{sp}}$ object $\mathcal{O}$, initialized to $S_\perp$, *R-op* sticks to $\mathcal{O}$ as before. However, as soon as *R-op* is applied for the second time, it "unsticks" and $\mathcal{O}$ starts behaving as though it had been stuck with *L-op* all along. The following is a trivial consequence of the definition of $\mathtt{T_{sp}}$.

Lemma 4.1 *Let $\mathcal{O}$ be an object of type $\mathtt{T_{sp}}$ initialized to $S_\perp$. Let E be an execution in which* R-op *is applied at most once on $\mathcal{O}$. Then, the following statements are true in E.*

1. *If r_1 and r_2 are the responses to any two operations on $\mathcal{O}$, then $r_1 = r_2$.*
2. *If $\mathcal{O}$ returns a response* D-first *($D \in \{L, R\}$), then an invocation of* D-op *precedes this response.*

4.1 Implementing consensus from $\{\mathtt{T_{sp}}, \mathtt{register}\}$ — upper bound

In this section, we show how to implement a consensus object for n processes using $(n-1)$ $\mathtt{T_{sp}}$ objects and $2(n-1)$ registers. Our implementation is recursive. Let $\mathcal{I}_j$ denote the

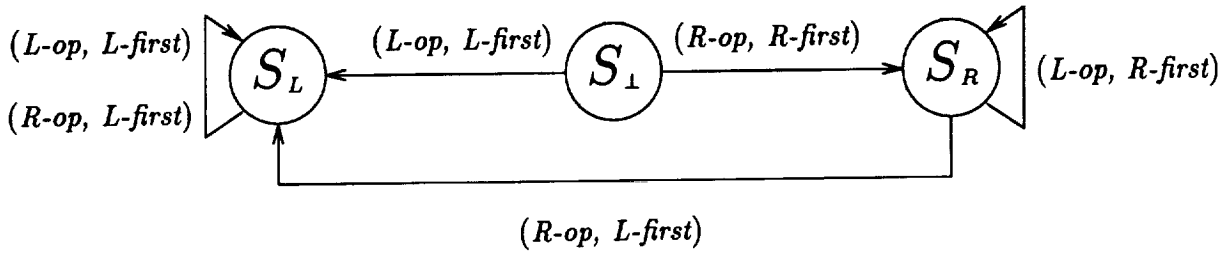

Figure 5: Object type $\mathtt{T_{sp}}$

$\mathcal{O}_{n-1}$: consensus object for $P_1, P_2, \ldots, P_{n-1}$, derived from $\mathcal{I}_{n-1}$
O_{sp}: $\mathtt{T_{sp}}$ object, initialized to $S_\perp$
L, R: binary registers

Apply(P_i, *propose* v_i, $\mathcal{O}_n$) (for $1 \leq i \leq n-1$)

1. $L :=$ Apply(P_i, *propose* v_i, $\mathcal{O}_{n-1}$)
2. if Apply(P_i, *L-op*, O_{sp}) = *L-first*
3. return(L)
4. else return(R)

Apply(P_n, *propose* v_n, $\mathcal{O}_n$)

$R := v_n$
if Apply(P_n, *R-op*, O_{sp}) = *L-first*
return(L)
else return(R)

Figure 6: Implementing **consensus** with $\mathtt{T_{sp}}$ and **register**

---

implementation of **consensus** from $\{\mathtt{T_{sp}}, \mathtt{register}\}$ for processes $P_1, P_2, \ldots, P_j$. The base case is to derive $\mathcal{I}_1$, implementation of **consensus** for the single process P_1, and is trivial: if $\mathcal{O}_1$ is a derived object of $\mathcal{I}_1$, Apply(P_1, *propose* v_1, $\mathcal{O}_1$) simply returns v_1. The recursive step of deriving $\mathcal{I}_n$ from $\mathcal{I}_{n-1}$ is presented in Figure 6.

Lemma 4.2 *The implementation $\mathcal{I}_n$ in Figure 6 is a correct implementation of* **consensus** *from $\{\mathtt{T_{sp}}, \mathtt{register}\}$ for processes $P_1, P_2, \ldots, P_n$. $\mathcal{I}_n$ requires $(n-1)$ objects of type $\mathtt{T_{sp}}$ and $2(n-1)$ registers.*

Proof We prove the correctness of $\mathcal{I}_n$ by induction. The following is the induction hypothesis: for $1 \leq j \leq n-1$, $\mathcal{I}_j$ is a correct implementation of **consensus** for processes $P_1, P_2, \ldots, P_j$. The base case, namely, that $\mathcal{I}_1$ (described above) is a correct implementation of **consensus** for P_1, is obvious. The induction step is proved through several simple

claims. Let $\mathcal{O}_n$ be a derived object of $\mathcal{T}_n$. Consider an execution E of the concurrent system $(P_1, P_2, \ldots, P_n; \mathcal{O}_n)$. Assume that each P_i executes Apply(P_i, *propose* v_i, $\mathcal{O}_n$) at most once in E.[4] We make the following claims about E. The proof of each claim follows its statement.

C1. For $D \in \{L, R\}$, the following holds:

1. Every process that writes the register D, writes the same value V in D.
2. If $D = L$, $V \in \{v_1, v_2, \ldots, v_{n-1}\}$. Otherwise, $V = v_n$.

For $D = R$, the claim is obvious since only P_n writes R. For $D = L$, the claim follows from the agreement and validity properties of $\mathcal{O}_{n-1}$.

C2. Some process completes a write on D before any process receives the response *D-first* from O_{sp}.

By Lemma 4.1, some process, say P_k, invokes *D-op* before any process receives the response *D-first*. By the implementation, this process P_k will have completed a write on the register D before invoking *D-op* on O_{sp}.

Consider, for arbitrary i, j and $i \neq j$, the executions of Apply(P_i, *propose* v_i, $\mathcal{O}_n$) and Apply(P_j, *propose* v_j, $\mathcal{O}_n$) in E. By Lemma 4.1, the responses received by P_i and P_j from O_{sp} (in Statement 2 of their respective executions) are the same. Let *D-first* be this response (for some $D \in \{L, R\}$). Thus, in Statement 3, both Apply(P_i, *propose* v_i, $\mathcal{O}_n$) and Apply(P_j, *propose* v_j, $\mathcal{O}_n$) read and return the value in the register D. From Claims **C2** and **C1**, it follows that both Apply(P_i, *propose* v_i, $\mathcal{O}_n$) and Apply(P_j, *propose* v_j, $\mathcal{O}_n$) read the same value V in D and that $V \in \{v_1, v_2, \ldots, v_n\}$. Thus, the value returned by both Apply(P_i, *propose* v_i, $\mathcal{O}_n$) and Apply(P_j, *propose* v_j, $\mathcal{O}_n$) is the same and is from $\{v_1, v_2, \ldots, v_n\}$. It is obvious that the implementation is wait-free. Hence the lemma. □

Corollary 4.1 $\mathbf{h}_{\mathbf{m}}^{\mathbf{r}}(\mathbf{T_{sp}}) = \infty$.

4.2 Implementing consensus from {$\mathbf{T_{sp}}$, register} — lower bound

The main technical result of this section states that any solution to n-process wait-free consensus using $\mathbf{T_{sp}}$ objects and registers requires at least $n - 1$ $\mathbf{T_{sp}}$ objects, regardless of how many registers are available. We prove this result by reducing the "1-resilient consensus problem for n processes communicating via registers[5]" to the "wait-free consensus problem for n processes communicating via registers and $(n-2)$ $\mathbf{T_{sp}}$ objects". The former problem is impossible to solve [LAA87]. Hence the impossibility of the latter. The reduction is based on the novel concept of *k-trap implementations.*

[4] This is not a limitation for the following reason. After P_i executes Apply(P_i, *propose* v_i, $\mathcal{O}_n$) once, it can record the return value in its local variable. Thereafter, when P_i needs to apply a *propose* operation on $\mathcal{O}_n$, it may simply return the value of this local variable as the response. This strategy works because $\mathcal{O}_n$ is a consensus object, and therefore must return the same response to every invocation.

[5] A protocol is *k-resilient* if it meets the problem specification despite the crash of k or fewer processes.

