Consistency Models with Global Operation Sequencing and their Composition

Alexey Gotsman\(^1\) and Sebastian Burckhardt\(^2\)

1 IMDEA Software Institute, Madrid, Spain
2 Microsoft Research, Redmond, 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

\(^*\) Alexey Gotsman was supported by an ERC Starting Grant RACCOON.
Consistency Models with Global Operation Sequencing and their Composition

Table 1 Specialising GSC.

<table>
<thead>
<tr>
<th>Implicit fences</th>
<th>pull</th>
<th>push</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSP [10]</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>TSO [24, 23]</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>dual TSO [2]</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>OSC [22]</td>
<td>updates</td>
<td>yes</td>
</tr>
<tr>
<td>linearizability [16]</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

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 [24, 23], 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 Table 1.

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 [22, 21].

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 [9, 8] (§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 [12, 3].

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 \( \text{Obj} = \{x, y, \ldots\} \). A finite number of clients interact with the service by performing operations on the objects, which are ranged over by \( \text{op} \) and come from a set \( \text{Op} \). 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 \( \text{Val} \); we can use a special member of \( \text{Val} \) to model operations that return no value. The sequential semantics of operations is defined by a function \( \text{eval} : \text{op}^* \times \text{Op} \to \text{Val} \) 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 \( \iota \in Id \) is the event identifier, \( x \in \text{Obj} \) is the object on which the event occurs, \( op \in \text{Op} \) is the operation done, \( a \in \text{Val} \) is its return value, and \( fen \subseteq \{\text{push, pull}\} \) gives the fences requested by the client. We use \( \text{obj}(e), \text{oper}(e), \text{rval}(e), \text{fences}(e) \) to select event components.

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

\[
\forall e_1, e_2, f_1, f_2, (e_1 \xrightarrow{R} e_2 \land f_1 \xrightarrow{R} f_2) \implies (e_1 \xrightarrow{R} f_2 \lor f_1 \xrightarrow{R} e_2).
\]

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 \( \mathcal{H} = (E, \text{so}, \text{rt}) \), where \( E \subseteq \text{Event} \); session order \( \text{so} \subseteq E \times E \) 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 \( \text{rt} \subseteq E \times E \) is a prefix-finite interval order such that \( \text{so} \subseteq \text{rt} \) and \( \forall e \in E. |\{f \in E \mid \neg(e \xrightarrow{\text{rt}} f)\}| < \infty \).

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 \( e \xrightarrow{\text{rt}} f \) tells us that the operation of \( e \) finished before the one of \( f \) started (the last restriction on \( \text{rt} \) 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 \( f \) [12, 3]. We denote components of histories and similar structures as in \( E_{\mathcal{H}} \) and \( \text{so}_{\mathcal{H}} \). A consistency model is defined by a set of histories.

### 3 Operational Specification

We define Global Sequence Consistency using the idealised protocol in Figure 1, 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 push\(_c\) and 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 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 pull\(_c\) function models
State for each client $c$:
- $\text{known}_c \in (\text{Id} \times \text{Op})^*$
- $\text{unacked}_c \in (\text{Id} \times \text{Op})^*$
- $\text{pending}_c \in (\text{Id} \times \text{Op})^*$

exec($c, \text{op}, \text{fen}$):
  if (pull $\in \text{fen}$)
    while ($\text{known}_c \neq \text{server}_\text{log}$) pull($c$)
    result :=
    eval(stripIds($\text{known}_c \cdot \text{unacked}_c \cdot \text{pending}_c, \text{op}$))
    if (push $\in \text{fen}$)
      while ($\text{pending}_c \neq []$) push($c$)
    return result

