Consistency Models with Global Operation Sequencing and their Composition

Alexey Gotsman[1] and Sebastian Burckhardt[2]

1 **IMDEA Software Institute, Spain**
2 **Microsoft Research, USA**

Abstract

Modern distributed systems often achieve availability and scalability by providing consistency guarantees about the data they manage weaker than linearizability. We consider a class of such consistency models that, despite this weakening, guarantee that clients eventually agree on a global sequence of operations, while seeing a subsequence of this final sequence at any given point of time. Examples of such models include the classical Total Store Order (TSO) and recently proposed dual TSO, Global Sequence Protocol (GSP) and Ordered Sequential Consistency.

We define a unified model, called *Global Sequence Consistency (GSC)*, that has the above models as its special cases, and investigate its key properties. First, we propose a condition under which multiple objects each satisfying GSC can be composed so that the whole set of objects satisfies GSC. Second, we prove an interesting relationship between special cases of GSC—GSP, TSO and dual TSO: we show that clients that do not communicate out-of-band cannot tell the difference between these models. To obtain these results, we propose a novel axiomatic specification of GSC and prove its equivalence to the operational definition of the model.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Consistency conditions, Weak memory models, Compositionality

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.23

1 Introduction

Modern distributed systems often achieve availability and scalability by providing consistency guarantees about the data they manage weaker than the gold standard of linearizability [16]. In this paper we consider a class of such consistency models that, despite this weakening, guarantee *global operation sequencing*: clients eventually agree on a global sequence of operations, while seeing a subsequence of this final sequence at any given point of time. An implementation of a service providing such a model may consist of a single *server* and multiple *clients*, each maintaining a replica of the data managed by the service. Clients accept operations from end-users, evaluate them on their local (possibly stale) data replica and forward the operations to the server. The server arranges all received operations into a totally ordered log and forwards them to clients in the order determined by the log. The server log thus establishes the desired global sequence of operations.

Such consistency models arise in different domains. For instance, clients may correspond to mobile devices, cloud servers or processor cores; the role of the server may be played by an elected leader, a replicated state machine [26], a reliable total-order broadcast [11] or the memory subsystem in a multiprocessor architecture [28]. Various models differ in whether the propagation of operations from clients to the server and vice versa is asynchronous or synchronous. Thus, in the *Global Sequence Protocol (GSP)* model [10], the propagation is asynchronous in both directions, which allows clients to execute operations even if they get

partitioned from the server [14]. This model is implemented in Microsoft's TouchDevelop system for mobile app programming, to support offline access [1], and in the Orleans actor framework [6], to support geo-replication [5]. In the *Total Store Order (TSO)* model [23, 24], implemented by SPARC and x86 multiprocessors, operation propagation from clients to the server is asynchronous, but the one from the server to clients is synchronous: clients *pull* all new operations from the server before evaluating each operation. Conversely, in the *dual TSO* model [2] operation propagation from the server to clients is asynchronous, but the one from the clients to the server is synchronous: clients *push* operations to the server immediately after they are executed. If we strengthen dual TSO by requiring that all update operations are propagated synchronously in both directions, we obtain *Ordered Sequential Consistency (OSC)* [22], which captures the semantics of coordination services such as ZooKeeper [18]. Finally, we obtain linearizability [16] when operation propagation is synchronous in both directions.

In this paper we study key properties of the consistency models from the above class. To this end, we consider a flexible model, called *Global Sequence Consistency (GSC)*, that has the above models as its special cases and obtain novel results about this model: a condition for safely composing multiple GSC services and a certain interesting relationship between the model's special cases. The GSC model is defined by the above client-server protocol where operation propagation is by default asynchronous, but operations may include two kinds of *fences*. The fences respectively force a client to pull all new operations from the server or push all outstanding local operations to the server (§3). Then we obtain various existing consistency models by systematically associating fences with operations as shown in Figure 1.

Implicit fences	pull	push
GSP [10]	no	no
TSO [23, 24]	yes	no
dual TSO [2]	no	yes
OSC [22]	updates	yes
linearizability [16]	yes	yes

Figure 1 Specialising GSC.

Like sequential consistency [20], GSC is not *composable* (aka *local*) [16]: objects satisfying GSC may fail to provide this consistency guarantee when combined. This is a problem because application programmers often want to distribute objects among multiple services, e.g., to place them in geographical locations where they are most likely to be updated and thereby minimise latency [21]. Non-composability does not allow programmers to easily predict the behavior of such a system. This is a particular issue in the Orleans implementation of geo-replication [5], which guarantees GSP only for each individual object.

To address this problem, we propose a condition under which multiple objects each satisfying GSC can be composed so that the whole set of objects satisfies GSC (§5). Informally, the condition requires using fences according to the following discipline: when switching between different objects, a client has to push the operations done on the old object and pull operations on the new object. Our result ensures that in this case clients interacting with multiple GSC services implementing different objects will behave as though they are interacting with a single GSC service. This result holds even when clients can communicate out-of-band, without using the GSC services. As its special cases, we obtain novel conditions for composing TSO and dual TSO objects, as well as a recently proposed condition for OSC [21, 22].

We also prove an interesting relationship between special cases of GSC—GSP, TSO and dual TSO (§4): we show that clients that do not communicate out-of-band cannot tell the difference between them. In particular, this result implies that a program without out-of-band communication written assuming TSO operates correctly under much weaker, fully asynchronous GSP. This equivalence has been previously conjectured without proof [10]; the

present paper confirms this conjecture. Assuming the absence of out-of-band communication is common for memory models, where clients are processors that do not communicate directly. However, this assumption is often not appropriate for distributed interactive applications, where clients can have external means of communication. In this setting, the above special cases of GSC are observably different.

Proving the above results about compositionality and equivalence is nontrivial due to the complexity of reasoning about the distributed protocol implementing GSC. Our main tool in tackling this complexity is an *axiomatic* specification of GSC, given in the style often used for consistency models in shared-memory [19] and distributed storage systems [8, 9] (§6). The specification represents service executions using several relations, declaratively describing how operations are processed by the GSC protocol; the consistency model is then defined by a set of axioms, constraining these relations. We prove that our axiomatic specification is equivalent to the operational one. A particular subtlety in formulating the axiomatic specification and proving this equivalence is the need for the specification to track the *real-time order* between operations, determining when one operation finishes before another one starts. This makes results established using the axiomatic specification applicable in the case when clients can communicate out-of-band [3, 12].

The axiomatic specification of GSC is instrumental in obtaining our results. A recurring challenge is to prove the existence of an execution that satisfies some conditions, e.g., is a composition of single-object executions in the proof of the compositionality criterion (§8). Constructing the desired execution is difficult to do directly on the operational model. Because of the wide-ranging effect of fences, such an execution cannot be obtained simply by local reordering of independent steps, as with simpler operational models. But via the axiomatic specification of GSC, we can solve this problem indirectly by formulating constraints on precedence of events in the execution as relations and then using algebraic techniques to prove that their union is acyclic, which guarantees that there exists an execution satisfying them. We hope that, in the future, the GSC model, with its two equivalent definitions, and our proof techniques will provide a solid foundation for obtaining further results about consistency models with global operation sequencing.

