# Abortable and Query-Abortable Objects and their Efficient Implementation

Marcos K. Aguilera
HP Laboratories
Palo Alto, California, USA

Svend Frolund*
Gatehouse A/S
Denmark

Vassos Hadzilacos†
Dept. of Computer Science
University of Toronto, Canada

Stephanie L. Horn†
Dept. of Computer Science
University of Toronto, Canada

Sam Toueg†
Dept. of Computer Science
University of Toronto, Canada

## ABSTRACT

We introduce *abortable* and *query-abortable* objects, intended for asynchronous shared-memory systems with low contention. These objects behave like ordinary objects when accessed sequentially, but may abort operations when accessed concurrently. An aborted operation may or may not take effect, i.e., cause a state transition, and it returns no indication of which possibility occurred. Since this uncertainty is problematic, a query-abortable object supports a QUERY operation that each process can use to determine its last non-QUERY operation on the object that caused a state transition, and the response associated with this state transition. Query-abortable objects can easily implement obstruction-free objects (introduced by Herlihy, Luchangco and Moir) and pausable objects (introduced by Attiya, Guerraoui and Kouznetsov).

We present universal constructions for abortable and query-abortable objects that use only *abortable registers* — registers that are strictly weaker than safe registers. Our universal constructions are novel and efficient in the number of registers used. Specifically, they are based on a simple timestamping mechanism for detecting concurrent executions, and in systems with $n$ processes, they use only $2n$ abortable registers. It is worth noting that our results imply that any obstruction-free object and any pausable object can be implemented using only $2n$ abortable registers.

Finally, we identify a potential problem with correctness properties based on step contention: with such properties, the composition of correct object implementations may result in an implementation that is *not* correct. In other words, implementations defined in terms of step contention are not always composable. To avoid this problem, we use interval contention to define the correct behaviour of abortable and query-abortable objects.

## Categories and Subject Descriptors

B.3.2 [**Memory structures**]: Design Styles—*Shared memory*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.4.1 [**Operating systems**]: Process Management—*Concurrency*; F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous

## General Terms

Algorithms, Design, Theory

## Keywords

Shared memory, memory contention, abortable types, non-triviality, universal constructions, obstruction-freedom

## 1. INTRODUCTION

We consider distributed systems in which asynchronous processes interact by accessing linearizable, wait-free shared objects [2]. A typical task in such a system is to implement a target object given certain "base objects". The complexity of such implementations is mostly due to the fact that multiple processes may access the target object concurrently. It has been observed that in some systems contention on shared objects is rare [3]. It is therefore tempting to try to simplify the task of implementing shared objects by allowing them to exhibit degraded behaviour in the (hopefully rare) event of contention.

In this paper we first introduce *abortable* objects. An abortable object behaves like an ordinary one when accessed sequentially, but may abort operations when accessed concurrently. An operation that aborts may or may not take effect, i.e., cause a state transition, but the object is not left in an inconsistent state. An operation that aborts returns the special value $\perp$; thus, the caller of an aborted operation does not know whether this operation took effect, and if it did, what the corresponding response should have been (this response is lost). For some objects, this uncertainty is tolerable: for example, for an abortable consensus object, the caller of an aborted "propose($v$)" operation can simply reapply this operation because there is no harm in doing so (it does not matter whether the aborted propose($v$) took effect or not). For many objects, however,

*Work done while author was at HP Laboratories.

†[vassos, slh, sam]@cs.toronto.edu. These authors were partially supported by the National Science and Engineering Research Council of Canada.


*PODC'07,* August 12–15, 2007, Portland, Oregon, USA.

not knowing whether an aborted operation took effect (and if it did take effect, losing the corresponding response) make them very difficult to use.

To solve this problem, we introduce *query-abortable* objects. Intuitively, a query-abortable object $O$ is an abortable object with an additional operation, called QUERY: each process can use QUERY to determine (a) its last non-QUERY operation on $O$ that caused a state transition, and (b) the response associated with this state transition. Of course, a QUERY operation can itself return $\perp$, if it happens to be concurrent with other operations. In this case, the user of a query-abortable object can apply QUERY repeatedly until it returns a non-$\perp$ response (this is guaranteed to occur when contention on this object stops).

For example, suppose a process $p$ applies a DEQUEUE operation on a query-abortable queue and the operation aborts. If $p$ now applies QUERY, the queue returns one of two possible values: (1) the last queue operation that $p$ applied that actually took effect and the associated response (from this reply $p$ can immediately deduce whether its aborted DEQUEUE took effect, and if it did, what element was dequeued), or (2) $\perp$ (this can occur only if there is contention); in this case, $p$ can reapply QUERY.

Query-abortable objects are related to the well-known concept of *obstruction-free* implementations, which was introduced by Herlihy, Luchangco and Moir in [3] as a weaker but simpler and cheaper alternative to wait-freedom. Intuitively, an implementation of an object is *obstruction-free* if it returns a response to each operation that eventually executes *in isolation*. It is easy to see that a query-abortable object of a given type can be used to implement an obstruction-free object of the same type (e.g., it is easy to use a query-abortable queue to derive an obstruction-free implementation of a queue). The converse, however, is not true: with an obstruction-free implementation of an object, an operation that encounters persistent contention may "starve", i.e., it may never return a response. This behaviour cannot occur with query-abortable objects: with such objects, operations may abort but they always respond.

In this paper, we prove that any abortable or query-abortable object can be implemented using only *abortable* registers. Such registers are quite weak; in fact, we show that abortable registers are strictly weaker than *safe* registers (defined by Lamport in [6]). The source of this weakness can be traced to the following behaviour: when a reader and a writer access a safe register concurrently, the read operation may fail but the write always succeeds, i.e., it always takes effect; in contrast, with an abortable register, the read may abort *and* the write may also abort (with or without taking effect).

Our universal constructions for abortable and query-abortable objects are simple, and they use only $2n$ abortable registers ($n$ is the number of processes in the system). To achieve this, they do not use the standard universal construction technique where "consensus-like" objects thread operations into a linear list. Instead, they use a new "counter-like" object, called *inc&read*, and do not keep any lists. The object *inc&read* provides a timestamping mechanism used to detect whether any two "code segments" are concurrent or whether one follows the other — a mechanism that appears to be of general use.

In this paper, we show that (abortable) *inc&read* can be implemented from (abortable) registers. In the full paper, we also show that abortable registers can be implemented from single-writer single-reader abortable registers.

Since (a) the universal construction for query-abortable objects uses only $2n$ abortable registers, and (b) it is easy to transform any query-abortable object into its obstruction-free counterpart and this can be done without using additional objects, we get a universal construction for *obstruction-free* objects that uses only $2n$ abortable registers. This is the first universal construction for obstruction-free objects that uses only a linear number of registers, and also the first one that uses registers that are strictly weaker than safe ones.

We conclude the paper by comparing two different types of contention — a concept that is central to abortable objects, obstruction-free objects, and other objects that are allowed to exhibit degraded behaviour during periods of contention. We consider two different notions of contention that have been used in previous work, namely, *interval* contention and *step* contention. Informally, an operation encounters interval contention if its execution interval intersects that of another operation; an operation encounters step contention if its execution interval contains steps of another operation. It is easy to see that step contention implies interval contention, but the converse is not true.

In defining the desirable behaviour of abortable and query-abortable objects, we are faced with the choice of whether to use step or interval contention. *Prima facie*, step contention appears to be the better choice: intuitively, with step contention there are fewer circumstances under which an abortable object can abort. This choice, however, can lead to anomalies regarding the composability of implementations. More precisely, we show that if the correctness properties of implementations are based on step contention, the composition of correct object implementations may result in an implementation that is not correct. To avoid this problem, we define the correct behaviour of abortable and query-abortable objects using interval contention, rather than step contention: we prove that with our definitions, implementations of abortable and query-abortable objects are indeed composable.

**Related work.** Our definition of abortable objects is related to the work of Attiya, Guerraoui and Kouznetsov [1]. We now explain this work and its relation to our results.

