On the Cost of Composing Shared-Memory Algorithms

Dan Alistarh
EPFL
dan.alistarh@epfl.ch

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Petr Kuznetsov
TU-Berlin/Telekom Innovation Labs
petr.kuznetsov@tu-berlin.de

Giuliano Losa
EPFL
giuliano.losa@epfl.ch

ABSTRACT

Decades of research in distributed computing have led to a variety of perspectives on what it means for a concurrent algorithm to be efficient, depending on model assumptions, progress guarantees, and complexity metrics. It is therefore natural to ask whether one could compose algorithms that perform efficiently under different conditions, so that the composition preserves the performance of the original components when their conditions are met.

In this paper, we evaluate the cost of composing shared-memory algorithms. First, we formally define the notion of *safely composable* algorithms and we show that every sequential type has a safely composable implementation, as long as enough state is transferred between modules. Since such generic implementations are inherently expensive, we present a more general light-weight specification that allows the designer to transfer very little state between modules, by taking advantage of the semantics of the implemented object. Using this framework, we implement a composed long-lived test-and-set object, with the property that each of its modules is asymptotically optimal with respect to the progress condition it ensures, while the entire implementation only uses objects with consensus number at most two. Thus, we show that the overhead of composition can be negligible in the case of some important shared-memory abstractions.

Keywords

Composition, Modularity, Complexity, Consensus, Test-and-Set

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

1. INTRODUCTION

Designing correct and efficient concurrent algorithms is a major challenge. There seems to be an agreement on what it means for a concurrent data structure to be correct [14, 15]; on the other hand, the situation is more complex when it comes to efficiency. Decades of research have led to a variety of algorithms designed for different models, providing different progress guarantees, and optimized for different complexity measures. It is hard to believe that we shall eventually agree on the *right* model of concurrent systems, the most beneficial progress condition, or the most important complexity criterion. All these depend on the evolving hardware specifics of modern multiprocessors or on unpredictable user computing demands.

To anticipate these uncertainties, it is tempting to look for *composable* solutions. The idea is that algorithms optimized for different complexity metrics and progress conditions could be composed to obtain new algorithms that maintain the properties of their components under a combination of the original model assumptions. Ideally, such a composed algorithm would have the ability to *adapt* to changing environment conditions, e.g. high contention, asynchrony, or failures, while maintaining correctness in all executions.

We focus on composing algorithms providing various *progress guarantees*. Thus, a composable algorithm is allowed to *abort* whenever the condition under which it is expected to make progress is violated. In case the currently employed algorithm aborts, the composition is free to switch to an algorithm that makes progress under a different set of assumptions. The resulting implementation has to guarantee correctness, regardless of the way it jumps between its underlying components. We call this property *safe composition*. Respectively, algorithms that allow such composition are called *safely composable*. Safely composable algorithms are appealing since they have the advantage of being able to optimize for the common case, while always guaranteeing correctness. Another advantage of such implementations is that they can be designed and analyzed in a modular way.

In this paper, we evaluate safely composable algorithms in terms of computational power and complexity cost, for implementations that guarantee progress in the absence of interval contention [2] or step contention [6]. First, we present a composable universal construction that allows for implementing any sequential type. Since such generic implementations are inherently expensive, we present an alternative light-weight specification that allows the designer to transfer very little state between modules, by taking advantage of the semantics of the implemented object. Using this framework, we implement a composed long-lived test-and-set object, with the property that each of its modules is asymptotically optimal with respect to its progress condition, while the entire implementation only uses objects with consensus number at most two. Thus, we show that the overhead of composition, both in terms of step and space complexity and computational power of underlying objects, can be made negligible for some shared-memory abstractions.

Our safely composable universal construction allows us to build and compose shared-memory implementations guaranteeing progress under different conditions. The construction extends the consensus-based algorithm of Herlihy [14] by using safely composable consensus objects: when such an object aborts, it provides the information about the "state" of the object being implemented that is used for the initialization of the next safely composable consensus object. For a generic object, this information is represented as a *history*, i.e. a sequence of operation requests that were previously submitted to the object. These histories do not have to be consistent across processes: only prefixes of *committed* (i.e., executed) operations must be consistent. A similar idea was used in [12, 20] to build an efficient abortable Byzantine fault-tolerant replicated state machine (Abstract) in message-passing systems.

A *light-weight* version of safely composable shared-memory implementations can be adapted and optimized to the object being implemented. When a safely composable implementation of the object aborts, it provides some *object-specific* information about the "state" of the object. This information is then used to initialize the next safely composable algorithm.

A natural correctness condition is that any safely composable object should be linearizable, when restricted to invocations and committed responses. Intuitively, there is a sharp trade-off between the amount of information transferred between the modules and the correctness of the composition: in particular, if too little information is transferred, the composed algorithm may no longer be linearizable. Our specification of a safely composable object deals with this issue in two ways: first, we ensure that all the values returned can be mapped to a matching set of histories of requests, which have to verify a series of consistency conditions. Second, we allow a module to restrict the possible interpretations of input values that it accepts.

The resulting specification (Section 5) has two key properties: first, the composition of any two safely composable modules is itself safely composable. Second, any safely composable module taken on its own is linearizable. These two properties greatly simplify the design and proof of composed algorithms. Finding an object specification that is both safe (guaranteeing correct composition) and general (allowing objects that export little state) is one of our main technical contributions.

We use our framework to build a speculative test-and-set (TAS) object that only uses registers in executions where there is low contention, and may revert to stronger hardware primitives in contended executions. The algorithm is built from two independent modules: the first module has constant step complexity, and ensures progress in the absence of step contention (i.e. it is obstruction-free). The second module reverts to a hardware implementation of the object and is therefore wait-free. The algorithm switches *forward* to a more contention-resilient module when contention is detected, and can also switch *back*, using the reset mechanism, to a more efficient speculative module if the algorithm is currently employing the expensive hardware object. The implementation is also *long-lived*, since the object can be reset once it has been won.

Our TAS algorithm is of independent interest for two reasons. First, the obstruction-free module shows that TAS can be implemented in constant time and space in the absence of interval contention, whereas the best known bound for obstruction-free consensus is linear [6]. A simple modification of our algorithm (described in the Appendix) yields the first *solo-fast* [6] TAS algorithm with constant step complexity for uncontended operations. Second, the entire algorithm can be seen as a simple efficient version of a *biased lock* [9], that uses only registers as long as a single process is using it, and reverts to the hardware implementation only under step contention, as opposed to interval contention for previous implementations [9, 19].

Our constructions have several implications concerning the cost of safe composition in shared memory. First, our speculative TAS implementation is wait-free, and results from the composition of two modules. One might expect that moving from a module that may abort to one that always makes progress would require consensus, since, intuitively, the processes need to "agree" on the value returned from a safely composable module before executing a wait-free one. Thus, the consensus number [14] of the resulting implementation should be n. We find that this is not necessarily the case. In particular, we show that if the semantic of the implemented object is known, the algorithm can speculate using only registers and a hardware implementation of the object, avoiding consensus. In particular, our composed TAS algorithm only uses objects with consensus number at most *two*.

Second, the step complexity overhead induced by composition is considerable in the case of generic implementations, since each process has to essentially obtain a snapshot of all previously performed requests. But if the semantics of the implemented object is known, as is the case of our TAS implementation, we show that the overhead of composition can be brought down to a small constant number of steps. In fact, our implementation is optimal in terms of fence complexity [7]. Therefore, safe composition does not have to imply an increase in time complexity.