2 Preliminaries

We consider a distributed service managing a collection of *objects* $\mathsf{Obj} = \{x, y, \ldots\}$. A finite number of clients interact with the service by performing *operations* on the objects, which are ranged over by op and come from a set Op. Parameters of operations, if any, are part of the operation name. For uniformity, we assume that all objects admit the same set of operations and that each operation returns one value from a set Val; we can use a special member of Val to model operations that return no value. The sequential semantics of operations is defined by a function $\mathsf{eval} : \mathsf{Op}^* \times \mathsf{Op} \to \mathsf{Val}$ that determines the return value of an operation on an object given the sequence of operations previously executed on this object.

The consistency model provided by the service defines the set of all possible interactions between the service and its clients. We now introduce a structure that records such interactions in a single computation, called a *history*. In it we denote client-service interactions using *events*, which are ranged over by e, f, g and come from an infinite countable set Event. Events have unique identifiers from a set Id. An event is of the form $e = (\iota, x, op, a, fen)$, where $\iota \in \mathsf{Id}$ is the event identifier, $x \in \mathsf{Obj}$ is the object on which the event occurs, $op \in \mathsf{Op}$ is the operation done, $a \in \mathsf{Val}$ is its return value, and $fen \subseteq \{\mathsf{push}, \mathsf{pull}\}$ gives the *fences* requested by the client. We use $\mathsf{obj}(e)$, $\mathsf{oper}(e)$, $\mathsf{rval}(e)$, $\mathsf{fences}(e)$ to select event components.

State for each client c:
  $known_c \in (\mathsf{Id} \times \mathsf{Op})^*$
  $unacked_c \in (\mathsf{Id} \times \mathsf{Op})^*$
  $pending_c \in (\mathsf{Id} \times \mathsf{Op})^*$
`exec`(c, op, fen):
  if $(\mathsf{pull} \in fen)$
    while $(known_c \neq server\_log)$ `pull`(c)
  $result :=$
    $\mathsf{eval}(\mathsf{stripIds}(known_c \cdot unacked_c \cdot pending_c), op)$
  $pending_c := pending_c \cdot (\mathsf{uniqueId}(), op)$
  if $(\mathsf{push} \in fen)$
    while $(pending_c \neq [\,])$ `push`(c)
  return $result$

Server state:
  $server\_log \in (\mathsf{Id} \times \mathsf{Op})^*$
`push`(c):
  if $(pending_c = (id, op) \cdot remaining_c)$
    $server\_log := server\_log \cdot (id, op)$
    $unacked_c := unacked_c \cdot (id, op)$
    $pending_c := remaining_c$
`pull`(c):
  if $(server\_log = known_c \cdot (id, op) \cdot \_)$
    $known_c := known_c \cdot (id, op)$
    if $(unacked_c = (id, op) \cdot remaining_c)$
      $unacked_c := remaining_c$

Figure 2 The pseudocode of the protocol defining the GSC model. We denote sequence concatenation by $\cdot$, an empty sequence by $[\,]$ and an irrelevant expression by _.

We use the following kinds of relations. A relation is a *strict partial order* if it is transitive and irreflexive. It is a *total order* if it additionally relates every two distinct elements one way or another. A relation is *prefix-finite* if each element is reachable along directed paths from at most finitely many others. A strict partial order R is an *interval order* if

$$\forall e_1, e_2, f_1, f_2.\, (e_1 \xrightarrow{R} e_2 \wedge f_1 \xrightarrow{R} f_2) \implies (e_1 \xrightarrow{R} f_2 \vee f_1 \xrightarrow{R} e_2).$$

Intuitively, an interval order R is consistent with an interpretation of events as segments of time during which the corresponding operations executed, with R ordering e before f if e finishes before f starts [13]. For example, the real-time order considered in linearizability [16] is an interval order.

A *history* is a triple $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$, where: $E \subseteq \mathsf{Event}$; *session order* $\mathsf{so} \subseteq E \times E$ is a union of prefix-finite total orders over a finite number of disjoint subsets of E (each corresponding to operations by the same client); and *real-time order* $\mathsf{rt} \subseteq E \times E$ is a prefix-finite interval order such that $\mathsf{so} \subseteq \mathsf{rt}$ and $\forall e \in E.\, |\{f \in E \mid \neg(e \xrightarrow{\mathsf{rt}} f)\}| < \infty$.

The set E defines all operations invoked by clients in a single computation and can be infinite. The session order arranges operations by the same client in the order in which they were executed. The real-time order $e \xrightarrow{\mathsf{rt}} f$ tells us that the operation of e finished before the one of f started (the last restriction on rt ensures that every operation finishes). Tracking this relationship is important because it allows the client who executed the operation of e to communicate its return value to the client executing f out-of-band, without using the service; the return value of e can then influence the operation executed by f [3, 12]. We denote components of histories and similar structures as in $E_{\mathcal{H}}$ and $\mathsf{so}_{\mathcal{H}}$. A consistency model is defined by a set of histories.

3 Operational Specification

We define Global Sequence Consistency using the idealised protocol in Figure 2, which is a generalisation of the Global Sequence Protocol (GSP) [10]. It assumes a single *server* and a finite number of *clients*. The server state is represented by a log $server\_log$ of operations received from clients, tagged with unique identifiers from Id. The state of each client c includes three logs: $known_c$ is the prefix of $server\_log$ that c knows about; $pending_c$ is the

log of operations by c that have not yet been pushed to the server; and $unacked_c$ is the log of operations by c that have been pushed to the server, but $known_c$ has not yet advanced enough to incorporate them.

The communication between the server and each client c is modeled by transitions $\texttt{push}(c)$ and $\texttt{pull}(c)$ that can fire nondeterministically at any time when the client is not executing an operation and atomically modify the client and the server state (implementations may refine this using asynchronous communication channels as in [10]). The $\texttt{push}(c)$ function models how the server processes the next operation by client c: it appends the oldest record in $pending_c$ to $server\_log$ and moves it to the end of $unacked_c$. The $\texttt{pull}(c)$ function models how the client c learns about the next entry in the server log: it appends to $known_c$ the next operation in $server\_log$ that is not yet part of $known_c$. If this operation is an echo of an operation previously executed by the same client c, we remove it from the $unacked_c$ log; the protocol ensures that in this case the operation is the first (oldest) one in $unacked_c$.

We model a client c executing an operation op with fences $fen \subseteq \{\mathsf{push}, \mathsf{pull}\}$ by $\texttt{exec}(c, op, fen)$. The body of $\texttt{exec}()$ is executed atomically, and only a single invocation of it can be in progress per client. At the beginning of $\texttt{exec}()$, we handle pull fences by repeatedly calling $\texttt{pull}(c)$ until the local $known_c$ matches $server\_log$. At the end of $\texttt{exec}()$, we handle push fences by repeatedly calling $\texttt{push}(c)$ until all $pending_c$ operations have been processed by the server. At the core of $\texttt{exec}()$, we first compute the result of the operation by conjoining the logs $known_c$, $unacked_c$ and $pending_c$, stripping identifiers using $\mathsf{stripIds}$ and applying the sequential semantics of operations defined by eval (§2). We then append the operation to the $pending_c$ with a unique identifier generated by $\mathsf{uniqueId}$. Since op is evaluated on a log that includes $unacked_c$ and $pending_c$, the client is always guaranteed to observe its own operations, even before they are acknowledged by the server (the "read-your-writes" property [29]). Note that when fen is empty, $\texttt{exec}(c, op, fen)$ returns immediately without communicating, so that in this case the protocol is partition-tolerant [14].