By definition, in an obstruction-free implementation, an operation that always encounters contention may never terminate. Attiya et al. were the first to suggest that, instead of starving, it may be better for an operation facing such persistent contention to return control to the caller. To do so, they introduced the notion of "pausable objects". When an operation is applied to such an object and encounters contention, the object may return a special value indicating that the operation is now "paused". A paused operation causes no contention, giving other operations a chance to terminate. The caller of a paused operation on an object can access this object again *only* by resuming the paused operation.

In contrast, with an abortable or query-abortable object, the caller of an aborted operation is not restricted in any way: it may reapply the same operation, but if applying a different operation on the object better suits its objectives, it is free to do so. Our algorithms actually take advantage of this flexibility: in many cases when an operation aborts, the caller just abandons the aborted operation, and later applies a *different* one.

The results in [1] are based on step contention. In contrast, we use interval contention and prove that implementations of abortable objects are composable under this notion of contention. As we mentioned above, implementations that are defined in terms of step contention are not always composable.

[1] and [3] showed that pausable and obstruction-free objects can be implemented using registers. Here we prove something stronger: query-abortable objects can be implemented using a linear number of abortable registers.

The term "abortable" has also been used in the context of mutual exclusion algorithms [5]. Roughly speaking, in an abortable mu-

tual exclusion algorithm, at any time while in the entry section, a process can choose to abort — i.e., it can give up its attempt to enter the critical section and return to the non-critical section. Abortable mutual exclusion is quite different from abortable objects, and cannot be used to implement them: while abortable objects are inherently fault-tolerant, abortable mutual exclusion algorithms are not (if a process crashes in the critical section, the other processes are blocked forever).

**Summary of contributions.** We define abortable and query-abortable objects as natural and clean extensions of standard wait-free linearizable objects. We give universal constructions for abortable and query-abortable objects that are simple and efficient — they use only a linear number of abortable registers. We introduce a new object, called *inc&read*, that can be used to detect concurrency and seems to be of general use. We prove that abortable registers are strictly weaker than safe registers. We show that abortable registers can be implemented from single-writer single-reader abortable registers. Finally, we identify a potential problem with step contention: the composition of implementations does not always preserve correctness properties based on step contention.

Due to space limitations, most proofs are omitted from this extended abstract.

# 2. MODEL

In this section, we describe a model in sufficient detail to present our results. Let $\Pi = \{p_1, \ldots, p_n\}$ be a set of $n$ asynchronous processes, each representing a sequential thread of control, and $\mathcal{O}$ be a set of *objects*, each representing a data structure shared by the processes. Processes in $\Pi$ interact with objects in $\mathcal{O}$ by invoking operations on the objects and receiving corresponding responses from them.

Each process $p \in \Pi$ executes *steps*, which are of three kinds: the invocation by $p$ of an operation $op$ on an object $O \in \mathcal{O}$, denoted $(\text{INV}, p, op, O)$; the receipt by $p$ of a response $res$ from an object $O \in \mathcal{O}$, denoted $(\text{RES}, p, res, O)$; or the crash of $p$, denoted $(\text{CRASH}, p)$. If $p$ executes a crash step, it does not subsequently execute other steps.

## 2.1 Histories over a set of objects

Let $\mathcal{O}' \subseteq \mathcal{O}$. A *history $H$ over $\mathcal{O}'$* is a (finite or infinite) sequence of invocation and response steps involving objects in $\mathcal{O}'$, as well as crash steps of processes that access objects in $\mathcal{O}'$. Such a history must satisfy the following well-formedness property: for each process $p \in \Pi$, the subsequence of $H$ involving the steps of $p$ consists of zero or more pairs of invocation and response steps, each pair accessing the same object (the invocation and response steps involved in each pair are said to be *matching*), possibly followed by an invocation step, possibly followed by $p$'s crash step. If step $(\text{CRASH}, p)$ is in $H$, then *$p$ crashes in $H$*; otherwise it is *correct in $H$*. *$H$ is a history of object $O$* if it is a history over $\{O\}$.

## 2.2 History completions

An *operation execution opx* in a history $H$ (over a set of objects) is either a pair consisting of an invocation of an operation and its matching response in $H$, in which case we say that *opx* is *complete* in $H$; or an invocation in $H$ that has no matching response in $H$, in which case we say that *opx* is *pending* in $H$. A history $H$ is *complete* if all the operation executions in $H$ are complete. A history $H$ is *wait-free* if all the operation executions invoked by *correct* processes in $H$ are complete.

Let $X$ be a set of operation executions that are pending in $H$. History $H'$ is an *$X$-completion of $H$* if it can be obtained from $H$ as follows. For each *opx* $\in X$, either remove the invocation of *opx*, or insert a matching response anywhere after the invocation of *opx*. If a matching response for *opx* is inserted and the process that invoked *opx* crashed in $H$, move this crash step to after the inserted response in $H'$; this ensures that $H'$ is well-formed.

$H'$ is a *completion of $H$* if it is an $X$-completion of $H$ where $X$ is the set of *all* pending operation executions in $H$. $H'$ is a *crash-completion of $H$* if it is an $X$-completion of $H$ where $X = \{$*opx* : *opx* is pending and the process that invoked *opx* crashes in $H\}$; i.e., $X$ is the set of operations executions that are pending due to the crashes of the invoking processes.

## 2.3 Object types

An *object type* $T$ is specified by a tuple $(OP, RES, Q, \delta)$, where $OP$ is a set of operations, $RES$ is a set of responses, $Q$ is a set of states, and $\delta \subseteq Q \times OP \times \{1, \ldots, n\} \times Q \times RES$ is a state transition relation. Intuitively, $(s, op, i, s', res)$ is in $\delta$ means that if process $p_i$ applies operation $op \in OP$ on an object $O$ of type $T$ that is in state $s$, $O$ may move to state $s'$ and return response $res$. We assume that $\delta$ is total, i.e., for every $s \in Q$, $op \in O$, and $i \in \{1, \ldots, n\}$, there is an $s' \in Q$ and a $res \in RES$ such that $(s, op, i, s', res) \in \delta$.

**Abortable object types.** Given any object type $T$, we now define the abortable counterpart $T_a$ of $T$. Intuitively, if a process applies an operation $op$ on an object $O$ of type $T_a$, this operation may complete "normally" or it may "abort". If $op$ completes normally, $O$ undergoes a state transition specified by $T$ and returns the corresponding response.[1] If $op$ "aborts", such a state transition may or may not occur, and $O$ returns a distinguished response $\perp$.

Formally, for each type $T$, the *abortable counterpart* of $T$, denoted $T_a$, is defined as follows. If $T = (OP, RES, Q, \delta)$, then $T_a = (OP, RES', Q, \delta')$ such that $RES' = RES \cup \{\perp\}$ where $\perp \notin RES$, and for every tuple $(s, op, i, s', res)$ in $\delta$, the state transition $\delta'$ contains the following tuples:

i. $(s, op, i, s', res)$,
ii. $(s, op, i, s', \perp)$, and
iii. $(s, op, i, s, \perp)$.

These three tuples of $\delta'$ correspond to $op$ completing normally, and $op$ aborting with or without the state transition having occurred. We say that $T_a$ is an *abortable type*.

With an abortable object type, if a process invokes an operation on an object and the operation aborts (i.e., returns $\perp$), then the process does not know whether the object underwent a state transition and if so, what the corresponding response would have been. This can be problematic, and motivates the introduction of *query*-abortable types.

**Query-abortable object types.** Given any object type $T$, we define the query-abortable counterpart $T_{qa}$ of $T$. Intuitively, an object $O$ of type $T_{qa}$ behaves like one of type $T_a$: each operation of $T$ may complete normally or abort, and each aborted operation may cause a state transition or not. In addition, object $O$ has an operation called QUERY that any process can invoke to determine its last non-QUERY operation on $O$ that caused a state transition, and the response associated with this state transition. As with all other operations, however, a QUERY operation may abort.

