On the robustness of the $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ hierarchy*

Tushar Chandra† Vassos Hadzilacos‡ Prasad Jayanti§ Sam Toueg¶

January 12, 1998

Abstract

The hierarchy $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ was proposed to classify object types according to their ability to support wait-free implementations of other object types. Recently there has been much research on the question of whether $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ is a robust hierarchy. Roughly speaking, a hierarchy is robust if each type is "stronger" than any combination of lower level types. For certain models of binding between processes and objects, we prove here that $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ is not a robust hierarchy.

1 Introduction

1.1 Background

Our study concerns concurrent systems in which asynchronous processes communicate via typed shared objects. A process applies operations on objects and receives a response for each operation it applies. An object's type specifies the possible states of the object, the operations that may be applied to it, and the legitimate state transitions and responses when operations are applied one after another, without overlap. In most systems, simple types of shared objects, such as registers and test&set objects, are supported in hardware, but more complex types of objects, such as queues, stacks, and sets, are not. Thus, complex shared objects must be implemented in software. It is therefore important to be able to answer the following *implementability* question: Given an set of types $\mathcal{B}$ directly supported by a system, and a new type T, is it possible to implement an object of type T using only objects that belong to types in $\mathcal{B}$?

An implementation of an object type is *linearisable* if every operation of the implemented object type appears to take effect instantaneously at some point between its invocation and its response [HW90]. 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

*The result in this paper appeared without proof in *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, 1994.

†IBM T. J. Watson, tushar@watson.ibm.com

‡University of Toronto, vassos@cs.utoronto.ca, partially supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

§Dartmouth College, prasad@cs.dartmouth.edu, supported by NSF RIA grant CCR-9410421 and Dartmouth College Startup grant.

¶Cornell University, sam@cs.cornell.edu, supported by NSF grant CCR-9102231.

of the remaining processes [Her91]. In this paper we are interested exclusively in linearisable and wait-free implementations.

A type T is called *n-universal* if, for all types T', an object of type T', shared by n processes, can be implemented using only registers and objects of type T [Her91]. It turns out that, for all n, there is a type that is n-universal, but is not $(n+1)$-universal [JT92]. Thus, types can differ in their ability to support implementations of other types. This observation has motivated research on classifying types into a hierarchy [Her91, Jay93]. A hierarchy assigns a level to each type, where a level is a positive integer or ∞. It is intended that types at higher levels are more powerful, in their ability to implement types, than those at lower levels.

The type hierarchy that has received the most attention is the *consensus hierarchy*, also known as $\mathtt{h}_\mathtt{m}^\mathtt{r}$ [Her91, Jay93]. In this hierarchy, a type T is mapped to level n if and only if n is the maximum integer such that T is n-universal; if no such maximum exists, then T is mapped to ∞. Knowing the levels of two types in this hierarchy allows us to compare their relative power: the one at a higher level is stronger than the one at a lower level, in the sense that the former can be used to implement the latter. But to answer the implementability question described earlier, we must be able to determine, not whether a *single* type is stronger than another, but whether a *set* of types $\mathcal{B}$ is stronger than a new type T. Does knowing the levels of the types in $\mathcal{B}$ and the level of T in the consensus hierarchy help us answer whether the former is stronger than (can implement) the latter? Jayanti observed that the answer to this question is affirmative if $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is *robust*: no object of a type at level n shared by n processes, can be implemented by *any* combination of types at levels strictly below n [Jay93]. In the same paper, Jayanti investigated $\mathtt{h}_\mathtt{m}^\mathtt{r}$ and some related hierarchies of types. He showed that the other hierarchies are not robust, but left open whether $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is.

It turns out that whether $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is robust depends on how types are modeled. The characteristics of types that influence this outcome are whether types are deterministic, whether they are oblivious, and how processes bind to objects. We informally explain these issues next; a more precise treatment is given in Section 2. A type is *deterministic* if the effect of each operation on the state and the response it returns are uniquely determined by the current state of the type. Each object type specifies a number of *ports* at which operations can be applied. If each operation has the same effect regarless of the port at which it is applied, the type is *oblivious*. Regarding the binding of processes to ports, each process may be allowed to bind to at most one port or to several ports of any object. Furthermore, the process-to-port binding may be permanent (for the duration of the process' life) or transient (for the duration of a single operation).

1.2 Statement of the result and related work

In this paper we prove that if types can be nondeterministic and nonoblivious, and each process is required to bind permanently to at most one port of an object, then $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is not robust.

Concurrently with the first appearance of our result [CHJT94], there appeared two papers proving the robustness of $\mathtt{h}_\mathtt{m}^\mathtt{r}$ in certain models [BGA94, PBN94]. Specifically, Borowsky, Gafni, and Afek show that if types must be deterministic and oblivious, each process may bind to several ports of the same object, and the process-to-port binding is transient, then $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is robust [BGA94]. Peterson, Bazzi, and Neiger show that if types must be deterministic, each process may bind to several ports of the same object, but the process-to-port binding is permanent, then $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is robust [PBN94].

More nonrobustness results have appeared recently. Moran and Rappoport [MR96] strengthen our result by proving that $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ is not robust even if types must be deterministic, as long as each process must be permanently bound to at most one port of an object. Lo and Hadzilacos [LH97] prove that $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ is not robust if types can be nondeterministic, regardless of whether they are oblivious and of how processes may bind to ports. Schenk proves a result similar to the one in [LH97], but his result requires infinite nondeterminism and applies only for a stronger definition of wait-free implementation [Sch96].

1.3 Overview of the proof

In the *consensus problem for n processes*, each process is initially given an input from the natural numbers. The problem requires each process to irrevocably decide a value so that no two processes decide different values and the value decided is the input of some process. Herlihy has shown that this problem plays a central role in wait-free implementations: if consensus for n processes can be solved using only objects of type T and registers, then T is n-universal [Her91]. Motivated by this fact, we define the n-`consensus` type. An object of this type allows a maximum of n processes to solve the consensus problem (see Section 2.1 for a precise specification of this type).

The crux of our proof of non-robustness of $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ lies in the definition of a new type called `booster`. The specification of `booster` is crafted so as to satisfy the following two properties:

(i) the type is weak enough that it is not possible to solve consensus for two processes using only registers and booster objects, and

(ii) the type is strong enough that it is possible to solve consensus for three processes using only registers, booster objects, and 2-consensus objects.

The intuition for how we obtain these properties may be described as follows. A booster object expects every pair of processes to show evidence that they have independent means for solving consensus between them. If such evidence is presented, the object provides "useful" responses to processes that help them reach consensus. On the other hand, if processes fail to present such evidence, then the object ensures that its responses cannot be of any use to processes in reaching consensus. Thus, if 2-consensus objects, which help pairs of processes solve consensus, are available, then it becomes possible to additionally use a booster object and solve consensus for three processes. On the other hand, if we have only booster objects and registers, with no means to solve consensus between pairs of processes, then the booster objects will be of no use. Properties (i) and (ii), together with the fact that it is not possible to solve consensus for three processes using only 2-consensus objects and registers, imply that $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ is not robust.

1.4 Organisation of the paper

The rest of this paper is organized as follows: In Section 2 we describe the model. In Section 3 we present the specification of type `booster`. In Section 4 we show that booster objects, together with 2-consensus objects and registers, can implement 3-consensus objects, while in Section 5 we show that booster objects and registers alone cannot implement 3-consensus objects. These two results imply that $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ is not robust.

2 Model and Definitions

2.1 Types

Let $\mathbb{N}$ denote the set of natural numbers, and, for any $n \in \mathbb{N}$, $\mathbb{N}_n = \{0, 1, \ldots, n-1\}$. A *type* is a tuple (Q, OP, n, RES, δ), where Q is a set of states, OP is a set of operations, n is a positive integer denoting the number of ports, RES is a set of responses, and $\delta \subseteq Q \times OP \times \mathbb{N}_n \times RES \times Q$ is a transition relation, known as the *sequential specification* of the type. Intuitively, if $(s, op, p, res, s') \in \delta$, the application of operation op at port p, when the current state is s, can change the state to s' and return the response res. The relation δ is required to satisfy the following property: For all $\sigma \in Q$, $op \in OP$, and $p \in \mathbb{N}_n$, there is at least one pair (σ', res) such that $(\sigma, op, p, \sigma', res) \in \delta$. This ensures that it is legitimate to apply any operation at any port in any state.

