A Generic Specification Framework for Weakly Consistent Replicated Data Types

Xue Jiang, Hengfeng Wei, Yu Huang
State Key Laboratory for Novel Software Technology, Nanjing University, China
xuejiang1225@gmail.com, {hfwei, yuhuang}@nju.edu.cn

***Abstract*—Recently Burckhardt et al. proposed a formal specification framework for eventually consistent replicated data types, denoted (vis, ar), based on the notions of visibility and arbitration relations. However, being specific to eventually consistent systems, this framework has two limitations. First, it does not cover non-convergent consistency models since arbitration ar is defined to be a total order over events in a computation. Second, it does not cover the consistency models in which each event is required to be aware of the return values of some or all events that are visible to it.**

In this paper, we extend the (vis, ar) specification framework into a more generic one called (vis, ar, V) for weakly consistent replicated data types. To specify non-convergent consistency models as well, we simply relax the arbitration relation ar to be a partial order. To overcome the second limitation, we allow to specify for each event e, a subset $V(e)$ of its visible set whose return values cannot be ignored when justifying the return value of e. To make it practically feasible, we provide candidates for the visibility and arbitration relations and the V function. By combining these candidates, we demonstrate how to specify various existing consistency models in the (vis, ar, V) framework. Moreover, it helps to discover new consistency models. As a case study, we prove that the causal consistency protocol of MongoDB database satisfies Causal Memory Convergence, a new causal consistency variant discovered in our framework.

***Keywords*—Replicated Data Types; Specification Framework; Weak Consistency Models; Causal Consistency; MongoDB**

I. Introduction

Geographically distributed systems often replicate data at multiple sites to achieve high availability and low latency, even under network partitions [1], [2]. According to the CAP theorem [3], [4] and the PACELC tradeoff [5], these systems often sacrifice strong consistency and choose to implement *weakly consistent replicated data types*.

Eventual consistency is one of the most widely used weak consistency models in distributed systems [6], [7]. It guarantees that "if clients stop issuing update requests, the replicas will eventually reach a consistent state." [7]. For example, to allow replicas to respond to user operations immediately, collaborative text editing systems [8]–[10] usually implement an *eventually consistent* replicated list object modelling the shared document. It requires the final lists at all replicas to be identical after executing the same set of user operations [8]. In principle, eventual consistency has two aspects:

1) At the *strong* aspect, it requires eventual *convergence* among replicas and thus the clients [1] will eventually obtain the same view of replica states.
2) At the *weak* aspect, it imposes no restrictions on the intermediate states and thus the clients may obtain (temporarily) inconsistent views of replica states.

[1]Following [2], we use clients, sessions, and processes interchangeably. We also use program order and session order interchangeably.

Burckhardt et al. [1], [2] proposed a formal specification framework for eventually consistent replicated data types. It is based on the *visibility* (denoted vis) and *arbitration* (denoted ar) relations over events of a history, and we call it the (vis, ar) framework.

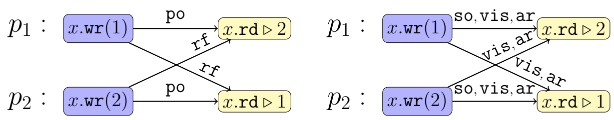
- The visibility relation is an acyclic relation that accounts for the relative timing of events. Intuitively, if an event e_1 is visible to another event e_2, it means that the effect of e_1 is visible to the client performing e_2 before e_2 is invoked. For example, e_2 may be a query returning the value written by update e_1 [11]. We call two events *concurrent* if they are invisible to each other.
- The arbitration relation is a *total order* over all events of a history. It indicates how the system resolves the conflicts due to concurrent events. For example, such a total order can be achieved using distributed timestamps [12].

Being specific to eventually consistent systems, the (vis, ar) framework is not general enough to cover some important weak consistency models such as PRAM [13] and Causal Memory [14]. Specifically, we identify two limitations of the (vis, ar) framework as follows, corresponding to the two aspects of eventual consistency mentioned above:

1) *Convergence vs. Non-Convergence.* The total ordering requirement of ar over all events aims to ensure eventual *convergence* of the clients' views of replica states. Therefore, the (vis, ar) framework does not cover the consistency models that do not enforce convergence.
2) *Awareness vs. Unawareness.* In the (vis, ar) framework, the return value of an event e is justified by the set $vis^{-1}(e)$ of events visible to e (arranged in the arbitration order `ar`), while *ignoring* all their return values. Therefore, the (vis, ar) framework does not cover the consistency models in which an event is required to be *aware* of the return values of some or all events that are visible it.

We illustrate both limitations with Causal Memory (*CM*) [14], [15] with respect to read/write registers, and motivate our first contribution of this paper which is a generalization of the (vis, ar) framework. Intuitively, *CM* ensures that processes agree on the relative ordering of operations that are *causally*

DOI 10.1109/SRDS51746.2020.00022



(a) History augmented with rf. (b) Abstract execution.