For the QUERY operation to be useful, the operations that $p$ applies on $O$ have to be distinguishable, i.e., unique. To see this, sup-

[1]The occurrence of a state transition does not imply a state change. For example, if a read operation completes normally, a state transition occurs, but the state of the object does not change.

pose $O$ is a query-abortable queue and consider the following execution. First $p$ applies ENQUEUE(3) on $O$, the operation completes normally, and $O$ returns *ok*. Then $p$ applies ENQUEUE(3) again, the operation aborts, and $O$ returns $\perp$. Now $p$ applies the QUERY operation to try to determine whether the second ENQUEUE(3) actually took effect, and $O$ replies "ENQUEUE(3) is the last operation that took effect and the corresponding response is *ok*". With such a reply, $p$ cannot determine whether its second ENQUEUE(3) took effect or not.

To address this problem, non-QUERY operations in query-abortable type $T_{qa}$ are of the form $\langle op, id \rangle$, where *op* is an operation of $T$ and *id* is an identifier: each process can use unique *id*s to distinguish the operations that it applies. In the example above, $p$ now applies $\langle \text{ENQUEUE}(3), 1\rangle$ followed by $\langle \text{ENQUEUE}(3), 2\rangle$, and then QUERY. The possible replies from this QUERY are now unambiguous: for example, if the reply is "$\langle \text{ENQUEUE}(3), 2\rangle$ is the last operation that took effect and the corresponding response is *ok*", $p$ knows that its aborted ENQUEUE actually took effect.

We can now precisely describe the behaviour of QUERY. Suppose a process $p$ applies a QUERY operation on an object $O$ of type $T_{qa}$. The state of $O$ does not change, and the response of $O$ to this QUERY is (non-deterministically chosen to be) either $\perp$ or a tuple $\langle op, id, res\rangle$ where: $\langle op, id \rangle$ is the last *non*-QUERY operation that $p$ applied to $O$ *that caused a state transition*, and $res$ is the response corresponding to this state transition (if $op$ does not exist then QUERY returns an arbitrary response).

Formally, for each type $T$, the *query-abortable counterpart* of $T$, denoted $T_{qa}$, is defined as follows. If $T = (OP, RES, Q, \delta)$, then $T_{qa} = (OP', RES', Q', \delta')$ such that:

1. $OP' = \{OP \times \mathbb{N}\} \cup \{\text{QUERY}\}$, where QUERY $\notin OP$.
2. $RES' = RES \cup \{OP \times \mathbb{N} \times RES\} \cup \{\perp\}$, where $\perp \notin RES$.
3. $Q' = Q \times (OP \times \mathbb{N} \times RES)^n$: intuitively, a state in $Q'$ records a state in $Q$, and for each process $p_i$, it also records the last non-QUERY operation by $p_i$ that caused a state transition (including its id), and the response corresponding to the state transition that occurred.
4. $\delta'$ is defined as follows.
   a. For every $(s, op, i, s', res) \in \delta$, every $R$ such that $(s, R) \in Q'$, and every *id* $\in \mathbb{N}$, $\delta'$ contains:
      i. $((s, R), (op, id), i, (s', R'), res)$,
      ii. $((s, R), (op, id), i, (s', R'), \perp)$, and
      iii. $((s, R), (op, id), i, (s, R), \perp)$,
      where $R'[i] = (op, id, res)$, and for all $j \neq i$, $R'[j] = R[j]$.
   b. For every $(s, R) \in Q'$, $\delta'$ contains:
      iv. $((s, R), \text{QUERY}, i, (s, R), R[i])$, and
      v. $((s, R), \text{QUERY}, i, (s, R), \perp)$.

The first three tuples of $\delta'$ correspond to a non-QUERY operation that completes normally, or aborts with or without the state transition having occurred. The fourth and fifth tuples of $\delta'$ correspond to a QUERY operation that completes normally or aborts.

It is worth noting that the query-abortable counterpart $T_{qa}$ of any type $T$ is also an abortable type, i.e., $T_{qa} = (T')_a$ for some type $T'$ (see Lemma 7 in Section 6).

## 2.4 Object behaviour

An object is defined by specifying how it should behave when operations are applied sequentially or concurrently. This is done by giving the object's type, the object's initial state, and some additional requirements that the object should satisfy, such as linearizability [4] and non-triviality (which we define later in this section).

To define object behaviour, it is useful to define the following concepts. Let *opx* and *opx*′ be operation executions in a history $H$ over a set of objects. We say that *opx precedes opx*′ *in* $H$ if *opx* is complete in $H$ and the response of *opx* precedes the invocation of *opx*′ in $H$. Two operation executions are *concurrent in* $H$ if neither precedes the other in $H$. A history $H$ is *sequential* if it has no concurrent operation executions.

Throughout this subsection, fix an object $O$ and a finite or infinite history $H$ of $O$.

**Conformity to a type.** Suppose $H$ is a complete and sequential history. $H$ induces a sequence of triples $(q_1, op_1, res_1)$, $(q_2, op_2, res_2), \ldots$, where $op_i$ is the $i$-th operation applied to $O$ in $H$, $q_i$ is the process that applied this operation, and $res_i$ is the response of $O$ to this operation. Let $T = (OP, RES, Q, \delta)$ be a type and $\sigma \in Q$ be a state of $T$. *$H$ conforms to type $T$ initialized to $\sigma$*, if there exists a sequence of states $\sigma_0, \sigma_1, \sigma_2, \ldots$ such that $\sigma_0 = \sigma$, and for each $i \geq 1$, $(\sigma_{i-1}, op_i, j, \sigma_i, res_i) \in \delta$, where $q_i = p_j$.

**Linearizability.** Intuitively, history $H$ is linearizable with respect to a type if every operation execution in $H$ appears to take effect instantaneously, at some point during the operation's execution interval, according to the type.

More precisely, we say that $H$ is *linearizable with respect to type $T$ initialized to $\sigma$* if the following holds. There is a completion $H'$ of $H$ and a sequential history $S$ such that: (a) $S$ and $H'$ have the same steps, (b) $S$ conforms to type $T$ initialized to $\sigma$, and (c) if an operation execution *opx* precedes an operation execution *opx*′ in $H'$ then *opx* also precedes *opx*′ in $S$.

**Non-triviality.** An object $O$ of abortable or query-abortable type can abort, i.e., return $\perp$, to any operation that is applied on it, *even if this operation is not concurrent with any other operation on $O$*. To prevent this trivial behaviour, we could require that an operation on $O$ return $\perp$ only if it is concurrent with some other operation on $O$.

This simplistic requirement, however, is not quite right. To see this, suppose a process invokes an operation and then crashes before the response. This results in an operation execution that is pending "forever", and thus, may be concurrent with an infinite number of other operation executions. So this requirement allows a single process crash to cause an infinite number of operations to abort; in other words, it allows a process that crashes to "interfere" forever with the execution of other operations. This is undesirable: abortable objects conforming to this definition would be very weak and trivial to implement using locks.

With our definition of non-triviality, a process that crashes may interfere only for a *finite* amount of time. To achieve this, given any history, we first complete every operation that is pending *due to the crash of the invoking process* in that history, or remove it altogether. We then require that, in this crash-completed history, each aborted operation is concurrent with some other operation. This effectively ensures that operations invoked by crashed processes have a finite duration and do not interfere forever.

More precisely, $H$ is *non-trivial* if there is a crash-completion $H'$ of $H$ such that every operation execution that returns $\perp$ in $H'$ is concurrent with some other operation execution in $H'$.

**Combining non-triviality and linearizability.** Suppose $H$ is linearizable and non-trivial. By the definitions above: (a) there is a completion of $H$ that satisfies some "linearizability" property, and (b) there is a crash-completion of $H$ that satisfies some "non-triviality" property. These two completions, however, may be incompatible. For example, the completion that satisfies linearizability may have removed a pending operation execution *opx* (because it did not take effect), while the completion that satisfies non-

triviality kept *opx* (because it caused another operation to abort). The question arises whether there is a single completion of $H$ that, in a precise sense, satisfies both properties at once. In the full paper we show that this is indeed the case, provided that all the operation executions in $H$ that are invoked by *correct* processes are complete, i.e., $H$ is wait-free.