A type is *n-ported* if the number of ports associated with it is n. A type is *deterministic* if, for all $s \in Q$, $op \in OP$ and $p \in \mathbb{N}_n$, there is at most one pair (res, s') such that $(s, op, p, res, s') \in \delta$. Thus, for deterministic types, δ can be regarded as a function $\delta : Q \times OP \times \mathbb{N}_n \rightarrow RES \times Q$. A type is *oblivious* if the effect of an operation does not depend on the port at which it is applied; more precisely, for all $s, s' \in Q$, $op \in OP$, $p, q \in \mathbb{N}_n$ and $res \in RES$, $(s, op, p, res, s') \in \delta$ if and only if $(s, op, q, res, s') \in \delta$. Thus, for oblivious types, the sequential specification can be given without reference to ports. In particular, for deterministic oblivious types, δ can be regarded as a function $\delta : Q \times OP \rightarrow RES \times Q$.

An example of a deterministic nonoblivious 2-ported type is the `1-reader 1-writer` V`-valued register` for which $Q = V$, $OP = \{read\} \cup \{write(v) : v \in V\}$, $n = 2$, $RES = \{ack\} \cup V$, and the sequential specification δ is as follows: For any $v, v' \in V$, $\delta(v, read, 1) = (v, v)$ (*read* at port 1 returns the state without modifying it), $\delta(v, write(v'), 1) = (ack, v)$ (*write* at port 1 has no effect), $\delta(v, read, 2) = (ack, v)$ (*read* at port 2 has no effect on the state and returns no useful information), and $\delta(v, write(v'), 2) = (ack, v')$ ($write(v')$ at port 2 changes the state to v').

Next we describe two deterministic oblivious types used in this paper, `register` and `consensus`. The type `register` for set of values V, has states $Q = V$, operations $OP = \{read\} \cup \{write(v) : v \in V\}$, responses $RES = \{ack\} \cup V$, and the following sequential specification: for any $v, v' \in V$, $\delta(read, v) = (v, v)$, and $\delta(write(v'), v) = (ack, v')$. Informally, the behaviour of the `consensus` type is as follows: For each $v \in \mathbb{N}$, there is an operation $propose(v)$. Assuming that the type is initially in a special state, denoted $\perp$, and the first operation applied to it is $propose(v)$, the type will enter state v and return v as a response. Any subsequent $propose(v')$ operation (for any $v' \in \mathbb{N}$) will return the current state, without affecting it. Thus, n processes $P_1, P_2, \ldots, P_n$ can use an n-ported object of this type to solve consensus among them: The object is initialised to state $\perp$, each process P_i applies the operation $propose(v_i)$ to the object, where v_i is the input of P_i, and decides whatever value it receives as a response. Formally, type `consensus` has states $Q = \{\perp\} \cup \mathbb{N}$, operations $OP = \{propose(v) : v \in \mathbb{N}\}$, responses $RES = \mathbb{N}$, and the following sequential specification: for each $v, v' \in \mathbb{N}$, $\delta(\perp, propose(v)) = (v, v)$; and $\delta(v, propose(v')) = (v, v)$. We write n`-consensus` to denote the n-ported `consensus` type.

The notion of type (in the sense presented above) was first described by Herlihy and Wing [HW90]. Their definition encompasses only oblivious types. The distinction between oblivious and nonoblivious types was made explicit by Kleinberg and Mullainathan [KM93]. The definitions given above are adapted from Bazzi, Neiger and Peterson [BNP94], Borowsky, Gafni and Afek [BGA94], and Jayanti [Jay95].

2.2 Processes and Objects

A *concurrent system* consists of processes and objects. Processes, objects, and the concurrent system can all be formally modeled using I/O automata [LT88], as described by Herlihy [Her91]. Our approach here is more informal.

An *object* is an instance of a type. It can be defined as an automaton of the same form as the type to which it belongs, except that each state and port is labeled with the object's unique name to distinguish them from the states and ports of other objects of the same type. A *process* is a program that accesses objects by invoking operations at ports of the object. After invoking an operation, the process waits until a response is returned by the object at that port. The way in which a process determines the port at which to access an object depends on the *binding scheme*. There are two properties that characterise such a scheme:

- The maximum number of ports of a given object to which a process may be bound at any one time. This could be one (*single-port* binding), or the number of ports of the object (*multiple-port* binding).
- The duration of the binding between a process and a port. This could be the entire lifetime of the process (*permanent* binding) or the duration of a single operation invocation (*transient* binding). With transient binding, a particular port may be used (at different times) by different processes.

Furthermore, if the process-to-port binding is single-port and permanent, we can distinguish between two binding schemes, depending on when the binding is determined. In *static* binding, the (unique) port to which a process is (permanently) bound is determined *a priori* and is the same in all executions involving this process. In *dynamic* binding, the port to which a process is bound may depend on the process' state, and may be different in different executions of the process.

Thus, there are five binding schemes. In all these schemes it is intended that there should never be two operations invoked simultaneously at the same port. Different binding schemes offer different degrees of flexibility to the implementor of a type. Least flexibility is afforded by single-port, permanent, static binding, and maximum flexibility by multiple-port, transient binding.

The plethora of binding schemes and object models can make the interpretation of results difficult. It is comforting to know that with certain classes of object models, the choice of binding scheme is immaterial. Specifically, in the case of oblivious types, the number of ports to which a process may bind is irrelevant: Since any operation has the same effect regardless of the port at which it is applied, the extra flexibility of allowing a process to bind to more than one port of an object serves no useful purpose. If, in addition to the types being oblivious, the number of ports associated with each type is at least as large as the number of processes in the system, then the details of the binding scheme are not an issue. In this case we can statically and permanently bind each process to one port of each object. The implementor of a type cannot get any mileage out of the additional flexibility of more complex binding schemes.

In this paper we consider an object model that admits nonoblivious (and nondeterministic) types. In this case the details of the binding scheme are relevant. Our results apply specifically to the single-port, permanent binding scheme. For ease of reference, we denote the static and dynamic versions of this scheme as $\mathcal{S}$ and $\mathcal{D}$, respectively.

2.3 Implementation

In this subsection we define what it means for a set of object types to implement another object type. A key concept that enters into this definition is that of linearisability [HW90]. We begin with some motivation and intuition for this concept. The sequential specification of a type describes the behaviour of an object of that type when operations are applied to the object one-at-a-time, with no overlap. In a concurrent system, however, different processes may invoke operations to different ports of the object at the same time. Linearisability is a property that describes the behaviour of objects in the presence of concurrent operations. It stipulates that the object should behave as if each operation were applied alone at some point between its invocation and its response.

Let O be an object of type T. An *access to* O, denoted as a triple $\langle o, p, r\rangle$, represents an invocation of operation o at port p of T that returns response r. Let $\sigma = \langle o_1, p_1, r_1\rangle, \ldots, \langle o_k, p_k, r_k\rangle$ be a sequence of accesses to O, and s be a state of T. We say that σ is *consistent with* T *initialised To* s if there are states $s_1, s_2, \ldots, s_{k+1}$ of T so that $s_1 = s$ and for each i, $1 \leq i \leq k$, $(s_i, o_i, p_i, r_i, s_{i+1})$ belongs to the transition relation of T. In other words, $r_1, \ldots, r_k$ (in that order) are legitimate responses of an object of type T, initialised to state s, to operations $o_1, \ldots, o_k$ applied (in that order) at ports $p_1, \ldots, p_k$.

Let $\mathcal{O}$ be a set of objects, called *base objects*. An *implementation of an object of type* T *initialised to state* s *from* $\mathcal{O}$ specifies two elements:

- For each base object in $\mathcal{O}$, an *initial state*. This is one of the states of the type to which the base object belongs.
- For each operation *op* and port p of T, a *procedure*. This is a program written in terms of operations to (some of the) base objects. This procedure specifies how to simulate *op* at p in terms of operations on the base objects.

An implementation must additionally satisfy a safety property (linearisability) and a liveness property (wait-freedom) defined below. Suppose all the base objects are initialised as specified by the implementation. Consider a run $\mathcal{R}$ of the implementation's procedures, containing perhaps repeated executions of some procedures. Recall that the procedures of the implementation are programs written in terms of operations to the base objects. Thus, each procedure execution in $\mathcal{R}$ results in accesses to the base objects. We assume that these accesses occur atomically, with no overlap. Thus, the run $\mathcal{R}$ may be viewed as a *sequence* of accesses to base objects, disjoint subsequences of which correspond to the procedure executions in $\mathcal{R}$. In contrast to base-object accesses, which are atomic, the executions of the procedures in $\mathcal{R}$ may overlap. More precisely, let e and e' be procedure executions in $\mathcal{R}$ (i.e., subsequences of $\mathcal{R}$). We say that e *precedes* e' (in $\mathcal{R}$) if the last base-object access of e occurs before the first base-object access of e' (in $\mathcal{R}$). Two procedure executions are *concurrent* (in $\mathcal{R}$) if neither precedes the other. We are interested only in runs in which there are no concurrent procedure executions at the *same* port; procedure executions at *different* ports, however, may be concurrent.