4.2.1 k-trap implementations

An implementation for processes $P_1, P_2, \ldots, P_n$ is a *k-trap implementation* if every derived object $\mathcal{O}$ of the implementation has the following property: in any execution of $(P_1, P_2, \ldots, P_n; \mathcal{O})$, regardless of the relative execution speeds of processes, all but up to k correct processes will be able to eventually complete their operations on $\mathcal{O}$. In other words, $\mathcal{O}$ appears wait-free to all but up to k correct processes.

We now contrast k-trap implementations with the familiar wait-free, non-blocking, and critical-section based implementations. Critical-section based implementations and non-blocking implementations (for n processes) are both $(n-1)$-trap implementations. A critical-section based implementation is $(n-1)$-trap because the crash of a single process in the critical section blocks the remaining $(n-1)$ processes. A non-blocking implementation is $(n-1)$-trap because repeated execution of operations by one process could cause the remaining processes to block. The converse does not hold: an $(n-1)$-trap implementation does not guarantee the properties of either a critical-section based implementation or a non-blocking implementation. To see this, suppose that exactly one process, say P, attempts to access the object, and suppose that P is correct. In the case of a critical-section based implementation or a non-blocking implementation, P is guaranteed to complete its operation on the object. But in a k-trap implementation ($k \geq 1$), P may block. Finally, note that a 0-trap implementation is the same as a wait-free implementation.

The following lemma establishes the utility of k-trap implementations in proving lower-bounds.

Lemma 4.3 *Let* T *be any object type such that for every state σ of* T*, there is a 1-trap implementation $\mathcal{I}_\sigma$ of* T*, initialized to σ, from* `register` *for n processes. Then, any wait-free implementation of* `consensus` *from* {T, `register`} *for n processes requires at least $n-1$ objects of type* T *(regardless of how many registers it uses).*

Proof Suppose that the lemma is false, and there is a wait-free implementation $\mathcal{J}$ of `consensus` from {T, `register`} for n processes such that $\mathcal{J}$ requires only $n-2$ objects of type T, initialized to states $\sigma_1, \sigma_2, \ldots, \sigma_{n-2}$ of T, and m registers (for some $m \geq 0$). Consider the protocol $\mathcal{P}$ in Figure 7. Clearly, processes communicate exclusively via registers in protocol $\mathcal{P}$. We argue below that $\mathcal{P}$ solves the consensus problem for processes $P_1, P_2, \ldots, P_n$ even if (at most) one of the processes may crash. By the impossibility result in [LAA87], such a protocol does not exist. Hence the lemma.

We claim that at most $(n-2)$ processes block on $\mathcal{O}$. This follows from the following facts:

1. $n-2$ base objects of $\mathcal{O}$ are 1-trap. So at most one process blocks on each of these.
2. No process blocks on the remaining base objects of $\mathcal{O}$, the registers $R_1, R_2, \ldots, R_m$.
3. $\mathcal{O}$ is derived from a wait-free implementation.

1. For $1 \leq i \leq n-2$, use $\mathcal{I}_{\sigma_i}$ to implement an object O_i of type T initialized to state σ_i.
2. Use $\mathcal{J}$ to implement a consensus object $\mathcal{O}$ from $O_1, O_2, \ldots, O_{n-2}$ and registers $R_1, R_2, \ldots, R_m$.
3. Let D be a 3-valued register initialized to $\perp$.
4. For $1 \leq i \leq n$, let v_i be the binary input value of process P_i for consensus. Process P_i executes the following procedure. We require that statements 1 and 2 are executed in a fair manner.

```
cobegin
    1.  D := Apply(P_i, propose v_i, O)
    2.  repeat until (D ≠ ⊥).
        decide D
coend
```

Figure 7: 1-resilient consensus protocol $\mathcal{P}$ for n processes

Therefore, if at most one of $P_1, P_2, \ldots, P_n$ crashes, there is still one process, call it P_k, that neither crashes nor blocks on $\mathcal{O}$. This process P_k eventually writes the response, call it V, returned by Apply(P_k, *propose* v_k, $\mathcal{O}$) in register D. Since $\mathcal{O}$ satisfies validity, we have $V \in \{v_1, v_2, \ldots, v_n\}$. Since $\mathcal{O}$ satisfies agreement, no process ever writes a value different from V in register D. Since Statements 1 and 2 are executed in a fair manner, every non-crashing process eventually reads V and decides V. In other words, $\mathcal{P}$ solves the consensus problem for $P_1, P_2, \ldots, P_n$ even if at most a single process may crash. □

4.2.2 1-trap implementation of $\mathtt{T_{sp}}$

Recall that $\mathtt{T_{sp}}$ has three states - $S_\perp, S_L$, and S_R. We now present a 1-trap implementation of $\mathtt{T_{sp}}$ initialized to $S_\perp$, and 0-trap implementations of $\mathtt{T_{sp}}$ initialized to S_L or S_R. These implementations use only registers as base objects. Thus, by Lemma 4.3, we have the desired lower bound.

A 1-trap implementation of $\mathtt{T_{sp}}$, initialized to $S_\perp$, from **register** for n processes is presented in Figure 8. This implementation is subtle. We present below an informal and intuitive argument of its correctness before proceeding to give the formal proof. Consider $\mathcal{O}$, a $\mathtt{T_{sp}}$ object derived from this implementation. Let H be a history of $\mathcal{O}$, and let *first-op* denote the first operation to complete in H. There are two cases. Case (1) corresponds to *first-op* being an *L-op* operation. Consider the linearization S which includes only the complete operations in H and sequences them in the order of their completion times. Thus,

$R[1 \ldots n]$: binary (1-writer, n-reader) registers initialized to 0

Apply(P_i, *L-op*, $\mathcal{O}$)	Apply(P_i, *R-op*, $\mathcal{O}$)
return(*L-first*)	1. if $(\forall k : R[k]= 0)$ then 2. $\quad R[i] := 1$ 3. $\quad$ repeat until $(\exists j < i : R[j]= 1)$ 4. return(*L-first*)

Figure 8: 1-trap implementation of $\mathtt{T_{sp}}$, initialized to $S_\perp$, from `register`

first-op, which is an *L-op* operation, becomes the the first operation in S. Furthermore, the response of every operation in S is *L-first* (this is obvious from the implementation). From the sequential specification of $\mathtt{T_{sp}}$ in Figure 5, it is obvious that S is legal from the state $S_\perp$ of $\mathtt{T_{sp}}$. Now consider Case (2), which corresponds to *first-op* being an *R-op* operation. The key observation is that if *first-op*, which is an *R-op* operation, completed in H, then by our implementation, there must be another *R-op* operation, call it *blocked-op*, from a different process which is concurrent with *first-op* and is blocked. Let us pretend that, although incomplete, *blocked-op* has indeed taken effect in H, and has *R-first* for its response. Consider the linearization S which sequences *blocked-op* first, *first-op* second, and the remaining complete operations in H in the order of their completion times. (*blocked-op* can be linearized before *first-op* since these two operations are concurrent.) Thus the first operation in the linearization S is a *R-op* operation with *R-first* as the associated response. The second operation in the linearization is also an *R-op* operation, and has *L-first* as the associated response. The remaining operations in the linearization have *L-first* as their response. From the sequential specification of $\mathtt{T_{sp}}$ in Figure 5, it is obvious that this linearization S is legal from the state $S_\perp$ of $\mathtt{T_{sp}}$. Hence the correctness of our implementation. We formalize the above arguments and present a more rigorous proof of correctness below. The proof is based on a series of claims.

Claim 4.1 *The implementation is 1-trap.*

Proof Clearly, a correct process P_i blocks if and only if the *repeat* $\cdots$ *until* loop (Statement 3 of Apply(P_i, *R-op*, $\mathcal{O}$)) never terminates. By Statement 2, such a P_i will have written the value 1 into $R[i]$.

Suppose that the claim is false, and two correct processes P_i and P_j (assume $j < i$) block on $\mathcal{O}$. It follows that $R[i] = R[j] = 1$ and each of P_i and P_j is caught in the *repeat* $\cdots$ *until* loop that never terminates. Process P_i eventually notices that $R[j] = 1$, and since $j < i$, P_i quits the *repeat* $\cdots$ *until* loop, and returns *L-first*. This contradicts the assumption that P_i

blocks on $\mathcal{O}$. □

The next claim asserts that if a process P_i successfully completes an *R-op* operation on $\mathcal{O}$, then a different process P_j is already blocked, unable to complete its *R-op* operation on $\mathcal{O}$.