We only consider computations of the protocol that adhere to certain *fairness constraints*: every operation by a client eventually gets pushed to the server, every operation received by the server eventually gets pulled by any client and every invocation of $\texttt{exec}()$ terminates.

The set of histories $(E, \mathsf{so}, \mathsf{rt})$ allowed by GSC is defined by considering all possible computations of the above protocol. The invocations of $\texttt{exec}()$ define the set of events E, the order in which they are invoked on clients defines so, and two events are related by rt if the $\texttt{exec}()$ function of the former finishes before the $\texttt{exec}()$ function the latter starts. We denote the set of histories defined in this way $\mathsf{HistGSC}$.

By systematically associating fences with operations in GSC we get various existing models as its special cases (Figure 1). If operations are executed without any fences, the GSC protocol exactly matches the one used to define GSP [10]. If every operation includes a pull fence, then the GSC protocol is isomorphic to one defining the *Total Store Order (TSO)* consistency model [23, 24]. In this case, operations are always evaluated based on an up-to-date state on the server, but are propagated to the server asynchronously. If every operation includes a push fence, then the GSC protocol is isomorphic to one defining a recently proposed *dual TSO* model [2]. In this case, all operations are pushed to the server immediately, but are evaluated on a client-local possibly stale state. If every operation includes both a pull and a push fence, then the GSC protocol produces exactly those histories that are linearizable [16] (we prove this in [15, §C]). Informally, in this case the total order in which the operations go into $server\_log$ defines a linearization of the execution, which preserves the real-time order between the operations.

As a subcase of dual TSO, we also obtain a recently proposed *Ordered Sequential*

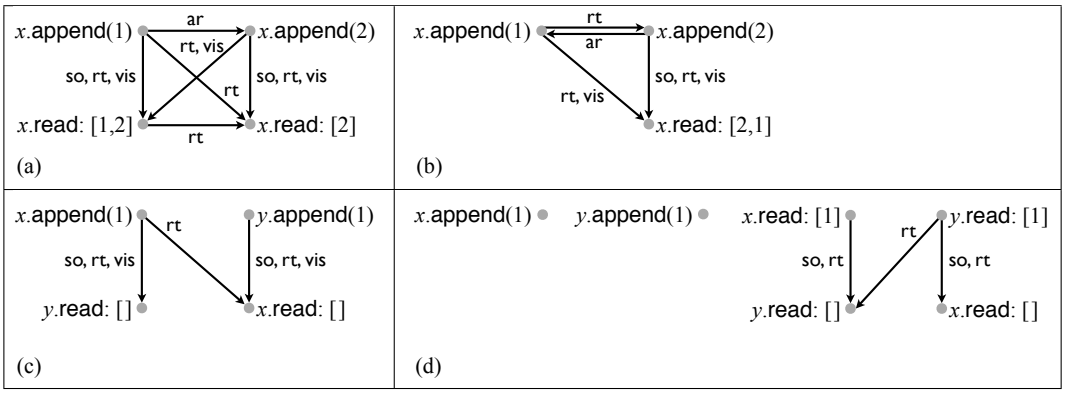

Figure 3 Examples of histories and abstract executions. Events do not include fences unless explicitly noted. Events by the same client are related by the session order so and laid out vertically. Thus, there are two clients in (a-c) and four in (d).

Consistency (OSC) [22], which captures the semantics of coordination services such as ZooKeeper [18]. OSC assumes a partitioning of all operations into read-only and update operations: $\mathsf{Op} = \mathsf{OpReadOnly} \uplus \mathsf{OpUpdate}$. Read-only operations do not change the state of an object: for any operation op and a sequence of operations ξ, we have $\mathsf{eval}(\xi, op) = \mathsf{eval}(\xi|_{\mathsf{OpUpdate}}, op)$, where $\xi|_{\mathsf{OpUpdate}}$ is the projection of ξ onto OpUpdate. In our setting, OSC is defined by requiring that every operation include a push fence (like in dual TSO) and all updates additionally include a pull fence. Thus, update operations are evaluated on an up-to-date state, whereas read-only operations can be evaluated on a stale state. We prove the correspondence to the original OSC definition in [15, §C].

With unrestricted fence placements, GSC is weaker than linearizability, as we illustrate by the example histories in Figures 3(a-c) (for now ignore the extra relations vis and ar). They use sequence objects x and y for which $\mathsf{eval}(\xi, \mathsf{read})$ returns the sequence of values in the append operations in ξ. The histories in Figures 3(a-c) can be produced by the GSC protocol, but are not linearizable: there does not exist a linearization of the events consistent with the real-time order and the sequential semantics of objects. In the following, we briefly describe how the GSC protocol produces these histories; the reader may wish to consult [15, §A], where we describe the corresponding protocol computations in detail.

In history (a) the read by the second client does not see 1, even though it happens after the read by the first client that does see 1. In the GSC protocol this can happen if the second client does not pull $\mathsf{append}(1)$ from the server before executing the read. This history is disallowed if the read by the second client is executed with a pull fence: since the read by the first client returns $[1, 2]$, at the time the read is executed, 1 must be in *known* and, hence, on the server; then the pull fence ensures that the later read by the second client sees 1.

In history (b) the return value of the read is $[2, 1]$ even though $\mathsf{append}(1)$ finishes before $\mathsf{append}(2)$ starts. This can happen if the latter operation is pushed to the server before the former. This outcome is disallowed if $\mathsf{append}(1)$ is executed with a push fence, so that it is pushed to the server before the operation finishes.

In history (c) each read does not see the append by the other client; this is a variant of the *store buffering* anomaly, characteristic of TSO [24]. It can be produced by the GSC protocol if the appends are pushed to the server only after the reads execute. The history is disallowed if the appends include push fences and the reads pull fences.

Finally, history (d) is a variant of the *independent reads of independent writes* anomaly [7]

and cannot be produced by the GSC protocol. There two clients concurrently append 1 to different sequence objects x and y. A third client sees the append to x, but not to y, and a fourth client sees the append to y, but not to x. Thus, from the perspective the latter two clients the updates to x and y happen in different orders. This outcome cannot happen in a GSC protocol computation, because there is a single order in which the append operations will be incorporated into the server log. If $x.\mathsf{append}(1)$ precedes $y.\mathsf{append}(1)$ in the log, then the read from x in the fourth client cannot return $[\,]$; otherwise, the read from y in the third client cannot return $[\,]$.

4 Equivalence between GSP, TSO and Dual TSO

We now establish a certain relationship between special cases of the GSC model: TSO [24] (all operations pull), dual TSO [2] (all operations push) and GSP [10] (operations neither pull nor push). We prove that the sets of histories allowed by these three models are the same modulo the real-time order, which means that the models are observationally equivalent to clients that cannot communicate out-of-band [3, 12].

Formally, for an event $e = (\iota, x, op, a, fen)$ let $\mathsf{mkPull}(e) = (\iota, x, op, a, \{\mathsf{pull}\})$ and $\mathsf{mkPush}(e) = (\iota, x, op, a, \{\mathsf{push}\})$. We lift mkPull and mkPush to sets of events and relations in the expected way. Let $\mathsf{EPush} = \{e \mid \mathsf{push} \in \mathsf{fences}(e)\}$ and $\mathsf{EPull} = \{e \mid \mathsf{pull} \in \mathsf{fences}(e)\}$.