Recall that a procedure specifies how to simulate an operation o at some port p of the implemented object. The execution of this procedure returns some response r. Thus, corresponding to each procedure execution in $\mathcal{R}$ there is an access $\langle o, p, r\rangle$ to the implemented object.

A run $\mathcal{R}$ of an implementation of object O is *linearisable with respect to type* T *initialised to* s if there is a sequence σ of the accesses to O that correspond to the procedure executions in $\mathcal{R}$

so that (a) σ is consistent with T initialised to s; and (b) σ respects the order of nonconcurrent procedure executions in $\mathcal{R}$ — i.e., if procedure execution e precedes procedure execution e' in $\mathcal{R}$, then the access that corresponds to e precedes the access that corresponds to e' in σ.

Definition 1 An implementation of an object of type T initialised to state s for k processes with binding scheme $\mathcal{M}$ is

- *linearisable* if every run of the implementation produced by k processes according to the rules of binding scheme $\mathcal{M}$ is linearisable with respect to T initialised to s.
- *wait-free* if there is no run of the implementation produced by k processes according to the rules of binding scheme $\mathcal{M}$ in which some procedure execution is infinite.

Note the role of the number of processes k and the binding scheme $\mathcal{M}$ in the preceding definition. These two elements affect the set of runs of the implementation that we are interested in. For example, a run that contains procedure executions at *three* ports cannot be produced by *two* processes with a single-port, permanent binding scheme. In contrast, such a run can be produced by two processes with a multi-port binding scheme. From now on "implementation" is short for "linearisable, wait-free implementation".

Definition 2 Let $\mathcal{B}$ be a set of types and T be a type. We say that $\mathcal{B}$ *implements* T *for* k *processes with binding scheme* $\mathcal{M}$ if for each state s of T, there is an implementation of an object of type T initialised to s from objects whose types belong to $\mathcal{B}$ for k processes with binding $\mathcal{M}$.

Definition 3 A set $\mathcal{B}$ of types is *n-universal* if $\mathcal{B} \cup \{\texttt{register}\}$ implements *any* n-ported type T for n processes.

2.4 Universality of consensus and the consensus hierarchy

Recall that a process is a program that accesses a set of objects. An *algorithm that uses a set of objects* $\mathcal{O}$ is a collection of processes, each of which accesses only (some subset of) the objects in $\mathcal{O}$. In a (wait-free) *consensus algorithm*, each process has an input value, drawn from $\mathbb{N}$, and may irrevocably decide a value, subject to the requirement that in each run of the algorithm the following properties hold:

Agreement No two processes decide different values.

Validity If a process decides v then v is some process' input.

Termination No process takes infinitely many steps without deciding.

We say that a set of types $\mathcal{B}$ *solves consensus among n processes* if there is an n-process consensus algorithm that uses a set of objects that belong only to types in $\mathcal{B}$. Herlihy established a fundamental link between the ability of a set of types to implement consensus, and their ability to implement other types.

Theorem 1 *(Herlihy [Her91]) Let $\mathcal{B}$ be any set of types that includes* `register`. *$\mathcal{B}$ solves consensus among n processes if and only if $\mathcal{B}$ is n-universal.*

The definition of "$\mathcal{B}$ solves consensus among n processes", like that of "$\mathcal{B}$ implements T for n processes", is relative to a binding scheme which constraints how processes can access objects. As it turns out, Theorem 1 holds regardless of the binding scheme — as long as the same binding scheme is used for solving consensus as in implementing a type.

Definition 4 The hierarchy $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is a function that maps each set of types $\mathcal{B}$ to a positive integer or to ∞. $\mathtt{h}_\mathtt{m}^\mathtt{r}(\mathcal{B})$ is the maximum n such that $\mathcal{B}$ is n-universal; if there is no such maximum, $\mathtt{h}_\mathtt{m}^\mathtt{r}(\mathcal{B})$ is ∞. If T is a type, we write $\mathtt{h}_\mathtt{m}^\mathtt{r}(T)$ to mean $\mathtt{h}_\mathtt{m}^\mathtt{r}(\{T\})$; $\mathtt{h}_\mathtt{m}^\mathtt{r}(T)$ is called the *consensus number* of T.

Thus, if $\mathtt{h}_\mathtt{m}^\mathtt{r}(\mathcal{B}) = n$, we can implement *any* n-ported type for n processes using only registers and objects whose types are in $\mathcal{B}$. Furthermore, there is at least one $(n+1)$-ported type that cannot be implemented for $n+1$ processes using only such objects. In this sense $\mathtt{h}_\mathtt{m}^\mathtt{r}(\mathcal{B})$ characterises the power of a set of types to support implementations of other types.

Since the relation "implements" (between a set of types $\mathcal{B}$ and a type T) depends on the binding scheme, the above definition is relative to a binding scheme. Unless otherwise noted, in this paper we assume a single-port, permanent binding scheme. When relevant, we specifically distinguish between the static and dynamic versions of this scheme, $\mathcal{S}$ and $\mathcal{D}$. From Definition 4 and Theorem 1 we get:

Corollary 1 *Let $\mathcal{B}$ be any set of types that includes* `register`. *$\mathcal{B}$ solves consensus among n processes if and only if $\mathtt{h}_\mathtt{m}^\mathtt{r}(\mathcal{B}) \geq n$.*

As described in Section 2.1, type n-`consensus`, together with registers, can solve consensus among n processes. On the other hand, using standard techniques (specifically, the bivalence argument [FLP85, Her91]) it can be shown that there is no algorithm that solves consensus among $n+1$ processes using only registers and objects that belong to deterministic types that have at most n ports. In particular, there is no algorithm that solves consensus among $n+1$ processes using only registers and n-consensus objects. These observations and Theorem 1 imply:

Theorem 2 *Type n-`consensus` is n-universal but not $(n+1)$-universal.*

From this and Definition 4 we get:

Corollary 2 *For any positive integer n, $\mathtt{h}_\mathtt{m}^\mathtt{r}(n\text{-}\mathtt{consensus}) = n$.*

2.5 Robustness of the consensus hierarchy

Definition 5 The hierarchy $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is *robust* if, for any set of types $\mathcal{B}$, $\mathtt{h}_\mathtt{m}^\mathtt{r}(\mathcal{B}) = \sup\{\mathtt{h}_\mathtt{m}^\mathtt{r}(T) : T \in \mathcal{B}\}$.

Thus, the hierarchy $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is robust if any set of types is (collectively) as strong as the strongest of its elements. Put in a different way, $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is robust if it is not possible to implement an object of a type at level n for n processes using any combination of objects that belong to types at levels strictly below n.

If $\mathtt{h}_\mathtt{m}^\mathtt{r}$ is robust, the consenus numbers of types hold the key to implementability question: Given a set $\mathcal{B}$ of base types, and an n-ported target type T, we can implement an object of type T for n processes using only objects of types in $\mathcal{B}$ if and only if the maximum consensus number of any type in $\mathcal{B}$ is at least n. Furthermore, if $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ is robust we can argue about the power of a system that supports a given set $\mathcal{B}$ of types in its hardware by reasoning about the power of each type in $\mathcal{B}$ individually. If, on the other hand, $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$ is not robust to determine the power of $\mathcal{B}$ we may have to take into account the interactions between types in $\mathcal{B}$.

3 Specification of the type `booster`

As mentioned in Section 1.3, our goal is to define a type an object of which allows three processes to solve consensus provided that each process is able to "provide evidence" to the object that it has access to external means by which it can solve consensus between itself and each of the remaining two processes. In addition, we want objects of this type to be weak enough that they (together with registers) cannot be used to solve consensus between two processes. Thus, objects of this type by themselves are too weak to solve consensus between two processes, but in the presence of `2-consensus` objects they become strong enough to solve consensus for three processes. In this section, we present a nondeterministic, nonoblivious type named `booster` with these properties.

The type `booster` is specified formally in Figure 1. The following informal description should be helpful in understanding the specification. A booster object has three ports, numbered 0, 1 and 2. Each port supports two operations — *enroll* and *tell-me-the-winner* (the latter hereafter abbreviated as *tmtw*.) The *enroll* operation takes no arguments and always returns the response *ack*. The *tmtw* operation takes two port numbers as arguments and returns a port number as the response.