Claim 4.2 *Let E be an execution of $(P_1, P_2, \ldots, P_n; \mathcal{O})$, and H be the corresponding history. Suppose that H contains the two events — an invocation $e_i^{inv} = inv(P_i,$ R-op, $\mathcal{O})$ and its matching response $e_i^{res} = resp(P_i,$ L-first, $\mathcal{O})$. Then H contains an invocation $e_j^{inv} = inv(P_j,$ R-op, $\mathcal{O})$ such that*

1. *$e_j^{inv} <_H e_i^{res}$, and*
2. *e_j^{inv} has no matching response in H.*

Proof The proof of this claim is based on the following observations:

O1. The predicate $\exists k : R[k] = 1$ is stable: that is, if it holds in some configuration of an execution, it holds in every subsequent configuration of that execution. Furthermore, this predicate must hold before a response can occur to any invocation of *R-op*.

The first part of this observation follows from the fact that once a 1 is written to a register, it is never changed. The second part is obvious from Statements 1 and 2 of the implementation.

O2. In H, let k be the smallest integer such that P_k has an invocation $e_k^{inv} = inv(P_k,$ *R-op*, $\mathcal{O})$ and P_k writes a 1 in $R[k]$. Then e_k^{inv} has no matching response in H.

To see this, notice that after writing a 1 in $R[k]$, P_k enters the *repeat* $\cdots$ *until* loop. This loop never terminates in H because of our premise that k is the smallest integer such that P_k writes a 1 in $R[k]$. Thus P_k does not return from Apply(P_k, *R-op*, $\mathcal{O}$).

O3. In H, if a process P_k writes 1 in $R[k]$ after an invocation $e_k^{inv} = inv(P_k,$ *R-op*, $\mathcal{O})$ and before its matching response, then $e_k^{inv} <_H e_i^{res}$.

Suppose not. Then $e_i^{res} <_H e_k^{inv}$. After the invocation e_k^{inv}, when P_k executes Statement 1 of the procedure Apply(P_k, *R-op*, $\mathcal{O}$), the guard $\forall k : R[k] = 0$ evaluates to *false* (by **O1**). Thus P_k returns the response *L-first* without writing into $R[k]$. This contradicts the premise that P_k writes 1 into $R[k]$ after the invocation e_k^{inv} and before its response.

To complete the proof of the claim, let S be the set of processes that invoke *R-op* on $\mathcal{O}$ and write 1 into a register in the execution E. Since H contains a response event e_i^{res}, by **O1**, S is non-empty. Let j be the smallest integer such that $P_j \in S$. By **O2**, P_j's invocation e_j^{inv} of *R-op* on $\mathcal{O}$ has no matching response in H. By **O3**, $e_j^{inv} <_H e_i^{res}$. Hence the claim. □

Claim 4.3 *Let E be an execution of $(P_1, \ldots, P_n; \mathcal{O})$, and H be the history of $\mathcal{O}$ in E. H is linearizable with respect to* $\mathtt{T_{sp}}$*, initialized to state $S_\perp$.*

Proof If H has no response events, then the claim is trivial: the empty sequence is a linearization of H and is legal from state $S_\perp$ of $\mathtt{T_{sp}}$. Assume, therefore, that H has one or more response events. Let $e_i^{res} = resp(P_i, L\text{-}first, \mathcal{O})$ be the earliest response in H. Let e_i^{inv} be the invocation whose matching response is e_i^{res}. There are two cases:

Case 1. $e_i^{inv} = inv(P_i, L\text{-}op, \mathcal{O})$

This corresponds to the case in which the first operation to complete is an *L-op* operation from process P_i. Define a sequential history S as follows:

1. S includes all complete operations in H.
2. If two operations op and op' are in S, $op <_S op'$ if and only if response of op precedes the response of op' in H.

It is obvious that (i) S is a linearization of H, and (ii) S is legal from the state $S_\perp$ of $\mathtt{T_{sp}}$.

Case 2. $e_i^{inv} = inv(P_i, R\text{-}op, \mathcal{O})$

This corresponds to the case in which the first operation to complete is an *R-op* from process P_i. By Claim 4.2, there is an invocation $e_j^{inv} = inv(P_j, R\text{-}op, \mathcal{O})$ such that $e_j^{inv} <_H e_i^{res}$ and e_j^{inv} has no matching response in H. Define a sequential history S as follows:

1. S includes all complete operations in H, and the operation (e_j^{inv}, e_j^{res}), where $e_j^{res} = resp(P_j, R\text{-}first, \mathcal{O})$.
2. The operation (e_j^{inv}, e_j^{res}) precedes all other operations in S.
3. If op and op' are operations in S different from (e_j^{inv}, e_j^{res}), $op <_S op'$ if and only if the response of op precedes the response of op' in H.

It is easy to verify that (i) S is a linearization of H, and (ii) S is legal from the state $S_\perp$ of $\mathtt{T_{sp}}$.

Hence the claim. □

Lemma 4.4 *Figure 8 presents a 1-trap implementation of* $\mathtt{T_{sp}}$*, initialized to* $S_\perp$*, from* `register` *for processes* $P_1, P_2, \ldots, P_n$.

Proof Follows from Claims 4.1 and 4.3. □

Lemma 4.5 *Figure 9 presents a 0-trap (wait-free) implementation of* $\mathtt{T_{sp}}$*, initialized to* S_R*, from* `register` *for processes* $P_1, P_2, \ldots, P_n$.

Proof Let E be an execution of $(P_1, P_2, \ldots, P_n; \mathcal{O})$, and let H_R and $H_\mathcal{O}$ be the histories of objects R and $\mathcal{O}$, respectively, in E. Let Σ_R be a linearization of H_R, which is legal from the state 0 of `register`. For every operation $op \in \Sigma_R$, define $f(op)$ as follows:

R: binary register initialized to 0

Apply(P_i, L-op, $\mathcal{O}$)

if ($R = 0$) then
    return(R-*first*)
else return(L-*first*)

Apply(P_i, R-op, $\mathcal{O}$)

$R := 1$
return(L-*first*)

Figure 9: 0-trap implementation of $\mathtt{T_{sp}}$, initialized to S_R, from `register`

if $op = (inv(P_i, read, R), resp(P_i, 0, R))$ **then**
    $f(op) = (inv(P_i, L\text{-}op, \mathcal{O}), resp(P_i, R\text{-}first, \mathcal{O}))$
else if $op = (inv(P_i, read, R), resp(P_i, 1, R))$ **then**
    $f(op) = (inv(P_i, L\text{-}op, \mathcal{O}), resp(P_i, L\text{-}first, \mathcal{O}))$
else if $op = (inv(P_i, write\ 1, R), resp(P_i, ack, R))$ **then**
    $f(op) = (inv(P_i, R\text{-}op, \mathcal{O}), resp(P_i, L\text{-}first, \mathcal{O}))$

Define a sequential history $\Sigma_\mathcal{O}$ as follows:

1. For every operation $op \in \Sigma_R$, include $f(op)$ in $\Sigma_\mathcal{O}$.
2. If $op, op' \in \Sigma_R$ and $op <_{\Sigma_R} op'$, then $f(op) <_{\Sigma_\mathcal{O}} f(op')$.

It is easy to verify that $\Sigma_\mathcal{O}$ is a linearization of $H_\mathcal{O}$, and is legal from the state S_R of $\mathtt{T_{sp}}$. □

Lemma 4.6 *Figure 10 presents a 0-trap (wait-free) implementation of* $\mathtt{T_{sp}}$, *initialized to* S_L, *from* `register` *for processes* $P_1, P_2, \ldots, P_n$.

Proof Obvious. □

Lemma 4.7 *Any wait-free implementation of* `consensus` *from* $\{\mathtt{T_{sp}}, \mathtt{register}\}$ *for* n *processes requires at least* $n - 1$ *objects of type* $\mathtt{T_{sp}}$.

Proof Follows from Lemma 4.3, and Claims 4.4, 4.5, and 4.6. □

Corollary 4.2 $\mathtt{h}_1^r(\mathtt{T_{sp}}) = 2$.

