Speculative Linearizability

Rachid Guerraoui   Viktor Kuncak   Giuliano Losa

School of Computer and Communication Sciences
Swiss Federal Institute of Technology Lausanne (EPFL)
rachid.guerraoui@epfl.ch   viktor.kuncak@epfl.ch   giuliano.losa@epfl.ch

Abstract

Linearizability is a key design methodology for reasoning about implementations of concurrent abstract data types in both shared memory and message passing systems. It provides the illusion that operations execute sequentially and fault-free, despite the asynchrony and faults inherent to a concurrent system, especially a distributed one. A key property of linearizability is inter-object composability: a system composed of linearizable objects is itself linearizable. However, devising linearizable objects is very difficult, requiring complex algorithms to work correctly under general circumstances, and often resulting in bad average-case behavior. Concurrent algorithm designers therefore resort to speculation: optimizing algorithms to handle common scenarios more efficiently. The outcome are even more complex protocols, for which it is no longer tractable to prove their correctness.

To simplify the design of efficient yet robust linearizable protocols, we propose a new notion: *speculative linearizability*. This property is as general as linearizability, yet it allows intra-object composability: the correctness of independent protocol phases implies the correctness of their composition. In particular, it allows the designer to focus solely on the proof of an optimization and derive the correctness of the overall protocol from the correctness of the existing, non-optimized one.

Our notion of protocol phases allows processes to independently switch from one phase to another, without requiring them to reach agreement to determine the change of a phase. To illustrate the applicability of our methodology, we show how examples of speculative algorithms for shared memory and asynchronous message passing naturally fit into our framework.

We rigorously define speculative linearizability and prove our intra-object composition theorem in a trace-based as well as an automaton-based model. To obtain a further degree of confidence, we also formalize and mechanically check the theorem in the automaton-based model, using the I/O automata framework within the Isabelle interactive proof assistant. We expect our framework to enable, for the first time, scalable specifications and mechanical proofs of speculative implementations of linearizable objects.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming

Keywords Speculation, Distributed Systems, Modularity

1. Introduction

The correctness of a system of processes communicating through linearizable objects [12, 17, 18] reduces to the correctness of the sequential executions of that system. In other words, linearizability reduces the difficult problem of reasoning about concurrent data types to that of reasoning about sequential ones. In this sense, the use of linearizable objects greatly simplifies the construction of concurrent systems. At first glance, the design and implementation of linearizable objects themselves looks also simple. One can focus on each object independently, design the underlying linearizable algorithm, implement and test it, and then compose it with algorithms ensuring the linearizability of the other objects of the system. In short, linearizability is preserved by *inter-object composition*: a set of objects is linearizable if and only if each object is linearizable when considered independently of the others.

However, it is extremely difficult to devise efficient and robust implementations of a linearizable object, even when considered independently from the others. The difficulty stems from the following dilemma. On the one hand, achieving robustness in a concurrent system requires assuming that the scheduling of processes, communication, and faults is completely nondeterministic. The object does not know which execution is going to unfold but needs to deliver a response to each of its method invocations no matter the execution. The object has to prepare for all scenarios, using up its resources for that task. This *conservative* approach typically yields *wait-free* [11] but inefficient strategies. On the other hand, achieving efficiency requires *speculating* that only a small subset of executions is likely to occur [23]. This is practically appealing because a real system typically spends most of its time in a common case and seldom experiences periods of alternative executions. Should the object focus on a very small subset of executions, it would save resources by preparing only for that restricted subset. Typically, the common cases that are usually considered in practice are those where the system is synchronous, there is no failure, contention is not very high, a specific conditional branch is to be considered, etc. At a high level, a speculative system repeats the following steps:

1. Speculate about the likelihood of certain executions and choose a corresponding algorithm.
2. Initialize the chosen algorithm.
3. Monitor the algorithm to estimate if the speculation was accurate. If not, abort the algorithm and recover a consistent state.

Examples of speculation include the Ethernet protocol, where processes speculatively occupy a single-user communication medium before backing off if collision is detected, or branch prediction in microprocessors, where the processor speculates that a particular branch in the code will be taken before discarding its computation if this is not the case. More recent instances of speculation include optimistic replication [15] or adaptive mutual exclusion [14]. In fact, most practical concurrent systems are speculative. In general, speculative systems may choose between many different options, or *speculation phases*, in order to closely match a changing common case.

To illustrate the challenges addressed in this work, consider a speculative algorithm that may switch between any two of n speculation phases in a safe manner. That is, recovering a consistent state of the first phase and then initializing the second. Doing so in an ad-hoc manner poses two major *scalability* problems to the design process. First, there are $O(n^2)$ different switching cases that need to be carefully handled in order to preserve linearizability. Second, adding a new phase to an object composed of n phases built ad-hoc may require deep changes to all of the n existing phases.

The case of State Machine Replication [16] (abbreviated SMR), used to build robust linearizable object implementations, illustrates those problems. Non-speculative SMR algorithms like Paxos [7] or PBFT [6] are notoriously hard to understand. The formal correctness proof of Disk Paxos [13] took about 7000 lines. Only an informal proof, 35 pages long, of a simplified version of PBFT is known to the authors [5]. Speculative SMR protocols are even harder. For instance, the Zyzzyva [15] protocol combines PBFT with a fast path implemented by a simple agreement protocol. The fast path is more efficient than PBFT when there are no failures. In the advent of failures, the fast path cannot make progress and Zyzzyva falls back to executing PBFT. The ad-hoc composition of the fast path with PBFT required deep changes to both algorithms and resulted in an entanglement that is hardly understandable. Although PBFT had previously been widely tested [5], Zyzzyva suffered from fragile implementations and no correctness proof has ever been proposed. In fact, Zyzzyva, being restricted to two phases, is very fragile [24]. If the common case is not what is expected by the fast path one falls-back to PBFT, making the optimization useless. An adversary can easily weaken the system by always making it abort the fast path and go through the slow one. Introducing a new dimension of speculation might make the protocol more robust but would require a new ad-hoc composition, including an alternative fast-path, at a cost comparable to the effort needed to build Zyzzyva from scratch, namely a Dantean effort. Given the diversity of situations encountered in practice, we are convinced that this ad-hoc approach is simply intractable.

Contributions. We propose in this paper a framework for devising, reasoning about, and mechanically verifying effective and robust linearizable implementations of concurrent data types in a scalable way. In short, we show how speculative implementations of linearizable objects can be devised and analyzed in an *incremental* manner. One can focus on each dimension of speculation independently of the others. For example, a programmer can first devise an algorithm with no speculation, implement, test, and prove it correct, then augment it later with an optimization that speculates that a certain subset of executions is more likely to occur (say there is no failure). This optimization sub-algorithm is composed with the original one as is. Later, another programmer can consider other dimensions of speculations (say there is no contention or asynchrony) and add new phases, still without modifying the original implementation and proofs.

At the heart of our framework lies our new notion, *speculative linearizability*. Speculative linearizability encompasses the notion of *switch actions*, which makes it significantly richer than linearizability, yet it reduces to linearizability if these actions are ignored. Speculative linearizability augments classical linearizability with a new aspect of composition. Not only a system of concurrent objects can be considered correct if each of them is correct (*inter-object composition*), but a set of algorithms implementing different speculation phases of the *same* object is correct if each of them is correct (*intra-object composition*). We express this new aspect

```
1: //Shared Register $V$, Initially $\perp$
2: **Function** *propose*$(val)$:
3: **if** $V = \perp$ **then**
4:     $V \leftarrow val$
5:     **return** $val$
6: **else**
7:     **return** $V$
8: **end if**
```

Figure 1. Consensus Specification

through a new composition theorem which we state and prove. Intuitively, speculative linearizability captures the idea of *safe* and *live abortability*. An implementation can abort if the assumptions behind speculation reveals wrong. When it does abort, it does so in a safe manner, by preserving the consistency (linearizability) of the object state. Moreover, the abort is also performed in a live manner, because a new protocol phase can resume and make progress. Processes can switch independently from one phase to another, without the need to reach agreement. Our notion of speculative linearizability is itself based on a new definition of linearizability. This definition is interesting in its own right because it enables a more local form of reasoning than the original definition and it allows for repeated events, which are the norm in practice. We have proved that our new definition of linearizability is equivalent to the classical definition.

The generality of our framework and its Isabelle/HOL formalization [10] sets our work apart from state of the art in the area [1, 2, 8]: we enable for the first time scalable mechanically-checked development of linearizable objects of arbitrary abstract data types.

Additional proofs and formalizations are available in [9, 10], as well as at `http://lara.epfl.ch/w/slin`.

2. Putting Speculative Linearizability to Work

In this section we provide intuition for speculative linearizability and its key property: intra-object composability. Our goal is to motivate correct-by-construction design of objects based on speculation phases. As an illustration, we present two speculative implementations of a consensus object. Each is composed of two speculation phases. The first applies to the message-passing computation model and the other to the shared-memory model. We analyze each phase of those speculative implementations in isolation and conclude, using the intra-object composition theorem, that their compositions are correct implementations of consensus.