Intuitively, each process using a booster object is to bind itself to a port of the object by applying an *enroll* operation at that port. The process then solves two pairwise consensus problems between itself and each of the other processes using the object. The process bound to port p uses p as its input value in each of the two consensus problems it has to solve. If the decision in a consensus problem is the port number p, we say that p is the winner of that consensus. The *tmtw* operation is used for a process to report to the booster object the outcomes of the two consensus problems that it solved. The two parameters to *tmtw* are supposed to indicate the winners of the two consensus problems that the process issuing the operation has solved. Specifically, if the process bound to port p invokes $tmtw(q, r)$, it is asserting that q (respectively, r) is the winner of the pairwise consensus between (the process bound to) p and (the process bound to) the smaller (respectively, larger) of the other two ports.

There are certain rules that processes are expected to respect in accessing a booster object. If these rules are not respected, the object gets "upset". Once upset, the object remains upset forever. If the object is upset and a *tmtw* operation is invoked, the object nondeterministically chooses a port number (0, 1 or 2) and returns it as the response. The three rules that determine when an object gets upset are stated below.

The **first rule** is that, at any port p, an *enroll* operation must be applied before a *tmtw* operation. The **second rule** is that if a $tmtw(q, r)$ operation is applied at some port, then an *enroll* operation has already been applied at ports q and r. In other words, a port cannot be reported to be the winner (of a pairwise consensus between two ports) unless the *enroll* operation has already been applied at that port. The **third rule** is that *tmtw* operations, applied at possibly different ports, should give compatible information concerning winners of pairwise consensus. Thus, for example, if $tmtw(0, 2)$ is applied at port 0 and $tmtw(1, 2)$ is applied at port 1, the two operations

S1. `booster` has three ports, numbered 0, 1 and 2.

S2. For a port number $p \in \{0, 1, 2\}$, define the "first pair of p", $FP(p)$, (respectively, "second pair of p", $SP(p)$) to be the (unordered) pair consisting of p and the minimum (respectively, maximum) of the other two port numbers. More precisely, $FP(p) = \{p, \min(\{0, 1, 2\} - \{p\})\}$, and $SP(p) = \{p, \max(\{0, 1, 2\} - \{p\})\}$.

S3. The following operations may be invoked at port p: *enroll* and *tell-me-the-winner*(q, r), where $q \in FP(p)$ and $r \in SP(p)$. Hereafter we will abbreviate *tell-me-the-winner* as *tmtw*.

S4. The response for an *enroll* operation is always *ack*. The response for a $tmtw(q, r)$ operation is a port number — 0, 1 or 2. (The sequential specification in **S7** makes this more specific.)

S5. The state of `booster` is represented by the following variables: $enrolled(p)$: `boolean` ($p \in \{0, 1, 2\}$), $PairwiseWinner(\{p, q\})$: $\{\perp, p, q\}$ ($p \neq q$ and $p, q \in \{0, 1, 2\}$), *upset*: `boolean`, *WinnerPort*: $\{\perp, 0, 1, 2\}$.

The intended meaning of these variables may be described as follows. The object sets $enrolled(p)$ to true when an *enroll* operation is invoked at port p. Thus, $enrolled(p)$ lets the object remember if an *enroll* operation has already been invoked at port p. When a process P invokes $tmtw(q, r)$ at port p, P is expected to report the winner of the pairwise consensus between the two ports in $FP(p)$ through the argument q. Similarly, P is expected to report the winner of the pairwise consensus between the two ports in $SP(p)$ through the argument r. The object remembers these reports of the pairwise winners between ports through the *PairwiseWinner* variables. Specifically, the variable $PairwiseWinner(\{p, q\})$ lets the object remember the reported winner between ports p and q. Certain sequences of operations provoke the object into setting the variable *upset* to true. Once set, this variable remains set forever. Furthermore, while $upset =$ true, the responses of the object to the *tmtw* operations are completely nondeterministic. If the object is not upset (i.e., $upset =$ false), it returns the same response to all *tmtw* operations. This response is a port number at which an *enroll* operation was already performed. The object remembers this response through the variable *WinnerPort*.

S6. Define the *fresh state* as the state in which $enrolled(p) =$ false for all $p \in \{0, 1, 2\}$, $PairwiseWinner(\{p, q\}) = \perp$ for all $p \neq q$ and $p, q \in \{0, 1, 2\}$, $upset =$ false, and $WinnerPort = \perp$. The states of the type `booster` are exactly those that are reachable from the fresh state by the application of *enroll* and *tmtw* operations.

S7. The following is the sequential specification of the type `booster`. Given a nonempty set S, $\textsc{choose}(S)$ nondeterministically chooses an element from S and returns it.

<u>*enroll* at port p</u>
$enrolled(p) :=$ true
return *ack*

<u>$tmtw(q, r)$ at port p</u>
if $\neg enrolled(p)$ **or** % violation of Rule 1
  $\neg enrolled(q)$ **or** $\neg enrolled(r)$ **or** % violation of Rule 2
  $PairwiseWinner(FP(p)) \notin \{\perp, q\}$ **or** $PairwiseWinner(SP(p)) \notin \{\perp, r\}$ **then** % violation of Rule 3
    $upset =$ true
if *upset* **then**
    return $\textsc{choose}(\{0, 1, 2\})$
else
    if $WinnerPort = \perp$ **then**
        $WinnerPort := \textsc{choose}(\{i : enrolled(i)\})$
        $PairwiseWinner(FP(p)) := q$
        $PairwiseWinner(SP(p)) := r$
    return *WinnerPort*

Figure 1: Object type `booster`

are incompatible. This is because the first arguments of the two operations are contradictory: the first *tmtw* operation asserts that port 0 is the winner of the pairwise consensus between ports 0 and 1, while the second *tmtw* operation asserts that port 1 is the winner of that consensus.

As already mentioned, the object gets upset as soon as processes violate any one of the above rules. Once upset, the object will remain upset forever. The response from an upset object is completely nondeterministic. Intuitively, the object refuses to be helpful to processes unless they respect the above rules while accessing it. These rules are designed in such a way that three processes will simply not have the ability to respect the rules unless 2-consensus objects are also made available to them. Thus, booster objects will be of no use in solving consensus among three processes unless 2-consensus objects are also available.

4 Type `booster` is strong in the presence of `2-consensus`

In this section we describe an algorithm that solves consensus between three processes, 0, 1 and 2, using only registers and objects of type `booster` and `2-consensus`.

---

shared objects:
B: `booster`, initialized to the fresh state
$C_{\{p,q\}}, p \neq q \in \{0,1,2\}$: `2-consensus`, initialized to $\perp$
$R_p, p \in \{0,1,2\}$: `register`, initialized arbitrarily

The ports of $C_{\{p,q\}}$ are renamed p and q (instead of 0 and 1), for each $p \neq q \in \{0,1,2\}$.
Process p is bound to port p of each object, $p \in \{0,1,2\}$.

Program of process $p \in \{0,1,2\}$
local variables: v, f, s, w, d

$v :=$ input value of process p
$\textsc{Apply}(p, \mathit{write}(v), R_p)$
$\textsc{Apply}(p, \mathit{enroll}, B)$
$f := \textsc{Apply}(p, \mathit{propose}(p), C_{FP(p)})$
$s := \textsc{Apply}(p, \mathit{propose}(p), C_{SP(p)})$
$w := \textsc{Apply}(p, \mathit{tmtw}(f,s), B)$
$d := \textsc{Apply}(p, \mathit{read}, R_w)$
decide d

Figure 2: A consensus algorithm for processes 0, 1 and 2.