▶ **Theorem 1.**

$$\begin{aligned}\forall E.\, \forall \mathsf{so}.\, E \cap (\mathsf{EPush} \cup \mathsf{EPull}) = \emptyset \implies ((\exists \mathsf{rt}.\, (E, \mathsf{so}, \mathsf{rt}) \in \mathsf{HistGSC}) \iff \\ (\exists \mathsf{rt}'.\, (\mathsf{mkPush}(E), \mathsf{mkPush}(\mathsf{so}), \mathsf{rt}') \in \mathsf{HistGSC}) \iff \\ (\exists \mathsf{rt}''.\, (\mathsf{mkPull}(E), \mathsf{mkPull}(\mathsf{so}), \mathsf{rt}'') \in \mathsf{HistGSC})).\end{aligned}$$

We prove Theorem 1 in §7 and [15, §C]. According to it, any GSP computation of the protocol, where operations are propagated asynchronously both from clients to the server and from the server to clients, can be transformed into an equivalent-modulo-rt computation where operations can be propagated asynchronously in only one direction. While the equivalence between TSO and dual TSO has been established before [2], the result about GSP was only conjectured [10], and its proof is a contribution of the present paper. Like proofs of other results of ours, this one exploits the axiomatic specification of GSC that we present in §6.

If we take the real-time order into account and, hence, allow clients to communicate out-of-band, then GSP is strictly weaker than TSO and dual TSO, and the latter two are incomparable. In particular, the above theorem does not hold if we additionally require $\mathsf{rt}' = \mathsf{rt}$ or $\mathsf{rt}'' = \mathsf{rt}$. Indeed, as we noted in §3, the history in Figure 3(a) is allowed by GSP, but is disallowed if the operations pull; hence, it is disallowed by TSO. However, the history is allowed if all operations push and, hence, is allowed by dual TSO. The history in Figure 3(b) is similarly allowed by GSP, but is disallowed if all operations push; hence, it is disallowed by dual TSO. On the other hand, it is allowed if all operations pull and, hence, is allowed by TSO. Finally, even modulo real-time order, GSP, TSO and dual TSO are strictly weaker than linearizability [16]: the history in Figure 3(c) is allowed by these models, but is not linearizable no matter how we change the real-time order.

5 Composing GSC Objects

GSC is not a *composable* (aka *local*) property [16]: objects satisfying GSC may fail to provide this consistency guarantee when combined. Indeed, consider the history in Figure 3(d). It is easy to see that the projections of the history to events on objects x or y yield GSC histories:

e.g., the projection to x can be produced by the GSC protocol if the rightmost client is slow to pull updates from the server. However, as we explained in §3, the overall history is not GSC. We now give a condition under which multiple objects each satisfying GSC behave such that the whole set of objects satisfies GSC. The condition requires using fences according to a certain discipline, formalised as follows. A history $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$ is *well-fenced* if

$$\forall e, f \in E.\, e \xrightarrow{\mathsf{so}} f \wedge \mathsf{obj}(e) \neq \mathsf{obj}(f) \implies \exists e' \in \mathsf{EPush}.\, \exists f' \in \mathsf{EPull}.\\ \mathsf{obj}(e') = \mathsf{obj}(e) \wedge \mathsf{obj}(f') = \mathsf{obj}(f) \wedge e \xrightarrow{\mathsf{so}^?} e' \xrightarrow{\mathsf{so}} f' \xrightarrow{\mathsf{so}^?} f,$$

where $R?$ is the reflexive closure of R. The above condition requires that, when switching between different objects, a client pushes to the server the operations done on the old object and pulls from the server operations on the new object. Let us denote by $\mathcal{H}|_x$ the projection of $\mathcal{H}$ to events on an object x. The following theorem is our main result (proved in §8).

▶ **Theorem 2.** *For a well-fenced history $\mathcal{H}$, we have* $(\forall x.\, \mathcal{H}|_x \in \mathsf{HistGSC}) \implies \mathcal{H} \in \mathsf{HistGSC}$.

The theorem ensures that well-fenced clients interacting with multiple GSC services, implementing different objects, behave as though they are interacting with a single GSC service. Since our histories track the real-time order between events, this result holds even when clients can communicate out-of-band, without using GSC services. Programmers can thus ensure consistency when accessing multiple GSC services by placing fences according to the proposed discipline. Even though fences are expensive (in particular, not partition-tolerant), clients only incur this overhead when switching between different services. A client accessing the same service incurs no overhead.

For example, assume we make the upper reads in Figure 3(d) push and the lower reads pull. Then the projection of the history to y is no longer GSC: since the lower read from y happens after the upper read from y and pulls operations from the server, it has to also observe 1. Hence, in this case the outcome shown in Figure 3(d) cannot happen when clients interact with multiple GSC services. (Actually, making the upper reads push is not required to ensure this, since they are read-only operations. Our results could be strengthened to incorporate such optimisations, but for simplicity we decided to treat all operations uniformly.)

As special cases of Theorem 2, we obtain novel criteria for composing TSO and dual TSO objects. Since in TSO all operations pull, we only need to require that a client pushes operations on an object before accessing a new one. Since in dual TSO all operations push, a client need only pull operations on the new object. As a subcase of dual TSO, we obtain the recently proposed criterion for composing OSC objects [22]. Recall that in OSC all operations push and update operations pull. Hence, in this case we require that a client start accessing a new object with an update operation. This can be ensured by adding dummy updates—a policy implemented by the ZooNet system [21] for composing ZooKeeper services [18]. Thus, our results generalise the compositionality criterion for OSC.

6 Axiomatic Specification

We now present the main technical tool we use to prove Theorems 1 and 2—an *axiomatic* specification of GSC, given in the style often used for consistency models in shared-memory [19] and distributed storage systems [8, 9]. It is based on the following notion. An *abstract execution* is a triple $\mathcal{A} = ((E, \mathsf{so}, \mathsf{rt}), \mathsf{vis}, \mathsf{ar})$, where: $(E, \mathsf{so}, \mathsf{rt})$ is a history; *visibility* $\mathsf{vis} \subseteq E \times E$ is a prefix-finite acyclic relation; and *arbitration* $\mathsf{ar} \subseteq E \times E$ is a prefix-finite total order such that $\mathsf{vis} \subseteq \mathsf{ar}$. Visibility and arbitration declaratively describe how the GSC protocol

processes the operations in E. Given a computation of the protocol, we have $e \xrightarrow{\mathsf{vis}} f$ if, when a client executed the operation of f, the operation of e was in one of its three local logs. We have $e \xrightarrow{\mathsf{ar}} f$ if the operation of e preceded the one of f in the server log. Figures 3(a-c) give examples of abstract executions (we omit some edges irrelevant for the following explanations).

To define the set of histories allowed by GSC, our specification constrains abstract executions using the *consistency axioms* in Figure 4, which declaratively describe guarantees the GSC protocol provides about operation processing and are explained in the following. In the axioms $R_1; R_2$ denotes the sequential composition of relations R_1 and R_2; we define $\mathsf{ctxt}_{\mathcal{A}}$ below. The axiomatic specification admits those histories that can be extended to an abstract execution satisfying the axioms. Denoting the latter set of executions ExecGSC, the corresponding set of histories is

$$\mathsf{HistGSC}_{\mathrm{ax}} = \{\mathcal{H} \mid \exists \mathsf{vis}, \mathsf{ar}.\ (\mathcal{H}, \mathsf{vis}, \mathsf{ar}) \in \mathsf{ExecGSC}\}.$$

As the following shows, the axiomatic specification is equivalent to the operational one.