Proof By Lemma 4.2, $\mathtt{h}_1^r(\mathtt{T_{sp}}) \geq 2$. By Lemma 4.7, $\mathtt{h}_1^r(\mathtt{T_{sp}}) \leq 2$. Hence the result. □

Apply(P_i, *L-op*, $\mathcal{O}$)	Apply(P_i, *R-op*, $\mathcal{O}$)
return(*L-first*)	return(*L-first*)

Figure 10: 0-trap implementation of $\mathtt{T_{sp}}$, initialized to S_L

Theorem 4.1 $\mathtt{h}_1^r$ *is neither tight nor robust.*

Proof Follows from Proposition 3.6 and Corollaries 4.1 and 4.2. □

Theorem 4.2 $\mathtt{h}_1$ *is neither tight nor robust.*

Proof From the definitions of $\mathtt{h}_1$ and $\mathtt{h}_1^r$, it is obvious that, for all types T, $\mathtt{h}_1(T) \leq \mathtt{h}_1^r(T)$. In particular, $\mathtt{h}_1(\mathtt{T_{sp}}) \leq \mathtt{h}_1^r(\mathtt{T_{sp}}) = 2 < \infty = \mathtt{h}_m^r(\mathtt{T_{sp}})$. Thus, by Proposition 3.6, $\mathtt{h}_1$ is neither tight nor robust. □

5 On the robustness of $\mathtt{h_m}$

The main result of this section is that $\mathtt{h_m}$ is not robust. We prove this result by presenting an infinite family $\mathtt{T}_{nd}^k$, $k \in \{2, 3, 4, \ldots\} \cup \{\infty\}$, of object types with the following properties:

1. There is an implementation of **consensus** from $\{\mathtt{T}_{nd}^k, \mathtt{register}\}$ for k processes, but not for $k+1$ processes.
2. There is no implementation of **consensus** from $\mathtt{T}_{nd}^k$ for two processes.

Property (1) implies that $\mathtt{h}_m^r(\mathtt{T}_{nd}^k) = k$. Property (2) implies that $\mathtt{h_m}(\mathtt{T}_{nd}^k) = 1$. Thus, $\mathtt{h_m} \neq \mathtt{h}_m^r$, and by Proposition 3.6, $\mathtt{h_m}$ is not robust.[6] This result is significant in the following sense. Registers by themselves are too weak to solve even 2-process consensus. So are $\mathtt{T}_{nd}^\infty$ objects. Combining these two types, however, lets us solve consensus among any number of processes!

The object type $\mathtt{T}_{nd}^k$ is specified in Figure 11. In this specification, *choose*(S) is assumed to choose an element from set S non-deterministically and return it. Notice that *upset* and *ahead*[i] are stable: once true, they remain true. Similarly, once *decision* $\in \{0, 1\}$, it does not change.

[6] A single member of the $\mathtt{T}_{nd}^k$ family is sufficient to establish that $\mathtt{h_m}$ is not robust. The existence of an entire family shows that there is not even a coarsening of $\mathtt{h_m}$ which is non-trivial and robust.

S1. T^{k}_{nd} supports operations in $\{\text{op}(i)|i = \{0,1\}\} \cup \{\texttt{give-decision}(i,b)|i \in \{0,1\}, b \in \{true, false\}\}$.

S2. The response for op(0) or op(1) is always *ack*. The response for **give-decision**$(-,-)$ is either 0 or 1.

S3. The state of T^{k}_{nd} is represented by the variables n_0, n_1, n_{gd} : **integer**; *decision* $\in \{\perp, 0, 1\}$; *ahead*[0..1], *upset* : **boolean**. Informally, n_0, n_1, n_{gd} count the number of executions of op(0), op(1), and **give-decision**, respectively. The variable *ahead*[i] is set to *true* if $n_i > 0$ and $n_{\bar{i}} = 0$ when **give-decision**$(i,-)$ is executed. The variable *upset* is set to *true* if one of the following happens: (i) op(1) is executed more than once (op(0) may be executed any number of times without upsetting a T^{k}_{nd} object); (ii) **give-decision** is executed more than k times; (iii) **give-decision**$(i,-)$ is executed with no prior execution of op(i); (iv) **give-decision**$(i, true)$ is executed with no prior execution of op($\bar{i}$); (v) **give-decision**$(i, false)$ is executed and *ahead*[$\bar{i}$] = *true*. If upset, a T^{k}_{nd} object returns 0 or 1 non-deterministically to an invocation of **give-decision**. If not upset, it sets *decision* irrevocably and non-deterministically (if not already set) to 0 or 1 such that $n_{decision} > 0$, and returns *decision*. See **S5** below for a formal sequential specification of T^{k}_{nd}.

S4. The state of T^{k}_{nd} corresponding to ($n_0 = n_1 = n_{gd} = 0$; *decision* $= \perp$; *ahead*[0..1] = *upset* $= false$) is known as the *fresh state*. The states of T^{k}_{nd} are *only* those that are reachable from the fresh state by the following specification.

S5. The sequential specification of T^{k}_{nd} is as follows:

```
op(i)                                   /*  i ∈ {0,1}  */
    n_i := n_i + 1
    if n_1 > 1 then upset := true
    return(ack)

give-decision(i, other-is-ahead)        /*  i ∈ {0,1}, other-is-ahead: boolean  */
    n_gd := n_gd + 1
    if (n_i > 0 ∧ n_ī = 0) then ahead[i] := true
    if (n_gd > k) ∨ (n_i = 0) ∨ (ahead[ī] ∧ ¬other-is-ahead) ∨ (n_ī = 0 ∧ other-is-ahead) then
        upset := true
    if upset then
        return(choose({0,1}))
    else if decision = ⊥ then
        decision := choose({j|n_j > 0})
    return(decision)
```

Figure 11: Object type T^{k}_{nd}

5.1 consensus from $\{T^k_{nd}, \texttt{register}\}$ — an implementation

In this section, we show, for $k \in \{2, 3, \ldots\} \cup \{\infty\}$, how to implement a consensus object for k processes using only T^k_{nd} objects and registers. Our implementation is recursive. Let $\mathcal{I}^k_n$ denote the implementation of **consensus** from $\{T^k_{nd}, \texttt{register}\}$ for processes $P_1, P_2, \ldots, P_n$. The base case is to derive $\mathcal{I}^k_0$, implementation of **consensus** for an empty set of processes, and is vacuous. The recursive step of deriving $\mathcal{I}^k_n$ from $\mathcal{I}^k_{n-1}$ is presented in Figure 12.

The implementation $\mathcal{I}^k_n$ works as follows. Processes $P_1 \ldots P_n$ split into two groups, G_0 and G_1. Group G_0 has $P_1 \ldots P_{n-1}$, and group G_1 has just P_n. Processes $P_1 \ldots P_{n-1}$ do consensus among themselves (recursively) and announce the outcome in $R[0]$. Process P_n announces its input value in $R[1]$. The rest of the protocol resolves which of the two groups is the winner. If G_0 wins, every process decides the value in $R[0]$. Similarly, if G_1 wins, every process decides the value in $R[1]$. The object O_{nd} is used to determine the winner of the two groups. Processes $P_1 \ldots P_{n-1}$ perform the operation op(0) on O_{nd}. Then they set the register $R'[0]$ to inform process P_n that op(0) has been executed on O_{nd}. Process P_n, on the other hand, performs op(1) on O_{nd}, and then sets $R'[1]$ to inform processes in G_0 that op(1) has been executed. Processes then perform the **give-decision** operation. The return value determines the winning group. For this strategy to work correctly, the arguments of the **give-decision** operation must be such that the O_{nd} object does not get upset. We urge the reader to understand how the registers $R'[0..1]$ are used to ensure that O_{nd} does not get upset. Finally, if O_{nd} returns v, a process assumes that the group G_v won and decides the value in $R[v]$.