The algorithm is shown in Figure 2. The informal description below may be helpful. The algorithm requires a booster object B, initialized to the fresh state; three 2-consensus objects, $C_{\{0,1\}}$, $C_{\{0,2\}}$ and $C_{\{1,2\}}$, each initialised to $\perp$; and three 3-ported registers R_0, R_1 and R_2, each initialised arbitrarily. For convenience, we label the two ports of the 2-consensus object $C_{\{p,q\}}$ as p and q (rather than the customary 0 and 1). Each process $p \in \{0,1,2\}$ binds itself to port p of each object used in the algorithm. This means that each process p is bound to one port of the booster object, one port of each of the registers, and one port in two out of the three 2-consensus objects. Specifically, p is bound to one port of $C_{FP(p)}$ and one port of $C_{SP(p)}$ (FP and SP are defined in rule **S2** of the specification of type `booster`, in Figure 1). Process p starts the algorithm by writing its input value into register R_p. It then enrolls with the booster object B, and enters a competition with each of the other two processes, to "win" the 2-consensus objects $C_{FP(p)}$ and $C_{SP(p)}$. To do so, p proposes its own name p to each of these objects. We say that p *wins* the competition between itself and the smaller (respectively, larger) of the other two processes, if p is the first to apply the *propose* operation to $C_{FP(p)}$ (respectively, $C_{SP(p)}$). Let f be the winner of the competition between p and the smaller of the other processes; similarly let s be the winner of the competition between p and the larger of the other processes. Process p invokes operation $tmtw(f, s)$ to the booster object, and regards the response w as the identity of the "winner" among the three processes. It then reads the input value of the winner process w (which was previously written by w into R_w) and decides that value. In Figure 2, we use the notation $\text{APPLY}(p, op, O)$ to indicate the procedure by which a process invokes operation op at port p of object O; the procedure returns the response of that invocation.

The crucial fact about this algorithm is that processes access the booster object B without upsetting it. This follows from the observations below:

1. Each process p invokes the *tmtw* operation only after it completes an *enroll* operation. Thus, at each port p, the *tmtw* operation is invoked only after an *enroll* operation completes. This means that Rule 1 is satisfied.

2. A process enters into the competition for winning the 2-consensus objects to which it is bound only after it has completed the *enroll* operation at its port. Thus, if process p wins $C_{FP(p)}$ or $C_{SP(p)}$, then *enroll* is already completed at port p of the booster object. It follows that if process p invokes $tmtw(f, s)$, then *enroll* has already been completed at ports f and s. This means that Rule 2 is satisfied.

3. For any pair of ports p and q, it is obvious from the specification of `2-consensus` that the same winner of the 2-consensus object $C_{\{p,q\}}$ is reported to B by process p (through an argument of its *tmtw* operation) as is reported to B by process q. This means that Rule 3 is satisfied.

Since B is not upset, by the specification of `booster`, all three processes obtain the same response, say w, to their *tmtw* operations. Thus, all processes decide by reading the same register, R_w. Furthermore, the specification of `booster` guarantees that before w is returned by a *tmtw* operation, $enrolled(w) = \text{true}$ — and hence that *enroll* was applied at port w of B. Since process w writes into R_w before it applies *enroll* at port w of B, it follows that all processes decide by reading R_w after the latter was written into by process w. Since that register is written only once, and the value written into it is the input value of process w, it follows that all processes decide on the same value, and that value was the input of one of the processes — as required by consensus. Since the algorithm in Figure 2 uses only booster, 2-consensus and register objects we have:

Lemma 1 {`booster`, `2-consensus`, `register`} *solves consensus for three processes.*

Note that in Figure 2 processes bind to ports in accordance with the rules of model $\mathcal{S}$ (and, *a fortiori*, in accordance to the rules of $\mathcal{D}$). Furthermore, the algorithm in Figure 2 uses only registers, one booster object and 2-consensus objects. By Corollary 1, we have

Corollary 3 *For each of the binding models $\mathcal{S}$ and $\mathcal{D}$,* $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\{\mathtt{booster}, \mathtt{2\text{-}consensus}\}) \geq 3$.

5 Type `booster` alone is weak

In this section we prove that, for binding scheme $\mathcal{D}$, it is impossible for two processes to solve consensus using only booster objects and registers. (The result applies, *a fortiori*, for binding scheme $\mathcal{S}$.) To that end, we show that `booster` can be implemented from registers *for two processes* with binding scheme $\mathcal{D}$. Thus, if there was a consensus protocol $\mathcal{P}$ for two processes that uses only booster objects and registers, then we could obtain another consensus protocol $\mathcal{P}'$ (for two processes) by replacing each booster object used in $\mathcal{P}$ with its implementation from registers. $\mathcal{P}'$ is a consensus protocol (for two processes) that uses *only* registers. This is well known to be impossible [CIL87, DDS87, LA87].

Let s be any state of the type `booster`. We claim that a booster object, initialised to state s and shared by only two processes, can be implemented using only registers. First we prove this claim for the easy case when s is not a fresh state. From the specification of `booster` it is clear that if s is not a fresh state, then one of the following is true: s is an upset state, or s is a state in which the state variable *WinnerPort* is not $\perp$, or s is a state in which *WinnerPort* is $\perp$ and $\mathit{enrolled}(p) = \mathit{true}$ for some p. We consider each of these cases:

- If s is an upset state, the implementation can simply return *ack* for each *enroll* operation and any of 0, 1 or 2 for each *tmtw* operation.
- If s is a state in which the state variable *WinnerPort* is w for some $w \in \{0, 1, 2\}$, the implementation can simply return *ack* for each *enroll* operation and w for each *tmtw* operation.
- If s is a state in which *WinnerPort* is $\perp$ and $\mathit{enrolled}(p) = \mathrm{true}$ for some p, the implementation can simply return *ack* for each *enroll* operation and p for each *tmtw* operation.

In the rest of this section, we show how to implement a booster object, initialised to the fresh state, from registers, for two processes using binding scheme $\mathcal{D}$. Recall that in this binding scheme each process dynamically chooses at most one port of the implemented booster object, and binds itself to it permanently. The implementation is presented in Figure 3, and informally explained below.

A process applies operation *op* at port p of the implemented booster object $\mathcal{O}$ by executing the procedure $\textsc{Apply}(p, op, \mathcal{O})$. $\mathcal{O}$ is implemented using three (binary-valued) registers R_0, R_1 and R_2, each initialized to 0. The fact that $R_p = 0$ indicates that no process has bound itself to port p, yet. As we shall see, the role of the registers is to help each process inform the other of the port through which it accesses the booster object. For $p \in \{0, 1, 2\}$, define $p^- = (p - 1) \mod 3$ and $p^+ = (p + 1) \mod 3$. In other words, p^- and p^+ are "the other" two ports, relative to port p.

To perform an *enroll* operation at port p, we set R_p to 1, and return *ack*. To perform a $\mathit{tmtw}(q, r)$ operation at port p, we first determine the port (if any) through which the other process may have accessed the booster. To do so, we examine the registers that correspond to the other two ports, R_{p^-} and R_{p^+}. There are three possible outcomes of this examination:

(a) Neither one of the other ports has been accessed. Then the $\mathit{tmtw}(q, r)$ operation returns p.

shared objects:
$R_p, p \in \{0,1,2\}$: `register`, initialized to 0

Procedure $\text{APPLY}(p, \mathit{enroll}, \mathcal{O})$
begin
$\text{APPLY}(p, \mathit{write}(1), R_p)$
return *ack*
end

Procedure $\text{APPLY}(p, \mathit{tmtw}(q,r), \mathcal{O})$
local variables: *otherport*
begin
if $\text{APPLY}(p, \mathit{read}, R_{p^-}) = 1$ **then** $\mathit{otherport} := p^-$
elsif $\text{APPLY}(p, \mathit{read}, R_{p^+}) = 1$ **then** $\mathit{otherport} := p^+$
else $\mathit{otherport} := \perp$
if $\mathit{otherport} = \perp$ **then return** p
else if $\mathit{otherport} \in FP(p)$ **then return** q
else return r
end

Figure 3: Implementing a booster object $\mathcal{O}$ initialized to the fresh state and shared by two processes

(b) One of the other two ports, p', has been accessed, and $p' \in FP(p)$. Then the $\mathit{tmtw}(q,r)$ operation returns q.

(c) One of the other two ports, p', has been accessed, and $p' \in SP(p)$. Then the $\mathit{tmtw}(q,r)$ operation returns r.

(Note that it is not possible that both of the other ports have been accessed, since we are assuming that there are only two processes accessing the booster, and they do so under the rules of binding model $\mathcal{D}$.) We now informally justify why the responses to booster operations given in this implementation are valid. A rigorous proof of correctness of the implementation is given in Lemma 2.

First of all, there is no question as to the validity of responses to *enroll* operations: the only possible response is *ack* and that is what our implementation returns. Consider now *tmtw* operations. Any port number may be returned by a *tmtw* operation applied when the object is upset (or that causes the object to become upset). Since our implementation always returns a port number there is, again, no question as to the validity of the response to any such *tmtw* operation. Thus, we restrict our attention to *tmtw* operations that are applied when the object is not upset and that do not cause it to become upset. To the first such operation, a valid response is any port to which an *enroll* operation has been applied. That port may be regarded as "the winner", and it is the only valid response to any subsequent *tmtw* operation — as long as the object is not upset.