2.1 Message-Passing Consensus

Our first example is a consensus implementation in a system composed of client and server processes which communicate by asynchronous message passing and which may crash at any point. Our goal is to implement consensus among the clients. Notable use cases of consensus in message-passing systems include Google's Chubby distributed lock service [4] and the Gaios reliable data store [3].

Consensus allows a set of clients to agree on a common value. Clients each propose a value and should receive a common decision value taken among their proposals. We consider implementations that offer an invocation function *propose*$(value)$ to each of its clients. The invocation *propose*(val) returns another value indicating the common decision. When a process returns from *propose*$(value)$ with a value d, we say the it *decides* value d. An algorithm is linearizable with respect to the consensus data type if all calls to *propose*(val) appear as if they executed atomically the algorithm in Figure 1 at some point after their invocation (we assume that proposal are different from $\perp$).

In an asynchronous message-passing system where processes may crash, there is no component which can reliably hold state. Hence, some crucial data may become unavailable due to the failure of a process. For example, a process may decide some value and then crash before informing any other process of its decision. In this case other processes have no way to know which value the crashed process decided and they cannot decide without risking to violate consensus.

Paxos [7] is an algorithm that implements consensus if less than half of the servers may fail. Paxos has a minimum latency of 3 message delays. However, suppose that executions are fault free and contention free (i.e. the time intervals delimited by corresponding invocations and responses do not overlap). Under this assumption, very simple algorithms can solve consensus much faster than Paxos, which still has a minimum latency of 3 message delays.

We would like to optimize Paxos for the fault-free and contention-free case. This could be done ad-hoc by adding a fast path inside Paxos. However, Paxos is a very intricate algorithm: the proof of Disk Paxos [7], a variant of Paxos, is 7000 lines long [13]. Adding a fast path to Paxos would require re-examining the correctness of the entire algorithm, because the fast path is interleaved with executions of the others parts. The entire proof would, in principle, need to be rewritten.

Our framework allows us to optimize Paxos without directly modifying the original algorithm, and enables reasoning about the correctness of the optimization independently of the basic Paxos itself. To use our framework, we wrap Paxos in a simple interface that allows composition with other speculation phases. This requires adding a trivial level of indirection and makes Paxos a speculation phase in its own right. Speculative linearizability ensures that any speculatively linearizable speculation phase can be composed with the Paxos speculation phase to form a correct implementation of consensus. Moreover, speculative linearizability, instantiated for consensus, is a property independent of Paxos. Therefore, from the point of view of algorithm designers, speculative linearizability hides the internal complexity of Paxos.

In this example we compose Paxos with the *Quorum* speculation phase, inspired from [8]. Quorum manages to decide on the value in only 2 message delays, whenever there is neither contention nor faults. If faults or contention happen, Quorum cannot decide and instead passes control to Paxos. In this case, we say that Quorum *switches* to Paxos. From then on, Paxos takes over the execution and can decide as long as less than half of the processes are faulty. To highlight that Paxos is used as a backup when optimization by Quorum is not possible, we call the Paxos speculation phase Backup. Using speculative linearizability, we will prove that the composition of Quorum and Backup is linearizable by reasoning about Quorum in isolation and by reusing an existing proof of Paxos.

By combining Quorum and Backup we obtain a system that is optimized for contention-free and fault-free loads while still remaining correct in all other conditions under which the Backup is correct. Such an approach has proved empirically successful in recent work [8]; the present paper provides the underlying theoretical foundations for such approaches.

We now describe more precisely the Quorum and Backup speculation phases. Their interface is that of a consensus object, as defined above, augmented with a *switch-to-backup*(sv) call. The *switch-to-backup*(sv) call is used by a client to switch from the Quorum phase to Backup, i.e. *switch-to-backup*(sv) is a procedure of Backup which may be called by a client executing Quorum. The sv parameter is called a *switch value*. A client calling *switch-to-backup*(sv) transfer its pending invocation to Backup and provides sv as an indication to Backup on how to take over Quorum's execution.

Informally, the Quorum algorithm works as follows:

- Upon *propose*(v), a client c broadcasts its proposal to all server processes and waits for accept messages. It also stores v in local variable $\mathit{proposal}_c$ of its memory and starts a local timer t_c.
- When a server process receives a proposal v from a client c:
    - If it has not sent any accept message, it sends an accept message $accept(v)$ to c.
    - If it has already sent an accept message $accept(v')$, it sends $accept(v')$ to c.
- If a client receives two different accept messages, it switches to Backup by calling and returning *switch-to-backup*$(\mathit{proposal}_c)$.
- If a client receives the same accept messages $accept(v)$ from all the servers, then it decides v.
- If timer t_c expires, then c waits for at least one message $accept(v')$ from a server, or selects one v' if it has some messages of the form $accept(v')$ already. It then switches to Backup by calling *switch-to-backup*(v').

The Backup phase is Lamport's Paxos algorithm where clients have the role of proposers and learners, while servers have the role of acceptors. Backup treats the switch calls from Quorum as regular proposals. In other words, upon a call *switch-to-backup*(v), a client process proposes the value v to the Paxos algorithm.

By composing Quorum and Backup we obtain a linearizable implementation of consensus that is optimized for crash-free and contention-free workloads. This new optimized implementation of consensus was obtained by composing a black-box implementation of Paxos with a much simpler protocol, Quorum, which makes progress only under particular conditions.

We next uncover the principles behind this approach and we will show the correctness of our optimized protocol by reasoning about each speculation phase in isolation, independently of each other. Thanks to our modular approach, we will be able to rely the existing proof of Paxos [13] to show the correctness of Backup with minor effort.

2.2 Linearizability

Before explaining *speculative* linearizability, we briefly review linearizability itself: we provide both its classical definition and our new formalization.

We consider a set of concurrent clients accessing an object through *invocation* procedures whose execution ends by returning a *response*, as in the consensus examples above. We assume that each client is sequential. In other words, a client does not invoke the object before having returned from its preceding invocation.

An object is accessed *sequentially* if every response to an invocation immediately follows the invocation. There must be no other response or invocation in between them. For example, the specification of a consensus object states that all processes return the same decision value and that this value must have been proposed by some process before it can be returned. Hence, in a sequential execution, the first proposed value must be returned to every process. Note that this corresponds to executing sequentially the algorithm in Figure 1: The first process executing will impose its value to all others.

We consider only deterministic objects. Therefore, in the sequential case, the response to a given invocation is determined by the sequence of past invocations that the object received. Therefore, we represent a sequential execution by its subsequence containing only and all invocations in the execution. We call such sequences of invocations *histories*.

When accessed concurrently, invocations and responses of different processes may be arbitrarily interleaved. The sequences of invocations and responses that may be observed at the interface of an object that is accessed concurrently are called *traces*. The occurrence of an invocation or a response in a trace is called an *event*, or *action*.

Linearizability is defined with respect to a sequential specification, which is a description of the allowed behaviors of the object when it is accessed sequentially. Intuitively, a trace is linearizable if each invocation appears to take effect at a single point in time after the invocation call and before the corresponding response. This intuitive definition can be restated as follows: A trace t is linearizable if we can associate, to each response r returned by a client c, a history h (called *commit history*) such that:

1. The response r is the response obtained in the sequential execution represented by history h.
2. All invocations in the history h are invoked in the trace t before the response r is returned.
3. The history h ends with the last invocation of client c.
4. For all commit histories h_1 and h_2, either h_1 is a prefix of h_2 or h_2 is a prefix of h_1.

This definition of linearizability is stated precisely in Section 4. In the terminology of Section 4, condition 1 above says that history h *explains* response r. Condition 1 is specific to a particular abstract data type. Conditions 2 and 3 correspond to *Validity*, and 4 to *Commit Order*. They do not depend on the particular ADT considered. We have proved [9] that this definition is equivalent to the original definition of linearizability [12].

As an example, consider a trace of a consensus object such that client c_1 proposes value v_1, then client c_2 proposes value v_2, then client c_2 returns v_2, and finally client c_1 returns v_2. This execution is linearizable because the returned values are as if the invocation of client c_2 was executed atomically before the invocation of client c_1. To show this, associate the history $[\mathit{propose}(v_2)]$ to the response of c_2 and the history $[\mathit{propose}(v_2), \mathit{propose}(v_1)]$ to the response of c_1. It is easy to check that all four conditions above are satisfied. Notably, when applied sequentially to a consensus object, both histories lead to response v_2. By the definition above, the trace is linearizable. Examples of traces that are *not* linearizable include:

$[c_1$ proposes v_1, c_2 proposes v_2, c_1 decides v_1, c_2 decides $v_2]$, as well as:

$[c_1$ proposes v_1, c_1 decides v_2, c_2 proposes v_2, c_2 decides $v_2]$.

2.3 Speculative Linearizability

We now consider modular implementations composed of two speculation phases (the definition generalizes to any number of phases). As in the composition of the Quorum and Backup algorithms, each client process starts by executing the first phase and may *switch* to the second phase using a procedure call, providing an argument that we call a *switch value* along with its pending invocation. In the examples of this section we omit the pending invocation from switch calls for the sake of conciseness.