Lemma 5.1 *For $1 \leq n \leq k$, the implementation $\mathcal{I}^k_n$ in Figure 12 is a correct implementation of* **consensus** *from $\{T^k_{nd}, \texttt{register}\}$ for processes $P_1, P_2, \ldots, P_n$.*

Proof Sketch By induction. Assume that $\mathcal{I}^k_{n-1}$ is correct. Let $\mathcal{O}_n$ be a derived object of the implementation in Figure 12. Consider an execution E of the concurrent system $(P_1, P_2, \ldots, P_n; \mathcal{O}_n)$ in which every process P_i has invoked $\texttt{Apply}(P_i, \textit{propose } v_i, \mathcal{O}_n)$ exactly once, and executed it to completion. The key claim is that O_{nd} is not upset in E. This follows from the following simple observations:

1. op(1) is executed only once.
2. For $v \in \{0, 1\}$, op(v) is executed before executing **give-decision**$(v, -)$.
3. **give-decision** is executed no more than n times. Since $n \leq k$, **give-decision** is executed no more than k times.
4. Suppose op(v) is ahead of op($\overline{v}$). That is, the operations op(v) and then **give-decision**$(v, -)$ are completed before the first invocation of op($\overline{v}$). Then, the use of the registers $R'[0..1]$ in the implementation $\mathcal{I}^k_{n-1}$ guarantees that when a process invokes **give-decision**$(\overline{v}, \textit{other-ahead})$, the second parameter, namely, *other-ahead*, is *true*.

base objects of the implementation $\mathcal{I}_n^k$

$\mathcal{O}_{n-1}$: consensus object for $P_1, P_2, \ldots, P_{n-1}$, derived from $\mathcal{I}_{n-1}^k$
O_{nd}: $\mathtt{T}_{\mathtt{nd}}^{\mathtt{k}}$ object, initialized to the fresh state
$R[0..1]$: binary registers
$R'[0..1]$: boolean registers, initialized to *false*

local variables of process P_i

$d_i, winner_i \in \{0, 1\}$
$other\text{-}ahead_i$: boolean

Apply(P_i, *propose* v_i, $\mathcal{O}_n$) (for $1 \leq i \leq n-1$)

1. $d_i :=$ **Apply**$(P_i, propose\ v_i, \mathcal{O}_{n-1})$
2. $R[0] := d_i$
3. **Apply**$(P_i, \mathtt{op}(0), O_{nd})$
4. $R'[0] := true$
5. $other\text{-}ahead_i := R'[1]$
6. $winner_i :=$
   Apply$(P_i, \mathtt{give\text{-}decision}(0, other\text{-}ahead_i), O_{nd})$
7. return$(R[winner_i])$

Apply(P_n, *propose* v_n, $\mathcal{O}_n$)

$d_n := v_n$
$R[1] := d_n$
Apply$(P_n, \mathtt{op}(1), O_{nd})$
$R'[1] := true$
$other\text{-}ahead_n := R'[0]$
$winner_n :=$
Apply$(P_n, \mathtt{give\text{-}decision}(1, other\text{-}ahead_n), O_{nd})$
return$(R[winner_n])$

Figure 12: Implementing **consensus** from $\{\mathtt{T}_{\mathtt{nd}}^{\mathtt{k}}, \mathtt{register}\}$

5. Suppose no process completes the operation op(v) before some process invokes **give-decision**($\overline{v}$, *other-ahead*). Then the use of the registers $R'[0..1]$ in the implementation $\mathcal{I}^k_{n-1}$ guarantees that the second parameter of **give-decision**, namely, *other-ahead*, is *false*.

Since O_{nd} is not upset in E, by the specification of $\mathbf{T^k_{nd}}$, we have:

1. Every **give-decision** operation on O_{nd} returns the same binary response. Let $winner \in \{0, 1\}$ denote this response.

2. Some process P_j invokes op($winner$) before O_{nd} returns $winner$ for the first time to a **give-decision** operation.

From the implementation, it is clear that P_j writes the value d_j in $R[winner]$ before invoking op($winner$). Furthermore, once a value is written by a process into a register $R[0]$ or $R[1]$, the value of that register never subsequently changes. For $R[0]$, this follows from the agreement property of $\mathcal{O}^k_{n-1}$, and for $R[1]$, this follows from the fact that only P_n writes $R[1]$ and writes it only once.

The above implies that for all i, **Apply**(P_i, *propose* v_i, $\mathcal{O}_n$) returns d_j. Thus, $\mathcal{O}_n$ satisfies agreement. If $j = n$, then $d_j = d_n = v_n$, and thus, $\mathcal{O}_n$ satisfies validity. If $j \neq n$, by the validity of $\mathcal{O}_{n-1}$, $d_j \in \{v_1, v_2, \ldots, v_{n-1}\}$. Thus, $\mathcal{O}_n$ satisfies validity. It is obvious that the implementation is wait-free. This concludes the proof of correctness of $\mathcal{I}^k_n$. □

5.2 consensus from $\{\mathbf{T^k_{nd}}$, register$\}$ — an impossibility result

In this section, we prove that $\mathbf{T^k_{nd}}$ objects and registers do not suffice to implement a consensus object for $k + 1$ processes. This impossibility result follows from a straight forward bivalency argument. The intuition behind why this impossibility result holds for $k+1$ processes, but not for k processes, is as follows. As we have seen, a $\mathbf{T^k_{nd}}$ object supports two kinds of operations: op and **give-decision**. The operation op(i) does not return any useful information to the invoking process. This is due to the fact that the response of op(i) is always *ack*. The operation **give-decision** does return useful information, but only to the first k invocations of the operation. Thereafter, its response is non-deterministic and hence is not helpful. Thus, k processes may gain useful information from a $\mathbf{T^k_{nd}}$ object, but $k + 1$ processes cannot. We now proceed to prove the impossibility result.

Let $\mathbf{T^k_d}$ be a deterministic object type whose specification is defined by replacing every expression of the form *choose*(S) in Figure 11 by *min*(S).[7] Thus, $\mathbf{T^k_d}$ is a deterministic restriction of $\mathbf{T^k_{nd}}$. Hence, if a history of an object is linearizable with respect to $\mathbf{T^k_d}$, then it is *a fortiori* linearizable with respect to $\mathbf{T^k_{nd}}$. We prove below that $\mathbf{T^k_d}$ objects and registers do not suffice to implement a consensus object for $k + 1$ processes. This trivially implies that $\mathbf{T^k_{nd}}$ objects and registers cannot implement a consensus object for $k + 1$ processes.

[7] *min*(S) is the minimum element in set S.

As mentioned, the proof uses a simple bivalency argument. Since bivalency arguments are standard, our definitions and the proof are informal. A configuration C of a concurrent system is *v-valent* (for $v \in \{0,1\}$) if there is no execution from C in which $\overline{v}$ is decided by some process. In other words, once the system is in configuration C, no matter how processes are scheduled, no process decides $\overline{v}$. A configuration is *monovalent* if it is either 0-valent or 1-valent. A configuration is *bivalent* if it is not monovalent. If E is a finite execution of a system $\mathcal{S}$ started in configuration C, $E(C)$ denotes the configuration of $\mathcal{S}$ at the end of the execution E. For the purposes of this section, a *step* of a process P consists of invoking an operation on an object O, receiving the response from O, and making an appropriate change in its state.

Lemma 5.2 *For all $k \in \{2,3,\ldots\}$, there is no implementation of* **consensus** *from* $\{\mathtt{T}_{\mathtt{d}}^{\mathtt{k}}, \mathtt{register}\}$ *for $k+1$ processes.*

Proof Assume $\mathcal{I}(O_1, O_2, \ldots, O_n)$ is an implementation of **consensus** from $\{\mathtt{T}_{\mathtt{d}}^{\mathtt{k}}, \mathtt{register}\}$ for processes $P_1, P_2, \ldots, P_{k+1}$. Let $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots, O_n)$. Consider the concurrent system $\mathcal{S} = (P_1, P_2, \ldots, P_{k+1}; \mathcal{O})$. Let C_0 be the initial configuration of $\mathcal{S}$. Assume that in C_0, each process P_i is about to execute $\mathtt{Apply}(P_i, \mathit{propose}\ v_i, \mathcal{O})$. Furthermore, assume that there are l, m $(1 \leq l, m \leq k+1)$ such that $v_l = 0$ and $v_m = 1$.

When P_l runs by itself from C_0, the validity and wait-freedom of $\mathcal{O}$ require that P_l decide $v_l = 0$. Similarly, when P_m runs by itself from C_0, it decides $v_m = 0$. Thus, C_0 is bivalent. Let E be an execution from C_0 such that (1) $C_{crit} = E(C_0)$ is bivalent, and (2) For all P_i, if P_i takes a step from C_{crit}, the resulting configuration is monovalent. Let S_v be the set of processes whose step from C_{crit} results in a v-valent configuration. Since C_{crit} is bivalent, neither S_0 nor S_1 is empty. Furthermore, $S_0 \cap S_1 = \emptyset$ and $|S_0 \cup S_1| = k+1 \geq 3$ (since $k \geq 2$). Without loss of generality, assume that $|S_0| \geq 2$ and $|S_1| \geq 1$. In particular, let $S_0 = \{P_1^0, P_2^0, \ldots, P_r^0\}$ and $S_1 = \{P_1^1, P_2^1, \ldots, P_s^1\}$, where $r \geq 2$ and $s \geq 1$.