Fig. 1. Motivating history for convergence and non-convergence: It satisfies *CM*, but it is non-convergent. We denote the operation of writing value v to x by $x.\mathtt{wr}(v) \triangleright \bot$ (where $\bot$ indicates that it returns no value and is omitted in figures) and the operation of reading value v from x by $x.\mathtt{rd} \triangleright v$. We will also write $x.\mathtt{rd} \triangleright \_$ to emphasize that the return value has been ignored. Here so denotes session order, formally defined in Definition 3.

related [14]. The *causality order* over operations is defined as the transitive closure of the union of program order po and the read-from relation rf which informally associates each read with a unique write from which it reads the value. A history satisfies *CM* (w.r.t. read/write registers) if for each process, the set of all operations on this process and all write operations on other processes can be arranged into an operation sequence which preserves the causality order such that each read reads the value from the most recently preceding write in this sequence on the same register.

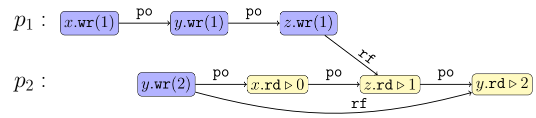
Example 1 (Convergence *vs.* Non-Convergence)**.** Consider the history in Fig. 1(a) consisting of two processes p_1 and p_2 which read from and write to a shared register x. This history satisfies *CM*: the witness operation sequences for p_1 and p_2 are $\langle x.\mathtt{wr}(1)\ x.\mathtt{wr}(2)\ x.\mathtt{rd}{\triangleright}2\rangle$ and $\langle x.\mathtt{wr}(2)\ x.\mathtt{wr}(1)\ x.\mathtt{rd}{\triangleright}1\rangle$, respectively. However, it is non-convergent: processes p_1 and p_2 cannot agree with a total order ar over $x.\mathtt{wr}(1)$ and $x.\mathtt{wr}(2)$. Particularly, it does *not* satisfy the convergent variant of causal consistency called *WCCv* [2], [15] defined in the (vis, ar) framework; see Fig. 1(b) and Definition 17.

To specify non-convergent consistency models as well, we can simply relax the arbitration relation ar to be a partial order.

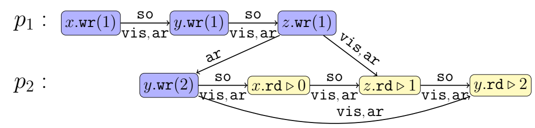
Example 2 (Awareness *vs.* Unawareness)**.** Consider the history in Fig. 2(a). It does not satisfy *CM*. In any operation sequence for process p_2, $x.\mathtt{wr}(1)$ must be placed after $x.\mathtt{rd}{\triangleright}0$ and $z.\mathtt{wr}(1)$ must be placed before $z.\mathtt{rd}{\triangleright}1$. As a consequence, $y.\mathtt{wr}(1)$ must be placed between $y.\mathtt{wr}(2)$ and $y.\mathtt{rd} \triangleright 2$. However, such sequences cannot be valid; in particular, the return value of $y.\mathtt{rd} \triangleright 2$ is not justified.

However, if the *return values* of the operations that causally precede $y.\mathtt{rd}{\triangleright}2$ can be ignored, $y.\mathtt{rd}{\triangleright}2$ can be justified by the operation sequence $\langle x.\mathtt{wr}(1)\ y.\mathtt{wr}(1)\ z.\mathtt{wr}(1)\ y.\mathtt{wr}(2)\ x.\mathtt{rd} \triangleright \_\ \ z.\mathtt{rd} \triangleright \_\ \ y.\mathtt{rd} \triangleright 2\rangle$. Similarly, $z.\mathtt{rd} \triangleright 1$ and $x.\mathtt{rd} \triangleright 0$ (0 is the initial value of x) can be justified by $\langle x.\mathtt{wr}(1)\ y.\mathtt{wr}(1)\ z.\mathtt{wr}(1)\ y.\mathtt{wr}(2)\ x.\mathtt{rd} \triangleright \_\ \ z.\mathtt{rd} \triangleright 1\rangle$ and $\langle y.\mathtt{wr}(2)\ x.\mathtt{rd}{\triangleright}0\rangle$, respectively. Actually, this history satisfies *WCCv* [2], [15] defined in the (vis, ar) framework in which return values are ignored; see Fig. 2(b) and Definition 17.

To allow an event e to be aware of the return values of some or all events in $\mathtt{vis}^{-1}(e)$, we introduce a function V defined on events. Specifically, $V(e)$ for event e is a subset



(a) History augmented with rf.



(b) Abstract execution.