Overall, this paper describes a novel framework for shared-memory composition. We present a TAS implementation that combines lightweight components, that only make progress under the absence of step contention, with a hardware TAS objects at *no cost*, either in the computational power of base objects or in step complexity. Is this possible for any object? If not, can we categorize objects based on the *cost* of their safely composable implementations, such as the power of the underlying model, complexity, or the amount state that must be transferred between the components? These are interesting directions for future research.

Roadmap. In Section 2 we present an overview of related work. Our model definitions are presented in Section 3. In Section 4 we describe a composable universal construction based on Abstract. We then present our light-weight framework for composable objects in Section 5, and showcase it by building a speculative test-and-set implementation. In the Appendix, we give abortable variants of shared-memory consensus algorithms. Due to space limitations, we only present sketches for some of the proofs. Detailed versions can be found in the full version of this paper [5].

2. RELATED WORK

Composing safe distributed algorithms optimized for the common case has been used to obtain efficient solutions to fundamental problems such as consensus [8, 10], Byzantine agreement [11, 12], mutual exclusion [17], or renaming [4].

In shared-memory, composition has either been used implicitly, by combining a fast-path algorithm with a (slower) wait-free one in an ad-hoc manner [6, 18], or explicitly, by requiring an algorithm to return an *abort* indication before a second algorithm can be called [3, 6]. Implicit solutions have focused on consensus implementations: Luchangco et al. [18] presented consensus and compare-and-swap implementations with constant step complexity in executions with no interval contention, that revert to hardware primitives otherwise; Attiya et al. [6] showed a consensus implementation with $O(n)$ step complexity in the absence of step contention, that reverts to hardware compare-and-swap otherwise, and an $\Omega(\log n)$ lower bound for the fast path of such implementations, for perturbable objects. Attiya et al. [6] also study consensus with *fails*, when a process may abort the current execution explicitly; however, their requirements on the abort condition are strictly stronger than the ones of safely composable implementations in Section 5. (For example, any instance of consensus with fails is shown to have consensus number 2, while a safely composable consensus implementation may have consensus number 1.) In this paper, we study the cost of composing such algorithms in a more general way–in particular, we analyze algorithms

that can be designed an proved independently, whose composition is always correct.

Aguilera et al. [3] define *abortable* and *query-abortable* shared-memory objects, that may return an *abort* indication under contention; if queried, these objects return the last operation of the querying process that caused a state transition and its response. The authors also introduce efficient universal constructions for such objects. Our safely composable objects always return a value together with the indication. However, this value may not be consistent with the object's actual state, and may be caused by another process's operation, therefore the two definitions are incomparable. Interestingly, reference [3] shows that abortable objects do not compose if their progress is based on step contention. Intuitively, this is because, the correctness properties of abortable objects of [3] allow an aborted operation to take effect *after* it aborts. In contrast, our definition ensures safe composition, irrespective of the progress predicate. Finally, reference [13] investigates formal specifications for composition, and their implications for the scalability of software verification. Our specification of a safely composable object can be seen as a generalization of the speculative framework given in that paper.

3. PRELIMINARIES

Model. We consider the standard wait-free asynchronous shared memory model with n processes, $n-1$ of which may fail by crashing. Processes communicate through multiple-writer-multiple-reader atomic registers. Any register R exports atomic read and write operations, with standard semantics.

Objects, Algorithms and Executions. We define an object as a quadruple $(\mathcal{Q}, s, \mathcal{I}, \mathcal{R}, \Delta)$, where $\mathcal{Q}$ is a set of states, s is a starting state, $\mathcal{I}$ is a set of requests, $\mathcal{R}$ is a set of responses, and $\Delta \subseteq \mathcal{Q} \times \mathcal{I} \times \mathcal{Q} \times \mathcal{R}$ is the *sequential specification* of the object [6]. We assume that object types are non-trivial, i.e., all these sets are non-empty. A *history* is defined as a sequence of inputs (elements of $\mathcal{I}$) that contains no duplicates. (For simplicity, we assume that each request has a unique identifier.)

For example, the (one-shot) *test-and-set* object has initial state 0, and is accessed by a test-and-set operation. The operation atomically reads the value of the test-and-set and sets it to 1. We say that the unique process that returns 0 from the test-and-set is the *winner*, while the processes that return 1 are *losers*.

To implement an object, processes follow an algorithm. Given a process and an input in the set $\mathcal{I}$, an algorithm determines a sequence of steps whose execution establish an output in $\mathcal{O}$. The steps taken may be either local steps or shared memory reads and writes. A process repeatedly chooses an arbitrary input and executes the sequence of steps described by the algorithm until it determines an output. When a process chooses an input m, we say the it *invokes* m. When it determines an output r, we say that it *commits* r. We say that an algorithm implements an object if and only if all the sequence of invocations and commits, ordered according to their real-time occurrences, that can possibly be observed in the system are linearizable [15].

We are interested in algorithms composed of a sequence of clearly separated modules. A module is similar to an algorithm, but it can be initialized and it can abort instead of committing. Two modules are composed by using the aborts of the first module as initialization values for the second module. A process starts by running the first module in the sequence of modules that compose the algorithm. Given an input, the first module determines a sequence of steps whose execution establish either an output or a *switch value* ranging over the set $\mathcal{V}$. If an output is committed, then a new input is invoked, as before. But, when a process determines a switch value v, we say that it *aborts* with the switch value v.

As for the first module, the second module determines the steps to be followed to either commit or abort an input. However it also determines a sequence of steps to execute given an input and a switch value. When a process running the first module aborts with the switch value v, the process executes the sequence of steps determined by the second module given its last input and the switch value v. In general, a process repeatedly chooses an input and executes the steps determined by its current module, until it aborts and switches to the next module.

We will be interested in properties of the sequence of aborts, invocations and commits, ordered according to their real-time occurrences, observed in the system. We call such sequences *traces*.

Let $\mathcal{V}$ be the set of *switch values*. We denote by $\mathcal{T}$ the set of tuples consisting of a request and a switch value, and we call such tuples *switch tokens*. A request m may be invoked as is or together with a proposed switch value $v \in \mathcal{V}$ for the object. In the first case an invocation is denoted by the tuple (invoke, m) and in the second case by (init, m, v). In an invocation, the purpose of the switch value is to initialize the current module of the object.

A reply may be of the form (commit, m, r) where m is the request being responded to and r is a response in $\mathcal{R}$ or (abort, m, v) where m is the request being responded to and v is a switch value in $\mathcal{V}$. In an abort response, the purpose of the switch value is to initialize a new module of the object.

Progress Conditions. Algorithms that ensure safety and progress in all executions are called *wait-free*. We will also consider implementations that ensure safety in all executions, but may not make progress if two or more processes access the implementation concurrently; we call such algorithms *contention-free*, as they ensure progress in the absence of *interval contention*. Also, we consider algorithms that guarantee that a process makes progress as long as no other process takes steps concurrently. Such algorithms are called *obstruction-free*, and they make progress in the absence of *step contention*. For more precise definitions of these progress conditions, please see [6].

4. A COMPOSABLE UNIVERSAL CONSTRUCTION

In this section, we consider implementing generic shared-memory objects in a composable way. We show that any sequential type can be implemented using only registers in executions when there is no step contention, and reverting to stronger compare-and-swap primitives otherwise. More precisely, we describe a composable universal construction, following the structure of an abortable replicated state machine (Abstract), introduced in [20] in the context of Byzantine fault-tolerant algorithms. "Light-weight" implementations that adapt to the semantics of the object are discussed in Section 5.