Consider a $tmtw(q, r)$ operation invoked at port p of the booster object $\mathcal{O}$. There are two cases, depending on the value of the local variable *otherport* (see lines 1–3 in Figure 3). This local variable records the port (if any) to which the other process is bound. If $otherport = \perp$ then it is possible that the other process has not yet applied *enroll* to any port of $\mathcal{O}$, and so that p is the only process to have applied an *enroll* operation. In this case, the only valid response is to return p — which is exactly what our implementation does (see line 4). If $otherport = p' \neq \perp$ then the other process has also enrolled to its port p'. In this case, the parameters of the operation $tmtw(q, r)$ being applied to $\mathcal{O}$ had better specify a legitimate "winner" between the two ports p and p' (which, in fact, are the only ports accessed in this execution, given the constraints of $\mathcal{D}$ binding, and the fact that there are only two processes using the booster object): If not then, by the definition of the `booster` type, the operation will cause the object to become upset (if it is not already upset). As mentioned before, in that case, any response is valid. Our implementation takes advantage of this observation to return an appropriate winner: If $p' \in FP(p)$, it is the first parameter of $tmtw(q, r)$ that is supposed to specify the winner between p and p'; thus, q is returned. If, on the other hand, $p' \in SP(p)$, it is the second parameter of $tmtw(q, r)$ that is supposed to specify the winner between p and p'; thus, r is returned. This completes our informal explanation of why the implementation is correct. A rigorous proof of correctness is given below.

Lemma 2 *The implementation shown in Figure 3 is a (wait-free and linearisable) implementation of a booster object $\mathcal{O}$, initialised to the fresh state, from registers, for two processes with binding scheme $\mathcal{D}$.*

PROOF. Since the pseudocode for each procedure is a straight-line program, the implementation is clearly wait-free. It remains to show that it is linearisable. Consider an arbitrary run $\mathcal{R}$, involving executions of the procedures shown in Figure 3 by two processes which are bound to ports according to the rules of binding scheme $\mathcal{D}$. We will show that $\mathcal{R}$ is linearisable with respect to type `booster` initialised to the fresh state. We assume (as required by the definition of linearisability) that before $\mathcal{R}$ starts, the base objects, R_0, R_1 and R_2, are initialised to 0 (the value specified by the implementation); and that the *read* and *write* operations applied to the registers in $\mathcal{R}$ are atomic. To show that $\mathcal{R}$ is linearisable with respect to type `booster` initialised to the fresh state, we must show that there is a total order of the accesses that correspond to the procedure invocations in $\mathcal{R}$ that (a) is consistent with respect to type `booster` initialised to the fresh state, and (b) respects the order of nonconcurrent procedure executions in $\mathcal{R}$. We will first specify a total ordering of the accesses that correspond to the procedure executions in $\mathcal{R}$ that respects the order of nonconcurrent procedure executions. We will then prove that the sequence of these accesses, in the specified order, is consistent with respect to `booster` initialised to the fresh state.

Since $\mathcal{R}$ is a run by two processes using binding model $\mathcal{D}$, there is at least one port, say u, at which no operation is applied in $\mathcal{R}$. (Thus, u is the "unused" port in $\mathcal{R}$.) For each procedure execution in $\mathcal{R}$, we will identify a single operation to a base object, called the *linearisation point* of the procedure execution. The total order of accesses that we want to define is the order of these linearisation points in $\mathcal{R}$. (This is well defined because we assume that accesses to the base objects in $\mathcal{R}$ are atomic.) Consider an execution of the *enroll* operation at port p in $\mathcal{R}$. It contains a single access to a base object, namely a *write* operation to register R_p. This is the linearisation point of this particular *enroll* procedure execution.

Consider an execution of the *tmtw* operation at port p in $\mathcal{R}$. This may contain exactly one access to a base object, namely a *read* operation of register R_{p^-} (cf. line 1 in Figure 3). If so, this is the linearisation point of the *tmtw* procedure execution. Alternatively, the execution of *tmtw* at port p may contain accesses to both R_{p^-} and R_{p^+} (cf. lines 1 and 2 in Figure 3). Note that one of p^- and p^+ is equal to the unused port, u. (This is because, by definition of u, $p \neq u$; thus one of the other two ports, p^- or p^+, must be u.) In this case, the linearisation point of the *tmtw* procedure execution is the reading of the base register that is *not* R_u. (Intuitively, in $\mathcal{R}$ the reading of R_u is useless because no process binds to port u, so no process applies *enroll* at u. Thus R_u is never written into and it always has its initial value, 0.)

Let σ be the sequence of accesses to the booster object $\mathcal{O}$ that correspond to the executions of *enroll* and *tmtw* procedures in $\mathcal{R}$, appearing in the order of their linearisation points. First we argue that this sequence respects the order of nonconcurrent procedure executions in $\mathcal{R}$. Let e and e' be two nonconcurrent procedure executions in $\mathcal{R}$. Without loss of generality, assume that e precedes e' in $\mathcal{R}$. In particular, the linearisation point of e precedes the linearisation point of e' in $\mathcal{R}$. Thus, the access that corresponds to e precedes the access that corresponds to e' in σ, as wanted. To complete the proof of the lemma, it remains to show that σ is consistent with respect type `booster` initialised to the fresh state.

Let $\langle o_i, p_i, r_i \rangle$ be the ith access in σ. (Recall, from Section 2, that this represents an invocation of operation o_i at port p_i, returning response r_i.) Let k be the number of invocations in σ. Define the sequence $s_1, s_2, \ldots, s_{k+1}$ of states of type `booster` so that s_1 is the fresh state, and for each i, $1 \leq i \leq k$, s_{i+1} is the state obtained if we apply operation o_i at port p_i when `booster` is in state s_i.

By the sequential specification of `booster`, s_{i+1} is uniquely defined except if o_i is a *tmtw*(q, r) operation applied at port p (p, q and r are not necessarily distinct), s_i is not an upset state, *WinnerPort* $= \perp$ in s_i, and *enrolled*(p) = *enrolled*(q) = *enrolled*(r) = true in s_i. In this case, the only state variable whose value is not uniquely determined by the sequential specification is the value to which *WinnerPort* is set. This value is nondeterministically chosen from the set of ports i such that *enrolled*(i) = true in s_i. Note that this set includes p, q and r, so any of these is a legitimate choice of value for *WinnerPort*. We make s_{i+1} unique even in this case by choosing the value of *WinnerPort* in s_{i+1} according to the following rule:

> **if** *enrolled*(p^-) = *enrolled*(p^+) = false **then** choose p
> **else if** *enrolled*(p^-) = true **then**
>   **if** $p^- \in FP(p)$ **then** choose q
>   **else** choose r
> **else**
>   **if** $p^+ \in FP(p)$ **then** choose q
>   **else** choose r

In this manner the states $s_1, s_2, \ldots, s_{k+1}$ are uniquely defined. Recall that $r_1, r_2, \ldots, r_k$ are the responses of the k accesses to `booster` in σ. We will show (see Claim 2 below) that $(s_i, o_i, p_i, r_i, s_{i+1})$ is in the state relation of `booster`, for each $1 \leq i \leq k$. This is the key to proving that $\mathcal{R}$ is linearisable with respect to `booster`. Before we do so, we need to establish some useful facts regarding the values of the state variables *WinnerPort* and *PairwiseWinner* in states $s_1, s_2, \ldots, s_{k+1}$. In what follows we say that access $\langle o_i, r_i, p_i \rangle$ in σ is a *tmtw* access (respectively, an *enroll* access), if $o_i = tmtw(q, r)$, for some ports q and r (respectively, $o_i = enroll$). We also say that this is an access at port p, if $p_i = p$.

Claim 1 *For any i, $1 \le i \le k+1$, if s_i is not an upset state and $WinnerPort \neq \bot$ in s_i, then one of the first $i-1$ accesses in σ is a tmtw access. Let $p^*, q^*, r^* \in \{0,1,2\}$ (not necessarily distinct) be such that the first tmtw access in σ is $tmtw(q^*, r^*)$ and it is applied at port p^*. Then the following hold in s_i:*

(a) If there is no enroll access at any port $\neq p^*$ that precedes the first tmtw access in σ, then $WinnerPort = p^* = q^* = r^*$.

(b) If there is an enroll access at some port $\pi \neq p^*$ that precedes the first tmtw access in σ, then

$$PairwiseWinner(\{p^*, \pi\}) = WinnerPort = \begin{cases} q^*, & \text{if } \pi \in FP(p^*) \\ r^*, & \text{if } \pi \in SP(p^*). \end{cases}$$

(c) $PairwiseWinner(FP(p^*)) = q^*$ and $PairwiseWinner(SP(p^*)) = r^*$.