We would like to reason about each speculation phase in isolation. For this purpose, we require that the switch values provided by the clients when changing phase be the only information passed from one speculation phase to the other. No side effects are permitted across speculation phases. Speculation phases are thus separated by a clear interface: a call to the phase's switch procedure with a switch value and a pending invocation as arguments.

Speculative linearizability is a property of speculation phases that relates the switch events and the invocations received by the speculation phase to the responses and switch events produced by the same phase. Moreover, no assumption other that trivial well-formedness conditions is made on the received switch events and invocations. This allows us to check that a phase is speculatively linearizable by reasoning about the phase independently of other phases it might be composed with.

Speculative linearizability is an extension of linearizability to traces that contain switch calls. In the definition of linearizability stated above, we associated to each response returned by a client a history of inputs with certain properties. In the same spirit, in addition to associating histories to responses returned, we now associate to each switch call in the trace a history of inputs. With respect to the first phase, we call these histories *abort histories*. With respect to the second phase, we call them *init histories*.

Such histories associated to switch calls represent possible linearizations of the execution of the first speculation phase. The idea is that, with the knowledge of how the execution of the first phase was linearized, the second phase can continue execution without violating linearizability. In order for a speculation phase to deduce a possible current linearization from a switch value, we suppose that all speculation phases agree on a common mapping from switch values to histories. We denote this mapping by r_{init}.

Depending on the particular data type being implemented, there are some sets of histories whose members all bring the system in the same logical state. In other words, the response to a new invocation is independent of which history of such a set was executed before. We call histories in such sets *equivalent* with respect to the data type. Consider our consensus example. Any set of histories whose members start with the same proposal is a set of equivalent histories: because the first proposal is the decision value, any new invocation will return the first proposed value. Processes invoking the object at a point after such a history has been executed cannot tell which history in the set actually happened. Because of this observation, we require that speculation phases agree on a mapping from switch values to *sets* of equivalent histories, as opposed to a single history. This allows us to use a compact representation of histories while ensuring that enough information is available for the second phase to take over the execution.

Two other principles, fault tolerance and avoidance of unnecessary synchronization, influenced our definition. For fault tolerance, we do not rely on just one client passing a single switch value. For efficiency reasons, we avoid the requirement that all clients switch with the same value because it would imply solving a potentially expensive agreement problem. Hence, the switch values of different clients may be different and no single switch call determines the execution of the second phase.

Applied to the first phase, speculative linearizability requires that the first phase be linearizable. Moreover, for all traces of the first phase, it must be possible to associate histories to the switch events so that each such history is a possible linearization of the trace up to the considered event and each association from switch event to history respects the common mapping that phases agree on. Finally, the longest common prefix of all abort histories must be a full linearization of the trace of the first phase. The last two requirements correspond to the *Abort Order* property of Section 5.

Applied to the second phase, speculative linearizability says that for all traces t of the second phase, for all possible associations of histories to switch values that respect the common mapping agreed on, by concatenating t to the trace represented by the longest common prefix of all init histories and by replacing switch calls with the pending invocation they contain, one should obtain a linearizable trace. This corresponds to the *Init Order* property of Section 5.

Having defined the new concept of speculative linearizability, we will show that the composition of any number of speculatively linearizable phases is a linearizable algorithm. This result is a corollary of our intra-object composition theorem, which states that the composition of two speculation phases is itself a speculation phase.

2.4 Applying Speculative Linearizability to Quorum+Backup

We next show that our message-passing speculation phases, Quorum and Backup, satisfy speculative linearizability. As explained above, we view the values passed when switching from Quorum to Backup as representing sets of histories equivalent to some linearization of Quorum's execution. To this end we assumed that speculation phases agreed on a common mapping from switch events to sets of histories. In this example, the mapping of a switch event of client c with switch value v is the set of all histories starting with invocation *propose*(v) from a client $c' \neq c$ and containing only invocations from clients other than c. Hence, a client switching to Backup with switch value v indicates to Backup that any history in the set mapped to v is a possible linearization of Quorum's execution.

Quorum is speculatively linearizable. We prove that Quorum is speculatively linearizable in two steps. First, we show that Quorum satisfies the following three invariants. Then we show that those invariants imply speculative linearizability.

I1: If some client c decides value v then all clients that switch, either before or after c decides, do so with value v.

I2: If some client c decides value v then all clients that decide do so with value v.

I3: All clients that switch or decide do so with a value that was proposed before they switch or decide.

In Quorum, if there are no faults nor contention, a client broadcasting a proposal will receive identical accept messages from all server processes and will return the value contained in the accept messages. However, if contention causes proposals from different clients to be received in different orders at different servers, or if a server fails or messages are lost, clients will switch to Backup because different accept messages will be received or not enough messages will be received before the timers expire.

Observe that if a client returns a value v in Quorum then all servers will reply with $accept(v)$ to all the other clients. This is because a client only returns value v if all servers send it accept messages with value v, and a server always responds with the same accept message. Hence, if some client decided, all the other clients will either return the same value v or will switch to Backup with the initialization value v in case message loss or server crashes cause their timers to expire. From these observation we can conclude that the invariants I1 and I2 are satisfied by any trace of Quorum.

Moreover, as servers respond with an accept message containing the first proposal received, client processes return or switch with a value that was proposed before. Hence, the invariant I3 holds.

We now prove that any trace satisfying I1, I2, and I3 satisfies speculative linearizability. Consider a trace t of Quorum that satisfies I1, I2, and I3.

According to the definition of speculative linearizability for the first phase, we begin by showing that t is linearizable. If no client decides then the trace t is trivially linearizable. Therefore assume that some clients decide. The invariant I2 implies that all decisions in the trace are the same. Let v be the common decision. The invariant I3 implies that some client, noted *winner*, proposed v before any client decided v. Let the history h be such that h starts with *winner*'s proposal and the sub-sequence of h starting at position 2 is equal to the subsequence of t containing all the proposals of the clients that decide and that are not *winner*. The history h represents a linearization of trace t. We satisfy our definition of linearizability by associating to each decision from a client c the history h truncated just after the proposal of c. The linearizability of Quorum follows.

To prove speculative linearizability, it remains to show that we can associate appropriate histories to each switch event in t. We consider two cases.

First, assume that some clients decided. Then, by invariant I1 we know that clients decide or switch with the same value v. Let history h be as defined above. To each switch event in t we associate h. By definition of h, h starts with the common decision v and all clients that switch do so with value v. Moreover the clients that switch do not decide; their invocations thus do not appear in h. Hence, the association from switch events to histories respects the common mapping. Moreover, h is a linearization of trace t. Therefore, each history associated to a switch event is a linearization of t. Finally, the longest common prefix of all histories associated to switch events is obviously h itself, and h is a linearization of trace t. We conclude that t satisfies speculative linearizability.

Now assume that no client decides. In this case the trace is trivially linearizable. Moreover, it is easy to associate histories to switch events, so that the longest common prefix of all such histories is empty. We thus trivially have that t satisfies speculative linearizability.

Note that a client that does not fail returns or switches at the latest when its timer expires. Quorum is thus wait-free.

Backup is speculatively linearizable. As for Quorum, we proceed by first abstracting Backup using simple invariants and then proving that those invariants imply speculative linearizability. Since Paxos has been proved correct in the past [13], we can trivially abstract Backup using the following two invariants:

I4 All clients decide the same value.

I5 All clients decide a switch value previously submitted by some client.

Consider a trace t satisfying invariants I4 and I5. We consider two cases. First assume that all switch values are the same and equal to v. Then, by definition of the common mapping from switch events to histories, for all associations of histories to switch events respecting the common mapping, the longest common prefix h of the histories associated to switch values is such that h starts with an invocation *propose*(v) from a client c that doesn't execute in t and contains only invocations by clients that don't execute in t. Let t_h be the trace represented by history h. By invariants I4 and I5 we know that all clients decide value v in t. Let t' be the trace t where switch calls are replaced by the pending invocation they contain. Consider the concatenation $t_h@t'$ of t_h and t'. Let the history h' be the concatenation of h and some ordering of the proposals in t'. Because h represents t_h and because all clients decide v in t, we have that h' is a linearization of $t_h@t'$. Hence, t satisfies speculative linearizability.

Now suppose that at least two switch values are different. Then for any association of histories to switch events respecting the common mapping, the longest common prefix of the histories associated to switch events is the empty history. From invariants I4 and I5 we have that all processes decide on one of the switch values submitted before any decision. Hence, t is linearizable, and so is the concatenation of the empty history and t. Therefore, t satisfies speculative linearizability.

We have proved that Quorum and Backup satisfy speculative linearizability. Therefore, using the intra-object composition theorem that we will prove below, we conclude that the composition of Quorum and Backup is a linearizable implementation of consensus.