## 2.5 Object implementations

An *implementation* $I$ of a *target object* $O$ from a list of *base objects* $(O_1, O_2, \ldots)$ is specified by a list of procedures $P_{op,i}$, one for every operation-process pair. Procedure $P_{op,i}$ specifies how the operation *op* applied by $p_i$ on $O$ should be carried out in terms of operations applied on the objects $O_1, O_2, \ldots$. Process $p_i$ invokes the operation *op* by starting to execute the procedure $P_{op,i}$. The value returned from this procedure is deemed to be the response from $O$. For the rest of this subsection, $I$ is an implementation of an object $O$ from objects $O_1, O_2, \ldots$.

In an *execution* $E$ of implementation $I$, each process $p_i$ applies a sequence of zero or more operations on the target object $O$, and carries out each operation by accessing base objects as prescribed by the procedures $P_{op,i}$. An execution $E$ of $I$ is *fair* if every correct process $p_i$ that executes a procedure $P_{op,i}$ of $I$ is not prevented by the scheduler from taking its next step in the procedure.

From each execution $E$ of $I$, we can extract the (finite or infinite) sequence $H$ of all the invocation and response steps involving objects $O$ and $O_1, O_2, \ldots$, as well as the crash steps of all the processes that invoke at least one operation on $O$; $H$ is called the *history induced by execution* $E$.

Let $H$ be the history induced by an execution $E$ of $I$, and $X$ be the target object or a base object of $I$. $H|X$ is the subsequence of all invocation and response steps in $H$ that involve $X$, and the crash steps in $H$ of all processes that invoke at least one operation on $X$. Clearly, $H|X$ is a history of object $X$.

We now explain what it means to implement a target type $T$ from a list of base types. Intuitively, such an implementation must be linearizable w.r.t. $T$, non-trivial, and wait-free, under the assumption that the base objects themselves are linearizable w.r.t. the base types, non-trivial, and wait-free.

More precisely, let $T$ be a type and $\sigma$ be a state of $T$. Let $I$ be an implementation of an object $O$ from base objects $(O_1, O_2, \ldots)$. We say that $I$ is *a (linearizable, non-trivial, and wait-free) implementation of $T$ initialized to $\sigma$ from types $(T_1, T_2, \ldots)$ initialized to* $(\sigma_1, \sigma_2, \ldots)$ if for every fair execution $E$ of implementation $I$, the history $H$ induced by $E$ satisfies the following property: If, for every base object $O_j$, $H|O_j$ is (a) linearizable w.r.t. type $T_j$ initialized to $\sigma_j$, (b) non-trivial, and (c) wait-free, then $H|O$ is (a) linearizable w.r.t. type $T$ initialized to $\sigma$, (b) non-trivial, and (c) wait-free.

# 3. ABORTABLE REGISTERS

The (multi-valued) *register* type supports operations READ and WRITE($v$), where $v$ is a value from some arbitrary domain. A WRITE($v$) operation changes the state of the register to $v$ and returns OK, and a READ operation returns the current state of the register.

With the *abortable register* type, a WRITE($v$) operation either completes normally, in which case it changes the state of the register to $v$ and returns OK, or it aborts and returns $\perp$. In the latter case, the WRITE($v$) operation may change the state of the register to $v$ or leave the state unchanged (note that the writer does not know whether the state changed). A READ operation returns either the current state of the register or $\perp$; in both cases, the state of the register does not change.

The following theorem implies that abortable registers are strictly weaker than registers.

THEOREM 1. *There is no implementation of the register type from abortable registers.*

Informally, we prove Theorem 1 by contradiction: we assume there is an implementation $I$ of register $R$ from abortable registers, and we construct an execution of $I$ where a write to $R$ never completes (because it requires an infinite number of operations on abortable registers); this contradicts the wait-freedom of $I$. More precisely, we construct an execution where a process $p_r$ performs an infinite number of reads on $R$, and another process $p_w$ performs a single write to $R$. In order for the write to $R$ to take effect, $p_w$ must change the state of at least one abortable register that is subsequently read by $p_r$ (otherwise, $p_r$ would never be able to determine the new state of $R$). The constructed execution, however, can ensure that $p_w$ only changes the state of abortable registers that are not subsequently read by $p_r$. Intuitively, when $p_w$ invokes an operation $op_w$ on some abortable register $R_a$, where $p_r$ later performs an operation $op_r$ on $R_a$, we can delay the response received by $p_w$ until $op_w$ is concurrent with $op_r$; thus, it is possible that $op_w$ returns $\perp$ and does not change the state of $R_a$. Furthermore, it is possible that $op_r$ returns a value that is not $\perp$, and thus, $p_r$ never notices that there are concurrent operations. Using this technique, we create an execution where the write to $R$ by $p_w$ never completes.

It is well-known that (multi-writer multi-reader multi-valued) registers can be implemented from single-writer single-reader boolean safe registers [6]. Together with Theorem 1 this implies:

COROLLARY 2. *There is no implementation of the single-writer single-reader boolean safe register type from abortable registers.*

In the full paper we show the following result.

THEOREM 3. *There is an implementation of the multi-writer multi-reader multi-valued abortable register type from single-writer single-reader multi-valued abortable registers.*

# 4. A TIMESTAMPING MECHANISM

Our universal constructions rely on a new object type called *inc&read*. Intuitively, *inc&read* returns timestamps that can be used to determine whether any two code segments execute concurrently or one after the other. We first define (abortable) *inc&read* and then show how it can be implemented using (abortable) registers.

## 4.1 The *inc&read* type

Intuitively, *inc&read* is a counter accessed via the following two operations: INC&READ, which increases the counter by some unknown arbitrary amount and returns the new value of the counter; and READ, which returns the value of the counter.

Formally, the type *inc&read* is the tuple $(OP, RES, Q, \delta)$, where $OP = \{\text{INC\&READ}, \text{READ}\}$, $RES = Q = \mathbb{Z}$, and the transition function $\delta$ contains the following tuples: $(s, \text{INC\&READ}, i, s+k, s+k)$ and $(s, \text{READ}, i, s, s)$, for every $s \in Q$, $1 \leq i \leq n$ and integer $k > 0$.

In this paper, we use the *abortable* counterpart of *inc&read*. With abortable *inc&read*, every operation behaves as described above or it aborts, i.e., it returns $\perp$. An INC&READ that aborts may or may not increase the counter; a READ that aborts does not change the counter.

**Algorithm 1** Abortable *inc&read* implementation

Shared variables and initial values:
$R_j = -1$ for $j = 1..n$

Code for process $p_i$:

**procedure** INC&READ()
1: **for** $j \leftarrow 1$ to $n$ **do**
2: $count_j \leftarrow$ READ($R_j$)
3: **if** $count_j = \perp$ **then return** $\perp$
4: $c \leftarrow \max_{1 \le j \le n}\{count_j\} + 1$
5: $status \leftarrow$ WRITE($R_i, c$)
6: **if** $status = \perp$ **then return** $\perp$
7: **return** $n \cdot c + i$

**procedure** READ()
1: **for** $j \leftarrow 1$ to $n$ **do**
2: $count_j \leftarrow$ READ($R_j$)
3: **if** $count_j = \perp$ **then return** $\perp$
4: $c \leftarrow \max_{1 \le j \le n}\{count_j\}$
5: *id* $\leftarrow \max_{1 \le j \le n}\{j \mid count_j = c\}$
6: **return** $n \cdot c +$ *id*

**Use of *inc&read*.** We use abortable *inc&read* to detect whether some segments of code are executed concurrently or one after the other. To do so, we "wrap" each such segment of code $\sigma$ by inserting an INC&READ and a READ just before and after $\sigma$, respectively: intuitively, for each execution of $\sigma$, INC&READ and READ return the "time" when this execution started and completed, respectively. This timestamping mechanism provides the following two useful properties. If a segment of code $\sigma$ together with its wrapper is executed in isolation, then this execution starts and terminates with the same, non-$\perp$ timestamp. Furthermore, if the execution of a segment $\sigma$ completes at "time" $t$ and the execution of a segment $\sigma'$ starts at "time" $t' > t$, then $\sigma$ is executed before $\sigma'$.