Fig. 2. Motivating history for awareness and unawareness. It violates *CM*. The arrows for relations (e.g., po, so, vis, and ar in this example) implied by transitivity are not drawn.

of $\mathtt{vis}^{-1}(e)$ whose return values cannot be ignored when justifying the return value of e.

Our First Contribution. In this paper, we extend the (vis, ar) specification framework for eventually consistent replicated data types into a generic one called (vis, ar, V) for weakly consistent replicated data types. On the one hand, by relaxing ar to be a partial order, the (vis, ar, V) framework is able to cover non-convergent consistency models such as PRAM [13] and Causal Memory [14]. On the other hand, by introducing the V function for events, the (vis, ar, V) framework is able to cover the consistency models in which each event is required to be aware of the return values of some or all events that are visible to it. To make it practically feasible, we provide common candidates for the three components of this generic framework, including the vis and ar relations and the V function.

Our Second Contribution. By combining candidates for each component, we are able to specify various existing consistency models in the (vis, ar, V) framework. Moreover, it helps to discover new consistency models. To demonstrate the usefulness of these new consistency models, we prove that the causal consistency protocol of MongoDB database satisfies *CMv* (Causal Memory Convergence), a new variant of causal consistency discovered in our framework.

Outline. Section II reviews the (vis, ar) specification framework for eventually consistent replicated data types. Section III presents our generic (vis, ar, V) specification framework for weakly consistent replicated data types, and provides recipes for it. Section III demonstrates how consistency models are specified in this framework, taking causal consistency variants as examples. Section V shows that the causal consistency protocol of MongoDB satisfies *CMv*. Section VI discusses related work and Section VII concludes.

II. Preliminaries

In this section, we review the formal specification framework called (vis, ar) for eventually consistent replicated data types proposed by Burckhardt et al. [1], [2].

A. Relations and Orderings

A binary relation R over a given set A is a subset of $A \times A$, i.e., $R \subseteq A \times A$. For $a, b \in A$, we use $(a, b) \in R$ and $a \xrightarrow{R} b$ interchangeably. The inverse relation of R is denoted by R^{-1}, i.e., $(a, b) \in R \iff (b, a) \in R^{-1}$. We use $R^{-1}(b)$ to denote the set $\{a \in A \mid (a, b) \in R\}$.

Given two binary relations R and S over set A, we define the composition of them as $R; S = \{(a, c) \mid \exists b \in A : a \xrightarrow{R} b \xrightarrow{S} c\}$. For $n \in \mathbb{N}^+$, R^n denotes the n-ary composition $R; R; \ldots; R$. The transitive closure of R is $R^+ \triangleq \bigcup_{n \geq 1} R^n$. For some subset $A' \subseteq A$, the restriction of R to A' is $R|_{A'} \triangleq R \cap (A' \times A)$. If $f : A \to B$ is a function (also a relation) from A to B, the restriction of f to $A' \subseteq A$ is $f|_{A'} \triangleq f \cap (A' \times B) = \{(a, f(a)) \mid a \in A'\}$.

A relation R is *natural* if $\forall x \in A : |R^{-1}(x)| < \infty$. A (strict) partial order is an irreflexive and transitive relation. A total order is a relation which is a partial order and total. For $A' \subseteq A$, $to(R, A')$ asserts that R is a total order over A'.

B. Abstract Data Types

We consider a replicated database storing *objects* of some abstract data types.

Definition 1 (Abstract Data Types)**.** An abstract data type $\tau \in \mathit{Type}$ is a pair $\tau = (\mathit{Op}, \mathit{Val})$ such that

- Op is the set of *operations* supported by τ;
- Val is the set of values allowed by τ. We assume $\perp \in \mathit{Val}$ to indicate that some operations may return no value.

Definition 2 (Sequential Semantics)**.** The *sequential semantics* of a type $\tau \in \mathit{Type}$ is defined by a function $\mathtt{eval}_\tau : \mathit{Op}^* \times \mathit{Op} \to \mathit{Val}$ that, given a sequence of operations S and an operation o, determines the return value $\mathtt{eval}_\tau(S, o) \in \mathit{Val}$ for o when o is performed after S.

In this paper, we use sequential semantics for illustration. In general, semantics of types can be defined so that the return value of an operation is determined by a graph (instead of a sequence) of prior operations, as in [1], [2]. Then, CRDTs (Conflict-free Replicated Data Types) [16] could be expressed.