4.1 Definition and Properties

We introduce the definition and properties of a composable universal construction, given as an Abstract [12, 20]. An Abstract encapsulates the specification of a state machine that may abort. We begin by recalling the definition and properties of an Abstract, as given in [20].

DEFINITION 1 (ABSTRACT, [20]). *An* Abstract *exports one operation* Invoke(m, h)*, that issues request* m *with initial history* h*. An* Abstract *exports two indications that may be returned to the client:* Commit(m, h)*, and* Abort(m, h)*. We say that the process or operation commits (resp. aborts) the request* m *with history* h*, where a* history h *is a sequence of requests that the process can use to compute a reply (resp., to recover). If the process commits (resp., aborts)* m *with history* h*, we refer to* h *as the commit history (resp., abort history).*
Abstract *ensures the following properties on its traces.*

1. *(Termination) If a correct process invokes a request* m*, then it eventually commits or aborts* m *with history* h*, and* h *contains* m*.*
2. *(Commit Order) Let* h *and* h' *be any two commit histories. Then either* h *is a strict prefix of* h' *or vice-versa.*
3. *(Abort Ordering) Every commit history is a prefix of every abort history.*
4. *(Validity) In every commit/abort history* h*, no request appears twice and every request was invoked by some process* before *the current operation returns.*
5. *(Non-Triviality) If a correct process invokes a request* m *and some predicate* NT *is satisfied, the process commits* m*. We say that the* Abstract guarantees progress *under predicate* NT*.*
6. *(Init Ordering) Any common prefix of init histories is a prefix of any commit or abort history.*

In short, an Abstract returns histories that represent the ordering of process requests. (For simplicity, we assume that every request has a unique identifier.) In case of a commit, this ordering is definitive and the result of the call is uniquely determined by the order of the requests in the history. In the case of an abort, we require that every abort history contains every commit history as its (non-strict) prefix. Thus, effectively, no request invoked after an abort can be committed.

Note that the above definition and properties hold for any system model. In this paper, we consider shared memory implementations. Thus, the non-triviality predicate NT for our implementations will be expressed in terms of different notions of contention in shared memory.

We will consider multiple Abstract instances composed to achieve wait-free object implementations whose performance may change depending on the adversarial setting. Intuitively, the *composition* of two Abstract instances A and B is an algorithm that first calls Abstract A, returning with history h if the call returns (commit, h); otherwise, if the call returns (abort, h), it calls Abstract B with initial history h. The central property of this framework is that its instances are inherently *composable*, i.e., the composition of any two Abstract instances generates a third Abstract instance.

THEOREM 1 (COMPOSITION, [20]). *The composition of any two* Abstract *instances is an* Abstract *instance.*

4.2 The Universal Construction

We build a universal construction following the Abstract specification. The construction is based on Herlihy's classic universal construction [14], replacing wait-free consensus with consensus instances that may abort in the presence of contention.

More precisely, an *abortable* consensus instance returns either a commit or an abort indication, together with a decision value v (in case of abort, the instance returns an empty value $\perp$). The instance guarantees to commit as long as a progress predicate NT holds. In the Appendix, we present abortable consensus algorithms ensuring progress in the absence of interval contention (SplitConsensus) and in the absence of step contention (AbortableBakery), which only use read-write atomic registers. These are abortable variants of algorithms already present in the literature [6, 18], respectively.

Description. Processes share an array of abortable consensus instances $Cons$ ensuring progress as long as predicate NT holds, an atomic register $Aborted$, a snapshot object $Reqs$, where process p_i adds its requests in component $Reqs[i]$, and an atomic counter C, used to assign timestamps to process requests. Each process maintains a list $lProp$ of proposed requests and a list $lPerf$ of performed requests. The process runs in parallel an instance of the universal construction and a task checkAbort that checks whether the $Aborted$ register has been set to true.

As long as the abortable consensus instances do not return an abort indication, the construction proceeds exactly as Herlihy's universal construction, using the $Cons$ vector to agree on the requests to be performed on each process' local copy of the object, and incrementing the counter C for each new request. On the other hand, if a process receives abort from a consensus instance $Cons[\ell]$ or reads $Aborted =$ true in the checkAbort task, then the process first sets the $Aborted$ register to true (in case the register is not already set), and reads $count$, the value of the counter C, to get the length of the abort history.

Then the process proceeds to compute a valid abort history. It starts from an empty history and appends, in order, all requests that have been decided in the $Cons$ vector, from 1 to $count$, irrespective of whether the requests have been committed or aborted. For consensus instances in $Cons$ to which it has not participated, the process can get a decision value by proposing $\perp$. Following this procedure, the process obtains an abort history of length at most $count$.

For initializing a new instance of the universal construction, each process proposes, in order, the requests in its (abort) history to the $Cons$ list of the new instance. The process then proceeds to execute the new instance as described above. (Note that a process may abort during this initialization step.)

Progress predicate. The above construction guarantees to commit requests as long as the progress predicate NT of the abortable consensus implementation holds. (This predicate could be the absence of interval contention, or the absence of step contention.) It follows that, given such a consensus algorithm, the construction verifies the properties of an Abstract with progress predicate NT.

LEMMA 1. *Given an abortable consensus algorithm* A *that always commits as long as predicate* NT *holds, the above construction is an* Abstract *with progress predicate* NT*.*

Contention-free, obstruction-free and wait-free variants. In the Appendix, we present abortable consensus algorithms that ensure progress in the absence of interval contention (SplitConsensus) and in the absence of step contention (AbortableBakery), which only use read-write atomic registers. Lemma 1 implies that these algorithms generate universal constructions with the corresponding progress predicates. It is easy to see that if the abortable consensus algorithm is replaced by a wait-free consensus algorithm we obtain a wait-free composable universal construction that never aborts. The composition of these three Abstracts is an Abstract that never aborts, and only uses registers in executions with no interval or step contention. Note that the *commit ordering* property of Abstract implies that the composition generates linearizable implementations for generic objects. We formalize this as follows.

PROPOSITION 1. *Every sequential type has an* Abstract *implementation, using only registers in the absence of interval or step contention, and employing compare-and-swap otherwise.*

Complexity Cost. On the other hand, the composition of generic object implementations comes at a price. Any wait-free universal Abstract implementation must have linear space and step complexity [16]. Moreover, we notice that *any* wait-free Abstract implementation of a non-trivial sequential type solves consensus, even if the original object has lower consensus number. This follows easily, since we can use the commit histories to reach a decision value among n processes: the process with the first committed request in the commit histories imposes its proposal value on the consensus object.

PROPOSITION 2. *Every* Abstract *implementation of a non-trivial sequential type guaranteeing wait-free progress solves wait-free consensus.*

In the next section, we examine how to avoid this cost.

5. SAFELY COMPOSABLE OBJECTS

5.1 Definitions

Given the definitions in Section 3, let $\mathcal{O} = (\mathcal{Q}, s, \mathcal{I}, \mathcal{R}, \Delta)$ be an object and $\mathcal{H}$ be the set of all possible histories (recall that a history is a sequence of requests in $\mathcal{I}$ that contains no duplicates). The definition of a safely composable algorithm takes two parameters: a set of switch values $\mathcal{V}$ and a *constraint* function $M : 2^{\mathcal{T}} \rightarrow 2^{\mathcal{H}}$, mapping every set of switch tokens into the set of possible histories that it encodes. (Recall that $\mathcal{T}$ denotes the set of switch tokens, i.e. pairs consisting of a request and a switch value.) Intuitively, a constraint function puts restrictions on the allowed interpretations of a given set of init or abort tokens.