Server state:
- $\text{server}_\text{log} \in (\text{Id} \times \text{Op})^*$
- push($c$):
  if ($\text{pending}_c = (\text{id}, \text{op}) \cdot \text{remaining}_c$)
    $\text{server}_\text{log} := \text{server}_\text{log} \cdot (\text{id}, \text{op})$
    $\text{unacked}_c := \text{unacked}_c \cdot (\text{id}, \text{op})$
    $\text{pending}_c := \text{remaining}_c$
  pull($c$):
  if ($\text{server}_\text{log} = \text{known}_c \cdot (\text{id}, \text{op}) \cdot _{-}$)
    $\text{known}_c := \text{known}_c \cdot (\text{id}, \text{op})$
  if ($\text{unacked}_c = (\text{id}, \text{op}) \cdot \text{remaining}_c$)
    $\text{unacked}_c := \text{remaining}_c$

Figure 1 The pseudocode of the protocol defining the GSC model. We denote sequence concatenation by ·, an empty sequence by [] and an irrelevant expression by _.

how the client $c$ learns about the next entry in the server log: it appends to $\text{known}_c$ the next operation in $\text{server}_\text{log}$ that is not yet part of $\text{known}_c$. If this operation is an echo of an operation previously executed by the same client $c$, we remove it from the $\text{unacked}_c$ log: the protocol ensures that in this case the operation is the first (oldest) one in $\text{unacked}_c$.

We model a client $c$ executing an operation $\text{op}$ with fences $\text{fen} \subseteq \{\text{push}, \text{pull}\}$ by exec($c, \text{op}, \text{fen}$). The body of exec() is executed atomically, and only a single invocation of it can be in progress per client. At the beginning of exec(), we handle pull fences by repeatedly calling pull($c$) until the local $\text{known}_c$ matches $\text{server}_\text{log}$. At the end of exec(), we handle push fences by repeatedly calling push($c$) until all $\text{pending}_c$ operations have been processed by the server. At the core of exec(), we first compute the result of the operation by conjoining the logs $\text{known}_c, \text{unacked}_c$ and $\text{pending}_c$, stripping identifiers using stripIds and applying the sequential semantics of operations defined by eval (§2). We then append the operation to the $\text{pending}_c$ with a unique identifier generated by uniqueld. Since $\text{op}$ is evaluated on a log that includes $\text{unacked}_c$ and $\text{pending}_c$, 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 $\text{fen}$ is empty, exec($c, \text{op}, \text{fen}$) 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 exec() terminates.

The set of histories ($E, \text{so}, \text{rt}$) allowed by GSC is defined by considering all possible computations of the above protocol. The invocations of 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 exec() function of the former finishes before the exec() function the latter starts. We denote the set of histories defined in this way HistGSC.

By systematically associating fences with operations in GSC we get various existing models as its special cases (Table 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 [24, 23]. 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 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: \(\text{Op} = \text{OpReadOnly} \uplus \text{OpUpdate}\). Read-only operations do not change the state of an object: for any operation \(\text{op}\) and a sequence of operations \(\xi\), we have \(\text{eval}(\xi, \text{op}) = \text{eval}(\xi_{\text{OpUpdate}}, \text{op})\), where \(\xi_{\text{OpUpdate}}\) is the projection of \(\xi\) onto \(\text{OpUpdate}\). 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 2(a-c) (for now ignore the extra relations \(\text{vis}\) and \(\text{ar}\)). They use sequence objects \(x\) and \(y\) for which \(\text{eval}(\xi, \text{read})\) returns the sequence of values in the append operations in \(\xi\). The histories in Figures 2(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 \(\text{append}(1)\) 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 \(\text{append}(1)\) finishes before \(\text{append}(2)\) starts. This can happen if the latter operation is pushed to the server before the
former. This outcome is disallowed if 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.append(1) precedes y.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 [12, 3].