PROOF. If the first $i-1$ accesses in σ are all *enroll* accesses, and given the fact that s_1 is the fresh state in which $WinnerPort = \bot$, a straightforward induction shows that $WinnerPort = \bot$ in s_i. Thus, one of the the first $i-1$ accesses in σ is a *tmtw* access. Let the first such access be the jth access in σ, for some $j < i$. Let p^*, q^*, r^* be as in the hypothesis of the claim; i.e., p^* is the port at which the first *tmtw* access in σ is applied, and q^* and r^* are the parameters of that *tmtw* operation. First we show that (a)-(c) hold in s_{j+1} (i.e., immediately after the first *tmtw* access in σ); later we also show that they hold in s_i. Since s_1 is the fresh state, and none of the first $j-1$ accesses in σ is a *tmtw*, the specification of `booster` implies that, in s_j, $WinnerPort = \bot$. Also since, by hypothesis of the claim, s_i is not upset and $j < i$, s_{j+1} is not upset either.

(a) Suppose there is no *enroll* access at any port $\neq p^*$ that precedes the first *tmtw* access in σ. Recall that this access is a $tmtw(q^*, r^*)$ operation applied at port p^*. In s_1, $enrolled(\pi) = \text{false}$, for all ports π, and none of the first j accesses in σ are *enroll* accesses to any port $\neq p^*$. Hence, for any port $\pi \neq p^*$, $enrolled(\pi) = \text{false}$, in s_j. Furthermore, since s_{j+1} is not an upset state, and the jth access in σ is a $tmtw(q^*, r^*)$ operation at port p^*, $enrolled(p^*) = \text{true}$ in s_j. By the specification of `booster`, applying $tmtw(q^*, r^*)$ to a state in which $enrolled(q^*)$ or $enrolled(r^*)$ is false, causes `booster` to enter an upset state. Since s_{j+1} is not upset, $p^* = q^* = r^*$. Also by the specification of `booster`, when $tmtw(q^*, r^*)$ at port p^* is applied to a state where $WinnerPort = \bot$, the state is changed to one in which $WinnerPort$ is set to one of p^*, q^* or r^*. Since all these are equal to each other, in s_{j+1}, $WinnerPort = p^* = q^* = r^*$, as wanted.

(b) Suppose there is an *enroll* access at some port $\pi \neq p^*$ that precedes the first *tmtw* access in σ (which is a $tmtw(q^*, r^*)$ operation at port p^*). Thus, $enrolled(\pi) = \text{true}$ in s_j. Since s_{j+1} is not an upset state, $enrolled(p^*) = \text{true}$ in s_j. Let π' be the remaining port of `booster` — i.e., π' is the unique element of $\{0,1,2\} - \{p^*, \pi\}$. Since $\mathcal{R}$ is a run produced by at most two processes bound to objects according to the rules of binding scheme $\mathcal{D}$, there is no *enroll* access at port π'. Thus, the set of ports i such that $enrolled(i) = \text{true}$ in s_{j-1} is $\{p^*, \pi\}$. By the rule, described earlier, for fixing the (nondeterministically chosen) value of $WinnerPort$ in s_{j+1} we have,

$$WinnerPort = \begin{cases} q^*, & \text{if } \pi \in FP(p^*) \\ r^*, & \text{if } \pi \in SP(p^*). \end{cases}$$

Also, by the specification of `booster`, in s_{j+1},

$$\mathit{PairwiseWinner}(\{p^*, \pi\}) = \begin{cases} q^*, & \text{if } \pi \in FP(p^*) \\ r^*, & \text{if } \pi \in SP(p^*). \end{cases}$$

Thus, in s_{j+1}, $\mathit{WinnerPort} = \mathit{PairwiseWinner}(\{p^*, \pi\})$, as wanted.

(c) Since $\mathit{WinnerPort} = \perp$ in s_j, the sequential specification of `booster` implies that, in s_{j+1}, $\mathit{PairwiseWinner}(FP(p^*)) = q^*$ and $\mathit{PairwiseWinner}(SP(p^*)) = r^*$, as wanted.

We have shown that (a)-(c) hold in s_{j+1}. From the specification of `booster` it is easy to verify that once $\mathit{WinnerPort} \neq \perp$, state variables *WinnerPort* and *PairwiseWinner* are not changed. Using this observation, a simple induction shows that (a)-(c) hold for $s_{i'}$, for any $i' \in \{j+1, \ldots, i\}$. In particular, they hold for s_i. This completes the proof of Claim 1. □

Claim 2 *For any i, $1 \leq i \leq k$, $(s_i, o_i, p_i, r_i, s_{i+1})$ belongs to the transition relation of* `booster`.

Proof. We must prove that if `booster` is in state s_i then applying o_i at port p_i may cause `booster` to enter state s_{i+1} and may return r_i. That `booster` may enter state s_{i+1} follows by the definition of the sequence $s_1, s_2, \ldots, s_{k+1}$. It remains to prove that the response returned by our implementation, r_i, is a legitimate response to operation o_i applied at port p_i when `booster` is in state s_i. This is obvious if o_i is an *enroll* operation or if s_{i+1} is an upset state. Thus, we may assume that o_i is a $\mathit{tmtw}(q, r)$ operation applied at port p, and that s_{i+1} is not an upset state.

Since s_{i+1} is not an upset state, $\mathit{enrolled}(p) = \mathit{enrolled}(q) = \mathit{enrolled}(r) = \mathtt{true}$ in s_i. If $\mathit{WinnerPort} = \perp$ in s_i, then by the sequential specification of `booster`, any of p, q or r is a legitimate response. The response returned by our implementation, r_i, is certainly one of these. So, in this case we are done.

It remains to consider the case where $\mathit{WinnerPort} \neq \perp$ in s_i. In this case, by the sequential specification of `booster`, the only legitimate response to the $\mathit{tmtw}(q, r)$ operation is the value of *WinnerPort* in s_i. Thus, we need to prove that r_i is equal to the value of *WinnerPort* in s_i.

By Claim 1, one of the first $i-1$ operation invocations in σ is to a *tmtw* operation. Let $p^*, q^*, r^* \in \{0, 1, 2\}$ be such that the first *tmtw* operation in σ is $\mathit{tmtw}(q^*, r^*)$ at port p^*. By Claim 1(a) and 1(b), we have that in s_i:

$$\mathit{WinnerPort} = \begin{cases} p^*, & \text{if no } \mathit{enroll} \text{ access at any port} \neq p^* \text{ precedes the first } \mathit{tmtw} \text{ in } \sigma \quad (*) \\ q^*, & \text{if some } \mathit{enroll} \text{ access at port } p' \neq p^* \text{ precedes the first } \mathit{tmtw} \text{ in } \sigma, \\ & \quad \text{and } p' \in FP(p^*) \quad (\dagger) \\ r^*, & \text{if some } \mathit{enroll} \text{ access at port } p' \neq p^* \text{ precedes the first } \mathit{tmtw} \text{ in } \sigma, \\ & \quad \text{and } p' \in SP(p^*) \quad (\ddagger) \end{cases}$$

There are two cases, depending on whether p and p^* are the same port.

Case 1. $p = p^*$. By Claim 1(c), in s_i, $\mathit{PairwiseWinner}(FP(p^*)) = q^*$ and $\mathit{PairwiseWinner}(SP(p^*)) = r^*$. By the specification of `booster`, since s_{i+1} is not an upset state, $\mathit{PairwiseWinner}(FP(p)) = q$ and $\mathit{PairwiseWinner}(SP(p)) = r$. Since $p = p^*$, in this case we have that $q = q^*$ and $r = r^*$. We distinguish two subcases.

SUBCASE 1(a). There is no *enroll* access to any port $\neq p^*$ that precedes the first *tmtw* access in σ. Then by Claim 1(a), $p^* = q^* = r^*$. Hence, in this subcase, $p = q = r = p^* = q^* = r^*$. Our implementation returns one of p, q or r. These are all the same and equal to the only possible value of *WinnerPort* in s_i (see $(*)$). Thus, in this subcase r_i is the value of *WinnerPort* in s_i, as wanted.