Let β be the function from histories to responses of $\mathcal{O}$ such that $\beta(h)$ is the last response obtained by applying h sequentially to $\mathcal{O}$. Given a history h and a requests m appearing in h, we define $\beta(h, m)$ as the response matching m in h.

Given a set I of requests, we define the equivalence relation $\equiv_I$ such that $h_1 \equiv_I h_2$ iff (i) both h_1 and h_2 contain all the requests in I, (ii) for all $h \in \mathcal{H}$, $\beta(h_1 h) = \beta(h_2 h)$, and (iii) for all requests $m \in I$, $\beta(h_1, m) = \beta(h_2, m)$. Intuitively, two histories composed of requests in I are equivalent if they "appear" the same (return the same responses) in all possible extensions.

Given a set of switch tokens T, we define $requests(T)$ as the set of requests found in the token in T and we define $eq(T, M)$ as the set of equivalence classes of the relation $\equiv_{requests(T)}$ partitioning the set $M(T)$. Given a history h we denote by $[h]_{T,M}$ the equivalence class of h according to $\equiv_{requests(T)}$ in $M(T)$.

Let τ be a trace of O, as defined in Section 3. Let an *index* be a position in a trace. An index i of τ might be either an *invoke*, *commit*, *abort*, or *init* index, depending on the event appearing at i in τ. Let $Ind(\tau)$ be the set of all indices in τ. Given a commit index i, let $response(i)$ be the response appearing at i. Let $inits(\tau)$ be the set of switch tokens found in the init requests of τ and $aborts(\tau)$ be the set of switch tokens found in the abort replies of τ.

An *interpretation* ϕ is a function $\phi : Ind(\tau) \rightarrow \mathcal{H}$. We denote by ϕ_τ the trace obtained from trace τ by replacing every commit or switch value appearing in τ at index i with $\phi(i)$. A trace τ is *valid* with respect to a mapping $M : 2^{\mathcal{T}} \rightarrow 2^{\mathcal{H}}$ iff $M(inits(\tau)) \neq \emptyset$.

An interpretation ϕ is *valid* with respect to trace τ, mapping M, and history $h_{abort} \in M(aborts(\tau))$ iff:

1. There exists $h_{init} \in M(inits(\tau))$ such that for every init index $i \in \tau$, $\phi(i) = h_{init}$.
2. For every abort index $i \in \tau$, $\phi(i) = h_{abort}$.
3. For every commit index $i \in \tau$, we have that $\beta(\phi(i)) = response(i)$.
4. ϕ_τ is a trace that satisfies the properties of an Abstract from Definition 1.

Intuitively, a valid interpretation ϕ for τ produces a trace ϕ_τ by replacing every commit value or switch value appearing at index i in τ with a history h so that the invocation, init events, commit and abort responses are globally consistent.

If ϕ is valid w.r.t. τ, M, and h_{abort}, we denote by $init(\phi, \tau)$ the unique history h such that for all init index i of τ, $\phi(i) = h$. Similarly we denote by $abort(\phi, \tau)$ the unique history h such that for all abort index i of τ, $\phi(i) = h$.

DEFINITION 2 (SAFE COMPOSITION). *An algorithm A is a* safely composable *implementation of an object $\mathcal{O}$ with respect to a set of switch values $\mathcal{V}$ and a constraint function M iff for every trace τ of A that is valid w.r.t. M and for every equivalence class $e \in eq(aborts(\tau), M)$, there exists a history $h_{abort} \in e$ and an interpretation ϕ that is valid with respect to τ, M, and h_{abort}. If $eq(aborts(\tau), M) = \emptyset$, then ϕ has to be valid with respect to τ, M, and the empty history $\perp$.*

5.2 Properties

The key property of this specification is that the composition of any two safely composable objects is also a safely composable object. (Recall that, intuitively, the composition of A and B runs algorithm B using A's outputs as inputs.) The proof of this result illustrates a trade-off between the strength of the specification and the difficulty of proving correctness of the composition: the more general the specification, the harder it is to prove that the composition is correct.

THEOREM 2 (COMPOSITION). *Let A and B be two algorithms that are safely composable implementations of an object O with respect to a set of switch values $\mathcal{V}$ and a constraint function $\mathcal{M}$. Then the composition of A and B is a safely composable implementation of object O with respect to $\mathcal{V}$ and $\mathcal{M}$.*

PROOF. We begin the proof by stating a couple of auxiliary results.

LEMMA 2. *For all set V of switch values, the relation $\equiv_V$ is a right congruence w.r.t. history concatenation. In other words, for all set V of switch values, if $h_1 \equiv_V h_2$ then for all history h, $h_1 h \equiv_V h_2 h$.*

This lemma is straightforward from the definition of $\equiv_V$. It means that we cannot distinguish whether history h_1 or history h_2 was executed if we only look at subsequent responses. This fact in crucial in the proof of the next lemma.

Consider now an arbitrary trace τ and suppose that ϕ is an interpretation of τ such that ϕ_τ satisfies the properties of Abstract, all init indexes are mapped to the same history, and all abort indexes are mapped to the same history. By the definition of Abstract, we know that for all init, commit, or abort indexes i of τ, there exists a history h_i such that $\phi(i) = init(\phi, \tau) h_i$. Given a history h, let $subst(\phi, h)$ be the interpretation of τ such that for all init, commit, or abort indexes i of τ, $subst(\phi, h)(i) = h h_i$. Intuitively, $subst(\phi, h)$ is obtained by replacing, in every history $\phi(i)$, the longest common prefix of init histories in ϕ_τ ($init(\phi, \tau)$) with the history h.

LEMMA 3. *Consider a trace τ and suppose that ϕ is a valid interpretation of τ with respect to a constraint function M and a history h_{abort}. Let $h_{init} = init(\phi, \tau)$. Then for all histories $h'_{init} \in M(inits(\tau))$ related to h_{init} (i.e. such that $h'_{init} \in [h_{init}]_{inits(\tau),M}$), there exists a history $h'_{abort} \in M(aborts(\tau))$ related to h_{abort} (i.e. such that $h'_{aborts} \in [h_{abort}]_{aborts(\tau)),M}$) and an interpretation ϕ' such that $init(\phi', \tau) = h'_{init}$ and the interpretation ϕ' is valid w.r.t τ, M, and h'_{abort}.*

Intuitively, the lemma implies that interpretations of safely composable objects compose: in particular, every valid interpretation of abort histories of A can be used to interpret the init histories of B.

PROOF. Let $\phi' = subst(\phi, h'_{init})$ and $h'_{abort} = abort(\phi', \tau)$. Since ϕ is an interpretation of τ such that ϕ_τ satisfies the properties of Abstract, ϕ' and h'_{abort} are well defined. Moreover we claim that ϕ' is valid w.r.t τ, M, and h'_{abort}:

- $h'_{init} \in M(inits(\tau))$ by definition and we trivially have that $init(\phi', \tau) = h'_{init}$.
- We trivially have that that for all abort index i, $\phi'(i) = h'_{abort}$.
- Lemma 2 implies that $\beta(\phi'(i)) = \beta(\phi(i)) = response(i)$.
- $\phi'(\tau)$ is a trace of Abstract because, given any history h, the four properties of Abstract are left invariant by substituting h, in every history, for the longest common prefix of init histories.