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

\begin{flushleft}
\textbf{Theorem 1.}
\end{flushleft}

\[ \forall E. \forall so. E \cap (EPush \cup EPull) = \emptyset \implies (\exists rt. (E, so, rt) \in HistGSC) \iff (\exists rt'. (mkPush(E), mkPush(so), rt') \in HistGSC) \iff (\exists rt''. (mkPull(E), mkPull(so), rt'') \in HistGSC)). \]

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 the 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 $rt' = rt$ or $rt'' = rt$. Indeed, as we noted in §3, the history in Figure 2(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 2(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.
Consistency Models with Global Operation Sequencing and their Composition

than linearizability [16]: the history in Figure 2(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 2(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 \( H = (E, so, rt) \) is well-fenced if

\[
\forall e, f \in E. e \xrightarrow{so} f \land \text{obj}(e) \neq \text{obj}(f) \implies \exists e' \in \text{EPush}. \exists f' \in \text{EPull}.
\]

\[
\text{obj}(e') = \text{obj}(e) \land \text{obj}(f') = \text{obj}(f) \land e \xrightarrow{so} e' \xrightarrow{so} f' \xrightarrow{so} f,
\]

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 \( H|_x \) the projection of \( H \) to events on an object \( x \). The following theorem is our main result (proved in §8).

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

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 2(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 2(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.
We now present the main technical tool we use to prove Theorems 1 and 2 – an axiomatic specification. As the following shows, the axiomatic specification is equivalent to the operational one.

\[ \text{RetVal. } \forall e \in E. \text{val}(e) = \text{eval} (\text{ctxt}_A(e), \text{oper}(e)). \]

\[ \text{RYW. } \text{so} \subseteq \text{vis}. \]

\[ \text{MonotonicView. } \text{vis} ; \text{so} \subseteq \text{vis}. \]

\[ \text{ObservedVis. } (\text{ar} ? ; (\text{vis} \setminus \text{so}) ; (\text{vis} \cap (\text{Event} \times \text{EPull})) \subseteq \text{vis}. \]

\[ \text{PushedVis. } (\text{ar} ? ; (\text{vis} \cap (\text{EPush} \times \text{EPull})) \subseteq \text{vis} \wedge \text{so} \cap \text{vis} \subseteq \text{vis}. \]

\[ \text{ObservedAr. } (\text{vis} \setminus \text{so}) ; \text{rt} \subseteq \text{ar}. \]

\[ \text{PushedAr. } \text{rt} \cap (\text{EPush} \times \text{Event}) \subseteq \text{ar}. \]

\[ \text{Eventual. } \forall e \in E. |\{ f \in E | \neg (e \overset{\text{vis}}{\rightarrow} f)\}| < \infty. \]

\[ \text{Figure 3} \text{ Axioms of the GSC model, constraining an execution } A = ((E, \text{so}, \text{rt}), \text{vis}, \text{ar}). \]

### 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 [9, 8]. It is based on the following notion. An abstract execution is a triple \( A = ((E, \text{so}, \text{rt}), \text{vis}, \text{ar}) \), where \((E, \text{so}, \text{rt})\) is a history; \text{visibility vis} \subseteq E \times E \) is a prefix-finite acyclic relation; and \text{arbitration ar} \subseteq E \times E \) is a prefix-finite total order such that \text{vis} \subseteq \text{ar} \). Visibility and arbitration declaratively describe how the GSC protocol processes the operations in \( E \). Given a computation of the protocol, we have \( e \overset{\text{vis}}{\rightarrow} f \) if, when a client executed the operation of \( e \), the operation of \( e \) was in one of its three local logs. We have \( e \overset{\text{ar}}{\rightarrow} f \) if the operation of \( e \) preceded the one of \( f \) in the server log. Figures 2(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 3, 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 \( \text{ctxt}_A \) below. The axiomatic specification admits those histories that can be extended to an abstract execution satisfying the axioms. Denoting the latter set of executions \( \text{ExecGSC} \), the corresponding set of histories is

\[ \text{HistGSC}_\text{ax} = \{ H | \exists \text{vis, ar. } (H, \text{vis}, \text{ar}) \in \text{ExecGSC} \}. \]