## 4.2 Implementing *inc&read* using registers

Algorithm 1 shows an implementation of (abortable) *inc&read* from (abortable) registers $R_1, \ldots, R_n$, where $R_i$ is written only by $p_i$ and read by all $n$ processes. Intuitively, $R_i$ contains the "local counter" of $p_i$ , and the value of the implemented *inc&read* counter is derived from the local counters as follows: it is an integer encoding of the pair $(c, i)$ where $c$ is the largest local counter and $i$ is the largest index of a process with local counter $c$. The encoding that we use here is $n \cdot c + i$.

To execute INC&READ, a process $p_i$ reads the local counters $R_1, \ldots, R_n$, determines the largest value and adds 1 to it. Then $p_i$ writes this new value, $c$, to register $R_i$ and returns $n \cdot c + i$. To execute READ, $p_i$ reads the local counters $R_1, \ldots, R_n$ and determines the largest local counter value $c$. Process $p_i$ then finds the largest index *id* of a process with local counter $c$ and it returns $n \cdot c +$ *id*. Operations INC&READ and READ abort if any read or write on any register $R_j$ aborts. This can occur only if the base registers $R_1, \ldots, R_n$ are abortable.

THEOREM 4. *Algorithm 1 is an implementation of inc&read initialized to* 0 *from registers* $R_1, \ldots, R_n$ *initialized to* $-1$. *Moreover, if* $R_1, \ldots, R_n$ *are abortable registers, then this algorithm implements abortable inc&read.*

By suitably choosing the initial values of the base registers $R_1, \ldots, R_n$ in Algorithm 1, we can initialize *inc&read* to any desired value.

**Algorithm 2** Universal construction for type $T_a$

Shared variables and initial values:
$R_j = \langle \sigma, 0 \rangle$ for $j = 1..n$
$C = 0$

Code for process $p_i$:

**procedure** EXECUTE(*op*)
1: *ts* $\leftarrow$ INC&READ($C$)
2: **if** $ts = \perp$ **then return** $\perp$
3: **for** $j \leftarrow 1$ to $n$ **do**
4: $r \leftarrow$ READ($R_j$)
5: **if** $r = \perp$ **then return** $\perp$
6: $\langle s_j, ts_j \rangle \leftarrow r$
7: let $\langle s_k, ts_k \rangle$ be such that $ts_k = \max_{1 \le j \le n}\{ts_j\}$
8: $\langle s, res \rangle \leftarrow$ APPLY$_T$($s_k$, *op*, $i$)
9: $status \leftarrow$ WRITE($R_i, \langle s,$ *ts*$\rangle$)
10: **if** $status = \perp$ **then return** $\perp$
11: *ts*$' \leftarrow$ READ($C$)
12: **if** $ts' \neq ts$ **then return** $\perp$
13: **return** *res*

# 5. IMPLEMENTING ABORTABLE TYPES

We now describe a universal construction for abortable types, denoted $\mathcal{U}_{T \to T_a}$. Given any type $T$, $\mathcal{U}_{T \to T_a}$ yields an implementation of $T_a$, the abortable counterpart of $T$. This universal construction is efficient: it uses only $2n$ abortable registers, and operations take $O(n)$ steps. It is also simpler than Herlihy's universal construction for arbitrary types from consensus and registers [2]. In Herlihy's construction, a linked list keeps a history of all operations applied to the object, and processes use consensus to agree on the next item on the list. In our universal construction, processes apply operations *in place*, without keeping a history.

$\mathcal{U}_{T \to T_a}$ is built modularly: it consists of Algorithm 2, which implements $T_a$ using $n$ abortable registers and one abortable *inc&read*, composed with Algorithm 1, which implements abortable *inc&read* using $n$ abortable registers.

As in Herlihy's universal construction, $\mathcal{U}_{T \to T_a}$ uses a procedure APPLY$_T$ that represents the sequential behaviour of type $T$: for every state $s$, operation *op*, and process $p_i$, APPLY$_T(s,$ *op*$, i)$ returns a pair $(s',$ *res*$)$ such that $(s,$ *op*$, i, s',$ *res*$) \in \delta_T$, where $\delta_T$ is the state transition function of $T$. Intuitively, if $p_i$ applies *op* on an object of type $T$ in state $s$, the object may change its state to $s'$ and reply *res*.

We now describe Algorithm 2. Given any type $T$, this algorithm implements $T_a$ from abortable registers $R_1, \ldots, R_n$ and abortable *inc&read* $C$. Registers $R_1, \ldots, R_n$ store the state of $T$ with a freshness timestamp, as observed by processes $p_1, \ldots, p_n$, respectively. To execute an operation *op*, a process $p_i$ first obtains a new timestamp $ts$ using INC&READ($C$). Next, $p_i$ reads registers $R_1, \ldots, R_n$ to find a state $s_k$ with the largest timestamp and applies *op* to $s_k$; this results in a new state $s$ and a response $res$. Then $p_i$ writes $s$ with timestamp $ts$ into $R_i$. Finally, $p_i$ executes READ($C$) to check whether $C$ is still equal to $ts$. If it is, $p_i$ returns response $res$; otherwise, $p_i$ aborts. During this algorithm's execution, if any operation on some register $R_j$ or on $C$ aborts, then $p_i$ also aborts.

We now provide some intuition as to why Algorithm 2 satisfies all the desirable properties, namely, wait-freedom, non-triviality, and linearizability, provided the base objects also satisfy these properties. The algorithm is wait-free because is has no unbounded loops. To see that it is non-trivial, suppose that $p$ executes procedure EXECUTE(*op*) in isolation. We show that this operation

execution does not return $\perp$. Since $p$'s operations on the base objects are executed in isolation and the base objects are non-trivial, they do not return $\perp$ in procedure EXECUTE(*op*). Thus, $p$'s operation execution does not return $\perp$ on lines 2, 5, or 10. Moreover, since $p$ executes lines 1–11 in isolation, by the first property of the timestamping mechanism of Section 4, $p$ gets the same timestamp from INC&READ($C$) on line 1 and READ($C$) on line 11. Thus, $p$'s operation execution does not return $\perp$ on line 12. So, $p$'s operation execution returns *res* on line 13, and *res* $\neq \perp$ since the value of *res* comes from APPLY$_T$ on line 8.

To see why Algorithm 2 is linearizable with respect to type $T_a$ initialized to state $\sigma$, consider any execution $E$ of the algorithm. In general, $E$ involves multiple and possibly concurrent executions of procedure EXECUTE(*op*). Operation executions in $E$ have a timestamp, the value returned by the INC&READ operation on line 1. We want to show that there is a linearization of $E$: an ordering of its operation executions that (a) conforms to $T_a$ initialized to $\sigma$, and (b) respects the order of non-concurrent operation executions. As we will see, ordering the operation executions by increasing timestamp yields such a linearization.

Roughly speaking, in Algorithm 2 a process determines the "current state" of type $T_a$, and based on this state, it produces a new state. We can therefore define a "dependency graph" for the operation executions that appear in $E$: in this graph, every operation execution is (represented by) a node. In what follows, we use the words "node" and "operation execution" interchangeably. Thus, each node has a timestamp, and is classified as either *successful* (if it returns a value other than $\perp$) or *unsuccessful* (if it returns $\perp$). We put a directed edge from node $u$ to node $v$ if and only if operation execution $v$ computes its new state on line 8 based on the state written by operation execution $u$ on line 9.

Since each node has at most one incoming edge and the dependency graph has no cycles, it is actually a forest. A source of the forest, i.e., a node with no incoming edge, is an operation execution that bases the new state it produces on the initial state $\sigma$. In the full paper we prove that (1) the dependency forest contains a path, called the *critical path*, that starts at a source, includes *all* successful nodes, and if the forest has only a finite number of successful nodes, ends at a successful node; and (2) the timestamps of nodes on the critical path increase monotonically.