Hence ϕ' is valid w.r.t. τ, M, and h'_{abort}.

It remains to show that $h'_{abort} \in [h_{abort}]_{aborts(\tau),M}$. By Lemma 2 we have that $h'_{abort} \in [h_{abort}]_{inits(\tau),M}$. Hence we know that (ii) for all $h \in \mathcal{H}$, $\beta(h'_{abort}h) = \beta(h_{abort}h)$.

Moreover because $\phi'(\tau)$ is a trace of Abstract, Validity holds of $\phi'(\tau)$. Hence (i) for all request m in $aborts(\tau)$, $m \in h'_{abort}$. By definition of h'_{abort}, we know that there exists h such that $h_{abort} = h_{init}h$ and $h'_{abort} = h'_{init}h$. We now consider two cases: Suppose $m \in h'_{init}$. By Validity we have that $m \in requests(inits(\tau))$. Then because $h'_{init} \in [h_{init}]_{inits(\tau),M}$ we have that $\beta(h_{abort}, m) = \beta(h'_{abort}, m)$. Suppose $m \in h$. Since $h'_{init} \in [h_{init}]_{inits(\tau),M}$ we have that for all h', $\beta(h_{init'}h') = \beta(h_{init}h')$. Hence (iii) $\beta(h_{init'}h, m) = \beta(h_{init}h, m)$. From (i), (ii), and (iii) we have that $h'_{abort} \in [h_{abort}]_{aborts(\tau),M}$. ☐

Returning to the proof of the Theorem, consider a trace τ of the composition of A and B that is valid w.r.t. M and let τ_A be the projection of τ onto the events of A and τ_B be the projection of τ onto the events of B. Consider an equivalence class $e \in eq(aborts(\tau_B), M)$. We show below that there exists $h \in e$ and an interpretation that is valid w.r.t. τ, M, and h.

First observe that since τ is valid w.r.t. M, then τ_A is valid w.r.t. M too. Thus, because A is a safely composable implementation of O w.r.t. $\mathcal{V}$ and $\mathcal{M}$, τ_B is valid w.r.t. $\mathcal{M}$. Hence, because B is a safely composable implementation of object O with respect to $\mathcal{V}$ and $\mathcal{M}$, we know that there exists a history $h^B_{abort} \in e$ and an interpretation ϕ_B of τ_B such that ϕ_B is valid with respect to τ_B, M, and h^B_{abort}.

Let $h^B_{init} = init(\phi_B, \tau_B)$, i.e. the history such that for all init index i of τ_B, $\phi_B(i) = h^B_{init}$. Observe that $h^B_{init} \in M(inits(\tau_B)) = M(aborts(\tau_A))$. Consider $e' = [h^B_{init}]_{inits(\tau_B),M}$, which is the equivalence class of h^B_{init} w.r.t. $\equiv_{inits(\tau_B)}$ in $M(inits(\tau_B))$. Because A is a safely composable implementation of object O with respect to $\mathcal{V}$ and $\mathcal{M}$, we know that there exists a history $h^A_{abort} \in e'$ and an interpretation ϕ_A of τ_A such that ϕ_A is valid with respect to τ_A, M, and h^A_{abort}.

Since $h^A_{abort} \in [h^B_{init}]_{inits(\tau_B),M}$, by Lemma 3 we obtain a history ${h^B_{abort}}' \in [h^B_{abort}]_{aborts(\tau_B),M} = e$ and an interpretation ϕ'_B that is valid w.r.t. τ_B, M, and ${h^B_{abort}}'$, and such that $init(\phi'_B, \tau_B) = h^A_{abort} = abort(\phi_A, \tau_A)$. Let ϕ be the interpretation of τ such that if i is an index of τ_A, then $\phi(i) = \phi_A(i)$ and else $\phi(i) = \phi'_B(i)$. Since ϕ_A and ϕ'_B coincide on the abort indices of τ_A (or equivalently on the init indices of τ_B), we have by the Abstract composition theorem that ϕ_τ is a trace of Abstract. Moreover since ϕ_A and ϕ'_B are valid w.r.t. τ_A and τ_B we have that for all commit index i, $\beta(\phi(i)) = response(i)$. Finally we have by validity of $\phi_A(\tau_A)$ and $\phi_B(\tau_B)$ that for every init index $i \in \tau$, $\phi(i) = init(\phi_A, \tau_A)$, and that for every abort index i in τ, $\phi(i) = {h^B_{abort}}'$.

In conclusion we obtained the interpretation ϕ which is valid w.r.t. τ, M, and ${h^B_{abort}}'$. With the fact that ${h^B_{abort}}' \in e$ this proves our goal.

Another property that follows from this specification is that any safely composable object which is not initialized by a previous module is linearizable. This follows since a valid interpretation ϕ of τ satisfies the Abstract Commit Order and Validity properties. It was proved in [13] that any trace satisfying these properties is linearizable.

THEOREM 3 (LINEARIZATION). *Consider a safely composable implementation A of object O with respect to a set of switch values $\mathcal{V}$ and a mapping $\mathcal{M}$. Consider a trace τ of A that contains no init requests. Then then projection of τ onto invoke and commit events is linearizable.*

Note that Abstract is a safely composable implementation of a generic object which responds to invocations with its full execution history. This can be seen by taking a constraint function M that maps a set of histories to their longest common prefix and by observing that in this case, the interpretation that associates to init requests the longest common prefix of all inits, to the abort request the longest common prefix of the abort requests, and to the responses the history contained in the response, is a valid interpretation.

6. A SPECULATIVE TEST-AND-SET ALGORITHM

We now present a speculative long-lived test-and-set implementation. The construction is based on two modules, composed to obtain a wait-free linearizable test-and-set (please see Figure 1 for an illustration). The first module, A1, uses only registers and has constant step complexity, ensuring progress in the absence of *step* contention. The second module A2 is essentially a hardware implementation of test-and-set. We consider an implementation where the two modules are composed in the increasing order of progress condition strength: a process first tries to execute the obstruction-free module; if this module aborts (because of contention), then it tries to execute the wait-free module A2. Upon a reset operation, the calling process also reverts the algorithm to an instance of the obstruction-free module A1, thus the module returns to speculative mode in case it was using the hardware implementation. We now describe each module in more detail.

6.1 The Obstruction-Free Module

In the first module, described in Algorithm 1, processes share four atomic registers: *aborted*, initially false, V, initially 0, and P

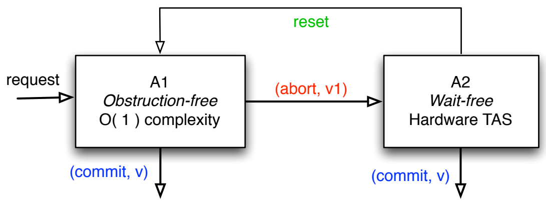

Figure 1: The structure of the test-and-set algorithm.

```
1  Shared:
2  Registers $P$ and $S$, initially $\perp$, register $aborted$, initially false,
   register $V$, initially 0
3  procedure A1-test-and-set( $val$ )$_i$
4    if $aborted$ = true then
5        if $V$.read() = 0 then return (abort, W)
6        else return (abort, L)
7    if $V$.read() = 1 or $val$ = L then
8        return (commit, loser)
9    if $P \neq \perp$ then return (commit, loser)
10   $P \leftarrow i$
11   if $S \neq \perp$ then return (commit, loser)
12   $S \leftarrow i$
13   if $P = i$ then
14       $V \leftarrow 1$
15       if $aborted$ = false then
16           return (commit, winner)
17       else return (abort, W)
18   else
19       $aborted \leftarrow$ true
20       $val \leftarrow V$
21       if $val = 1$ then
22           return (commit, loser)
23       return (abort, W)
```

Algorithm 1: The obstruction-free module.

and S, initially $\perp$. The *aborted* register indicates whether the current instance has been aborted, whereas V holds the current value of the object. Registers P and S are used to indicate whether other processes are taking steps in the current execution. In short, we ensure that each process either reaches a winner/loser decision in the absence of interval contention, or that the process detects interval contention, and aborts.