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

**Theorem 3.** \( \text{HistGSC} = \text{HistGSC}_\text{ax} \).

We now explain the axioms in Figure 3 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 \( H = (E, \text{so}, \text{rt}) \). To prove the soundness result, we extract \text{vis} and \text{ar} from the computation as described above and show that the resulting abstract execution satisfies all the axioms in Figure 3. \text{RetVal} says that the result of an operation \( e \) is computed by applying its sequential semantics to the sequence of operations given by \( \text{ctxt}_A(e) \), which is obtained by arranging the operations invoked by the events in the set \( \{ f | f \overset{\text{vis}}{\rightarrow} e \wedge \text{obj}(e) = \text{obj}(f) \} \) according to \text{ar}. For example, the execution in Figure 2(b) satisfies \text{RetVal}: the read returns [2,1] because both appends are visible to it and \( x.\text{append}(2) \overset{\text{ar}}{\rightarrow} x.\text{append}(1) \). \text{RYW} formalises the “read-your-writes” guarantee from §3: a client observes all operations it has executed before. \text{MonotonicView} similarly ensures that a client observes all operations it has observed before.
The axioms $\text{ObservedVis}$ to $\text{PushedAr}$ are more subtle, and we thus give detailed justifications for their soundness. They constrain $\text{vis} \lor \text{ar}$ based on the fact that, by a certain moment, a particular operation was guaranteed to have been pushed to the server. In $\text{ObservedVis}$ and $\text{ObservedAr}$ this is the case because the operation was observed by a client other the one that that executed it (expressed in the axioms using $\text{vis} \setminus \text{so}$); in $\text{PushedVis}$ and $\text{PushedAr}$ this is the case because the operation included a push fence (expressed using $\text{EPush}$). In more detail, these axioms are justified as follows:

- $\text{ObservedVis}$. Assume $e_1 \xrightarrow{ar} e_2 \xrightarrow{vis} e_3 \xrightarrow{rt (\text{Event} \times \text{EPull})} e_4$. Since $e_2 \xrightarrow{vis} e_3$, 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 $e_1 \xrightarrow{ar} e_2$, 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, $e_1 \xrightarrow{\text{vis}} e_4$. If $e_3 \xrightarrow{rt (\text{Event} \times \text{EPull})} e_4$, then after $e_3$ finished, the client executing $e_4$ pulled all updates from the server, which must have included $e_1$. Hence, $e_1 \xrightarrow{\text{vis}} e_4$ again.

- $\text{PushedVis}$. Assume $e_1 \xrightarrow{ar} e_2 \xrightarrow{vis} e_3, e_2 \in \text{EPush}$ and $e_3 \in \text{EPull}$. Since $e_2 \in \text{EPush}$, $e_2$ was on the server after its operation finished. Since $e_1 \xrightarrow{ar} e_2$, so was $e_1$. If $e_2 = e_3$, we trivially have $e_1 \xrightarrow{vis} e_3$. Otherwise, since $e_2 \xrightarrow{vis} e_3$, $e_1$ was also on the server before $e_3$ started. Since $e_3 \in \text{EPull}$, $e_3$ pulled all operations from the server, including $e_1$. Hence, $e_1 \xrightarrow{\text{vis}} e_3$.

- $\text{ObservedAr}$. Assume $e_1 \xrightarrow{vis} e_2 \xrightarrow{ar} e_3$. Since $e_1 \xrightarrow{vis} e_2$, $e_1$ must have been on the server by the time $e_2$ finished. Since $e_2 \xrightarrow{ar} e_3$, $e_3$ started after $e_2$ finished and thus must follow $e_1$ in the server log. Hence, $e_1 \xrightarrow{ar} e_3$.

- $\text{PushedAr}$. Assume $e_1 \xrightarrow{ar} e_2$ and $e_1 \in \text{EPush}$. 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 $e_1 \xrightarrow{ar} e_2$.

Finally, the $\text{Eventual}$ 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 $A$ satisfies $\text{MonotonicView}$ and $\text{ObservedVis}$, then $\text{vis}_A$ is transitive. If $A$ satisfies $\text{ObservedAr}$, then $\text{vis}_A \cup \text{rt}_A$ is acyclic.