▶ **Theorem 3.** $\mathsf{HistGSC} = \mathsf{HistGSC}_{\mathrm{ax}}$.

RetVal. $\forall e \in E.\ \mathsf{rval}(e) = \mathsf{eval}(\mathsf{ctxt}_{\mathcal{A}}(e), \mathsf{oper}(e))$.
RYW. $\mathsf{so} \subseteq \mathsf{vis}$.
MonotonicView. $\mathsf{vis}\ ;\ \mathsf{so} \subseteq \mathsf{vis}$.
ObservedVis.
$\quad \mathsf{ar}?\ ;\ (\mathsf{vis} \setminus \mathsf{so})\ ;\ (\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))? \subseteq \mathsf{vis}$.
PushedVis. $\mathsf{ar}?\ ;\ (\mathsf{rt}? \cap (\mathsf{EPush} \times \mathsf{EPull})) \subseteq \mathsf{vis}?$.
ObservedAr. $(\mathsf{vis} \setminus \mathsf{so})\ ;\ \mathsf{rt} \subseteq \mathsf{ar}$.
PushedAr. $\mathsf{rt} \cap (\mathsf{EPush} \times \mathsf{Event}) \subseteq \mathsf{ar}$.
Eventual. $\forall e \in E.\ |\{f \in E \mid \neg(e \xrightarrow{\mathsf{vis}} f)\}| < \infty$.

Figure 4 Axioms of the GSC model, constraining an execution $\mathcal{A} = ((E, \mathsf{so}, \mathsf{rt}), \mathsf{vis}, \mathsf{ar})$.

We now explain the axioms in Figure 4 and, on the way, give the key ideas for the proof of the "$\subseteq$" direction of the theorem, showing the *soundness* of the axiomatic specification. Consider a computation of the GSC protocol producing a history $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$. To prove the soundness result, we extract vis and ar from the computation as described above and show that the resulting abstract execution satisfies all the axioms in Figure 4. RetVal says that the result of an operation e is computed by applying its sequential semantics to the sequence of operations given by $\mathsf{ctxt}_{\mathcal{A}}(e)$, which is obtained by arranging the operations invoked by the events in the set $\{f \mid f \xrightarrow{\mathsf{vis}} e \wedge \mathsf{obj}(e) = \mathsf{obj}(f)\}$ according to ar. For example, the execution in Figure 3(b) satisfies RetVal: the read returns $[2, 1]$ because both appends are visible to it and $x.\mathsf{append}(2) \xrightarrow{\mathsf{ar}} x.\mathsf{append}(1)$. RYW formalises the "read-your-writes" guarantee from §3: a client observes all operations it has executed before. MonotonicView similarly ensures that a client observes all operations it has observed before.

The axioms ObservedVis to PushedAr are more subtle, and we thus give detailed justifications for their soundness. They constrain vis or ar based on the fact that, by a certain moment, a particular operation was guaranteed to have been pushed to the server. In ObservedVis and ObservedAr this is the case because the operation was observed by a client other the one that that executed it (expressed in the axioms using $\mathsf{vis} \setminus \mathsf{so}$); in PushedVis and PushedAr this is the case because the operation included a push fence (expressed using EPush). In more detail, these axioms are justified as follows:

- ObservedVis. Assume $e_1 \xrightarrow{\mathsf{ar}?} e_2 \xrightarrow{\mathsf{vis}\setminus\mathsf{so}} e_3 \xrightarrow{\mathsf{rt}\cap(\mathsf{Event}\times\mathsf{EPull})?} e_4$. Since $e_2 \xrightarrow{\mathsf{vis}\setminus\mathsf{so}} e_3$, when a client executed e_3, it was aware of the event e_2 by a different client. The client could only find out about e_2 from the server, so by the time e_3 finished, e_2 was on the server.

Since $e_1 \xrightarrow{\mathsf{ar}?} e_2$, so was e_1. If $e_3 = e_4$, then the client executing this event was also aware of e_1, since clients pull operations in the order of the server log. Hence, $e_1 \xrightarrow{\mathsf{vis}} e_4$. If $e_3 \xrightarrow{\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull})} e_4$, then after e_3 finished, the client executing e_4 pulled all updates from the server, which must have included e_1. Hence, $e_1 \xrightarrow{\mathsf{vis}} e_4$ again.

- PushedVis. Assume $e_1 \xrightarrow{\mathsf{ar}?} e_2 \xrightarrow{\mathsf{rt}?} e_3$, $e_2 \in \mathsf{EPush}$ and $e_3 \in \mathsf{EPull}$. Since $e_2 \in \mathsf{EPush}$, e_2 was on the server after its operation finished. Since $e_1 \xrightarrow{\mathsf{ar}?} e_2$, so was e_1. If $e_1 = e_3$, we trivially have $e_1 \xrightarrow{\mathsf{vis}?} e_3$. Otherwise, since $e_2 \xrightarrow{\mathsf{rt}?} e_3$, e_1 was also on the server before e_3 started. Since $e_3 \in \mathsf{EPull}$, e_3 pulled all operations from the server, including e_1. Hence, $e_1 \xrightarrow{\mathsf{vis}} e_3$.
- ObservedAr. Assume $e_1 \xrightarrow{\mathsf{vis} \setminus \mathsf{so}} e_2 \xrightarrow{\mathsf{rt}} e_3$. Since $e_1 \xrightarrow{\mathsf{vis} \setminus \mathsf{so}} e_2$, e_1 must have been on the server by the time e_2 finished. Since $e_2 \xrightarrow{\mathsf{rt}} e_3$, e_3 started after e_2 finished and thus must follow e_1 in the server log. Hence, $e_1 \xrightarrow{\mathsf{ar}} e_3$.
- PushedAr. Assume $e_1 \xrightarrow{\mathsf{rt}} e_2$ and $e_1 \in \mathsf{EPush}$. Then e_1 was pushed to the server before e_2 started. Hence, e_2 was pushed onto the server after e_1, so that $e_1 \xrightarrow{\mathsf{ar}} e_2$.

Finally, the Eventual axiom guarantees that an event e can be invisible to at most finitely many other events f. Its soundness is ensured by the fairness constraints in the GSC protocol (§3). The axioms imply more properties of the relations in an execution.

▶ **Proposition 4.** *If $\mathcal{A}$ satisfies* MonotonicView *and* ObservedVis*, then $\mathsf{vis}_{\mathcal{A}}$ is transitive. If $\mathcal{A}$ satisfies* ObservedAr*, then $\mathsf{vis}_{\mathcal{A}} \cup \mathsf{rt}_{\mathcal{A}}$ is acyclic.*

The executions in Figures 3(a-c) satisfy all the axioms. On the other hand, the history in Figure 3(d) cannot be extended to an execution satisfying the axioms. Indeed, for the return values of the upper reads to be consistent with RetVal, we must have $x.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} x.\mathsf{read} : [1]$ and $y.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} y.\mathsf{read} : [1]$. Arbitration has to order the two appends one way or another. If, for example, we have $x.\mathsf{append}(1) \xrightarrow{\mathsf{ar}} y.\mathsf{append}(2)$, then by ObservedVis we must also have $x.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} x.\mathsf{read} : [\,]$, contradicting RetVal.