More precisely, processes first check the value of the *aborted* register. If it has been set to true, then the process aborts with state W, if the V register has not been set, or with state L, dropping from contention, if V has been set. Otherwise, if $aborted$ = false, competing processes race to write their value to both registers P and S, in order. If one of these registers has already been written by another process, then the current process can safely commit loser (lines 9 and 11). Otherwise, the process checks if its identifier is still the value in P (line 12). If this holds, then we are certain that the process is the only one in this state. Next, the process sets the register V to 1, modifying the value of the object (line 14). This step will cause other processes to subsequently return loser. Finally, it checks whether the instance has been aborted by another process. If not, then the process returns winner. Otherwise, the process will abort with state W, signifying that the object has not yet been won (line 17). On the other hand, if register P's value is no longer i, then interval contention occurred. In this case, the process either returns loser, if V has already been set to 1 by a concurrent process, or it aborts with state W if V is still 0 (lines 18–23).

Analysis. We show that A1 is a correct safely composable object, whose progress predicate is the absence of step contention. It is easy to see that the algorithm has constant time and space complexity. The main technical point in the proof is mapping algorithm traces to Abstract histories. We start by defining the set of input values $\mathcal{V}$ and the mapping M with respect to which A1 is safely composable. In short, $\mathcal{V}$ is the set of abort values W and L, while M maps any set of replies into a set of possible histories that all start with the same request that returned W, and contain all the requests that caused the replies.

DEFINITION 3. *We define the set* $\mathcal{V} = \{\mathsf{W}, \mathsf{L}\}$. *Then the mapping* $M : 2^{\mathcal{T}} \to 2^{\mathcal{H}}$ *is defined as follows. Consider a set of replies* $S = \{(r_1, v_1), \ldots, (r_\ell, v_\ell)\}$. *If S contains a reply with value* $v_i = \mathsf{W}$, *then* $M(S) := \{h \in \mathcal{H} \mid (\exists i \in \{1, \ldots, \ell\}$ such that $v_i = \mathsf{W}$ and $\mathsf{head}(h) = r_i)$ and $(\forall j \in \{1, \ldots, \ell\}.r_j \in h)\}$.

Otherwise, M(S) contains histories that start with an arbitrary request r not in S, and contain all requests in S, i.e. $M(S) := \{h \in \mathcal{H} \mid (h \neq \perp$ and $\mathsf{head}(h) \in \mathcal{R} \backslash S)$ and $(\forall j \in \{1, \ldots, \ell\}, r_j \in h)\}$.

Next, we prove that module A1 is safely composable with respect this definition.

LEMMA 4 (OBSTRUCTION-FREE MODULE). *The module* A1 *in Figure 1 is a safely composable test-and-set implementation with respect to the sets $\mathcal{V}$ and M from Definition 3.*

PROOF. The proof is structured as follows: we first prove invariants on the algorithm A1, after which we use these invariants to build the mapping to an Abstract required by the definition of a safely composable object.

The following invariants hold in every execution of A1 (their proof can be found in the full version of this paper [5]):

1. At most one process may execute lines 14-17. Also, at most one process may commit winner in A1.
2. If a process commits winner, then no process aborts with W in A1.
3. In every execution, there exists at least one process p such that 1) p either a) crashes, or b) commits winner, or c) aborts with W, *and* 2) p's request is invoked before any operation commits loser.
4. No operation that aborts with W may start after an operation commits loser.
5. All operations that start after an operation aborts will abort. All operations that start after an operation aborts with L abort with L.

We now consider a trace τ of A1 that is consistent with the mapping M, i.e. that $M(inits(\tau)) \neq \emptyset$. In particular, this implies that the trace τ is non-empty. We will consider arbitrary equivalence classes of histories $e \in eq(M(aborts(\tau)))$. For each e, we show that there exists a history $h_{abort} \in e$ and an interpretation ϕ of events in τ such that ϕ is valid with respect to τ, M, and h_{abort}.

We build h_{abort} as follows. Let A be the set of requests in τ that either commit winner or abort with W. If this set is empty, then, by Invariant 3, there exists a process p that crashes, whose operation started before any operation commits loser. If A is empty, then we add the corresponding request to A, so that A is now non-empty. Next, we define the set B to contain all the requests in τ that committed loser. Finally, we define the set C to contain all the requests in τ that aborted with L. We order requests in sets B and C, respectively, based on the linearization order when reading register *aborted* in line 3. Notice that, by Invariants 4 and 5, we obtain that the concatenation, in order, of requests in sets A, B, and C respects the order of non-overlapping operations. The only ordering which is not fixed at this point is that on members of the set A of requests.

Therefore, consider an arbitrary equivalence class e from the set $eq(M(aborts(\tau)))$. We consider three cases. First, if $aborts(\tau)$ contains no requests, then all the requests committed in τ. In this case, e is trivial, h_{abort} is empty, and the set A is a singleton. To the request in A we append all requests in B, ordered as described above, to obtain history h. For each committed request r we define $\phi(r)$ as the prefix of h that ends with r. It is clear that $\beta(\phi(r)) = response(r)$ for all requests r. We also assign history h to any $init$ requests in τ. It then follows that the resulting trace ϕ_τ satisfies the Abstract properties.

Hence we can assume that $aborts(\tau)$ is not empty. If $aborts(\tau)$ contains no requests that aborted with W, then, by Definition 3, $M(aborts(\tau))$ is the set of histories starting with any request $r \notin aborts(\tau)$ that contain each request in $aborts(\tau)$. All such histories are in the same equivalence class with respect to $\equiv_{aborts(\tau)}$, therefore there exists only one possible choice for e. On the other hand, since there are no requests aborting with W, the set A is a singleton. We build the history h_{abort} by concatenating the sets A, B, and C. Since, by definition, the request in A is not in $aborts(\tau)$, we obtain that $h_{abort} \in e$ as required. To build ϕ, we associate h_{abort} to each aborting and $init$ request; to every committed request r, we associate the prefix of h_{abort} up to r. The resulting mapping ϕ verifies the Abstract properties.

Finally, we consider the case where $aborts(\tau)$ contains at least one request aborting with W. Let S be the set of these requests. By Definition 3, for each request $r \in S$, there exists an equivalence class $e_r \in eq(M(aborts(\tau)))$ with the property that each history in e_r starts with request r. In order to build the history h_{abort} for each such class, first notice that in trace τ the set A must *equal* the set S, by Invariant 2. Hence, given an equivalence class e_r for $r \in S$, we place r first in h_{abort}, after which we place the rest of the requests in S (in the linearization order when reading register $aborted$ on line 3), then requests in B and in C, in the order described above. Clearly, $h_{abort} \in e_r$. To build ϕ, we associate h_{abort} to each aborting and $init$ request; to every committed request r, we associate the prefix of h_{abort} up to r. It is straightforward to check that the resulting mapping verifies the Abstract properties.