Example 3 (Register)**.** An integer read/write register reg supports two operations: $\mathtt{wr}(v)$ writes value $v \in \mathbb{Z}$ to the register and $\mathtt{rd}$ reads the value from it. A $\mathtt{rd}$ operation returns the value of the last preceding $\mathtt{wr}$, or the initial value 0 if there are no prior writes. Formally, for any operation sequence S,

$$\begin{aligned}
&\mathtt{eval}_{\mathsf{reg}}(S, \mathtt{wr}(v)) = \perp, \\
&\quad \mathtt{eval}_{\mathsf{reg}}(S, \mathtt{rd}) = v, \text{ if } \mathtt{wr}(0)\ S = S_1\ \mathtt{wr}(v)\ S_2 \\
&\qquad\qquad \text{and } S_2 \text{ contains no } \mathtt{wr} \text{ operations.}
\end{aligned}$$

Example 4 (Key-value Store)**.** A key-value store kvs supports two operations: $\textsc{put}(k, v)$ writes value v to key k and $\textsc{get}(k)$ reads value (which may be the initial value 0) from key k. Formally, for any operation sequence S,

$$\begin{aligned}
&\mathtt{eval}_{\mathsf{kvs}}(S, \textsc{put}(k, v)) = \perp, \\
&\quad \mathtt{eval}_{\mathsf{kvs}}(S, \textsc{get}(k)) = v, \text{ if } \textsc{put}(k, 0)\ S = S_1\ \textsc{put}(k, v)\ S_2 \\
&\qquad\qquad \text{and } S_2 \text{ contains no } \textsc{put} \text{ operations on } k.
\end{aligned}$$

Example 3 (Queue)**.** An integer FIFO queue fq supports two operations: $\mathtt{enq}(v)$ adds value $v \in \mathbb{N}$ to the tail of the queue. $\mathtt{deq}$ removes and returns the element v at the head of the queue; if the queue is empty, we let $v = \perp$. Formally, for any operation sequence S,

$$\begin{aligned}
&\mathtt{eval}_{\mathsf{fq}}(S, \mathtt{enq}(v)) = \perp, \\
&\quad \mathtt{eval}_{\mathsf{fq}}(S, \mathtt{deq}) = v, \text{ if } S = S_1\ \mathtt{enq}(v)\ S_2 \\
&\qquad\qquad \text{and } \mathtt{eval}_{\mathsf{fq}}(S_1, \mathtt{deq}) = \perp \\
&\qquad\qquad \text{and } S_2 \text{ contains no } \mathtt{deq} \text{ operations.}
\end{aligned}$$

C. Histories

Clients interact with the replicated database by performing operations on objects. We use a *history* to record such interactions in a computation.

Definition 3 (Histories)**.** A *history* is a tuple $H = (E, \mathtt{op}, \mathtt{rval}, \mathtt{so})$ [2] such that

- E is the set of all *events* of operations invoked by clients in a single computation;
- $\mathtt{op} : E \to \mathit{Op}$ describes the operation of an event;
- $\mathtt{rval} : E \to \mathit{Val}$ describes the value returned by the operation $\mathtt{op}(e)$ of an event e;
- $\mathtt{so} \subseteq E \times E$ is a partial order over E, called the *session order*. It relates operations within a session in the order they were invoked by clients.

We lift $\mathtt{op}$ to sets of events by defining $\mathtt{op}(F) = \{\mathtt{op}(e) \mid e \in F\}$ for $F \subseteq E$. See Figs. 1(b) and 2(b) for examples of histories (for now ignore the relations $\mathtt{vis}$ and $\mathtt{ar}$).

D. Abstract Executions

To justify the return values of all events in a history, we need to know how these events are related to each other. Following [1], [2], this is captured declaratively by the *visibility* and *arbitration* relations.

Definition 4 (Abstract Executions)**.** An *abstract execution* is a triple $A = ((E, \mathtt{op}, \mathtt{rval}, \mathtt{so}), \mathtt{vis}, \mathtt{ar})$ such that

- $(E, \mathtt{op}, \mathtt{rval}, \mathtt{so})$ is a history;
- Visibility $\mathtt{vis} \subseteq E \times E$ is an acyclic and natural relation;
- Arbitration $\mathtt{ar} \subseteq E \times E$ is a total order.

Figs. 1(b) and 2(b) show examples of abstract executions. In both abstract executions, the visibility relations $\mathtt{vis}$ correspond to the read-from $\mathtt{rf}$ relations. The arbitration relation $z.\mathtt{wr}(1) \xrightarrow{\mathtt{ar}} y.\mathtt{wr}(2)$ in Fig. 2(b) enforces an order between these two concurrent operations.

[2]For simplicity, we do not record the *returns-before* relation over events, which captures the real-time ordering of non-overlapping operations [2].

E. Consistency Models

Definition 5 (Consistency Models)**.** A *consistency model* is a set of consistency predicates on abstract executions.

We write $A \models P$ if the consistency predicate P is true on the abstract execution A.

Definition 6 (Satisfication (Abstract Execution))**.** An abstract execution A *satisfies* consistency model $\mathcal{C} = \{P_1, \ldots, P_n\}$, denoted $A \models \mathcal{C}$, if each consistency predicate in $\mathcal{C}$ is true on A. Formally, $A \models \mathcal{C} \iff A \models P_1 \wedge \cdots \wedge A \models P_n$.