Recall from §3 that GSC disallows the history in Figure 3(a) if the read in the second client is a pull. Accordingly, there is no abstract execution that extends the resulting history and satisfies the axioms: by ObservedVis, in such an execution we would have $x.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} x.\mathsf{read} : [2]$, contradicting RetVal. Similarly, there is no execution that extends the history in Figure 3(b) assuming $x.\mathsf{append}(1)$ is a push. This is because by PushedAr in such an execution we must have $x.\mathsf{append}(1) \xrightarrow{\mathsf{ar}} x.\mathsf{append}(2)$, so that by RetVal the read must return $[1, 2]$. Finally, there is no execution for the history in Figure 3(c) assuming the appends push and the reads pull: by PushedVis we must have $x.\mathsf{append}(1) \xrightarrow{\mathsf{vis}} x.\mathsf{read} : [\,]$, contradicting RetVal.

As follows from the "$\supseteq$" direction of Theorem 3, the axioms in Figure 4 are also *complete*: given an abstract execution $(\mathcal{H}, \mathsf{vis}, \mathsf{ar})$, we can construct a computation of the GSC protocol producing the history $\mathcal{H}$. Due to space constraints, we defer the detailed proof of Theorem 3 to [15, §B]. The completeness part of the proof is nontrivial, but uses similar techniques to the proof of the compositionality criterion that we present in §8.

7 Proof of Model Equivalence

As a simple illustration of the use of the axiomatic specification of GSC, we prove the first "$\Longleftrightarrow$" in Theorem 1, showing that GSP and dual TSO are equivalent modulo real-time order (the rest of the proof is given in [15, §C]). Consider E and so such that $E \cap (\mathsf{EPush} \cup \mathsf{EPull}) = \emptyset$.

The "$\Longleftarrow$" direction. It is easy to see that

$$\forall \mathsf{rt}.\, (\mathsf{mkPush}(E), \mathsf{mkPush}(\mathsf{so}), \mathsf{mkPush}(\mathsf{rt})) \in \mathsf{HistGSC} \implies (E, \mathsf{so}, \mathsf{rt}) \in \mathsf{HistGSC},$$

since erasing fences from events does not invalidate any axioms.

The "$\Longrightarrow$" direction. Assume rt such that $(E, \mathsf{so}, \mathsf{rt}) \in \mathsf{HistGSC}$. Then for some vis and ar we have $\mathcal{A} \triangleq ((E, \mathsf{so}, \mathsf{rt}), \mathsf{vis}, \mathsf{ar}) \in \mathsf{ExecGSC}$. Let $\mathsf{rt}' = \mathsf{mkPush}(\mathsf{ar})$. Then

$$\mathcal{A}' \triangleq ((\mathsf{mkPush}(E), \mathsf{mkPush}(\mathsf{so}), \mathsf{rt}'), \mathsf{mkPush}(\mathsf{vis}), \mathsf{mkPush}(\mathsf{ar}))$$

is an abstract execution. Further, since $\mathcal{A}$ satisfies all GSC axioms, so does $\mathcal{A}'$. In particular, $\mathcal{A}'$ satisfies ObservedVis and PushedVis because $\mathsf{mkPush}(E) \cap \mathsf{EPull} = \emptyset$, and ObservedAr and PushedAr by the choice of rt'. This completes the proof.

Thus, our axiomatic specification allows easily proving the above model equivalence by picking a witness for the real-time order and checking axiom validity. Such a proof would be much more challenging with the operational specification, as it would require devising a nontrivial transformation of one execution of the GSC protocol into another.

8 Proof of the Compositionality Criterion

We next show how to use our axiomatic specification of the GSC model to prove Theorem 2. Here we give only the key ideas and defer the complete proof to [15, §D]. Consider a well-fenced history $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$ such that $\forall x.\, \mathcal{H}|_x \in \mathsf{HistGSC}$. Then for any x there is an execution $\mathcal{A}_x = (\mathcal{H}|_x, \mathsf{vis}_x, \mathsf{ar}_x) \in \mathsf{ExecGSC}$. We need to show $\mathcal{H} \in \mathsf{HistGSC}$, to which end we construct an execution $\mathcal{A} = (\mathcal{H}, \mathsf{vis}, \mathsf{ar}) \in \mathsf{ExecGSC}$.

Let $\mathsf{so}_0 = \bigcup_{x \in \mathsf{Obj}} \mathsf{so}_{\mathcal{H}|_x}$, $\mathsf{vis}_0 = \bigcup_{x \in \mathsf{Obj}} \mathsf{vis}_x$ and $\mathsf{ar}_0 = \bigcup_{x \in \mathsf{Obj}} \mathsf{ar}_x$. It is reasonable to expect vis and ar to extend the corresponding per-object orders in $\mathcal{A}_x$, so we should have $\mathsf{vis}_0 \subseteq \mathsf{vis}$ and $\mathsf{ar}_0 \subseteq \mathsf{ar}$. The most difficult part is to construct ar; once this is done, we construct vis as the smallest relation containing vis_0 that is a solution to the system of inequalities given by the axioms RYW-PushedVis in Figure 4. The following lemma gives a closed form for this solution. Let $\mathsf{Id} = \{(e, e) \mid e \in E\}$.

▶ **Lemma 5.** *Given any arbitration order* $\mathsf{ar} \supseteq \mathsf{ar}_0$*, the relation*

$$\mathsf{vis} = \mathsf{so} \cup (\mathsf{ar}? \,;\, (\mathsf{vis}_0 \setminus \mathsf{so}) \,;\, (\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))? \,;\, \mathsf{so}?) \cup ((\mathsf{ar}? \,;\, (\mathsf{rt}? \cap (\mathsf{EPush} \times \mathsf{EPull})) \,;\, \mathsf{so}?) \setminus \mathsf{Id})$$

is the smallest one such that $\mathsf{vis}_0 \subseteq \mathsf{vis}$ *and* $(\mathcal{H}, \mathsf{vis}, \mathsf{ar})$ *satisfies* RYW-PushedVis.

The first component of vis is meant to validate RYW, the second ObservedVis and the third PushedVis. Appending $\mathsf{so}?$ at the end of the last two components validates MonotonicView.