By a standard argument, the enabled step of every process in configuration C_{crit} must be on the same base object O of $\mathcal{O}$. Furthermore, again by a standard argument, O is not a register. Thus, the enabled step of every process in configuration C_{crit} is on O, an object of type $\mathtt{T}_{\mathtt{d}}^{\mathtt{k}}$. Let s_2^0 and s_1^1 denote the enabled steps of P_2^0 and P_1^1, respectively, in configuration C_{crit}. Consider the following scenarios $\mathtt{S}_0$ and $\mathtt{S}_1$, each starting from the configuration C_{crit}.

- In Scenario $\mathtt{S}_0$, P_2^0 takes the step s_2^0. Then, P_1^1 takes a step. Let D_0 be the resulting configuration. Clearly D_0 is a 0-valent configuration.
- In Scenario $\mathtt{S}_1$, P_1^1 takes the step s_1^1. Then, P_2^0 takes a step. Let D_1 be the resulting configuration. Clearly D_1 is a 1-valent configuration.

Processes P_2^0 and P_1^1 have to distinguish Scenario $\mathtt{S}_0$ from Scenario $\mathtt{S}_1$, since they must decide 0 in (every extension of) $\mathtt{S}_0$, and decide 1 in (every extension of) $\mathtt{S}_1$. Observe that unless the operation applied by P_2^0 (resp. P_1^1) in step s_2^0 (resp. s_1^1) is a **give-decision** operation, it must eventually apply a **give-decision** operation on O in order to distinguish $\mathtt{S}_0$ from $\mathtt{S}_1$. Thus, we extend Scenarios $\mathtt{S}_0$ and $\mathtt{S}_1$ as follows:

- If the operation applied by P_2^0 on O in step s_2^0 is not a **give-decision** operation, run P_2^0 (in both scenarios) exactly until P_2^0 completes a step in which it applies a **give-decision** operation on O.
- If the operation applied by P_1^1 on O in step s_1^1 is not a **give-decision** operation, run P_1^1 (in both scenarios) exactly until P_1^1 completes a step in which it applies a **give-decision** operation on O.

A process $P \in \{P_1, \ldots, P_{k+1}\} - \{P_1^0, P_2^0, P_1^1\}$ has to distinguish Scenario S_0 from Scenario S_1, since P must decide 0 in (every extension of) S_0, and decide 1 in (every extension of) S_1. Observe, however, that P cannot distinguish S_0 from S_1 until it applies a **give-decision** operation on O. Thus, we extend Scenarios S_0 and S_1 as follows:

- For each $P \in \{P_1, \ldots, P_{k+1}\} - \{P_1^0, P_2^0, P_1^1\}$, run P (in both scenarios) exactly until P completes a step in which it applies a **give-decision** operation on O.

We make the following observations: (1) The process P_1^0 is in the same state in Scenarios S_0 and S_1. (2) Every base object except O is in the same state in S_0 and S_1. (3) In both S_0 and S_1, a **give-decision** operation is applied on O at least k times (once by each process in $\{P_1, \ldots, P_{k+1}\} - \{P_1^0\}$, in the execution from C_{crit}). The second observation, together with the specification of T_d^k, implies that every subsequent **give-decision** operation on O returns 0 in either scenario. Extend Scenarios S_0 and S_1 by letting P_1^0 run by itself. By the above observations, P_1^0 cannot distinguish whether it is running in S_0 or S_1. Yet it must decide 0 in S_0 and 1 in S_1. This is impossible. Hence the lemma. □

Corollary 5.1 *For all* $k \in \{2, 3, \ldots\} \cup \{\infty\}$, $h_m^r(T_{nd}^k) = k$.

Proof Follows from Lemmas 5.1 and 5.2. □

5.3 h_m is not robust

In this section, we prove that $h_m(T_{nd}^k) = 1$. Thus, h_m is different from h_m^r and, hence, is not robust. We begin with a simple technical lemma that will be useful in proving $h_m(T_{nd}^k) = 1$. The lemma states that it is trivial to implement T_{nd}^k, initialized to any state different from the fresh state. In the following, let $\sigma[v]$ denote the value of state variable v in state σ.

Lemma 5.3 *Let* σ *be any state of* T_{nd}^k *different from the fresh state. Figure 13 is an implementation of* T_{nd}^k, *initialized to* σ, *from* $\emptyset$.[8]

Proof If σ is different from the fresh state, then it is easy to verify that $(\sigma[decision] \in \{0,1\}) \vee (\sigma[n_0] > 0) \vee (\sigma[n_1] > 0) \vee \sigma[upset]$. From this and the specification of T_{nd}^k, the correctness of the implementation is obvious. □.

[8] Thus, the implementation requires no base objects, not even registers.

op(i)	give-decision(i, b)

return(ack)	**if** $\sigma[decision] \in \{0,1\}$ **then**
	  return($\sigma[decision]$)
	else if ($\sigma[upset] \vee \sigma[n_0] > 0$) **then**
	  return(0)
	else return(1)

Figure 13: Implementing $\mathtt{T}^{\mathtt{k}}_{\mathtt{nd}}$, initialized to a non-fresh state σ

The following lemma states that it is impossible to implement a consensus object for two processes using just $\mathtt{T}^{\mathtt{k}}_{\mathtt{nd}}$ objects. Intuitively, $\mathtt{T}^{\mathtt{k}}_{\mathtt{nd}}$ objects are so weak that a process cannot use these objects to leave its "foot marks" behind. Thus, if a process P_0 runs first, and then a different process P_1 runs, P_1 does not realize that P_0 ran before it started. This can cause P_1 to decide a value which is not consistent with the decision of P_0. The proof below formalizes this argument. The details of the argument are subtle due to the non-determinism of the $\mathtt{T}^{\mathtt{k}}_{\mathtt{nd}}$ objects.

Lemma 5.4 *For all* $k \in \{2,3,\ldots\} \cup \{\infty\}$, $\mathtt{h_m}(\mathtt{T}^{\mathtt{k}}_{\mathtt{nd}}) = 1$.

Proof To prove this lemma, we must show that it is impossible to implement a consensus object for two processes using just $\mathtt{T}^{\mathtt{k}}_{\mathtt{nd}}$ objects. We show this by contradiction. Let $\mathcal{I}(O_1, O_2, \ldots, O_n)$ be an implementation of **consensus** from $\mathtt{T}^{\mathtt{k}}_{\mathtt{nd}}$ for processes P_0 and P_1, which is *resource optimal*: *i.e.*, if $\mathcal{I}'$ is another implementation of **consensus** from $\mathtt{T}^{\mathtt{k}}_{\mathtt{nd}}$ for two processes, then $\mathcal{I}'$ requires at least n base objects. From Lemma 5.3, it follows that every base object of $\mathcal{I}$ is initialized to the fresh state.

Consider a derived consensus object $\mathcal{O}$ of the implementation $\mathcal{I}$. Let $O_1, O_2, \ldots, O_n$ be the base objects of $\mathcal{O}$. In other words, $\mathcal{O} = \mathcal{I}(O_1, O_2, \ldots, O_n)$. In the following, we present two scenarios, $\mathtt{S}_0$ and $\mathtt{S}_1$, which are indistinguishable to P_1, but require P_1 to take different actions.