Thus, we have verified the safely composable properties in each case, and therefore module A1 is a safely composable implementation of test-and-set under $\mathcal{V}$ and M. □

6.2 The Wait-Free Module

The wait-free module A2, whose pseudocode is given in Algorithm 2, uses a hardware test-and-set T, whose value is initially 0. Participating processes entering the module with $val =$ L automatically return loser; every other participant calls T, and commits the value obtained. We prove that this module is also safely composable. The proof is a simplified version of Lemma 4.

LEMMA 5 (WAIT-FREE MODULE). *The module* A2 *in Figure 2 is a safely composable test-and-set implementation with respect to the sets $\mathcal{V}$ and M given in Definition 3.*

PROOF SKETCH. We consider an arbitrary trace τ of A2. Since τ is valid w.r.t. M, τ must be non-empty. Since the module A2 never aborts requests, it follows that $aborts(\tau) = \emptyset$, therefore the history h_{abort} is empty. Therefore the only requirement is to find a valid interpretation ϕ. It is straightforward that this interpretation is given by the linearization order at the test-and-set object T, after which we append requests with $val = L$, respecting the order of non-overlapping operations. □

```
1  Shared: An array $TAS[\,]$ of speculative test-and-set objects
2  A register $Count$, initially 0
3  procedure reset( )$_i$
4      if $crtWinner$ = true then
5          $Count \leftarrow Count$.read() + 1
6          $crtWinner \leftarrow$ false
7      return
8  procedure test-and-set( )$_i$
9      $c \leftarrow Count$.read()
10     $(res, val) \leftarrow TAS[c].A1$.test-and-set($\perp$)
11     if $res$ = abort then
12         $(res, val) \leftarrow TAS[c].A2$.test-and-set($val$)
13     if $val$ = winner then
14         $crtWinner \leftarrow$ true
15     return $val$

16 Shared: Test-and-set object $T$
17 procedure A2-test-and-set( $val$ )$_i$
18     if $val$ = L then return (commit, loser)
19     return (commit, $T$.test-and-set())
```

Algorithm 2: The resettable test-and-set object.

6.3 Composing and Resetting the Modules

Composing the modules. The above modules have the property that they can be composed in any order to yield a linearizable implementation (in particular, module $A1$ can also be composed with itself). In this extended abstract, we consider A1 composed with A2, as in Figure 1. The composition yields a linearizable wait-free test-and-set object that uses only registers in the absence of step contention, as we show in Lemma 7.

Resetting the Object. The reset mechanism reverts the object to 0 once it has been set, and is also used to revert to a speculative module from the more expensive wait-free module. Practically, this mechanism ensures the *back edge* in the diagram from Figure 1.

The procedure, whose pseudocode is in Algorithm 2, works as follows: we use an array $TAS[\,]$ of wait-free test-and-set objects, each implemented using A1 and A2 and a shared register $Count$, which will be used as a counter. Since the test-and-set is well formed, only the current winner may reset the object [1]. When the winner calls reset, it increments the value of $Count$ by one. Each process wishing to participate in the test-and-set first reads the value of $Count$, and then participates in $TAS[Count]$, accessing each of the modules in this wait-free implementation if necessary. We note that a similar construction was used by Afek et al. [1] to obtain multi-use randomized test-and-set from single-writer registers.

6.4 Proof of Correctness

We first prove that a request may abort from A1 only in the presence of step contention.

LEMMA 6 (PROGRESS). *The algorithm* A1 *never aborts in the absence of step contention.*

PROOF. There are three possibilities for a process p to abort: either on line 17, or on line 23, or on lines 5–6. If p aborts on line 23, then the presence of step contention is straightforward: there must exist another process q that wrote to register P *after* p wrote to P. On the other hand, if p aborts on line 17, then there must exist a process q *concurrent* with p that sets the register $aborted$ to true. While such a process q must exist, it is not immediate that q takes a step during p's execution. Assume for contradiction that q takes no steps during p's execution, in particular that it wrote to the $aborted$ register *before* p invoked its request. In this case, however, the register P cannot equal $\perp$ during p's execution, therefore p will never reach line 17, a contradiction. Finally, if the process aborts on lines 5 or 6, then necessarily another process q wrote to the register $aborted$. It follows that process q experienced step contention, as required. □

We conclude with the proof of correctness of the long-lived implementation given in Algorithm 2. We first prove that the composition of A1 and A2 is a linearizable one-shot test-and-set: from Lemma 4 and Lemma 5 plus Theorem 2, we get that the composition is safely composable. By Theorem 3, the composition is linearizable, since no request ever aborts from the composition. Second, we prove that the long-lived version can be linearized starting from the single-use linearizations.

THEOREM 4. *The composition of modules* A1 *and* A2 *given in Algorithm 2 is a correct wait-free linearizable long-lived test-and-set implementation, which uses only registers and ensures constant step complexity in obstruction-free executions.*

PROOF. First, notice that we can put together Lemma 4, Lemma 5, Theorem 2, and Theorem 3 to obtain the following result.

LEMMA 7. *The composition of modules* A1 *and* A2 *in lines 9-15 of Figure 2 is a linearizable one-shot test-and-set.*

Given this Lemma, consider an arbitrary trace of the long-lived algorithm. For each index i in the array $TAS[\,]$, let Op_i be the operations invoked on $TAS[i]$. From Lemma 7, we know that, for any i, we can linearize the operations in $idOp_i$ (the reset operation is simply added to the linearization order given by the Lemma) to obtain a history h_i of requests. We will then obtain a linearization order for the long-lived object by concatenating the linearization orders h_i in increasing index order. The resulting history respects the consistency requirements since it is valid on each object, and respects the order of non-overlapping operations because the register $Count$ is atomic. The second claim follows immediately from Lemma 6. □

7. CONCLUSIONS AND FUTURE WORK

We have presented a framework for building safely composable shared memory algorithms, and analyzed the cost of composing such algorithms. Our results suggest that this cost is negligible when expressed in terms of step complexity or computational power, if the implementation may take advantage of the semantics of the implemented object. Our framework provides a simple way to design and prove the correctness of speculative concurrent algorithms. One direction for future work would be to apply our framework to implementations of more complex objects, such as queues or fetch-and-increment registers, and to see whether it can yield practical algorithms.

8. REFERENCES

[1] Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *WDAG '92*, pages 85–94, London, UK, 1992. Springer-Verlag.

[2] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[3] Marcos K. Aguilera, Svend Frolund, Vassos Hadzilacos, Stephanie L. Horn, and Sam Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC '07*, pages 23–32, New York, NY, USA, 2007. ACM.

[4] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Generating fast indulgent algorithms. In *ICDCN'11*, pages 41–52, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Dan Alistarh, Rachid Guerraoui, Petr Kuznetsov, and Giuliano Losa. On the complexity of composing shared-memory algorithms. Technical report, EPFL, 2012.

[6] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56:24:1–24:33, July 2009.

[7] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.

[8] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing paxos. *SIGACT News*, 34:47–67, March 2003.

[9] David Dice, Mark Moir, and William Scherer. Quickly reacquirable locks. Technical report, Sun Microsystems, 2003.