We are concerned with histories that satisfy some consistency model.

Definition 7 (Satisfication (History))**.** A history H *satisfies* consistency model $\mathcal{C} = \{P_1, \ldots, P_n\}$, denoted $H \models \mathcal{C}$, if it can be extended to an abstract execution that satisfies $\mathcal{C}$. Formally, $H \models \mathcal{C} \iff \exists\, \mathtt{vis}, \mathtt{ar}.\ (H, \mathtt{vis}, \mathtt{ar}) \models \mathcal{C}$.

F. Return Value Consistency

A common consistency predicate is the consistency of return values defined for a given data type τ. In an abstract execution A, the return value of an event e is determined by its *operation context*, denoted $\mathtt{ctxt}_A(e)$, which is the restriction of A to the set $\mathtt{vis}^{-1}(e)$ of events visible to e.

Definition 8 (Operation Context)**.** The *operation context* of an event $e \in E$ in an abstract execution $A = ((E, \mathtt{op}, \mathtt{rval}, \mathtt{so}), \mathtt{vis}, \mathtt{ar})$ is defined as

$$\mathtt{ctxt}_A(e) \triangleq A|_{\mathtt{vis}^{-1}(e), \mathtt{op}, \mathtt{vis}, \mathtt{ar}}.$$

Definition 9 (Return Value Consistency)**.** For a data type τ, its return value consistency predicate on an abstract execution A is $\text{RVAL}(\tau) \triangleq \forall e \in E.\ \mathtt{rval}(e) = \mathtt{eval}_\tau\big(\mathtt{ctxt}_A(e), \mathtt{op}(e)\big)$.

RVAL requires that the return value $\mathtt{rval}(e)$ of an event e should agree with the result computed by applying the sequential semantics to the operation sequence given by $\mathtt{ctxt}_A(e)$, which is obtained by arranging the events in $\mathtt{vis}^{-1}(e)$ according to $\mathtt{ar}$, and the operation $\mathtt{op}(e)$ [17].

Note that $\mathtt{ctxt}_A(e)$ ignores $\mathtt{rval}$ in the original history. Consequently, when justifying the return value of event e, it is *not* required for $\mathtt{eval}$ to be consistent with the return values of the events visible to e. As shown in Example 2, the abstract execution in Fig. 2(b) satisfies RVAL($\mathtt{reg}$).

III. A Generic Specification Framework for Weakly Consistent Replicated Data Types

In this section, we present our generic (vis, ar, V) specification framework for weakly consistent replicated data types. It not only covers more existing consistency models than the (vis, ar) framework does, but also helps to discover new ones. We first define the (vis, ar, V) framework in a general way. Then, we provide candidates for the visibility and arbitration relations and the V function.

A. The (vis, ar, V) Specification Framework

The (vis, ar, V) framework generalizes (vis, ar) in two ways, overcoming the two limitations of (vis, ar):

1) *Convergence vs. Non-Convergence.* The arbitration relation $\mathtt{ar}$ in (vis, ar, V) is a partial order over events. Therefore, (vis, ar, V) is able to cover classic non-convergent consistency models such as PRAM [13] and Causal Memory [14].
2) *Awareness vs. Unawareness.* In (vis, ar, V), we allow to specify for each event e, a subset $V(e)$ of $\mathtt{vis}^{-1}(e)$ whose return values cannot be ignored when justifying the return value of e. Therefore, (vis, ar, V) is able to cover the consistency models in which each event is required to be aware of the return values of some or all events that are visible to it.

To cover non-convergent consistency models, we reformulate the definition of abstract executions by relaxing $\mathtt{ar}$ to be a partial order.

Definition 10 (Abstract Executions in (vis, ar, V) Framework)**.** An *abstract execution* in the (vis, ar, V) framework is a triple $A = ((E, \mathtt{op}, \mathtt{rval}, \mathtt{so}), \mathtt{vis}, \mathtt{ar})$ such that

- $(E, \mathtt{op}, \mathtt{rval}, \mathtt{so})$ is a history;
- Visibility $\mathtt{vis} \subseteq E \times E$ is an acyclic and natural relation;
- Arbitration $\mathtt{ar} \subseteq E \times E$ is a partial order.

To be awareness of return values is a bit more involved. We first reformulate operation context by selectively unhiding the return values of some or all visible events.