In Scenario $\mathtt{S}_0$, P_0 invokes $\mathtt{Apply}(P_0, propose\,0, \mathcal{O})$ and executes it to completion. (Execution to completion is possible since $\mathcal{I}$ is a wait-free implementation.) Assume that during the execution of $\mathtt{Apply}(P_0, propose\,0, \mathcal{O})$, every base object behaves like a $\mathtt{T}^{\mathtt{k}}_{\mathtt{d}}$ object. That is, the history of each base object in the execution of $\mathtt{Apply}(P_0, propose\,0, \mathcal{O})$ is linearizable with respect to $\mathtt{T}^{\mathtt{k}}_{\mathtt{d}}$. We will refer to this as Assumption **A1**. By the validity property of $\mathcal{O}$, $\mathtt{Apply}(P_0, propose\,0, \mathcal{O})$ returns 0. Let $\mathcal{S}$ be the set of base objects which are in the fresh state in Scenario $\mathtt{S}_0$ at the completion of $\mathtt{Apply}(P_0, propose\,0, \mathcal{O})$. Continue Scenario $\mathtt{S}_0$, and begin Scenario $\mathtt{S}_1$, by letting P_1 invoke $\mathtt{Apply}(P_1, propose\,1, \mathcal{O})$ and run by itself in either scenario. (See Figure 14 for a depiction of Scenarios $\mathtt{S}_0$ and $\mathtt{S}_1$.) Assume that each

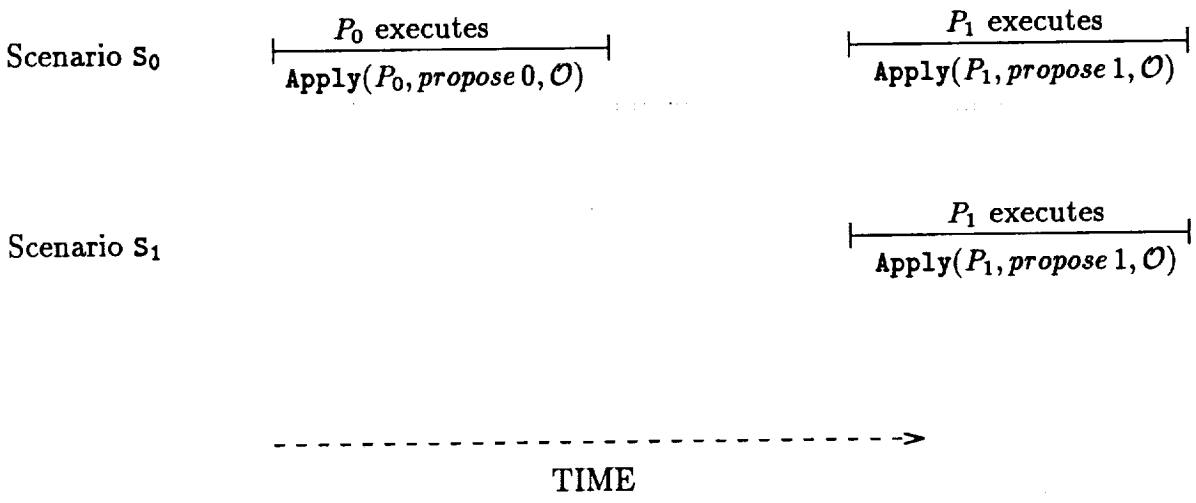

Figure 14: Scenarios S_0 and S_1

---

base object in $\mathcal{S}$ behaves deterministically, consistent with T_d^k, in both scenarios. We will refer to this as Assumption **A2**. We prove the following statement inductively: the base objects in $\{O_1, O_2, \ldots, O_n\} - \mathcal{S}$ can choose among the non-deterministic alternatives (when applicable) such that for all $i \geq 0$, P_1 cannot distinguish S_0 from S_1 in i steps. The base case for $i = 0$ is trivial. To prove the induction step, assume the hypothesis for $i \leq m$.

Consider the $(m+1)^{st}$ step. Let *oper* be the operation that P_1 performs in this step in Scenario S_0, and let O be the base object on which it performs *oper*. From the induction hypothesis and the fact that the implementation is deterministic, it follows that P_1 performs *oper* on O in its $(m+1)^{st}$ step in Scenario S_1 too.

Suppose *oper* $\in \{\text{op}(0), \text{op}(1)\}$. Then, the response is *ack* in either scenario. Thus, S_0 and S_1 remain indistinguishable to P_1 after $m+1$ steps. Hence the induction step.

Suppose that *oper* is **give-decision**$(-,-)$. We make a case analysis to prove the induction step.

Case 0. $O \in \mathcal{S}$

O is fresh in both S_0 and S_1 just before the invocation of Apply(P_1, *propose* 1, $\mathcal{O}$). For S_0, this follows from the definition of $\mathcal{S}$, and for S_1, from the fact that every base object is initialized to the fresh state. By Assumption **A2**, O behaves deterministically (consistent with T_d^k) in both scenarios. The above facts, together with induction hypothesis, guarantee that (i) O is in the same state in both scenarios at the end of m steps of P_1, and (ii) O returns the same response to *oper* in both scenarios. Thus, S_0 and S_1 remain indistinguishable to P_1 after $m+1$ steps. Hence the induction step.

Case 1. Case 0 does not apply and the following holds: In at least one of S_0 and S_1, O is upset in the first $m+1$ steps of P_1.

Let S_i be a scenario in which O is upset in the first $m+1$ steps of P_1. By the specification of T^k_{nd}, O is free to return 0 or 1 to *oper* in Scenario S_i. Suppose that O uses this freedom and returns the same response to *oper* in S_i as it does in $S_{\overline{i}}$. Then S_0 and S_1 remain indistinguishable to P_1 after $m+1$ steps. Hence the induction step.

Case 2. Neither Case 0 nor Case 1 applies. In other words, O is not fresh in S_0 just before the invocation of $\texttt{Apply}(P_1, propose\ 1, O)$ and, in both S_0 and S_1, O is not upset at the end of $m+1$ steps of P_1.

We prove the induction step by contradiction. Assume that it is not possible to keep Scenarios S_0 and S_1 indistinguishable to P_1 at the end of $m+1$ steps. We will refer to this as Assumption **A3**. We arrive at a contradiction after a series of claims. Let σ_0^k and σ_1^k denote the state of O at the end of k steps of P_1 in Scenarios S_0 and S_1 respectively.

C1. $\sigma_1^m[n_{gd}] = 0$. In other words, P_1 does not apply a **give-decision** operation on O in its first m steps.

Suppose that the claim is false. Let $k \leq m$ be the smallest integer such that $\sigma_1^k[n_{gd}] = 1$. That is, **give-decision** is executed on O for the first time by P_1 in its k^{th} step in Scenario S_1. Since O is not upset in S_1, this implies that $\sigma_1^k[decision] \in \{0,1\}$, and this value is returned by O in the k^{th} step of P_1 in S_1. By inductive hypothesis, the same value $\sigma_1^k[decision]$ is returned by O in the k^{th} step of P_1 even in S_0. Since O is not upset in S_0, this implies that $\sigma_0^k[decision] = \sigma_1^k[decision]$. Since *decision* is irrevocable, it follows that $\sigma_0^m[decision] = \sigma_0^k[decision] = \sigma_1^k[decision] = \sigma_1^m[decision] \in \{0,1\}$. Since O is not upset in either scenario, the responses $\sigma_0^m[decision]$ and $\sigma_1^m[decision]$ of O to *oper* in Scenarios S_0 and S_1, respectively, are identical. Thus, S_0 and S_1 remain indistinguishable to P_1 after $m+1$ steps. This contradicts Assumption **A3**.

C2. There is a $v \in \{0,1\}$ such that $\sigma_1^m[n_v] > 0$ and $\sigma_1^m[n_{\overline{v}}] = 0$. In other words, P_1 executes $\texttt{op}(v)$, but not $\texttt{op}(\overline{v})$ in its first m steps in S_1.

Suppose $\sigma_1^m[n_0] = \sigma_1^m[n_1] = 0$. Then, by the specification of T^k_{nd}, when P_1 applies $oper \equiv \texttt{give-decision}(-,-)$ in the $(m+1)^{st}$ step in S_1, it upsets O. This contradicts the case we are considering. Suppose $\sigma_1^m[n_0] > 0$ and $\sigma_1^m[n_1] > 0$. Since $\sigma_1^m[n_{gd}] = 0$ (by **C1**), by the specification of T^k_{nd}, O is free to return either 0 or 1 in S_1. Suppose that O uses this freedom and returns the same response to *oper* in S_1 as it does in S_0. Then S_0 and S_1 remain indistinguishable to P_1 after $m+1$ steps. This contradicts Assumption **A3**.

C3. P_1 executes $\texttt{op}(v)$ on O at least once in its first m steps in S_0.

Follows from **C2** and the induction hypothesis.

C4. $oper \equiv \texttt{give-decision}(v, false)$.

Suppose $oper \equiv \texttt{give-decision}(\overline{v}, -)$ or $oper \equiv \texttt{give-decision}(v, true)$. Since $\sigma_1^m[n_{\overline{v}}] = 0$ (by **C2**), O will be upset in S_1 when *oper* is invoked in the $(m+1)^{st}$ step. This contradicts the case we are considering.

C5. $\sigma_0^m[ahead[\overline{v}]] = false$.