Using speculative linearizability, we have optimized the Paxos algorithm to obtain an algorithm that has a latency of two message delays in the absence of faults and contention. Implementing this optimization by modifying the Paxos algorithm would have meant modifying a notoriously intricate distributed algorithm. Instead, by using our framework, we were able to optimize Paxos without modifying it. Moreover, thanks to the intra-object composition theorem, we easily proved the optimized protocol correct just by proving the optimization speculatively linearizable and by relying on the correctness of Paxos.

2.5 Shared-Memory Consensus

To illustrate the broad applicability of our framework we now apply it to an algorithm in the shared-memory model. We consider a consensus implementation in an asynchronous shared-memory system.

Consensus can be implemented on modern microprocessors using the wait-free compare-and-swap (CAS) instruction, but this instruction may be slower than an atomic register access. It has been proved [11] that wait-free consensus cannot be implemented with registers.

However, assuming that all executions are contention free (i.e. sequential), Figure 1 implements consensus using only registers. Hence the following question: is it possible to devise an object that uses only registers in contention-free executions but that always executes correctly? We obtain such an object by composing a register-based speculation phase called *RCons*, shown in Figure 2, and a CAS-based speculation phase called *CASCons*, shown in Figure 3. The CAS-based speculation phase is a straightforward implementation of consensus using the CAS operation. Both speculation phase are inspired by [25]. It turns out that these speculation phases, like many other speculation-based objects, are instances of our framework.

The register-based consensus uses a wait-free splitter algorithm [19]. The splitter guarantees that at most one process returns with true and all others return with false. Moreover, it guarantees that, in the absence of contention, exactly one process returns with true. A splitter can be implemented using only registers, as in our example implementation in Figure 2.

A client invokes the register-based speculation phase using the function *propose*(val), where val is the value that it proposes. To simplify the notation, we assume that, at the time of the call, the caller identifier is stored in the special variable c. The register-based speculation phase, *RCons*, can return the content of register V, or it can switch to the CAS-based speculation phase, *CASCons*, by invoking the function *switch-to-CASCons*(V) and returning its result. In the latter case, we say that a client *switches* with value v, where v is the content of register V when it was last read. Such switching is the basic mechanism of composing speculation phases in our framework.

As for Quorum and Backup, we will show that the composition of the phases *RCons* and *CASCons* form a linearizable implementation of consensus by a modular reasoning. Thanks to the intra-object composition theorem, we will get the correctness of the composition of *RCons* and *CASCons* by analyzing each in isolation.

Let us now show that the *RCons* and *CASCons* algorithms are speculatively linearizable.

As for Quorum and Backup, we will first consider *RCons* in isolation and, given an arbitrary execution of *RCons*, we will show that we can associate appropriate histories to the responses returned and to the switch calls. We will then consider *CASCons* in isolation and show that, given an arbitrary execution of *CASCons* and an arbitrary association of histories to the switch calls of *CASCons* (respecting the common mapping), we can associate appropriate histories to the responses appearing in the trace.

We first show that the invariants I1, I2, and I3 defined for Quorum also hold of the traces of *RCons*. To establish these invariants, we first observe the following: By property of the splitter, at most one client executes lines 15 to 18. Moreover, if one client c does so and returns, it makes sure that other clients will either return with its value at line 12 or switch with its value at line 27. Indeed, if the variable *Contention* is false at line 16, it means that no process got past line 23. Therefore, all processes that switch will find c's value in variable *V* at line 25. We conclude that the invariants I1 and I2 hold on all executions of *RCons*. If some client c returns value v then all clients that switch, either before or after c returns, do so with value v and if some client c returns value v then all clients that return do so with value v. Next, observe that all clients return or switch with their own value or with the content of register V, which can only contain proposed values. Consequently, the invariant I3 is also an invariant on all executions of *RCons*: All clients that switch do so with a value that was proposed before they switch.

Similarly, the *CASCons* phase can be abstracted by the same invariants that we used to abstract the Backup speculation phase: All clients return the same value and all clients return a switch value previously submitted by some client.

When we proved that Quorum and Backup are speculatively linearizable, we showed that the invariants I1, I2, and I3 imply speculative linearizability of the first phase and that the invariants I4 and I5 imply the speculative linearizability of the second phase.

```
 1: Object RCons
 2: //Shared Register $V$, Initially $\perp$; and $D$, Initially $\perp$
 3: //Shared Register $Contention \in \{true, false\}$, Initially false
 4: //Shared Registers: $Y \in \{true, false\}$, Initially false; and $X$
 5: //Local Variable $v$; local constant $c$ denoting the caller's identifier
 6: Function propose($val$):
 7: $v \leftarrow val$
 8: if $D \neq \perp$ then
 9:     return $D$
10: end if
11: if splitter() $=$ true then
12:     $V \leftarrow v$
13:     if $\neg Contention$ then
14:         $D \leftarrow v$
15:         return $v$
16:     else
17:         return switch-to-CASCons($v$)
18:     end if
19: else
20:     $Contention = true$
21:     if $V \neq \perp$ then
22:         $v \leftarrow V$
23:     end if
24:     return switch-to-CASCons($v$)
25: end if
26: Function splitter():
27: $X \leftarrow c$ //Remember that $c$ is the caller's identifier
28: if $Y =$ true then
29:     return false
30: end if
31: $Y \leftarrow$ true
32: if $X = c$ then
33:     return true
34: else
35:     return false
36: end if
```

Figure 2. Register-Based Speculative Consensus

```
1: Object CASCons
2: //Shared Register $D$ Initially $\perp$
3: Function switch-to-CASCons($val$):
4: return $CAS(D, \perp, val)$
5: Function propose($val$):
6: //Since processes have to call switch-to-CASCons first, we know that
   the consensus has already been won, hence just return $D$.
7: return $D$
```

Figure 3. CAS-Based Speculative Consensus

Since we established that the traces of *RCons* satisfy I1, I2, and I3 and that the traces of *CASCons* satisfy I4 and I5 we can conclude that *RCons* and *CASCons* satisfy speculative linearizability.

We will now formally define speculative linearizability.

3. Trace Properties

In this section we define trace properties and the operations they support. Trace properties are our model of distributed systems.

A trace property is a set of finite sequences of events. We use trace properties to describe the set of all finite sequences that a system can generate, as well as the desired properties of systems, such as linearizability with respect to a given abstract data type. We restrict ourselves to finite sequences because our work deals with safety properties only.

Sequences. We write $[1..n]$ for the set $\{1, 2, \ldots, n\}$. A *sequence* s of length n with elements from a set E is a function $s : [1..n] \rightarrow E$. We denote n by $|s|$. We write E^* for the set of all sequences of elements in E. We write $[e_1, e_2, \ldots, e_n]$ for the sequence s such that $s(1) = e_1$, $s(2) = e_2$, …, and $s(n) = e_n$. If $s = [e_1, e_2, \ldots, e_n]$ then $s::e_{n+1}$ is the sequence $[e_1, e_2, \ldots, e_n, e_{n+1}]$. If $s = [e_1, e_2, \ldots, e_n]$ and $s' = [e'_1, e'_2, \ldots, e'_n]$ then $s:::s' = [e_1, \ldots, e_n, e'_1, \ldots, e'_n]$ and, if $m < |s|$, we write $s_{|m}$ for the the sequence $[e_1, e_2, \ldots, e_m]$. We say that a sequence s is a *prefix* of a sequence s' iff there exists a sequence s'' such that $s' = s:::s''$. We say s is a *strict prefix* of s' iff the s'' is non-empty. Given a set of sequences P, the longest common prefix of set P is the longest among the sequences s' such that for all sequence $s \in P$, s' is a prefix of s.

Multisets. We represent multisets of elements of set E by a multiplicity function $E \rightarrow \mathbb{N}$. Given two multisets m_1 and m_2 and $e \in E$, we define $(m_1 \cup m_2)(e) = max(m_1(e), m_2(e))$ and $(m_1 \uplus m_2)(e) = m_1(e) + m_2(e)$. Moreover $m_1 \subseteq m_2$ iff for all $e \in E$, $m_1(e) \leq m_2(e)$. Let $elems : E^* \rightarrow (E \rightarrow \mathbb{N})$ be a function that given a sequence returns a multiset representing all the elements found in the sequence and their number of occurences. Given a sequence s we write $e \in s$ iff $elems(s)(e) > 0$.

Trace Properties. A trace is a sequence of *actions* which represents *events* happening at the interface between a system and its environment. An event occurs at some point in time and has no duration. We can approximate invoking a method and starting to execute it as one action. The execution of a method from start to finish, however, is typically not an action because it has a positive duration in time. We would represent it with an invocation action and a return action.

We classify actions into input and output actions. This classification is especially important to define composition of systems, because we need to know which actions of a component affect another component. The classification is described by a *signature*. A signature *sig* is a pair (in, out) where in is a set of input actions and out is a disjoint set of output actions. We require in and out to be disjoint. We denote by $acts(sig)$ the set of all actions in signature *sig*. When all the actions of a trace t belong to $acts(sig)$ we say that t is a trace in *sig*.

Definition 1. *A* trace property *is a pair* $P = (sig, Traces)$ *where sig is a signature and Traces is a set of traces in* $acts(sig)$.