Definition 11 (Operation Context in (vis, ar, V) Framework)**.** Let $A = ((E, \mathtt{op}, \mathtt{rval}, \mathtt{so}), \mathtt{vis}, \mathtt{ar})$ be an abstract execution in the (vis, ar, V) framework. The *operation context* of $e \in E$ in A is defined as

$$\mathtt{ctxt}_A(e, V) \triangleq A|_{\mathtt{vis}^{-1}(e), \mathtt{op}, \mathtt{rval}|_{V(e)}, \mathtt{vis}, \mathtt{ar}},$$

where $V : E \to 2^E$ specifies a subset of $\mathtt{vis}^{-1}(e)$. We let *Ctxt* be the set of all operation context, ranged over by $\mathbb{C}$. We use $V_{\mathbb{C}}$ to select the set $V(e)$ in $\mathbb{C}$.

Accordingly, the RVAL consistency predicate should be adapted to use $\mathtt{ctxt}_A(e, V)$.

Definition 12 (Return Value Consistency in (vis, ar, V) Framework)**.** For a data type τ, its return value consistency predicate on an abstract execution A is

$$\text{RVAL}(\tau, V) \triangleq \forall e \in E.\ \mathtt{rval}(e) \in \mathtt{eval}_\tau\big(\mathtt{ctxt}_A(e, V), \mathtt{op}(e)\big).$$

In the definition above, we need to redefine the sequential semantics of τ, i.e., the $\mathtt{eval}_\tau$ function. On the one hand, since $\mathtt{ar}$ is a partial order in $\mathtt{ctxt}_A(e, V)$, there may be a set of operation sequences over $\mathtt{vis}^{-1}(e)$ to evaluate. Thus, we regard an operation context as a *set of serializations*, which are linear extensions of $\mathtt{ar}$ over $\mathtt{vis}^{-1}(e)$. This is why we use '$\in$' instead of '$=$' in the definition of $\mathtt{ctxt}_A(e, V)$. On the other hand, besides justifying the return value $\mathtt{rval}(e)$ of event e, $\mathtt{eval}_\tau$ is required to preserve the unhidden return values specified by $\texttt{rval}|_{V(e)}$ in a given serialization obtained from $\texttt{ctxt}_A(e,V)$. Such serializations are considered *valid.*

Definition 13 (Sequential Semantics in (vis, ar, V) Framework)**.** The *sequential semantics* of a type $\tau \in \mathit{Type}$ is defined by a function $\texttt{eval}_\tau : \mathit{Ctxt} \times \mathit{Op} \to 2^{\mathit{Val}}$ that, given an operation context $\mathbb{C} \in \mathit{Ctxt}$ and an operation $o \in \mathit{Op}$, determines the possible return values $\texttt{eval}_\tau(\mathbb{C}, o) \subseteq \mathit{Val}$ for o when o is performed in the context $\mathbb{C}$. Specifically, $\texttt{eval}_\tau(\mathbb{C}, o)$ is computed as follows [3]:

$$\begin{aligned}\texttt{eval}_\tau(\mathbb{C}, o) = \big\{ & v \in \mathit{Val} \mid \exists S \in \mathbb{C} : \\ & \big(\texttt{eval}_\tau(\texttt{op}(S), o) = v \wedge \\ & \forall e \in V_{\mathbb{C}} : \texttt{rval}(e) = \texttt{eval}_\tau\big(\texttt{op}(S_{\prec e}), \texttt{op}(e)\big)\big)\big\},\end{aligned}$$

where $S_{\prec e}$ is the prefix of S before e. Given a serialization $S \in \mathbb{C}$, the first conjunction computes the return value for o when o is performed after operation sequence $\texttt{op}(S)$, and the second one ensures that S is valid by checking the unhidden return values w.r.t. corresponding prefixes of S.

Example 4 (Convergence *vs.* Non-Convergence)**.** We argue that the history in Fig. 1 satisfies *CM* in the (vis, ar, V) framework. (*CM* is formally defined in Section IV.) In this example, we choose an $\texttt{ar}$ which does not arbitrate between $x.\texttt{wr}(1)$ and $x.\texttt{wr}(2)$. It is easy to check that the resulting abstract execution in Fig. 1(b) satisfies RVAL(reg).

Example 5 (Awareness *vs.* Unawareness)**.** We argue that the history in Fig. 2 does not satisfy *CM* in the (vis, ar, V) framework. Consider the abstract execution in Fig. 2(b), obtained by augmenting the history with $\texttt{vis}$ and $\texttt{ar}$ as in Example 2. Besides $\texttt{vis} \subseteq \texttt{ar}$, *CM* requires $\texttt{eval}_{\texttt{reg}}$ to preserve the unhidden return values of the visible events in $V(e) = \texttt{so}^{-1}(e)$. By similar argument to that in Example 2, there is no way of justifying $y.\texttt{rd} \triangleright 2$ while preserving the return values of $x.\texttt{rd} \triangleright 0$ and $z.\texttt{rd} \triangleright 1$.

B. Recipes for the (vis, ar, V) Specification Framework

The (vis, ar, V) framework is parametric w.r.t. three components, i.e., the vis and ar relations and the V function. By combining different consistency predicates on them, we can specify various consistency models in this framework. To make it practically feasible, we provide a number of candidates for each component, as summarized in Table I.