Consider now the first requirement of a linearization, i.e., ordering the operation executions so that the resulting sequence conforms to type $T_a$ initialized to $\sigma$. Intuitively, an edge from node $u$ to node $v$ in the dependency forest means that $v$ is applied on the state that $u$ produces. Thus, if $v$ is a successful operation execution, it should be ordered after $u$; moreover, $u$ must cause a state transition (to produce the state seen by $v$), even if it is an unsuccessful operation execution. Therefore, the nodes on the critical path form a sequence that conforms to type $T_a$ initialized to $\sigma$, since we can think of the unsuccessful nodes on the critical path as aborted operation executions that cause a state transition. We need, however, to produce a sequence of *all* nodes of the dependency forest, not only those on the critical path. By (1), all nodes outside the critical path are unsuccessful and not depended upon by any successful node. So these unsuccessful nodes have no visible effect. Thus, we can think of them as aborted operation executions that do not cause a state transition, and can order them arbitrarily among the critical path nodes, with the resulting sequence still conforming to type $T_a$ initialized to $\sigma$.

By (2), nodes on the critical path are in timestamp order. Thus, if we order all nodes in the dependency forest by increasing timestamp, the resulting sequence conforms to type $T_a$ initialized to $\sigma$. This sequence also satisfies the second requirement of a linearization, i.e., respecting the order of non-concurrent operation executions. This is because if an operation execution completes before another starts, the former has a smaller timestamp than the latter. Thus, we can linearize $E$ by ordering its operation executions in timestamp order.

THEOREM 5. *For any type $T$, Algorithm 2 is an implementation of abortable type $T_a$ initialized to $\sigma$ from $n$ abortable registers initialized to $\langle \sigma, 0 \rangle$ and an abortable inc&read initialized to* 0.

By composing Algorithm 2 with Algorithm 1, which implements abortable *inc&read* from $n$ abortable registers (see Theorem 4), we have:

THEOREM 6. *There is a universal construction $\mathcal{U}_{T \to T_a}$ which, for any type $T$, implements $T_a$ from $2n$ abortable registers.*

# 6. IMPLEMENTING QUERY-ABORTABLE TYPES

We now present a universal construction for query-abortable types, denoted $\mathcal{U}_{T \to T_{qa}}$. Given any type $T$, $\mathcal{U}_{T \to T_{qa}}$ yields an implementation of $T_{qa}$, the query-abortable counterpart of $T$. To describe $\mathcal{U}_{T \to T_{qa}}$, we first prove that every type $T$ has a "queriable" counterpart $T_q$ such that $T_{qa} = (T_q)_a$, i.e., $T_{qa}$ is the abortable counterpart of $T_q$. Thus, given any type $T$ (with an associated APPLY$_T$ procedure) we can derive an implementation of $T_{qa}$ in two steps: (a) we first use the given APPLY$_T$ procedure to derive an APPLY$_{T_q}$ procedure for type $T_q$, and (b) we then use APPLY$_{T_q}$ in any universal construction for abortable types, such as $\mathcal{U}_{T \to T_a}$ described in Section 5. This results in an implementation of $(T_q)_a$, i.e., of $T_{qa}$.

**Queriable object types.** Given any object type $T$, we now define the queriable counterpart $T_q$ of $T$. Intuitively, $T_q$ is similar to the query-abortable counterpart $T_{qa}$ of $T$ except that operations do not abort. As in $T_{qa}$, the non-QUERY operations of $T_q$ are of the form $\langle op, id \rangle$, where *op* is an operation of $T$ and *id* is an integer identifier. More precisely, an object $O$ of type $T_q$ behaves like one of type $T$, but it has an additional operation called QUERY: if a process applies QUERY to $O$, object $O$ returns a tuple $\langle op, id, res \rangle$, where $\langle op, id \rangle$ is the last *non*-QUERY operation that $p$ applied to $O$ and $res$ is the corresponding response. The formal specification of $T_q$, given below, is derived from the specification of $T_{qa}$ by removing the option to abort.

For each type $T$, the *queriable counterpart* of $T$, denoted $T_q$, is defined as follows. If $T = (OP, RES, Q, \delta)$, then $T_q = (OP', RES', Q', \delta')$ such that:

1. $OP' = \{OP \times \mathbb{N}\} \cup \{\text{QUERY}\}$, where QUERY $\notin OP$.
2. $RES' = RES \cup \{OP \times \mathbb{N} \times RES\}$.
3. $Q' = Q \times (OP \times \mathbb{N} \times RES)^n$.
4. $\delta'$ is defined as follows.
   a. For every $(s, op, i, s', res) \in \delta$, every $R$ such that $(s, R) \in Q'$, and every *id* $\in \mathbb{N}$, $\delta'$ contains the tuple $((s, R), (op, id), i, (s', R'), res)$, where $R'[i] = (op, id, res)$, and for all $j \neq i$, $R'[j] = R[j]$.
   b. For every $(s, R) \in Q'$, $\delta'$ contains the tuple $((s, R), \text{QUERY}, i, (s, R), R[i])$.

From the definitions of the queriable, abortable, and query-abortable counterparts of any type $T$, it is straightforward to verify that:

LEMMA 7. *For every type $T$, $T_{qa} = (T_q)_a$.*

**Algorithm 3** Procedure $\text{APPLY}_{T_q}$ derived from $\text{APPLY}_T$

**procedure** $\text{APPLY}_{T_q}(s, op, i)$
1: $\langle s', R\rangle \leftarrow s$
2: **if** $op = \text{QUERY}$ **then return** $\langle s, R[i]\rangle$
3: $\langle op', id\rangle \leftarrow op$
4: $\langle s', res\rangle \leftarrow \text{APPLY}_T(s', op', i)$
5: $R[i] \leftarrow \langle op', id, res\rangle$
6: $s \leftarrow \langle s', R\rangle$
7: **return** $\langle s, res\rangle$

**Using $T_q$ to implement universal construction $\mathcal{U}_{T\to T_{qa}}$.** Given any type $T$ with an associated procedure $\text{APPLY}_T$, we can get an implementation of $T_{qa}$ as follows. We first use the given $\text{APPLY}_T$ procedure to derive an $\text{APPLY}_{T_q}$ procedure for type $T_q$ (the construction of $\text{APPLY}_{T_q}$ from $\text{APPLY}_T$, given by Algorithm 3, is straightforward). We then use $\text{APPLY}_{T_q}$ in any universal construction for abortable types to obtain the abortable counterpart of $T_q$. This results in an implementation of $(T_q)_a$, and by Lemma 7, $(T_q)_a = T_{qa}$.

If the universal construction for abortable types used above is $\mathcal{U}_{T\to T_a}$ (described in Section 5) then by Theorem 6, we get a universal construction for *query*-abortable types that uses $2n$ abortable registers. Thus:

THEOREM 8. *There is a universal construction $\mathcal{U}_{T\to T_{qa}}$ which, for any type $T$, implements $T_{qa}$ from $2n$ abortable registers.*

# 7. ALTERNATE QUERY-ABORTABLE TYPE

In this section we describe and show how to implement $T_{QA}$ — a natural variant of the query-abortable type $T_{qa}$. Like $T_{qa}$, type $T_{QA}$ is strong enough to support the implementation of obstruction-free and pausable types, but it is simpler and slightly easier to use than $T_{qa}$. In particular, processes that use objects of type $T_{QA}$ can apply the original operations of $T$ *without* padding them with unique *id*s (recall that this padding is required to use objects of type $T_{qa}$). There is a trade-off, however: in contrast to $T_{qa}$, $T_{QA}$ is *not* an abortable type. More precisely, there is no type $T'$ such that $T_{QA} = (T')_a$. We now describe $T_{QA}$ and how to implement it using abortable registers.