We denote *sig* by $sig(P)$ and *Traces* by $Traces(P)$. We allow to write $in(P)$ instead of $in(sig(P))$ and similarly for the other sets of actions defined above.

We say that a trace property Q satisfies a trace property P, denoted $Q \models P$, if Q's signature is equal to P's signature and the set of trace of Q is a subset of the set of traces of P:

$$Q \models P \Leftrightarrow \quad (sig(Q) = sig(P) \wedge Traces(Q) \subseteq Traces(P)) .$$

Composition of Trace Properties. Trace properties may be composed only if their signatures are *compatible*. Signatures sig_1 and sig_2 are compatible if and only if they have no output actions in common, that is $out(sig_1)$ is disjoint from $out(sig_2)$. As a result, $out(sig_1) \cap acts(sig_2) \subseteq in(sig_2)$ and vice versa. We define the *projection* $proj(t, A)$ of a trace t onto a set A of actions as the sequence obtained by removing all actions of t that are not in A. For example, $proj([x, y, x', z, y', z, y, z, y], \{x', y'\}) = [x', y']$.

Definition 2. *If* sig_1 *is compatible with* sig_2 *then the composition* $P = P_1||P_2$ *is such that:*

- *The signature* $sig(P)$ *is such that* $in(P) = (in(P_1) \cup in(P_2)) \setminus (out(P_1) \cup out(P_2))$ *and* $out(P) = out(P_1) \cup out(P_2)$.
- *The set of traces* $Traces(P)$ *is the largest set of traces in* $acts(P)$ *such that if* $t \in Traces(P)$ *then* $proj(t, acts(P_1)) \in Traces(P_1)$ *and* $proj(t, acts(P_2)) \in Traces(P_2)$

Intuitively, composition requires components to execute at the same time the actions that appear as input of one component and output of the other. Actions that appear in a unique component are executed asynchronously from the other components. Composition has the following property:

Property 1. *If* $Q_1 \models P_1$ *and* $Q_2 \models P_2$ *then* $Q_1||Q_2 \models P_1||P_2$.

Projection of Trace Properties.

Definition 3. *Given a trace property* P *we define its projection* $proj(P, A)$ *onto a set of actions* A *such that* $sig(proj(P, A)) = (in(P) \cap A, out(P) \cap A)$ *and* $Traces(proj(P, A)) = \{t \mid \exists t' \in Traces(P). t = proj(t', A)\}$

Applying the *proj* operator to a trace property P amounts to removing all the actions not in A from P's interface.

4. Linearizability

Our notion of speculative linearizability is itself based on a new definition of linearizability. This definition is interesting in its own right because it enables a more local form of reasoning than the original definition. Moreover, it allows for repeated events, which are the norm in practice. The technical report [9] contains a proof that our new definition of linearizability is equivalent to the classical definition.

4.1 Abstract Data Types

Abstract Data Types (ADTs) formalize our intuitive understanding of how sequential objects may behave.

In order to simplify our discussion in the rest of the paper, we deviate from the usual, state-machine based, definition of abstract data types. However we do so without loss of generality. Traditionally, the allowed behaviors are described using an ad-hoc state machine that given a state and an input transitions to a new state and produces an output. Instead, we determine outputs from the input history seen so far using an *output function*. Computation of the output function amounts to replaying the execution of the state-machine description.

Definition 4. *An ADT* T *is a tuple* $T = (I_T, O_T, f_T)$ *where* I_T *is a non-empty set, whose members we call* inputs*,* O_T *is a disjoint non-empty set whose members we call* outputs*, and* $f_T : I_T^* \rightarrow O_T$ *is an output function.*

In the rest of the paper, we consider objects of some fixed ADT $T = (I_T, O_T, f_T)$.

Example 1. *We consider a consensus object with operations of the form* $(p(v), d(v'))$ *with* v *and* v' *belonging to some set* V_{Cons}. $p(v)$ *is a shorthand for "propose(v)" and* $d(v)$ *for "decide(v)". The consensus object must guarantee that a unique value* v *may be decided and that at any point a decided value has been pro-*

posed before. Hence, because we are considering sequential executions, the first proposed value must be decided in all subsequent operations. Formally, we define the ADT Cons such that $I_{Cons} = \{p(v) \mid v \in V_{Cons}\}$, $O_{Cons} = \{d(v) \mid v \in V_{Cons}\}$, $f_{Cons}([p(v_1), p(v_2), \ldots, p(v_n)]) = d(v_1)$

4.2 A Trace Model of Concurrent Objects

We consider a set C of asynchronous sequential processes called clients. Clients use a concurrent object of ADT T by calling the object's invocation function with one of the ADT's inputs, and the invocation function returns one of the ADT's outputs. We denote a sequence of interaction between clients and object as follows. Suppose that the following sequence of interactions between clients and object is observed: Client c_1 invokes input in_1; Client c_2 invokes input in_2; Client c_2 returns with output out_2; Finally client c_1 returns with output out_1. This sequence of interactions is denoted by the sequence of actions

$$[inv(c_1, in_1), inv(c_2, in_2), res(c_2, in_2, out_2), res(c_1, in_1, out_1)]$$

Signature of a Concurrent Object Formally, a sequence of interactions between a concurrent object and its clients is a trace of actions in the signature sig_T, where the inputs $in(sig_T)$, called invocation actions, are of the form $inv(c, n, in)$ and the outputs $out(sig_T)$, called response actions, are of the form $res(c, n, in, out)$ for some client c, natural number n, $in \in I_T$, and $out \in O_T$.

The reader will notice that the second parameter of the actions is not present in our preceding example. It will be used in Subsection 5 to define the signature of a speculation phase.

In Section 4 we define the trace property Lin_T such that a concurrent object described by a trace property O is linearizable w.r.t. T if and only if $O \subseteq Lin_T$.

Underscore Notation. Throughout the paper we will use the underscore symbol "_" in mathematical expressions. Each occurrence of this symbol is to be replaced by a fixed fresh variable. For example, we might say an action a is an invocation action if $a = inv(\_, \_, \_)$, or equivalently, that if $a = inv(c, n, in)$ for some c, n, and in, then a is an invocation action.

4.3 A New Definition of Classical Linearizability

In this section we define a trace property called linearizability, and noted Lin_T. Informally, a trace in sig_T is said to be linearizable iff its actions may be reordered to form a *sequential trace* that:

1. Respects the semantics of the ADT T.
2. Preserves the order of non-overlapping operations.

Property 1 implies that clients accessing a linearizable object cannot tell whether the object produces sequential or concurrent traces. This is a crucial property because it hides concurrency from the view of application developers. Property 2 makes linearizability a *local* property. In other words, a system composed of linearizable objects is itself linearizable. We refer the reader to [12] for an in-depth discussion about linearizability.

Instead of using the definition above we propose a new definition of linearizability that will simplify our task when we define speculation phases. It directly establishes a relationship between the linearizable traces and the ADT, without the intermediary of sequential traces.

Note that some definitions of linearizability [12] assume more or less explicitly that all inputs submitted are unique. This seems unwarranted as many algorithms qualified as implementing a linearizable object are deemed correct without requiring this assumption. Our definition does not rely on this assumption and it coincides with the other definitions on traces satisfying the assumption.

Consider a trace t in sig_T.

Definition 5. *Trace t is linearizable iff it is* well-formed *and it admits a* linearization function $g : [1..|t|] \to I_T^*$.

We define linearization functions and well-formed traces below.

To ensure that our definition corresponds to the usual one, in the technical report [9] we also formalize the usual definition of linearizability [12], by defining when a trace is linearizable*, even in case of repeated events. We then prove the equivalence between the two definitions.

Theorem 1. *A trace in sig_T is linearizable if and only if it is linearizable*.*

4.4 Linearization Functions

In the rest of the paper we call sequences of inputs histories. A linearization function maps the indices of t to histories which must satisfy two properties:

First, applying the ADT's output function to the history associated with a response index must yield the output contained in the response. Hence a linearization function may be seen as giving an explanation of the outputs observed, in term of the ADT's semantics.

Second, the history associated to a response index must only contain inputs invoked before index i and for all pairs of histories corresponding to response indices, one is a strict prefix of the other. This second condition gives us a total order on the response indices of t (the prefix order on histories) which corresponds to the reordering needed in the original definition.

A history corresponding to some response index i in a trace t according to a linearization function is said to be a linearization of the (concurrent) trace $t_{|i}$.

We now state this definition formally:

Definition 6. ***Linearization Function.*** *A function g is a linearization function for t iff*

- *Function g* explains *trace t.*
- *Function g and trace t satisfy the* validity*(t,g) and* commit-order*(t,g) predicates.*

Definition 7. ***Explains.*** *We say that function g explains trace t iff for all index $i \in [1..|t|]$, if $t(i) = res(\_, \_, \_, out)$ then $out = f_T(g(i))$.*

Consider a function $g : [1..n] \to I_T^*$ that explains trace t.