SUBCASE 1(b). There is an *enroll* access to some port $p' \neq p^*$ that precedes the first *tmtw* access in σ. Consider the procedure executions in run $\mathcal{R}$ (implemented as in Figure 3) that correspond to these two accesses. By definition of σ, in $\mathcal{R}$, the *write* access that writes 1 into $R_{p'}$ (contained in the execution of the procedure that corresponds to *enroll* at port p') precedes the *read* access that reads 1 in $R_{p'}$ (contained in the execution of the procedure that corresponds to the first *tmtw*). According to our implementation of *tmtw* then,

$$r_i = \begin{cases} q, & \text{if } p' \in FP(p) \\ r, & \text{if } p' \in SP(p). \end{cases}$$

Recalling that in this case $q = q^*$ and $r = r^*$, we conclude that r_i is precisely the value of *WinnerPort* in s_i (see (†) and (‡)), as wanted.

CASE 2. $p \neq p^*$. First note that p and p^* are the only ports at which operations are applied. This is because $\mathcal{R}$ is a run produced by at most two processes bound to ports in accordance with the rules of binding scheme $\mathcal{D}$. Since there are operations applied at ports p and p^*, and $p \neq p^*$, there cannot be operations to any other ports. Next we show that, in s_i, $\mathit{WinnerPort} = \mathit{PairwiseWinner}(\{p^*, p\})$. We distinguish two possibilities:

- There is no *enroll* access to any port $\neq p^*$ that precedes the first *tmtw* access in σ. Then by Claim 1(a) and (c), $\mathit{PairwiseWinner}(FP(p^*)) = \mathit{PairwiseWinner}(SP(p^*)) = \mathit{WinnerPort} = p^*$. Since $\{p^*, p\}$ is either $FP(p^*)$ or $SP(p^*)$, it follows that $\mathit{WinnerPort} = \mathit{PairwiseWinner}(\{p^*, p\})$.
- There is an *enroll* access to some port $\pi \neq p^*$. By the earlier observation that p and p^* are the only ports to which operations are applied, it must be that $\pi = p$. Thus, by Claim 1(b), we have that $\mathit{PairwiseWinner}(\{p, p^*\}) = \mathit{WinnerPort}$.

In either case, we have that $\mathit{WinnerPort} = \mathit{PairwiseWinner}(\{p^*, p\})$.

Since s_{i+1} is not an upset state, an *enroll* access at port p^* precedes the first *tmtw* access in σ (which, by assumption, is applied at port p^*). Thus, an *enroll* access at port p^* precedes in σ the $\mathit{tmtw}(q, r)$ access applied at port p. Thus, in $\mathcal{R}$, the linearisation point of the procedure execution that corresponds to the *enroll* access at port p^* precedes the linearisation point of the procedure execution that corresponds to the $\mathit{tmtw}(q, r)$ access at port p. More specifically, in $\mathcal{R}$, the *write* access that is the linearisation point of *enroll* at p^*, precedes the *read* access that is the linearisation point of $\mathit{tmtw}(q, r)$ at p. Hence, in our implemention of $\mathit{tmtw}(q, r)$, the value assigned to *otherport* is p^*. Hence, the response to $\mathit{tmtw}(q, r)$ returned by our implementation is

$$r_i = \begin{cases} q, & \text{if } p^* \in FP(p) \\ r, & \text{if } p^* \in SP(p). \end{cases}$$

By the sequential specification of `booster`, the application of $\mathit{tmtw}(q, r)$ at port p in a state where $\mathit{PairwiseWinner}(FP(p)) \neq q$ or $\mathit{PairwiseWinner}(SP(p)) \neq r$ causes `booster` to enter an upset state. Since s_{i+1} is not an upset state, it follows that in s_i:

$$\mathit{PairwiseWinner}(\{p^*, p\}) = \begin{cases} q, & \text{if } p^* \in FP(p) \\ r, & \text{if } p^* \in SP(p). \end{cases}$$

As shown above, in s_i, $PairwiseWinner(\{p^*, p\}) = WinnerPort$. Therefore, in s_i, $r_i = WinnerPort$, as wanted. This completes the proof of Claim 2. $\square$

By Claim 2, σ is consistent with respect to `booster` initialised to the fresh state (s_1). We have therefore shown that an arbitrary run $\mathcal{R}$ consisting of executions of the procedures in Figure 3 is linearisable with respect to `booster` initialised to the fresh state. Hence, Figure 3 is a linearisable and wait-free implementation of a booster object initialised to the fresh state for two processes with binding scheme $\mathcal{D}$. This completes the proof of Lemma 2. $\square$

We have already argued that there are also implementations, using only register objects, of a booster object initialised to any state other than the fresh state. Thus we have,

Lemma 3 `register` *implements* `booster` *for two processes with binding scheme* $\mathcal{D}$ *— and,* a fortiori, *with binding scheme* $\mathcal{S}$.

Lemma 4 *For each of the binding schemes* $\mathcal{S}$ *and* $\mathcal{D}$, $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathtt{booster}) = 1$.

PROOF. Suppose, for contradiction, that $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathtt{booster}) \geq 2$. By definition of $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}$, there is an implementation $\mathcal{I}$ of `2-consensus` from $\{\mathtt{booster}, \mathtt{register}\}$ for two processes with binding scheme $\mathcal{D}$ (or $\mathcal{S}$). By Lemma 3, we can replace the booster objects used in $\mathcal{I}$ with their register implementations. This results in a new implementation of `2-consensus` from `register` for two processes with binding scheme $\mathcal{D}$ (or $\mathcal{S}$). By Theorems 1 and 2, it follows that there is an algorithm that solves consensus between two processes and uses only registers. This contradicts the well-known fact that no such algorithm exists [CIL87, DDS87, LA87]. $\square$

We can now prove our main result:

Theorem 3 *If types may be nondeterministic and nonoblivious, for each of the binding schemes* $\mathcal{S}$ *and* $\mathcal{D}$*, the hierarchy* $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}$ *is not robust.*

PROOF. (All statements regarding $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}$ in this proof are valid with either binding scheme $\mathcal{S}$ or $\mathcal{D}$.) By Corollary 3, $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\{\mathtt{booster}, \mathtt{2\text{-}consensus}\}) \geq 3$. By Lemma 4, $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathtt{booster}) = 1$. By Corollary 2, $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathtt{2\text{-}consensus}) = 2$. Thus, $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\{\mathtt{booster}, \mathtt{2\text{-}consensus}\}) > \max(\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathtt{booster}), \mathtt{h}^{\mathtt{r}}_{\mathtt{m}}(\mathtt{2\text{-}consensus}))$, which means that $\mathtt{h}^{\mathtt{r}}_{\mathtt{m}}$ is not robust. $\square$

References

[BGA94] E. Borowsky, E. Gafni, and Y. Afek. Consensus power makes (some) sense. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 363–372, August 1994.

[BNP94] R. Bazzi, G. Neiger, and G. Peterson. On the use of registers in achieving wait-free consensus. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 354–363, 1994.

[CHJT94] T. Chandra, V. Hadzilacos, P. Jayanti, and S. Toueg. Wait-freedom vs. t-resiliency and the robustness of wait-free hierarchies. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 334–343, 1994.

[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.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

[Her91] 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.

[Jay93] P. Jayanti. On the robustness of Herlihy's hierarchy. In *Proceedings of the 12th Annual Symposium on Principles of Distributed Computing*, August 1993.

[Jay95] P. Jayanti. Solvability of consensus: composition breaks down for non-deterministic types. *SIAM Journal of Computing*, 1995. Accepted for publication.

[JT92] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the 6th Workshop on Distributed Algorithms, Haifa, Israel*, November 1992. Appeared in Lecture Notes in Computer Science, Springer-Verlag, No: 647.

[KM93] J. Kleinberg and S. Mullainathan. Resource bounds and combinations of consensus objects. In *Proceedings of the 12th Annual Symposium on Principles of Distributed Computing*, August 1993.

[LA87] M.C. Loui and H.H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research*, 4:163–183, 1987.

[LH97] Wai-Kau Lo and Vassos Hadzilacos. All of us are smarter than any of us: wait-free hierarchies are not robust. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, May 1997.

[LT88] N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT, MIT Laboratory for Computer Science, 1988.

[MR96] S. Moran and L. Rappoport. On the robustness of $\mathtt{h}_{\mathtt{m}}^{\mathtt{r}}$. In *Proceedings of the 10th Workshop on Distributed Algorithms*, October 1996.

[PBN94] G. Peterson, R. Bazzi, and G. Neiger. A gap theorem for consensus types. In *Proceedings of the 13th Annual Symposium on Principles of Distributed Computing*, pages 344–353, August 1994.

[Sch96] Eric Schenk. *Computability and Complexity Results for Agreement Problems in Shared Memory Distributed Systems*. PhD thesis, University of Toronto, 1996.