[10] Partha Dutta and Rachid Guerraoui. The inherent price of indulgence. In *PODC '02*, pages 88–97, New York, NY, USA, 2002. ACM.

[11] Oded Goldreich and Erez Petrank. The best of both worlds: Guaranteeing termination in fast randomized byzantine agreement protocols. *Inf. Process. Lett.*, 36(1):45–49, 1990.

[12] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *EuroSys '10*, pages 363–376, New York, NY, USA, 2010. ACM.

[13] Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. Speculative linearizability. Technical report, EPFL, 2011. Accepted for publication at PLDI 2012, available at http://lara.epfl.ch/w/slin.

[14] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[15] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[16] Prasad Jayanti. A lower bound on the local time complexity of universal constructions. In *PODC*, pages 183–192, 1998.

[17] Prasad Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC*, pages 295–304, 2003.

[18] Victor Luchangco, Mark Moir, and Nir Shavit. On the uncontended complexity of consensus. In *Proc. of the 17th International Conference on Distributed Computing*, pages 45–59, 2003.

[19] Nalini Vasudevan, Kedar S. Namjoshi, and Stephen A. Edwards. Simple and fast biased locks. In *PACT '10*, pages 65–74, New York, NY, USA, 2010. ACM.

[20] Marko Vukolic. *Abstractions for asynchronous distributed computing with malicious players*. PhD thesis, EPFL, 2008.

APPENDIX

A. ALGORITHMS FOR ABORTABLE CONSENSUS

The SplitConsensus **Algorithm.** The propose procedure contains

```
Shared: $S$, a splitter object, $V, C, D$, registers, initially $\bot$
procedure init( $old$ )$_i$
    $(ind, res) \leftarrow$ propose($old$)
    return $(ind, res)$
procedure propose( $v$ )$_i$
    if splitter.$get()$ = stop then
        if $V \neq \bot$ then
            if $C$ = false then return (commit, $V$)
            else return (abort, $V$)
        $V \leftarrow v$
        if $C$ = false then
            splitter.$reset()$
            return (commit, $v$)
    else
        $C$ = true
        $ret \leftarrow V$
        return (abort, $ret$)
procedure SplitConsensus( $old, v$ )$_i$
    $(ind, res) \leftarrow$ init($old$)
    if $ind$ = abort then return $(abort, old)$
    else
        if $res = \bot$ then return propose($v$)
        else return (commit, $res$)
```

Algorithm 3: The SplitConsensus Algorithm.

the main body of the consensus protocol, first given in [18]. Each process proposes a value and receives commit/abort indication, together with a tentative decision value. If the indication is commit, then processes agree on the tentative value; if the indication is abort, agreement is not guaranteed.

Processes share a splitter object S and atomic registers V, which holds the tentative decided value, and C, a boolean flag signaling contention. See Algorithm 3 for the pseudocode. Each process first accesses the splitter S. If the process successfully acquires the splitter, i.e. returns stop from it, then it proceeds to read the shared value V. If V has already been updated, then the process returns (commit, V) or (abort, V), respectively, depending on whether contention has been detected by reading the flag C or not.

If the register V has not been updated by other processes, then the process updates it with its initial value v. If the contention flag C is false, then the process resets the splitter object and returns a commit to its initial value v. Otherwise, if contention is detected or the process cannot acquire the splitter, it sets the contention flag C to true, and aborts with the current value of the shared value V.

In order to compose several instances of consensus protocols, we introduce a wrapper SplitConsensus function around the propose procedure. This allows the process to suggest, besides its proposal value v for the consensus object, a value old for the object that it may have inherited from another instance of abortable consensus. (If this is the first invocation of a consensus protocol by the process, or if no value is inherited, then the process has $old = \bot$.)

The AbortableBakery **Algorithm.** We focus on the description of the propose procedure, since the wrapper is identical to the previous algorithm. The algorithm is an abortable variant of the solo-fast consensus algorithm presented in [6]. Processes share register arrays (A_i) and (B_i) with $i \in \{1, \ldots, n\}$, initially $\bot$. Process p_j is assigned registers A_j and B_j. Each process tries to impose its input value as the decision by associating it with the highest timestamp in the arrays (A_i) and (B_i). The process will always succeed if there is no step contention; otherwise, it may abort.

Each process p_i first performs a collect on the (A_i) array. The local variable k_i is the minimal value k such that the registers A_i contain no values with timestamp larger than k, and no distinct values with timestamp k. If such a timestamp exists in (A_i), then p_i sets its estimate to the associated value v. Otherwise, process p_i collects the contents of the array (B_i). If the collect is not empty, then it sets its estimate v_i to the value in V' with largest timestamp. If the collect is empty, then p_i keeps its input value as its estimate.

```
Shared:
    Registers $(A_i), (B_i), i \in \{1, n\}$, initially $\bot$
    Register $Quit$, initially false, and $Dec$, initially $\bot$
procedure propose( $input_i$ )$_i$
    $V \leftarrow$ collect$A_i$
    $k_i \leftarrow$ minimal $k$ such that the registers $A_i$ contain no values
    with timestamp $> k$,
    and no different values with timestamp $k$
    if $\exists(k_i, v) \in V$ then $v_i \leftarrow v$
    else
        $V' \leftarrow$ collect$B_i$
        if $V' \neq \emptyset$ then $v_i \leftarrow u \in V$ with highest timestamp in
        $V'$
        else $v_i \leftarrow input_i$
    $A_i \leftarrow (k_i, v_i)$
    $V \leftarrow$ collect$A_i$
    if there are no timestamps larger than $k_i$ and no values other
    than $v_i$ with timestamp $k_i$ in $V$ then
        $B_i \leftarrow (k_i, v_i)$
        $V \leftarrow$ collect$A_i$
        if there are no timestamps larger than $k_i$ and no values
        other than $v_i$ with timestamp $k_i$ in $V$ then
            if $Quit$ = false then
                $Dec = v_i$
                return (commit, $v_i$)
    $Quit \leftarrow$ true
    return (abort, $Dec$)
procedure init( $old$ )$_i$
    $(ind, res) \leftarrow$ propose($old$)
    return $(ind, res)$
procedure AbortableBakery( $old, v$ )$_i$
    $(ind, res) \leftarrow$ init($old$)
    if $ind$ = abort then return $(abort, old)$
    else
        if $res = \bot$ then return propose($v$)
        else return (commit, $res$)
```

Algorithm 4: The AbortableBakery consensus algorithm.

Next, the process writes its current (k_i, v_i) combination in its slot in (A_i), and collects the contents of the array. If there are no timestamps larger than k_i and no values other than v_i with timestamp k_i in the collect, then the process writes (k_i, v_i) in B_i. The process then checks again whether any process wrote a timestamp larger than k_i or a value other than v_i with timestamp k_i in A_i. Otherwise, the process sets the decision value to v_i and returns it. If any of the previous checks fails, then the process has experienced step contention and aborts by setting the $Quit$ register to true, and returns an abort indication together with the current value of the Dec register.

B. SOLO-FAST TEST-AND-SET

The algorithm composed of modules A1 and A2 can be transformed into a solo-fast algorithm by removing the code in the if clause on line 3 of module A1. The resulting algorithm has the property that a process uses the hardware object only when itself encountering step contention, whereas in the current version a process may abort if *another* process experiences step contention. Given this modification, the composed algorithm remains correct, but the proof that A1 is a safely composable object becomes more involved.