Definition 8. ***Commit Index and Commit Histories.*** *If $i \in [1..n]$ is such that $t(i) = res(\_, \_, \_, \_)$ then we say that i is a* commit index *and that $g(i)$ is a (t, g)-commit history.*

Definition 9. ***Sequence of Previous Inputs.*** *We define the sequence of previous inputs at i in t, $inputs(t, i)$, as the sequence of all inputs submitted before index i in trace t. Formally, for all $i \in [1..|t|]$, $|inputs(t, i)| = |proj(t_{|i}, in(sig_T))|$, and for all $j \in [1..|inputs(t, i)|]$, $proj(t_{|i}, in(sig_T))(j) = inv(\_, \_, inputs(t, i)(j))$.*

Definition 10. ***Valid Commit Index*** *For all index $i \in [1..|t|]$ such that $t(i) = res(\_, \_, in, \_)$, we say that i is (t, g)-*valid *iff $elems(g(i)) \subseteq elems(inputs(t, i))$ and the input in is the last element of the history $g(i)$.*

In other words, if $t(i) = res(\_, \_, in, \_)$ then $g(i)$ is a permutation of the sequence of previous inputs at i and ends with the input in.

We now define validity and commit-order:

Definition 11. ***Validity****(t,g). All commit indices in t are (t, g)-valid.*

Definition 12. ***Commit Order****(t,g): For all pairs of distinct commit indices (i, j), either $g(i)$ is a strict prefix of $g(j)$ or $g(j)$ is a strict prefix of $g(i)$.*

Example 2. *Consider the following trace t:*

$[inv(c, in_1), inv(c', in_2), res(c', in_2, f_T([in_2])), res(c, in_1, f_T([in_2, in_1]))]$.

Consider a function g such that $g(3) = [in_2]$ *and* $g(4) = [in_2, in_1]$*. The function g explains t. Moreover t and g satisfy validity and commit-order. Hence g is a linearization function for t.*

4.5 Well-Formed Traces

The well-formedness of traces is a basic requirement. For example, it states that a response should be given to a client only if it previously issued an invocation.

Given a client c, let $Act_T(c)$ be the union of the sets $\{inv(c, \_, in) | in \in I_T\}$ and $\{res(c, \_, in, out) | in \in I_T \wedge out \in O_T\}$.

Definition 13. ***Client sub-trace.*** *The client sub-trace* $sub(t, c)$ *of trace t in* sig_T *is the trace* $sub(t, c) = proj(t, Act_T(c))$*.*

Definition 14. ***Well-formed client sub-trace.*** *A client sub-trace t is well-formed iff* $t(1)$ *is such that* $t(1) = inv(\_, \_, \_)$ *and for all* $i \in [1..|t|-1]$*, if* $t(i) = inv(\_, \_, in)$*, then* $t(i+1) = res(\_, \_, in, \_)$*.*

Definition 15. ***Well-formed traces.*** *We say that trace t in* sig_T *is* well-formed *if and only if all its client sub-traces are well-formed.*

Note that invocations with no corresponding response may appear in a well-formed trace. We say that those invocations are pending.

4.6 The Lin_T Trace Property

We define the trace property Lin_T such that the signature $sig(Lin_T)$ is sig_T and the set $Traces(Lin_T)$ is the set of all traces satisfying linearizability.

Finally, consider a system S whose behavior is denoted by a trace property P. We say that the system S implements the ADT T if and only if $proj(S, sig_T) \models Lin_T$.

5. Speculative Linearizability

We next present our main results: a precise definition of speculative linearizability, and the proof that a composition of implementations that are speculatively linearizable implementations is itself speculatively linearizable. This section presents a trace-based formalization. We summarize an alternative, automata-based, formalization in Section 6.

5.1 Speculation Phases

We now consider concurrent objects implemented by several speculation phases. Clients use a speculation phase by calling the object's invocation functions and the invocation functions return outputs to the clients. However, a speculation phase also accepts initialization calls from clients and it may switch to another speculation phase.

Moreover, each speculation phases is identified by a natural number and a speculation phase identified by n may only switch to a speculation phase identified by $n + 1$. Clients start by accessing speculation phase 1 and continue to do so as long as they return in the phase 1. However, speculation phase 1 may switch to phase 2. In this case it passes an initialization value and its pending input to phase 2. A client which returns in phase 2 stops accessing speculation phase 1 and instead uses phase 2 for its next invocations.

Let S_1 and S_2 be two consecutive speculation phases. Suppose that client c_1 invokes input in_1 on S_1; client c_2 invokes input in_2 on S_1; client c_2 switches from S_1 to S_2 providing initialization value v; client c_1 returns response out_1 from S_1; finally client c_2 returns response out_2 from S_2. This sequence of interactions is denoted by this sequence of events, from now on called actions:

$$[inv(c_1, 1, in_1), inv(c_2, 1, in_2), swi(c_2, 2, in_2, v),$$
$$res(c_1, 1, in_1, out_1), res(c_2, 2, in_2, out_2)]$$

Instead of formalizing the signature of a single speculation phase identified by a natural number we generalize to the signature of a composition of speculation phases. We now assume that a speculation phase may be composed of several other speculation phases.

Let $\mathbb{N}^2_< = \{(m, n) \mid (m, n) \in \mathbb{N}^2 \wedge m < n\}$. We denote a speculation phase by an element of $\mathbb{N}^2_<$. A speculation phase $(m, n) \in \mathbb{N}^2_<$ describes the possible behaviors of the composition of speculation phases from m to n.

We now explain why we make this generalization.

We would like to define a property $P(n)$ of speculation phase n, for $n \in \mathbb{N}$, such that for all $m \in \mathbb{N}$ the composition $P(1)||P(2)|| \dots ||P(m)$ implements Lin_T. To prove that

$$P(1)||P(2)|| \dots ||P(m) \models Lin_T$$

we generalize $P(n)$ to $P(m, n)$ in order to use induction.

We would like $P(m, n)$ to be such that:

$$\text{For all natural number } n > 1,\ P(1, n) \models Lin_T \tag{1}$$

$$\text{For all } (m, n) \in \mathbb{N}^2_< \text{ and } (n, o) \in \mathbb{N}^2_<,$$
$$P(m, n)||P(n, o) \models P(m, o) \tag{2}$$

Using induction and equation 2 we have that for all $n > 1$,

$$P(1, 2)||P(2, 3)|| \dots ||P(n-1, n) \models P(1, n).$$

Thus, with equation 1, we have that for all $n > 1$,

$$P(1, 2)||P(2, 3)|| \dots ||P(n-1, n) \models Lin_T \tag{3}$$

Now suppose that we have a family of trace properties $Q_1, Q_2, \dots, Q_n$, representing implementations of speculation phases, such that $Q_i \models P(i, i+1)$ for all $i \in [1..n]$. Let $Comp = Q_1||Q_2|| \dots ||Q_n$. By property 1 and equation 3, we have that $Comp \models Lin_T$.

Signature of a Speculation Phase We now define a signature $sig_T(m, n, Init)$ which models the interface of a speculation phase (m, n) that uses initialization values from the set *Init*. For all $m \in \mathbb{N}$ and any arbitrary non-empty set *Init* of initialization values, we define the set of *switch actions* as formed of all actions of the form $swi(c, m, in, v)$ where c is a client, $m \in \mathbb{N}$, $in \in I_T$, and $v \in Init$.

Definition 16. ***Signature of a speculation phase.*** *For all* $(m, n) \in \mathbb{N}^2_<$ *and any non-empty set Init of initialization values we define the signature* $sig_T(m, n, Init)$ *as the union of the sets of all invocation actions* $inv(\_, o, \_)$*, of all response actions* $res(\_, o, \_, \_)$*, and of all switch actions* $swi(\_, o, \_, \_)$*, where* $o \in [m..n]$*.*

5.2 Specification of a Speculation Phase

In this section we propose a specification for speculation phases. We consider a set of initialization values *Init*, and a relation $r_{init} \in Init \times I_T^*$ such that r_{init}^{-1} is a total onto function. The relation r_{init} associates a set of input sequences to any value in *Init*. Remember that the init values convey information about the execution of one phase to the next phase. We think of this information in terms of input histories which represent possible linearizations of the concurrent trace of a phase. We say the set of input sequences associated by r_{init} with a value is the value's possible interpretations.

We will now define a trace property $SLin_T(m, n)$, such that

- For all natural number m, $proj(SLin_T(1, m), acts(sig_T)) \models Lin_T$.
- For all $(m, n) \in \mathbb{N}^2_<$ and $(n, o) \in \mathbb{N}^2_<$, $SLin_T(m, n)||SLin_T(n, o) \models SLin_T(m, o)$.

We define the property $SLin_T(m, n)$ in the same vein as Lin_T. Consider a trace t in $sig_T(m, n, Init)$.