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

Recall from §3 that GSC disallows the history in Figure 2(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 $\text{ObservedVis}$, in such an execution we would have $x.\text{append}(1) \xrightarrow{\text{vis}} x.\text{read} : [2]$, contradicting $\text{RetVal}$. Similarly, there is no execution that extends the history in Figure 2(b) assuming $x.\text{append}(1)$ is a push. This is because by $\text{PushedAr}$ in such an execution we must have $x.\text{append}(1) \xrightarrow{ar} x.\text{append}(2)$, so that by $\text{RetVal}$ the read must return $[1,2]$. Finally, there is no execution for the history in Figure 2(c) assuming the appends push and the reads pull: by $\text{PushedVis}$ we must have $x.\text{append}(1) \xrightarrow{\text{vis}} x.\text{read} : []$, contradicting $\text{RetVal}$.
As follows from the “⇒” direction of Theorem 3, the axioms in Figure 3 are also complete: given an abstract execution \((H, \text{vis}, \text{ar})\), we can construct a computation of the GSC protocol producing the history \(H\). 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 “⇐” 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 \(E \cap (\text{EPush} \cup \text{EPull}) = \emptyset\). The “⇐” direction. It is easy to see that

\[
\forall r. (\text{mkPush}(E), \text{mkPush}(so), \text{mkPush}(rt)) \in \text{HistGSC} \implies (E, so, rt) \in \text{HistGSC},
\]

since erasing fences from events does not invalidate any axioms.

The “⇒” direction. Assume rt such that \((E, so, rt) \in \text{HistGSC}\). Then for some vis and ar we have \(A \models ((E, so, rt), \text{vis}, \text{ar}) \in \text{ExecGSC}\). Let \(rt' = \text{mkPush}(ar)\). Then

\[
A' \models ((\text{mkPush}(E), \text{mkPush}(so), rt'), \text{mkPush}(\text{vis}), \text{mkPush}(ar))
\]

is an abstract execution. Further, since \(A\) satisfies all GSC axioms, so does \(A'\). In particular, \(A'\) satisfies \text{ObservedVis} and \text{PushedVis} because \(\text{mkPush}(E) \cap \text{EPull} = \emptyset\), and \text{ObservedAr} and \text{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 \(H = (E, so, rt)\) such that \(\forall x. H|_x \in \text{HistGSC}\). Then for any \(x\) there is an execution \(A_x = (H|_x, \text{vis}_x, \text{ar}_x) \in \text{ExecGSC}\). We need to show \(H \in \text{HistGSC}\), to which end we construct an execution \(A = (H, \text{vis}, \text{ar}) \in \text{ExecGSC}\).

Let \(so_0 = \bigcup_{x \in \text{Obj}} so_{H|_x}\), \(\text{vis}_0 = \bigcup_{x \in \text{Obj}} \text{vis}_x\), and \(\text{ar}_0 = \bigcup_{x \in \text{Obj}} \text{ar}_x\). It is reasonable to expect \text{vis} and \text{ar} to extend the corresponding per-object orders in \(A_x\), so we should have \(\text{vis}_0 \subseteq \text{vis}\) and \(\text{ar}_0 \subseteq \text{ar}\). The most difficult part is to construct \text{ar}; once this is done, we construct \text{vis} as the smallest relation containing \(\text{vis}_0\) that is a solution to the system of inequalities given by the axioms \text{RYW-PushedVis} in Figure 3. The following lemma gives a closed form for this solution. Let \(\text{Id} = \{(e, e) \mid e \in E\}\).

Lemma 5. Given any arbitration order \(\text{ar} \supseteq \text{ar}_0\), the relation

\[
\text{vis} = so \cup (\text{ar}?; (\text{vis}_0 \setminus so); (\text{rt} \cap (\text{Event} \times \text{EPull}))?; so?) \cup ((\text{ar}?; \text{rt}? \cap (\text{EPush} \times \text{EPull}))?; so?) \setminus \text{Id})
\]