We now describe the construction of ar. This order needs to include several relations. Since $\mathsf{vis}_0 \subseteq \mathsf{vis}$ and $\mathcal{A}$ should satisfy ObservedAr, we must have $(\mathsf{vis}_0 \setminus \mathsf{so}) \,;\, \mathsf{rt} \subseteq \mathsf{ar}$. Since $\mathcal{A}$ should satisfy PushedAr we must have $\overline{\mathsf{rt}} \triangleq \mathsf{rt} \cap (\mathsf{EPush} \times \mathsf{Event}) \subseteq \mathsf{ar}$. Since $\mathcal{A}$ should satisfy RYW and $\mathsf{vis} \subseteq \mathsf{ar}$, we must have $\mathsf{so} \subseteq \mathsf{ar}_0$. Finally, for $\mathcal{A}$ to satisfy RetVal, ar should include one more relation that is more subtle. We illustrate the need for it using the example in Figure 5. Assume that we have the solid edges in the figure. If we arbitrate between the

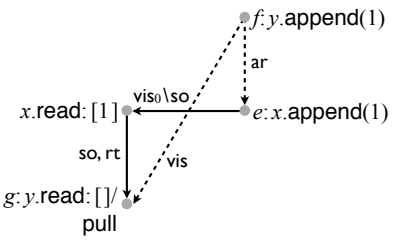

Figure 5 Motivation for $\prec$.

two appends as shown by the dashed edge $f \xrightarrow{\mathsf{ar}} e$, then according to the construction in Lemma 5 we will also have the dashed edge $f \xrightarrow{\mathsf{vis}} g$ (needed for $\mathcal{A}$ to satisfy ObservedVis). But then the resulting $\mathcal{A}$ will violate RetVal. We therefore include the following relation into ar, which ensures that such situations do not happen:

$$\begin{aligned} e \prec f \iff \exists g.\, \mathsf{obj}(f) = \mathsf{obj}(g) \wedge (f,g) \notin \mathsf{vis}_0 \wedge \\ (e,g) \in (\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ (\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))\ ;\ \mathsf{so}_0? \cup (\mathsf{rt} \cap (\mathsf{EPush} \times \mathsf{EPull}))\ ;\ \mathsf{so}_0?. \end{aligned}$$

If $e \prec f$, then adding an edge $f \xrightarrow{\mathsf{ar}} e$ would create a visibility edge $f \xrightarrow{\mathsf{vis}} g$ between events on the same object that is not in vis_0. Note that the expression covering (e,g) above is more specific than the one in Lemma 5: we have so_0 instead of so, and rt must be used. This is crucial for the proof (specifically, Lemma 6 below) and, as we show, is still sufficient to validate RetVal because the history $\mathcal{H}$ is well-fenced.

Thus, we need to construct an ar that includes $R \triangleq \overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}) \cup \prec$. For this to be possible, R has to be acyclic.

▶ **Lemma 6.** *$\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}) \cup \prec$ is acyclic.*

Establishing this lemma is the most subtle part of the proof. To do this, we construct a closed-form expression covering the transitive closure of R.

▶ **Lemma 7.**

$$\begin{aligned} &(\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}) \cup \prec)^+ \\ &= (\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}))^+ \cup (\prec \cup \mathsf{ar}_0\ ;\ \prec)\ ;\ (\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}))^* \quad \textit{and} \\ &(\overline{\mathsf{rt}} \cup \mathsf{so} \cup \mathsf{ar}_0 \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}))^+ \\ &\subseteq \overline{\mathsf{rt}} \cup \mathsf{ar}_0 \cup (\mathsf{ar}_0\ ;\ \overline{\mathsf{rt}}) \cup (\overline{\mathsf{rt}}\ ;\ \mathsf{ar}_0) \cup (\mathsf{ar}_0\ ;\ \overline{\mathsf{rt}}\ ;\ \mathsf{ar}_0) \cup ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt}) \cup \\ &(\mathsf{ar}_0\ ;\ ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt})) \cup (((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt})\ ;\ \mathsf{ar}_0) \cup (\mathsf{ar}_0\ ;\ ((\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ \mathsf{rt})\ ;\ \mathsf{ar}_0). \end{aligned}$$

The proof Lemma 7 relies on establishing that components of R satisfy various algebraic properties, some of which exploit the fact that the history $\mathcal{H}$ is well-fenced. For example, we prove that $\prec$ is a strict partial order, i.e., transitive and irreflexive.

To prove Lemma 6, it is thus sufficient to prove that the relation covering R^+ in Lemma 7 is irreflexive. This relation describes only particular paths in R of length at most 5. Its irreflexivity is then established by a case analysis on these paths.

Using Lemma 6, we can extend R to a prefix-finite total order, which we take as ar; then vis is defined by Lemma 5. We can then show that vis defined in this way is prefix-finite, acyclic and $\mathsf{vis} \subseteq \mathsf{ar}$, so that $\mathcal{A} = (\mathcal{H}, \mathsf{vis}, \mathsf{ar})$ is an abstract execution. By Lemma 5, $\mathcal{A}$ satisfies RYW-PushedVis. It satisfies PushedAr because $\overline{\mathsf{rt}} \subseteq \mathsf{ar}$, and it is also easy to check that it satisfies ObservedAr.

We next argue that $\mathcal{A}$ satisfies RetVal, which exploits the particular way in which we constructed ar. To this end, we show that for any object x we have $\mathsf{vis}|_x = \mathsf{vis}_x$, where $\mathsf{vis}|_x$ is the projection of vis to events on x. Then since for any x we have $\mathsf{ar}_x \subseteq \mathsf{ar}$ and $\mathcal{A}_x$ satisfies RetVal, so does $\mathcal{A}$. Since $\mathsf{vis}_x \subseteq \mathsf{vis}$ by construction, we only need to show $\mathsf{vis}|_x \subseteq \mathsf{vis}_x$. Consider arbitrary $f, g \in E$ such that $\mathsf{obj}(f) = \mathsf{obj}(g) = x$ and $f \xrightarrow{\mathsf{vis}} g$. To show $f \xrightarrow{\mathsf{vis}_x} g$ our proof considers several cases corresponding to which of the components of the union defining vis in Lemma 5 the edge (f,g) belongs to. For illustration, here we only consider a single case when (f,g) comes from the following instance of the second component of the union, which uses an rt edge: $(f,g) \in \mathsf{ar}?\ ;\ (\mathsf{vis}_0 \setminus \mathsf{so})\ ;\ (\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))\ ;\ \mathsf{so}?$. Then for some g' we have

$$f \xrightarrow{\mathsf{ar}?;(\mathsf{vis}_0 \setminus \mathsf{so});(\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))} g' \xrightarrow{\mathsf{so}?} g.$$

Figure 5 illustrates the case when $g' = g$. If $\mathsf{obj}(g') \neq \mathsf{obj}(g)$, then since the history $\mathcal{H}$ is well-fenced, for some $g'' \in \mathsf{EPull}$ we have $g' \xrightarrow{\mathsf{so}} g'' \xrightarrow{\mathsf{so}_0?} g$. Since $\mathsf{so} \subseteq \mathsf{rt}$, this implies $g' \xrightarrow{\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull})} g'' \xrightarrow{\mathsf{so}_0?} g$. Hence,

$$f \xrightarrow{\mathsf{ar}?;(\mathsf{vis}_0 \setminus \mathsf{so});(\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))} g'' \xrightarrow{\mathsf{so}_0?} g. \tag{1}$$

If $\mathsf{obj}(g') = \mathsf{obj}(g)$, then $g' \xrightarrow{\mathsf{so}_0?} g$ and we again have (1) for $g'' = g'$. Thus, in all cases (1) holds for some g''. Then for some e we have

$$f \xrightarrow{\mathsf{ar}?} e \xrightarrow{(\mathsf{vis}_0 \setminus \mathsf{so});(\mathsf{rt} \cap (\mathsf{Event} \times \mathsf{EPull}))} g'' \xrightarrow{\mathsf{so}_0?} g.$$

Now if $\neg(f \xrightarrow{\mathsf{vis}_x} g)$, then $e \prec f$, contradicting $\prec \subseteq \mathsf{ar}$. Hence, $f \xrightarrow{\mathsf{vis}_x} g$, as required.