Definition 17. ***Interpretation of Init Actions.*** *We say that a function $f : [1..|t|] \to I^*$ is an interpretation of the init actions of t iff for all index $i \in [1..|t|]$, if $t(i) = swi(\_, m, \_, v)$ then $(v, f(i)) \in r_{init}$.*

Definition 18. ***Interpretation of Abort Actions.*** *We say that a function $f : [1..|t|] \to I^*$ is an interpretation of the abort actions of t iff for all index $i \in [1..|t|]$, if $t(i) = swi(\_, n, \_, v)$ then $(v, f(i)) \in r_{init}$.*

Definition 19. ***Speculatively Linearizable.*** *We say that trace t is (m, n)-speculatively linearizable iff trace t is (m, n)-well-formed and for all interpretation f_{init} of the init actions of t, there exists two functions $g : [1..|t|] \to I_T^*$ and $f_{abort} : [1..|t|] \to I_T^*$ such that:*

- *f_{abort} is an interpretation of the abort actions of t.*
- *g is an $(f_{init}, f_{abort}, m, n)$-speculative linearization function for t.*

5.3 Speculative Linearization Functions

Intuition Behind the Definitions. Consider a speculation phase S_1 whose clients switch to speculation phase S_2. We require clients to switch with a value whose set of possible interpretations includes a possible linearization of the concurrent trace at the point where the client switches. Note that every history (1) of which all commit histories are prefix and (2) that contains only values that have been invoked is a possible linearization. This is the intuition behind the definition of Abort-Order property (Definition 32).

To ensure that phase S_2 continues executing consistently with what phase S_1 did, we require all its commit histories to have as a prefix one of the possible linearizations of S_1's trace. This is the intuition behind the Init Order property (Definition 31). However S_2 receives switch values that represent sets of histories, and cannot determine which history in this set is a possible linearization. Hence the quantifier alternation in Definition 19: for all possible interpretations of the init values, phase S_2 needs to ensure the existence of a proper linearization function.

Definition 20. ***Speculative linearization function.*** *A function $g : [1..|t|] \to I_T^*$ is a $(f_{init}, f_{abort}, m, n)$-speculative linearization function for t iff:*

- *The function g explains trace t.*
- *The predicates Validity$(t, g, f_{init}, f_{abort})$, Commit-Order$(t, g)$, Init-Order$(t, g, f_{init}, f_{abort})$, and Abort-Order$(t, g, f_{abort})$ hold.*

Definition 21. ***Explains.*** *We say that the function g explains trace t iff for all index $i \in [1..|t|]$, if $t(i) = res(\_, \_, \_, out)$ then $out = f_T(g(i))$*

Consider a function $g : [1..|t|] \to I_T^*$ that explains trace t and consider two functions $f_{init} \subseteq r_{init}$ and $f_{abort} \subseteq r_{init}$. In order to define validity, commit-order, init-order, and abort-order we need the definitions 22 to 28 below.

Definition 22. ***Commit Indices and Commit Histories.*** *If $i \in [1..n]$ is such that $t(i) = res(\_, \_, \_, \_)$ then we say that i is a (t, m, n)-*commit index *and that $g(i)$ is a (t, g)-*commit history.

Definition 23. ***Init Indices and Init Histories.*** *If $i \in [1..n]$ is such that $t(i) = swi(\_, m, \_, v)$ we say that i is a (t, m, n)-*init index *and that $f_{init}(i)$ is a (t, m, n, f_{init})-*init history.

Definition 24. ***Abort Indices and Abort Histories.*** *If $i \in [1..n]$ is such that $t(i) = swi(\_, n, \_, v)$ we say that i is a (t, m, n)-*abort index *and that $f_{abort}(i)$ is a (t, m, n, f_{abort})-*abort history.

The multiset $ivi(m, t, f_{init}, i)$ contains all the inputs known to to have been invoked in phases (k, l), where $l \le m$:

Definition 25. ***Initially Valid Inputs.***

$$ivi(m, t, f_{init}, i) = \bigcup \{elems(f_{init}(v)) \cup \{in\} \mid \exists j < i.t(i) = swi(\_, m, in, v)\}.$$

Note that, in the previous definition, $\bigcup$ is the multiset union (defined in Section 3) and corresponds to pointwise maximum of representation functions.

Now recall that $inputs(t, i)$ is the sequence of inputs that were invoked before i in t.

Definition 26. ***Valid Inputs.*** *We define*

$$vi(m, t, f_{init}, i) = ivi(m, t, f_{init}, i) \uplus elems(inputs(t, i))$$

Definition 27. ***Valid Commit Index.*** *For all index i such that $t(i) = res(\_, \_, in, \_)$, we say that i is (t, g, f_{init})-*valid *if and only if $elems(g(i)) \subseteq vi(m, t, f_{init}, i)$ and the input in is the last element of history $g(i)$.*

Definition 28. ***Valid Abort Index.*** *For all index i such that $t(i) = swi(\_, n, in, v)$, we say that i is (t, f_{init}, f_{abort})-*valid *if and only if $elems(f_{abort}(v)) \cup \{in\} \subseteq vi(m, t, f_{init}, i)$.*

We now define validity, commit-order, init-order, and abort-order:

Definition 29. ***Validity****$(t, g, f_{init}, f_{abort})$: All (t, m, n)-commit indices are (t, g, f_{init})-valid and all (t, m, n)-abort indices are (t, f_{init}, f_{abort})-valid.*

Definition 30. ***Commit Order****(t,g): For all pair of distinct commit indices i, j, either $g(j)$ is a strict prefix of $g(i)$ or $g(i)$ is a strict prefix of $g(j)$.*

Definition 31. ***Init Order****$(t, g, f_{init}, f_{abort})$: The longest common prefix of all (t, m, n, f_{init})-init histories is a strict prefix of all (t, g)-commit histories and of all (t, m, n, f_{abort})-abort histories.*

We assume that the longest common prefix of an empty set of histories is the empty history.

Definition 32. ***Abort Order****(t, g, f_{abort}): All (t, g)-commit histories are prefix of all (t, m, n, f_{abort})-abort histories.*

Note that if $m = 1$, then t has no init histories. Hence Init Order does not constrain commit and abort histories.

5.4 Well-Formed Traces

The well-formedness condition defines basic requirements on traces of a concurrent object.

Given a client c let $Act_T(c, m, n)$ be the set of actions such that

$$\begin{aligned}Act_T(c, m, n) = &\{inv(c, o, in) | o \in [m..n] \wedge in \in I_T\}\\ &\cup \{res(c, o, in, r) | o \in [m..n] \wedge in \in I_T \wedge r \in O_T\}\\ &\cup \{swi(c, o, in, v) | o \in \{m, n\} \wedge in \in I_T \wedge r \in O_T \wedge v \in Init\}\end{aligned}$$

Note that the switch actions whose second parameter is not m or n are projected away.

Definition 33. ***Client sub-trace.*** *The (m, n)-client-sub-trace of the trace t is the trace $sub(t, m, n, c) = proj(t, Act_T(c, m, n))$.*

Definition 34. ***Well-formed client sub-trace.*** *A (m, n)-client subtrace t_c is well-formed iff it is empty or:*

- *For all $i \in [1..|t_c| - 1]$, if $t(i) = inv(\_, \_, in)$ or $t(i) = swi(\_, m, in, \_)$, then $t(i + 1) = res(\_, \_, in, \_)$ or $t(i + 1) = swi(\_, n, in, \_)$.*
- *If t_c contains an abort action then it is the last element of t_c.*
- *If $m \ne 1$ then $t_c(1)$ is an init action and there are no other init actions in t_c.*
- *If $m = 1$ then $t_c(1)$ is an invocation and there are no init actions in t_c.*

Definition 35. ***Well-formed traces.*** *We say that the trace t is (m,n)-well-formed if and only if all its (m,n)-client sub-traces are well-formed.*

5.5 The $SLin_T$ Trace Property

We define the trace property $SLin_T(m,n)$ as follows:

Definition 36. *The $SLin_T(m,n)$ is such that the signature $sig(SLin_T(m,n))$ is $sig_T(m,n,Init)$ and the set $Traces(SLin_T(m,n))$ is the set of all traces satisfying (m,n)-speculative linearizability.*

Finally, consider a system S whose behavior is denoted by a trace property P. We say that the system S is a (m,n)-speculation phase if and only if $proj(P, sig(m,n)) \models ALin(m,n)$.

Theorem 2. *For all natural number m, any initialization set Init, and any initialization relation r_{init},*

$$proj(SLin_T(1,m), acts(sig_T)) = Lin_T$$

Proof. Compared to linearizability, speculative linearizability only adds constraints on the actions that are not in the signature sig_T. $\square$

5.6 Intra-Object Composition Theorem

Our main theorem states that the composition of two speculation phases is also a speculation phase.

Theorem 3. *Suppose that $S_1 \models SLin_T(m,n)$ and $S_2 \models SLin_T(n,o)$, then $proj(S_1||S_2, sig_T(m,o,Init)) \models SLin_T(m,o)$.*

The proof of this result in the technical report [9]. It consists of two pages in the current format. Among the key steps is the construction of a speculative linearization function g that explains the traces of the composition of two phases. The function is constructed by merging two speculative linearization functions. To show that it is indeed a speculative linearization function we use a form of transitive reasoning to relate commit histories between different phases through abort histories. Among the properties of speculative linearizability, Validity is the most difficult to show, in part because it is sensitive to the fact that we allow duplicate events in our traces (something that even most existing developments of classic linearizability ignore).