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

The first component of \text{vis} is meant to validate \text{RYW}, the second \text{ObservedVis} and the third \text{PushedVis}. Appending \(so_0\) at the end of the last two components validates \text{MonotonicView}.
We now describe the construction of \( \mathcal{A} \). This order needs to include several relations. Since \( \text{vis}_0 \subseteq \text{vis} \) and \( \mathcal{A} \) should satisfy \texttt{ObservedAR}, we must have \((\text{vis}_0 \setminus \text{so}) : \text{rt} \subseteq \mathcal{A}\). Since \( \mathcal{A} \) should satisfy \texttt{PushedAR} we must have \( \mathcal{A} = \text{rt} \cap (\text{EPush} \times \text{Event}) \subseteq \mathcal{A} \). Since \( \mathcal{A} \) should satisfy \texttt{RYW} and \( \text{vis} \subseteq \mathcal{A} \), we must have \( \text{so} \subseteq \mathcal{A}_0 \). Finally, for \( \mathcal{A} \) to satisfy \texttt{RETVAL}, \( \mathcal{A} \) should include one more relation that is more subtle. We illustrate the need for it using the example in Figure 4. Assume that we have the solid edges in the figure. If we arbitrate between the two appends as shown by the dashed edge \( f \xrightarrow{\text{vis}} e \), then according to the construction in Lemma 5 we will also have the dashed edge \( f \xrightarrow{\text{vis}} g \) (needed for \( \mathcal{A} \) to satisfy \texttt{ObservedVis}). But then the resulting \( \mathcal{A} \) will violate \texttt{RETVAL}. We therefore include the following relation into \( \mathcal{A} \), which ensures that such situations do not happen:

\[
  e \prec f \iff \exists g. \text{obj}(f) = \text{obj}(g) \land (f, g) \not\in \text{vis}_0 \land \\
  (e, g) \in ((\text{vis}_0 \setminus \text{so}) \cup (\text{rt} \cap (\text{Event} \times \text{EPush}))) \cup (\text{so} \cup (\text{rt} \cap (\text{EPush} \times \text{EPull})) : \text{so}_0 ?) ?.
\]

If \( e \prec f \), then adding an edge \( f \xrightarrow{\text{vis}} e \) would create a visibility edge \( f \xrightarrow{\text{vis}} g \) between events on the same object that is not in \( \text{vis}_0 \). Note that the expression covering \((e, g)\) above is more specific than the one in Lemma 5: we have \( \text{so}_0 \) instead of \( \text{so} \), and \( \text{rt} \) must be used. This is crucial for the proof (specifically, Lemma 6 below) and, as we show, is still sufficient to validate \texttt{RETVAL} because the history \( \mathcal{H} \) is well-fenced.

Thus, we need to construct an \( \mathcal{A} \) that includes \( \mathcal{A} \triangleq \mathcal{A} \cup \text{so} \cup \text{ar}_0 \cup ((\text{vis}_0 \setminus \text{so}) : \text{rt}) \cup \prec \). For this to be possible, \( \mathcal{A} \) has to be acyclic.

Lemma 6. \( \mathcal{A} \cup \text{so} \cup \text{ar}_0 \cup ((\text{vis}_0 \setminus \text{so}) ; \text{rt}) \cup \prec \) 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 \( \mathcal{A} \).

Lemma 7.

\[
  (\mathcal{A} \cup \text{so} \cup \text{ar}_0 \cup ((\text{vis}_0 \setminus \text{so}) ; \text{rt}) \cup \prec)^+ \\
  = (\mathcal{A} \cup \text{so} \cup \text{ar}_0 \cup ((\text{vis}_0 \setminus \text{so}) ; \text{rt}))^+ \cup (\mathcal{A} \cup \text{so} \cup \text{ar}_0 \cup ((\text{vis}_0 \setminus \text{so}) ; \text{rt}))^+ \cup \\
  (\mathcal{A} \cup \text{so} \cup \text{ar}_0 \cup ((\text{vis}_0 \setminus \text{so}) ; \text{rt}))^+ \\
  \subseteq \mathcal{A} \cup \text{ar}_0 \cup \text{ar}_0 \cup (\text{ar}_0 \cup \text{ar}_0 \cup (\text{vis}_0 \setminus \text{so}) ; \text{rt}) \cup \\
  (\text{ar}_0 \cup (\text{vis}_0 \setminus \text{so}) ; \text{rt}) \cup (\text{ar}_0 \cup (\text{vis}_0 \setminus \text{so}) ; \text{rt}) \cup \text{ar}_0 \cup (\text{vis}_0 \setminus \text{so}) ; \text{rt}) \cup \text{ar}_0.
\]