Thus, $\mathcal{A}$ satisfies all GSC axioms except for possibly Eventual. Since $\forall x.\, \mathsf{vis}|_x = \mathsf{vis}_x$ and $\mathcal{A}_x$ satisfies Eventual, we have

$$\forall e \in E.\, |\{f \in E \mid \mathsf{obj}(e) = \mathsf{obj}(f) \wedge \neg(e \xrightarrow{\mathsf{vis}} f)\}| < \infty, \tag{2}$$

i.e., an event e cannot be invisible to infinitely many events f on the same object. Then, as the following lemma shows, we can extend vis so as to validate Eventual without invalidating any of the other axioms.

▶ **Lemma 8.** *Let $\mathcal{H} = (E, \mathsf{so}, \mathsf{rt})$ and $\mathcal{A} = (\mathcal{H}, \mathsf{vis}, \mathsf{ar})$ be an execution that satisfies all GSC axioms except for possibly* Eventual. *Assume (2) holds. Then there exists $\mathsf{vis}' \supseteq \mathsf{vis}$ such that $(\mathcal{H}, \mathsf{vis}', \mathsf{ar}) \in \mathsf{ExecGSC}$.*

We thus construct an execution $(\mathcal{H}, \mathsf{vis}', \mathsf{ar}) \in \mathsf{ExecGSC}$, which shows that $\mathcal{H} \in \mathsf{HistGSC}$ and thereby establishes Theorem 2.

The axiomatic specification of GSC plays an important role in the above proof. It allows us to concisely state constraints that the global order on operations represented by ar needs to satisfy for the global execution to be GSC. We can then show that the desired global order exists by proving algebraic properties over relations, as exemplified by Lemma 7.

9 Related Work and Discussion

Lev-Ari et al. [22] have proposed a criterion for composing objects providing Ordered Sequential Consistency (OSC), which is a special case of our results (§5). In comparison to them, we handle a more complex consistency model, which requires a different proof approach: specifying the consistency model axiomatically and reasoning about it using algebraic techniques. Lev-Ari et al. have also implemented their criterion in a library for composing ZooKeeper instances and showed that it has a competitive performance [21]. We hope that our results will enable similar practical implementations for systems providing other consistency models from the family we considered. In particular, the implementation of GSP in Orleans [5] provides only per-object consistency guarantees, and our results should allow its clients to use multiple objects while preserving the consistency model.

There are other widely used consistency models that are in general non-composable, such as sequential consistency [20]. Perrin et al. [25] proposed conditions on the use of sequentially consistent concurrent objects under which a composition of multiple objects stays sequentially consistent. Our compositionality result is similar in spirit, but handles a family of more complex consistency models implemented in modern systems [10, 18, 23]. Vitenberg and Friedman [30] showed that combining sequential consistency with any composable property

yields a non-composable property. Our compositionality criterion does not contradict this result, since well-fencedness of histories is not a composable property.

Our operational specification of the GSC model generalizes the GSP protocol [10], with significant differences. First, GSP allows only pure read and update operations, while GSC permits mixed operations that both modify the state and return a value to the caller. Second, GSP does not support push and pull fences that are attached to operations. Rather, its original proposal [10] investigated stronger synchronization primitives, such as standalone fences and transactions, which cannot be used to define TSO, dual TSO and OSC as special cases. Therefore, GSP is unsuitable to serve as a unifying model that clarifies the relationship between these instances.

Axiomatic specifications have been previously proposed for consistency models in shared-memory [19, 23] and distributed storage systems [8, 9]. Our GSC specification uses the same framework as for the latter. Researchers have proposed axiomatic specifications for TSO-like models and proved their equivalence to operational ones [17, 23]. However, our specifications are the first to formalise the role of the real-time order in distinguishing between these models. Including real-time order into axiomatic models [8] is important in a distributed setting because of the possibility of out-of-band communication between clients; without this one cannot safely substitute implementations for specifications [3, 12].

We have exploited the axiomatic specification of GSC to establish a compositionality criterion and an equivalence between GSP and TSO/dual TSO. However, axiomatic specifications of consistency models have been shown useful to obtain other kinds of results, such as criteria for *robustness*—checking when an application running on a weak consistency model behaves as if it runs on a strong one [4, 27]. We hence hope that our specifications will allow obtaining such results for consistency models with global operation sequencing.

Acknowledgements. We thank Idit Keidar, Kfir Lev-Ari and Matthieu Perrin for helpful comments. Gotsman was supported by an ERC Starting Grant RACCOON.

References

1 Microsoft TouchDevelop. https://www.touchdevelop.com/.

2 P. A. Abdulla, M. F. Atig, A. Bouajjani, and T. P. Ngo. The benefits of duality in verifying concurrent programs under TSO. In *CONCUR: International Conference on Concurrency Theory*, 2016.

3 H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In *PODC: Symposium on Principles of Distributed Computing*, 2013.

4 G. Bernardi and A. Gotsman. Robustness against consistency models with atomic visibility. In *CONCUR: International Conference on Concurrency Theory*, 2016.

5 P. Bernstein, S. Burckhardt, S. Bykov, N. Crooks, J. Faleiro, G. Kliot, A. Kumbhare, M. R. Rahman, V. Shah, A. Szekeres, and J. Thelin. Geo-distribution of actor-based services. Technical Report MSR-TR-2017-3, Microsoft Research, 2017.

6 P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, Microsoft Research, 2014.

7 H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI: Conference on Programming Language Design and Implementation*, 2008.

8 S. Burckhardt. *Principles of Eventual Consistency*. now publishers, 2014.

9 S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *POPL: Symposium on Principles of Programming Languages*, 2014.

10 S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In *ECOOP: European Conference on Object-Oriented Programming*, 2015.

11 X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4), 2004.

12 I. Filipovic, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52), 2010.

13 P. C. Fishburn. Intransitive indifference with unequal indifference intervals. *Journal of Mathematical Psychology*, 7, 1970.

14 S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

15 A. Gotsman and S. Burckhardt. Consistency models with global operation sequencing and their composition (extended version). *arXiv CoRR*, 1707.09242, 2017.

16 M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

17 L. Higham and J. Kawash. Memory consistency and process coordination for sparc multiprocessors. In *HiPC: International Conference on High Performance Computing*, 2000.

18 P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC: USENIX Annual Technical Conference*, 2010.

19 ISO/IEC Standard. *Programming Languages – C++, 14882:2011*. 2011.

20 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), 1979.

21 K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer. Modular composition of coordination services. In *USENIX ATC: USENIX Annual Technical Conference*, 2016.

22 K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer. Composing ordered sequential consistency. *Information Processing Letters*, 123, 2017.

23 S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs: International Conference on Theorem Proving in Higher Order Logics*, 2009.

24 S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO. In *SPAA'95: Symposium on Parallel Algorithms and Architectures*, 1995.

25 M. Perrin, M. Petrolia, A. Mostéfaoui, and C. Jard. On composition and implementation of sequential consistency. In *DISC: International Symposium on Distributed Computing*, 2016.

26 F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22, 1990.

27 D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2), 1988.

28 D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

29 D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS: Conference on Parallel and Distributed Information Systems*, 1994.

30 R. Vitenberg and R. Friedman. On the locality of consistency conditions. In *DISC: International Symposium on Distributed Computing*, 2003.