6. Automata Specification of Speculative Linearizability and Isabelle/HOL Proof

We have also formalized in Isabelle/HOL an executable version of the specification of speculative linearizability. Our formalization is based on the theory of I/O-automata [21], which is formalized in Isabelle's IOA framework [22]. Our specification automaton corresponds to speculative linearizability instantiated for State Machine Replication [16] protocols (abbreviated SMR protocols). We wrote a mechanically-checked proof of the intra-object composition theorem for the automata specification. Thanks to this proof, it is now possible to obtain mechanically-checked proofs of speculative SMR protocols from the mechanically-checked proofs of its speculation phases taken in isolation. Moreover, our proof of the intra-object composition theorem shows that refinement proofs in our framework are practical.

The speculative approach to SMR protocols has been shown to yield some of the most efficient SMR protocols in practice [8]. Therefore, our formalization and the proof of the intra-object composition theorem enable for the first time scalable mechanically-checked proofs of practical SMR protocols. A proof document and the corresponding Isabelle theories are available at the Archive of Formal Proofs [10].

Informal Description of the Automata Specification. The automata specification is a formalization of speculative linearizability for an ADT that we call the universal ADT. The output function of the universal ADT is the identity function. In other words, this ADT responds to an invocation with its full trace, in the form of a history. The universal ADT can be used as an abstraction for generic SMR protocols [8] because, given a linearizable implementation, it suffices to apply the output function of another ADT A to the responses in order obtain an implementation of A. We formalize the case where histories of inputs are used as initialization values and where the relation r_{init} maps a history h to the singleton set $\{h\}$. The specification automaton can be seen as an implementation of speculative linearizability in which all clients reside on a unique, centralized, process.

The automaton receives invocations and switch calls as inputs and produces responses and switch call outputs.

The specification automaton maintains a state with the following components:

- A history *hist*, representing the longest linearization made visible to a client (i.e. the longest commit history);
- for each client c, a phase $phase(c) \in \{$ Sleep, Pending, Ready, or Aborted $\}$;
- for each client c, $pending(c)$, the last input submitted by c;
- a set *InitHists* containing the init histories received;
- two booleans: *aborted* and *initialized*.

We say that an input is pending if it is the last submitted input of a client c whose phase is "Pending" and if it is not present in *hist*.

The automaton starts in a state where $hist$ is the empty history, *InitHists* is the empty set, *aborted* and *initialized* are set to *false*, and for all clients c, $phase(c) =$ "Sleep".

We now describe how the automaton reacts to the inputs it receives, namely invocations and switch calls from the previous phase. When receiving a switch call from the previous phase by client c, if $phase(c)$ equals "Sleep" then the automaton adds the provided init history to the set *InitHists* and sets $pending(c)$ to the provided input. When receiving a new invocation by client c, if $phase(c)$ equals "Ready", the automaton sets $pending(c)$ to the input received. In both cases $phase(c)$ is then set to "Pending". This denotes the fact that client c is waiting for a response to input $pending(c)$.

Moreover, the automaton nondeterministically performs one of the following actions:

A1 If *initialized* $=$ *false* and if there is at least one client c such that $phase(c)$ is not "Sleep", then the automaton computes the longest common prefix of the histories in *InitHists*, assigns it to *hist*, and sets *initialized* to *true*.

A2 It selects a pending input, say from client c, appends it to *hist*, emits an output for client c containing the new value of *hist* as response, and sets $phase(c)$ to "Ready".

A3 It sets *aborted* to *true*.

A4 If *aborted* $=$ *true* then a client c such that $phase(c)$ is not "Aborted" is selected is $phase(c)$ is set to "Aborted". Moreover, the automaton emits a switch action containing an abort value h' such that *hist* is a prefix of h' and the inputs of h' which are not in *hist* are pending.

The precise definition of our specification automaton is available in the ALM entry [10] of the Archive of Formal Proofs.

Relation to the Trace Based Specification. The trace-based specification requires associating histories to responses (commit histories) and switch actions (init and abort histories). Because our ADT's output function is the identity and because $r_{init}(h) = \{h\}$, we have no choice but to associate to a response or switch action the very history it contains (remember that outputs and switch values are histories in this section). Hence commit histories are obtained by truncating *hist* at a pending request. Since *hist* grows by appending to it, we have that Commit Order is satisfied. Moreover, abort histories are obtained by appending some pending requests to *hist*, and at this point *hist* does not grow anymore. Hence Abort Order holds. Since *hist* is initialized with the longest common prefix of the init histories seen, Init Order also holds. Finally, *hist* grows by appending pending requests to it, Validity is also satisfied.

To gain more insight into speculative linearizability, one may also observe that any extension of history *hist* with some pending requests is a linearization of the current trace. Thanks to this observation, step A2 may be interpreted as selecting a possible linearization and producing an output that realizes it. Step A4 can be seen as passing one of the possible linearizations to the next speculation phase. Finally, in step A1, the automaton computes the weakest (given the init histories it received) under-approximation of the set of possible linearizations of the previous phase and selects one of them to initialize its execution.

Proof of the Intra-Object Composition Theorem. To prove the intra-object composition theorem in this formulation, we construct a refinement mapping [20] between the composition of two speculation phases and a single speculation phase. We prove the refinement mapping correct with the help of 15 state invariants about the composed automaton. Using the meta-theorem about refinement mappings (included in the IOA theory packaged with Isabelle/HOL), we conclude that the set of traces of the composition of two speculation phases is included in the set of traces of a single speculation phase. The proof is written in the structured proof language Isar [26] and consists of roughly of 500 proof steps. With the specification, it forms a total of 1600 lines of Isabelle/HOL code.

Our automata specification can be used as the basis for mechanically-checked refinement proofs of distributed protocols. Our proof of the composition is a good example of such a refinement proof and shows the practicality of the approach.

7. Concluding Remarks

We have presented speculative linearizability, an extension of the theory of linearizability. Our extension allows us to implement linearizable objects by composing independently devised speculation phases that are optimized for different situations. This form of intra-object composition complements the classical concept of inter-object composition, inherent to linearizability in its traditional form. We propose a formalized framework that enables, for the first time, scalable mechanically-checked development of linearizable objects of arbitrary abstract data types.

Our work can be viewed as generalization of [1] and [2]. The former work focused on contention-free executions and did not address composition: a protocol that aborts simply stops executing with no possibility of switching. In the latter work, a classification of the trace properties that are switchable is proposed. A global synchronization is required for switching. In contrast, our notion of speculative linearizability does not require the switching protocols to solve agreement and is, in that sense, much more general. Our work can also be viewed as a formalization of a generalized form of *Abortable State Machine Replication (Abstract)*, an abstraction presented in an intuitive form in [8] to allow the construction of speculative Byzantine Fault Tolerant SMR protocols by composing independently designed speculation phases. Compared to [8] , our work concerns arbitrary abstract data types, including one-shot ones, and generalizes the idea to linearizability. Being strictly more general, our framework can be used to reason about the fault-tolerant algorithms developed following those approaches [1, 2, 8].

References

[1] M. K. Aguilera, S. Frolund, V. Hadzilacos, S. L. Horn, and S. Toueg. Abortable and query-abortable objects and their efficient implementation. In *PODC*, 2007.

[2] M. Bickford, C. Kreitz, R. v. Renesse, and X. Liu. Proving hybrid protocols correct. In *TPHOLs '01*, 2001.

[3] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. NSDI*. USENIX Assoc., 2011.

[4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI*. USENIX Assoc., 2006.

[5] M. Castro and B. Liskov. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, MIT, 1999.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[7] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[8] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. The Next 700 BFT Protocols. In *EUROSYS*, 2010.

[9] R. Guerraoui, V. Kuncak, and G. Losa. Speculative Linearizability. Technical Report 170038, EPFL, 2011.

[10] R. Guerraoui, V. Kuncak, and G. Losa. Abortable linearizable modules. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Abortable_Linearizable_Modules.shtml`, March 2012. Formal proof development.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

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

[13] M. Jaskelioff and S. Merz. Proving the correctness of Disk Paxos. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/DiskPaxos.shtml`, June 2005. Formal proof development.

[14] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC*, 2003.

[15] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *SOSP*, 2007.

[16] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[17] L. Lamport. On interprocess communication. part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[18] L. Lamport. On interprocess communication. part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[19] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[20] N. Lynch and F. Vaandrager. Forward and backward simulations I: untimed systems. *Inf. Comput.*, 121:214–233, September 1995.

[21] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[22] O. Müller. I/O automata and beyond: Temporal logic and abstraction in Isabelle. In *TPHOLs*, pages 331–348, 1998.

[23] F. Pedone. Boosting system performance with optimistic distributed protocols. *Computer*, 34(12):80–86, 2001.

[24] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *NSDI*, 2008.

[25] M. M. V. Luchangco and N. Shavit. On the uncontended complexity of consensus. In *ICDCS*, pages 45–59, 2003.

[26] M. Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *TPHOLs*, pages 167–184, 1999.