The proof Lemma 7 relies on establishing that components of \( \mathcal{A} \) 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 \( \mathcal{A}^+ \) in Lemma 7 is irreflexive. This relation describes only particular paths in \( \mathcal{A} \) 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 \( vis \subseteq ar \), so that \( A = (H, vis, ar) \) is an abstract execution. By Lemma 5, \( A \) satisfies \( RYW-PUSHEDVis \). It satisfies \( PUSHEDAR \) because \( \Pi \subseteq ar \), and it is also easy to check that it satisfies \( OBSERVEDAR \).

We next argue that \( A \) satisfies \( RETVAL \), which exploits the particular way in which we constructed \( ar \). To this end, we show that for any object \( x \) we have \( vis_x \subseteq vis_x \), where \( vis_x \) is the projection of \( vis \) to events on \( x \). Then since for any \( x \) we have \( ar_x \subseteq ar \) and \( A_x \) satisfies \( RETVAL \), so does \( A \). Since \( vis_x \subseteq vis \) by construction, we only need to show \( vis_x \subseteq vis_x \).

Consider arbitrary \( f, g \in E \) such that \( \text{obj}(f) = \text{obj}(g) = x \) and \( f \xrightarrow{vis} g \). To show \( f \xrightarrow{vis} g \) 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: \((f, g) \in ar? ; (vis_0 \setminus so) ; (rt \cap (\text{Event} \times \text{EPull})) ; so? \). Then for some \( g' \) we have

\[
f \xrightarrow{ar? ; (vis_0 \setminus so) ; (rt \cap (\text{Event} \times \text{EPull}))} g' \xrightarrow{so?} g.
\]

Figure 4 illustrates the case when \( g' = g \). If \( \text{obj}(g') \neq \text{obj}(g) \), then since the history \( H \) is well-fenced, for some \( g'' \in \text{EPull} \) we have \( g' \xrightarrow{so} g'' \xrightarrow{so?} g \). Since \( so \subseteq rt \), this implies \( g' \xrightarrow{rt \cap (\text{Event} \times \text{EPull})} g'' \xrightarrow{so?} g \). Hence,

\[
f \xrightarrow{ar? ; (vis_0 \setminus so) ; (rt \cap (\text{Event} \times \text{EPull}))} g'' \xrightarrow{so?} g.
\]

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

\[
f \xrightarrow{ar? ; e} \xrightarrow{vis_0 ; so?} g.
\]

Now if \( \neg(f \xrightarrow{vis_0} g) \), then \( e \prec f \), contradicting \( \prec \subseteq ar \). Hence, \( f \xrightarrow{vis_0} g \), as required.

Thus, \( A \) satisfies all GSC axioms except for possibly \( EVENTUAL \). Since \( \forall x . vis_x \subseteq vis_x \) and \( A_x \) satisfies \( EVENTUAL \), we have

\[
\forall e \in E . \{\{ f \in E \mid \text{obj}(e) = \text{obj}(f) \land \neg(e \xrightarrow{vis} f)\}\} < \infty,
\]

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.

\begin{lemma}
Let \( H = (E, so, rt) \) and \( A = (H, vis, ar) \) be an execution that satisfies all GSC axioms except for possibly \( EVENTUAL \). Assume (2) holds. Then there exists \( vis' \supseteq vis \) such that \((H, vis', ar) \in \text{ExecGSC} \).
\end{lemma}

We thus construct an execution \((H, vis', ar) \in \text{ExecGSC} \), which shows that \( H \in \text{HistGSC} \) 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, 23, 18]. 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 [23, 19] and distributed storage systems [9, 8]. 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 [23, 17]. 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 [12, 3].

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 [27, 4]. 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.
References

Consistency Models with Global Operation Sequencing and their Composition