1) Recipe for the Visibility Relation: We identify a number of common consistency predicates on $\texttt{vis}$ in roughly the order they induce larger and larger visible sets consisting of the events visible to some event.

- The weakest one does not enforce an event to observe any particular set of events, not even the events previously performed on the same session. Formally, $\emptyset \subseteq \texttt{vis}$.
- The most basic ingredient for visibility is the session order $\texttt{so}$. The consistency predicate $\texttt{so} \subseteq \texttt{vis}$ requires each event see all the previous events *in the same session.*

[3] Note that we overload $\texttt{eval}_\tau$ with different type signatures.

TABLE I
RECIPES FOR $\texttt{vis}$, $\texttt{ar}$, AND V IN (vis, ar, V) FRAMEWORK.

vis	$\emptyset \subseteq \texttt{vis}$
	$\texttt{so} \subseteq \texttt{vis}$
	$\texttt{vis};\texttt{so} \subseteq \texttt{vis}$
	$\texttt{so};\texttt{vis} \subseteq \texttt{vis}$
	$\texttt{so};\texttt{vis};\texttt{so} \subseteq \texttt{vis}$
	$\texttt{vis};\texttt{so};\texttt{vis} \subseteq \texttt{vis}$
	$(\texttt{so} \cup \texttt{vis})^+ \subseteq \texttt{vis}$
ar	$\emptyset \subseteq \texttt{ar}$
	$\texttt{so} \subseteq \texttt{ar}$
	$\texttt{vis} \subseteq \texttt{ar}$
	$\texttt{vis};\texttt{so} \subseteq \texttt{ar}$
	$\mathit{to}(\texttt{ar}, E)$
$V(e)$	$V(e) = \emptyset$
	$V(e) = \texttt{so}^{-1}(e) \cap \texttt{vis}^{-1}(e)$
	$V(e) = \texttt{vis}^{-1}(e)$

- To allow an event to see the events *in different sessions* as well, it is necessary to compose $\texttt{so}$ with $\texttt{vis}$ in some ways. There are four basic kinds of compositions, namely $\texttt{vis};\texttt{so} \subseteq \texttt{vis}$, $\texttt{so};\texttt{vis} \subseteq \texttt{vis}$, $\texttt{so};\texttt{vis};\texttt{so} \subseteq \texttt{vis}$, and $\texttt{vis};\texttt{so};\texttt{vis} \subseteq \texttt{vis}$.
- To further allow an event to see the events in different sessions through an arbitrarily long chain of compositions of $\texttt{vis}$ and $\texttt{so}$, we rely on the transitive closure over $\texttt{vis}$ and $\texttt{so}$. That is, we have $(\texttt{so} \cup \texttt{vis})^+ \subseteq \texttt{vis}$, where $\texttt{hb} \triangleq (\texttt{so} \cup \texttt{vis})^+$ is the well-known *happens-before order* [1], [2] first proposed by Lamport [12]. Note that $\texttt{hb} \subseteq \texttt{vis}$ implies that $\texttt{vis}$ is transitive.

2) Recipe for the Arbitration Relation: When justifying the return value of event e with respect to the sequential semantics $\texttt{eval}$, we rely on the arbitration relation $\texttt{ar}$ to resolve conflicts caused by concurrent events in the set $\texttt{vis}^{-1}(e)$ of events visible to e. In the (vis, ar, V) framework, $\texttt{ar}$ is a partial order over events. In the following, we identify a number of common consistency predicate on $\texttt{ar}$ in roughly the order they are able to resolve more and more conflicts.

- The weakest one does not impose any constraints on how the conflicts should be resolved. Formally, $\emptyset \subseteq \texttt{ar}$.
- A slightly stronger arbitration orders the events in $\texttt{vis}^{-1}(e)$ (for some event e) according to the session order [1]. Formally, $\texttt{so} \subseteq \texttt{ar}$. Note that $\texttt{so}$ may be a *proper* subset of $\texttt{vis}$.
- To resolve all conflicts in $\texttt{vis}^{-1}(e)$, we need $\texttt{vis} \subseteq \texttt{ar}$.
- $(\texttt{vis};\texttt{so}) \subseteq \texttt{ar}$ orders an event after other ones previously observed in the same session [1].
- Finally, convergent consistency models often require $\texttt{ar}$ to be a total order over E, denoted $\mathit{to}(\texttt{ar}, E)$, as in the (vis, ar) framework [1], [2].