**Alternate query-abortable types.** Roughly speaking, given a type $T$, an object of type $T_{QA}$ behaves like one of type $T$ except that its operations may abort (with or without taking effect) and there is an additional operation called QUERY. A process can call QUERY to determine the fate of its *last* non-QUERY operation *op* on the object: if *op* took effect, then QUERY returns the response that should have been returned by *op*; otherwise, QUERY returns a special value $\mathcal{F}$. As with all other operations, QUERY can also abort.

Formally, for each type $T$, type $T_{QA}$, is defined as follows. If $T = (OP, RES, Q, \delta)$, then $T_{QA} = (OP', RES', Q', \delta')$ such that:

1. $OP' = OP \cup \{\text{QUERY}\}$, where $\text{QUERY} \notin OP$.
2. $RES' = RES \cup \{\perp, \mathcal{F}\}$, where $\perp \notin RES$ and $\mathcal{F} \notin RES$.
3. $Q' = Q \times (RES \cup \{\mathcal{F}\})^n$.
4. $\delta'$ is defined as follows.
   a. For every $(s, op, i, s', res) \in \delta$ and every $R$ such that $(s, R) \in Q'$, $\delta'$ contains:
      i. $((s, R), op, i, (s', R'), res)$, where $R'[i] = res$, and for all $j \neq i$, $R'[j] = R[j]$
      ii. $((s, R), op, i, (s', R'), \perp)$, where $R'[i] = res$, and for all $j \neq i$, $R'[j] = R[j]$; and
      iii. $((s, R), op, i, (s, R'), \perp)$, where $R'[i] = \mathcal{F}$, and for all $j \neq i$, $R'[j] = R[j]$.
   b. For every $(s, R) \in Q'$, $\delta'$ contains:
      iv. $((s, R), \text{QUERY}, i, (s, R), R[i])$
      v. $((s, R), \text{QUERY}, i, (s, R), \perp)$

**Implementing type $T_{QA}$.** Given any type $T$, it is easy to implement $T_{QA}$ from $T_{qa}$ as follows. Let $O_{QA}$ be the target object of type $T_{QA}$ and $O_{qa}$ be a base object of type $T_{qa}$. To apply a non-QUERY operation *op* on $O_{QA}$, $p_i$ increments a local counter $c_i$, applies $\langle op, c_i\rangle$ on $O_{qa}$, and returns the response of $O_{qa}$. To apply QUERY on $O_{QA}$, $p_i$ applies QUERY on $O_{qa}$. If $O_{qa}$ returns $\perp$ then $p_i$ returns $\perp$; if $O_{qa}$ returns a tuple $\langle op', k, res\rangle$ with $k = c_i$ then $p_i$ returns *res*, otherwise $p_i$ returns $\mathcal{F}$. Since $O_{qa}$ is of type $T_{qa}$, it can be implemented from $2n$ abortable registers (Theorem 8), and therefore:

THEOREM 9. *There is a universal construction $\mathcal{U}_{T\to T_{QA}}$ which, for any type $T$, implements $T_{QA}$ from $2n$ abortable registers.*

# 8. IMPLEMENTATION COMPOSABILITY

We now explain how to compose implementations, and give a desirable property of such compositions. Let $I$ be an implementation of an object $O$ from base objects $(O_1, O_2, \ldots)$.[2] It is easy to see that by replacing each base object $O_j$ used in implementation $I$, with an implementation $I_j$ of $O_j$ from some base objects $(O_1^j, O_2^j, \ldots)$, we get an implementation $I'$ of $O$ from base objects $(O_1^1, O_2^1, \ldots, O_1^2, O_2^2, \ldots)$. We say that $I'$ is obtained by composing $I$ and $I_1, I_2, \ldots$.

Intuitively, such a composition should preserve correctness, i.e., it should preserve linearizability, non-triviality, and wait-freedom. More precisely, we say that *compositions preserve correctness* if the following holds.

Let $I$, $I_1, I_2, \ldots$, and $I'$ be implementations as described above. Suppose that:

1. $I$ is a (linearizable, non-trivial, and wait-free) implementation of a type $T$ initialized to $\sigma$ from some base types $(T_1, T_2, \ldots)$ initialized to $(\sigma_1, \sigma_2, \ldots)$, and
2. for every base type $T_j$ in $I$, $I_j$ is a (linearizable, non-trivial, and wait-free) implementation of $T_j$ initialized to $\sigma_j$ from some types $(T_1^j, T_2^j, \ldots)$ initialized to $(\sigma_1^j, \sigma_2^j, \ldots)$.

Then, $I'$ is a (linearizable, non-trivial, and wait-free) implementation of type $T$ initialized to $\sigma$ from types $(T_1^1, T_2^1, \ldots, T_1^2, T_2^2, \ldots)$ initialized to $(\sigma_1^1, \sigma_2^1, \ldots, \sigma_1^2, \sigma_2^2, \ldots)$.

THEOREM 10. *Under our definition of non-triviality (based on* interval *contention), compositions preserve correctness.*

We now consider a different definition of non-triviality — one based on step contention instead of interval contention. When non-triviality is based on step contention, an operation on a non-trivial object $O$ can abort only if it is not executed "solo". More precisely, an operation execution *opx* on a non-trivial object $O$ invoked by some process can return $\perp$ only if, during the execution interval of *opx*, there is at least one step of another process executing an operation on $O$.

[2]Recall that $I$ is given by a list of procedures $P_{op,i}$, one for every operation-process pair, where $P_{op,i}$ specifies how the operation $op$ applied by $p_i$ on $O$ should be carried out in terms of operations applied on the objects $O_1, O_2, \ldots$.

**Algorithm 4** Implementation $I$ of $WL_a$ from registers $X$ and $Y$

Shared variables and initial values:
$X = 0$; $Y = 0$

| Code for process $p$: | Code for process $q$: |
|---|---|
| **procedure** COMPETE | **procedure** COMPETE |
| 1: $t \leftarrow$ READ($Y$) | 1: $t \leftarrow$ READ($X$) |
| 2: WRITE($X, 1$) | 2: WRITE($Y, 1$) |
| 3: $t' \leftarrow$ READ($Y$) | 3: $t' \leftarrow$ READ($X$) |
| 4: **if** $t = 0$ **and** $t' = 0$ **then return** "winner" | 4: **if** $t = 0$ **and** $t' = 0$ **then return** "winner" |
| 5: **else if** $t = 1$ **and** $t' = 1$ **then return** "loser" | 5: **else if** $t = 1$ **and** $t' = 1$ **then return** "loser" |
| 6: **else return** $\perp$ | 6: **else return** $\perp$ |

THEOREM 11. *Under the definition of non-triviality based on* step *contention, compositions do not preserve correctness.*

PROOF. In this proof we assume that non-triviality is defined in terms of step contention, and give an example of a composition of implementations that does *not* preserve correctness under this definition: even though we compose implementations that are correct (i.e., linearizable, non-trivial, and wait-free) the implementation that we obtain violates non-triviality.

Consider the *Winner-Loser* type *WL* which intuitively allows two processes $p$ and $q$ to compete and determine who is the "winner" and who is the "loser" (similar to two-process name consensus). To do so, $p$ and $q$ can each apply an operation called COMPETE: the first application returns "winner" and the second one returns "loser". This type is intended for a single competition: if a process applies the operation COMPETE more than once, the type gets "upset" and thereafter it non-deterministically returns either "winner" or "loser". Let $WL_a$ be the abortable counterpart of the Winner-Loser type, as defined in Section 2.3. Recall that in $WL_a$, each COMPETE operation may also abort and return $\perp$. A correct implementation of this type must be non-trivial, i.e., a COMPETE operation *op* by one process can return $\perp$ only if the other process takes a step during the execution of *op*.

In Algorithm 4, we give an implementation $I$ of an object $O$ of type $WL_a$ from atomic registers $X$ and $Y$. $X$ is "owned" by $p$ and $Y$ by $q$, and both registers are initialized to 0. To execute a COMPETE operation, a process reads the other process's register, writes 1 to its own register, and then reads again the other process's register. If a process reads 0 both times, it is the winner; if it reads 1 both times, it is the loser; otherwise, it detects a concurrent COMPETE operation and aborts. It can be shown that this implementation is correct: intuitively, the target object $O$ is wait-free, linearizable w.r.t. $WL_a$, and non-trivial, provided that $X$ and $Y$ behave like atomic registers. In particular, with this implementation, a COMPETE operation *op* by one process can return $\perp$ only if the other process takes a step during the execution of *op*.