Suppose $\sigma_0^m[ahead[\overline{v}]] = true$. Then, when P_1 executes $oper \equiv$ **give-decision**$(v, false)$ (guaranteed by **C4**) in its $(m+1)^{st}$ step in S_0, it upsets O. This contradicts the case we are considering.

C6. $v = 1$ implies $\sigma_0^0[n_{gd}] = 0$. In other words, if $v = 1$, then P_0 never executed a **give-decision** operation on O in S_0.

Suppose $v = 1$ and P_0 executed **give-decision**$(1, -)$ on O in S_0. Since O is not upset in S_0, it follows that P_0 executed op(1) on O before executing **give-decision**$(1, -)$. By **C3** and the assumption that $v = 1$, P_1 executed op(1) in S_0. Thus op(1) was executed at least twice on O in S_0. By the specification of T_{nd}^k, O would be upset in S_0. This contradicts the case we are considering.

Suppose $v = 1$ and P_0 executed **give-decision**$(0, -)$ on O in S_0. Since O is not upset in S_0, it follows that P_0 executed op(0) on O before executing **give-decision**$(0, -)$. By **C5** and the assumption that $v = 1$, $\sigma_0^m[ahead[0]] = false$. This implies that P_0 executed op(1) on O before executing **give-decision**$(0, -)$. By **C3** and the assumption that $v = 1$, P_1 executed op(1) in S_0. Thus op(1) was executed at least twice on O in S_0. By the specification of T_{nd}^k, O would be upset in S_0. This contradicts the case we are considering.

C7. $v = 0$.

Suppose $v = 1$. Then, we can infer: (1) $\sigma_1^m[n_{gd}] = 0$ (by **C1**), (2) $\sigma_0^m[n_{gd}] = 0$ (by **C1**, induction hypothesis, and **C6**), (3) $\sigma_1^m[n_1] > 0$ (by **C2**), (4) $\sigma_0^m[n_1] > 0$ (by **C3**). These four facts, together with the specification of T_{nd}^k, imply that O is free to return 0 to *oper* in both S_0 and S_1. Suppose that O does this. Then S_0 and S_1 remain indistinguishable to P_1 after $m + 1$ steps. This contradicts Assumption **A3**.

C8. O returns 0 to *oper* (in the $(m+1)^{st}$ step of P_1) in Scenario S_1.

C2 and **C6** imply that $\sigma_1^m[n_0] > 0$ and $\sigma_1^m[n_1] = 0$. Further, by the case we are considering, O is not upset in the first $m + 1$ steps of P_1 in Scenario S_1. The above facts imply that the only legal value that O can return to *oper* is 0.

C9. If P_0 executed **give-decision**$(1, -)$ on O (in S_0), it did so only after executing op(0) on O.

Suppose P_0 executed **give-decision**$(1, -)$ on O (in S_0). Since O is not upset in S_0, this implies that P_0 executed op(1) on O before executing **give-decision**$(1, -)$. If P_0 did not execute op(0) before executing **give-decision**$(1, -)$, then the execution of **give-decision**$(0, -)$ would set *ahead*[1] to *true*. This, together with the fact that *ahead*[1] is stable, implies that $\sigma_0^m[ahead[1]] = true$. This contradicts the conjunction of **C5** and **C7**.

C10. Every execution of the operation **give-decision**$(-, -)$ on O by P_0 in Scenario S_0 returns the response 0.

Consider the earliest execution e of **give-decision**$(w, -)$ on O by P_0 in S_0. If $w = 1$, **C9** implies that P_0 executes op(0) before e. If $w = 0$, the fact that O is not upset in S_0 implies that P_0 executes op(0) before e. Thus, we conclude that

P_0 executes op(0) before e. This, together with Assumption **A1**, implies that e returns 0. From this and the fact that O is not upset in S_0, it follows that every execution of **give-decision**$(-,-)$ on O in S_0 returns the response 0.

C11. P_0 never executes **give-decision**$(-,-)$ on O (in S_0).
Suppose that the claim is false. Then, from **C10** and the fact that O is not upset in S_0, it follows that O returns 0 to *oper* in the $(m+1)^{st}$ step of P_1 in Scenario S_0. Thus, by **C8**, S_0 and S_1 remain indistinguishable to P_1 after $m+1$ steps. This contradicts Assumption **A3**.

We have: (1) $\sigma_1^m[n_0] > 0$. This follows from **C3** and **C7**. (2) $\sigma_0^m[n_0] > 0$. This follows from (1) and induction hypothesis. (3) $\sigma_0^m[n_{gd}] = 0$. This follows from **C1**, induction hypothesis, and **C11**. From (2), (3), and the specification of $\mathbf{T}_{\mathbf{nd}}^{\mathbf{k}}$, it is clear that O is free to return 0 to *oper* (in the $(m+1)^{st}$ step of P_1) in Scenario S_0. Suppose that it does. Then, by **C8**, S_0 and S_1 remain indistinguishable to P_1 after $m+1$ steps. This contradicts Assumption **A3**. Hence the induction step.

This completes the proof of the induction step.

Since $\mathcal{I}$ is a wait-free implementation, Apply$(P_1, propose\,1, \mathcal{O})$ terminates in S_0 after a finite number of steps, returning some value $val \in \{0,1\}$. Since S_1 is indistinguishable to P_1 from S_0, Apply$(P_1, propose\,1, \mathcal{O})$ terminates in S_1 after the same number of steps, also returning val. If $val = 0$, validity of consensus is violated in S_1. If $val = 1$, agreement of consensus is violated in S_0. Thus, $\mathcal{I}$ is not a correct implementation, a contradiction. □

Theorem 5.1 $\mathbf{h_m}$ *is neither tight nor robust.*

Proof Follows from Proposition 3.6, Corollary 5.1, and Lemma 5.4. □

6 Conclusion

It is well known that shared primitives, depending on their type, vary widely in their ability to support inter-process synchronization. Recent research focussed on analyzing the power of individual primitives. In this paper, we ask whether, from our understanding of the power of the individual primitives, we can infer the power of a set of primitives. For instance, is it impossible to implement a universal primitive from non-universal primitives? The answer is not clear. It is conceivable that clever protocols for such implementations exist. Besides being of theoretical interest, these issues have implications to multi-processor architectures. To make a systematic study of these issues possible, we define the property of robustness for wait-free hierarchies. Contrary to popular belief, we show that Herlihy's wait-free hierarchy is not robust. We also show that some natural variants of Herlihy's hierarchy are also not robust. This raises the obvious question of whether there is a non-trivial robust wait-free hierarchy at all. We do not know the answer yet. However, we observe that such a hierarchy, if it exists, is either $\mathbf{h_m^r}$ or some coarsening of it. Thus, further research on the structure

of h_m^r is essential to resolving this open question. As explained in the paper, the answer to this question, regardless of whether it is affirmative or negative, has useful implications. We close with the conjecture that h_m^r is not robust.

Acknowledgement

I had innumerable discussions with my advisor Sam Toueg on this subject. They were very helpful in crystalizing my ideas, and in discovering some of these results. The "swap object" that Jon Kleinberg and Sendhil Mullainathan showed me helped me discover T_{sp}. I am grateful to Sam, Jon, and Sendhil for sharing their insights with me. I thank Tushar Chandra for reading parts of this paper and providing helpful comments. Aparna helped me with typing. Little Sucharita never complained the travel between home and school, no matter what time of the night and how cold.

References

[AGTV92] Y. Afek, E. Gafni, J. Tromp, and P. Vitanyi. Wait-free test&set. In *Proceedings of the 6th Workshop on Distributed Algorithms, Haifa, Israel*, November 1992. (Appeared in Lecture Notes in Computer Science, Springer-Verlag, No: 647).

[AR92] Y. Aumann and M.O. Rabin. Clock construction in fully asynchronous parallel systems and pram simulation. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, October 1992.

[CHP71] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, 1971.

[CIL87] B. Chor, A. Israeli, and M. Li. On processor coordination using asynchronous hardware. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 86–97, August 1987.

[Her88] M.P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, 1988.

[Her91a] M.P. Herlihy. Impossibility results for asynchronous pram. In *Proceedings of the 3rd ACM Symposium on Parallel Architectures and Algorithms*, July 1991.

[Her91b] M.P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[HW90] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[LAA87] M.C. Loui and Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research*, 4:163–183, 1987.

[Lam77] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

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