Figure 1(a) illustrates an execution of this implementation. In this execution, there are steps of $q$ during the operation of $p$; so $p$ is allowed to abort, and in fact, it returns $\perp$. On the other hand, there are no steps of $p$ during $q$'s operation, and so $q$'s operation *cannot* abort (and in fact, it returns "loser").

We now implement atomic register $X$ using three atomic registers $x_1, x_2, x_3$, based on a standard "quorum" technique. These three registers are each initialized to $(0, 0)$. To write $v$ to $X$, $p$ writes the pair $(v, ts)$, where *ts* is a timestamp higher than any timestamp that $p$ has used before, to any two of $x_1, x_2, x_3$. Process $q$, the reader, maintains a pair $(v', ts')$ — the pair with the highest timestamp that $q$ has ever seen. To read $X$, $q$ reads any two of $x_1, x_2, x_3$, updates $(v', ts')$ if necessary, and returns $v'$. This is clearly a correct implementation of atomic register $X$ (written by $p$ and read by $q$) from atomic registers $x_1, x_2, x_3$. We also implement atomic register $Y$ (written by $q$ and read by $p$) using three atomic registers $y_1, y_2, y_3$, in the same way.

In the implementation $I$ of $O$ of type $WL_a$ from registers $X$ and $Y$ (given in Figure 4), we now replace the base registers $X$ and $Y$ with their "quorum" implementations described above. If implementations were composable, this would yield a *correct* implementation $I'$ of $O$ of type $WL_a$ (from registers $x_1, x_2, x_3$ and $y_1, y_2, y_3$), that is, $I'$ would be linearizable, non-trivial, and wait-free. This, however, is not the case: $I'$ has an execution that violates non-triviality.

An execution of $I'$ that violates non-triviality is shown in Figure 1(b). It is similar to the execution of $I$ shown in Figure 1(a), except that the operations on $X$ and $Y$ are now executed using the "quorum" implementations of these registers. To perform its first read of $Y$, $p$ reads base registers $y_1$ and $y_2$; it finds that both of them contain $(0, 0)$, and so it returns 0 as the read in $Y$. Then, to write 1 to $X$, $p$ writes $(1, 1)$ to base register $x_1$ and then goes to sleep (before writing $(1, 1)$ to $x_2$). At this point, to do its first read of $X$, $q$ reads base registers $x_2$ and $x_3$; it finds both of these to contain $(0, 0)$, and so it returns 0 as the value read in $X$. Process $q$ then writes $(1, 1)$ to $y_1$ and $y_2$. For its second read of $X$, suppose that $q$ now reads base registers $x_1$ and $x_2$. Since $x_1$ contains $(1, 1)$ and $x_2$ contains $(0, 0)$, $q$ returns 1 as the value read in $X$. Since the two high-level reads of $X$ by $q$ returned different values, $q$'s COMPETE operation now returns $\perp$. Note that $q$'s COMPETE operation aborts *even though during its entire execution interval there were no steps of* $p$. This violates the non-triviality requirement. (Process $p$ can now complete its remaining operations as shown in Figure 1(b), and return $\perp$ just as it did in Figure 1(a).) □

The execution that violates non-triviality described in Figure 1(b) also shows that a process $p$ that has stopped taking steps (because it fell asleep or crashed) can interfere with another process $q$ that is executing alone, and *this interference by* $p$ *can manifest itself arbitrarily long after* $p$ *stopped taking steps.* In our example, the writing of 1 to $X$ by the sleeping process $p$ takes effect when $q$ reads $x_1$ in its second read of $X$, and this occurs without $p$ taking any steps. Thus, in general, $q$ may perceive interference and abort even in the absence of step contention.

## 9. CONCLUDING REMARKS

One of the contributions of this paper is the introduction of a new type of register, namely, the abortable register. Even though this register appears to be very weak (and in fact, is weaker than a safe register) it can still be used to support interesting implementations, e.g., it can be used to obtain obstruction-free implementations of any object. Further research is needed to study the power and limitations of abortable registers.

**Acknowledgments.** We thank the anonymous referees for their helpful comments.

## 10. REFERENCES

[1] Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In *DISC' 05: 19th International Symposium on Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2005.

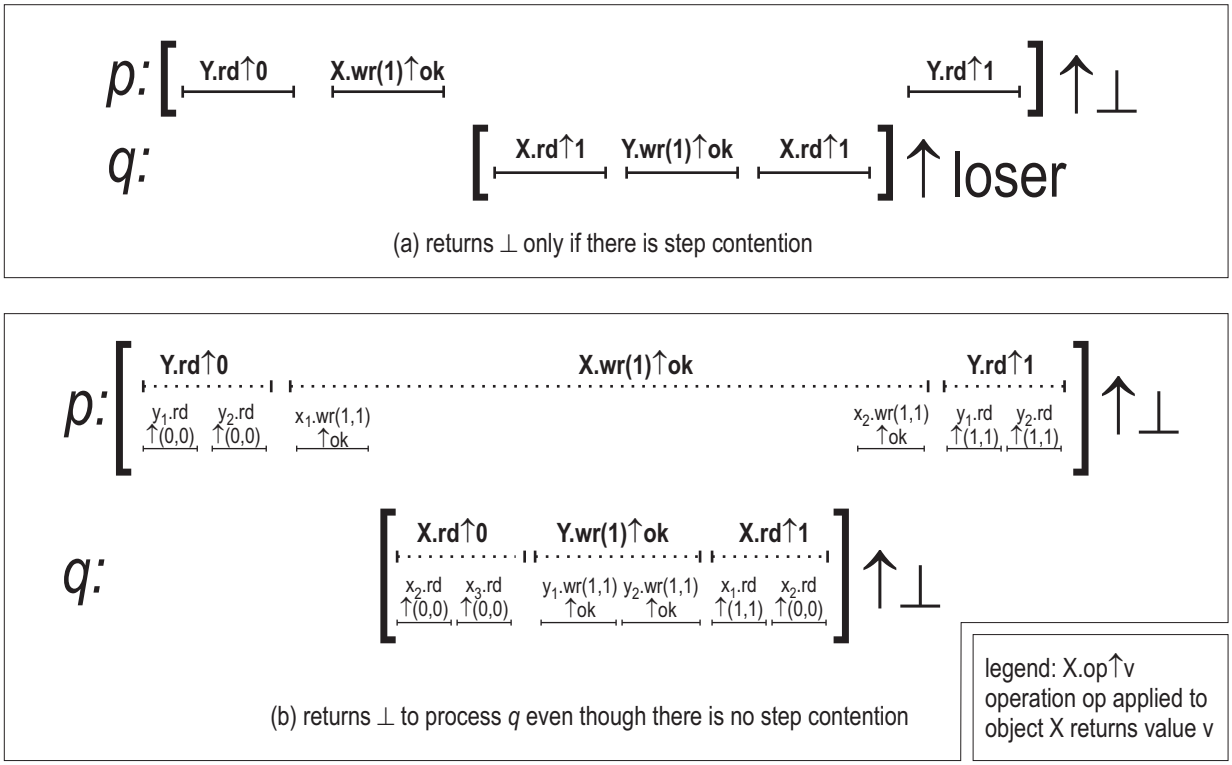


**Figure 1: (a) Execution of $I$ satisfying non-triviality. (b) Execution of $I'$ violating non-triviality.**


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

[3] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522–529, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Maurice Herlihy and Jeannette Wing. Axioms for concurrent objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 13–26, New York, NY, USA, 1987. ACM Press.

[5] Prasad Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 295–304, New York, NY, USA, 2003. ACM Press.

[6] Leslie Lamport. On interprocess communication. Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.