3) Recipe for the V Function: By definition, $V(e)$ is a subset of $\texttt{vis}^{-1}(e)$. We identify three common consistency predicates on $V(e)$:

- The strongest one requires e to be aware of the return values of *all* events visible to it. Formally, $V(e) = \mathtt{vis}^{-1}(e)$;
- Consistency models like Causal Memory [14] allow e to ignore the return values of its visible events *in different sessions* while being aware of those in its own session. Formally, $V(e) = \mathtt{so}^{-1}(e) \cap \mathtt{vis}^{-1}(e)$;
- The weakest one allows e to ignore the return values of any visible events, as in the (vis, ar) framework. Formally, $V(e) = \emptyset$.

IV. Consistency Models

By combining different consistency predicates for each component, we can specify various existing consistency models in the (vis, ar, V) framework. Moreover, it helps to discover new consistency models. In this section, we demonstrate how various consistency models are specified in this framework. Due to space limit, we take causal consistency variants as examples.

A. Causal Consistency: Overview

Causal consistency is one of the most widely used weak consistency models in distributed systems [14], [19]. The key notion is the *happens-before* order $\mathtt{hb}$ over events [2], [12]. Intuitively, causal consistency ensures that if an event e_1 happens before event e_2, then all sessions must observe e_1 before e_2. However, concurrent events may be observed in different orders by different sessions.

In the literature, there are several causal consistency variants with subtle differences [2], [15], [18]. In this section, we consider six variants that we call Weak Causal Consistency (*WCC*), Weak Causal Convergence (*WCCv*), Causal Memory (*CM*), Causal Memory Convergence (*CMv*), Strong Causal Consistency (*SCC*), and Strong Causal Convergence (*SCCv*), as defined in Table II. It is worthwhile to note that these variants may have different names in related work, as summarized in Table II. This table also highlights that *CMv*, *SCC*, and *SCCv* are new variants discovered in our framework.

In terms of visibility, all causal consistency variants require $\mathtt{hb} = (\mathtt{so} \cup \mathtt{vis})^{+} \subseteq \mathtt{vis}$ to capture the happens-before order over events. In terms of arbitration, they all require $\mathtt{vis} \subseteq \mathtt{ar}$ to enforce the happens-before order over events in $\mathtt{vis}^{-1}(e)$ when justifying the return value of the event e in its operation context. However, they may differ in two aspects:

- How large is the function V for specifying the subset of visible events whose return values must be respected? Note that in causal consistency models, $\mathtt{so}^{-1}(e) \subseteq \mathtt{vis}^{-1}(e)$ for any event e (since $\mathtt{so} \subseteq \mathtt{vis}$). Thus, the candidate $V(e) = \mathtt{so}^{-1}(e) \cap \mathtt{vis}^{-1}(e)$ is equivalent to $V(e) = \mathtt{so}^{-1}(e)$.
- How strong is the arbitration relation $\mathtt{ar}$ for resolving conflicts? We distinguish between two cases according to whether $\mathtt{ar}$ is a total order or not, given $\mathtt{vis} \subseteq \mathtt{ar}$.

Fig. 3 gives examples on objects of FIFO queues.

Remark. To exclude the trivial implementations in which replicas update its own local objects without communicating with others at all (i.e., $\mathtt{vis} = \mathtt{so}$), we may *additionally* require these variants to satisfy the *eventual visibility* (*EV*) predicate [2], [15]. *EV* guarantees that an event can be invisible to at most finitely many other events.

B. Weak Causal Consistency

Weak causal consistency (*WCC*), recently proposed in [15], is the weakest variant of causal consistency we consider. Informally speaking, an abstract execution satisfies *WCC* as long as the return value $\mathtt{rval}(e)$ of each event e can be justified by some serialization of its visible set $\mathtt{vis}^{-1}(e)$, while ignoring their return values. More specifically, in terms of arbitration, *WCC* makes no extra restrictions on convergence and thus concurrent events may be observed in different orders by different sessions. In terms of $V(e)$, *WCC* allows each event e to ignore all the return values of its visible events.

Definition 14 (Weak Causal Consistency)**.**

$$WCC \triangleq (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar}) \land (V(e) = \emptyset) \land \text{RVAL}.$$

Example 6 (Weak Causal Consistency)**.** The history of Fig. 3(a) does not satisfy *WCC*. Intuitively, event $q.\mathtt{enq}(2)$ (resp. $p.\mathtt{enq}(2)$) must be visible to event $q.\mathtt{deq} \triangleright 2$ (resp. $p.\mathtt{deq} \triangleright 2$). By transitivity of $\mathtt{vis}$, event $p.\mathtt{enq}(1)$ is visible to event $p.\mathtt{enq}(2)$. Since $\mathtt{vis} \subseteq \mathtt{ar}$, it is impossible for p_3 to dequeue 2 before 1 from the FIFO queue p.

The history of Fig. 3(b) satisfy *WCC*. For example, the return value of $b : \mathtt{deq} \triangleright 2$ can be justified by the serialization $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ a : \mathtt{deq} \triangleright \_\ b : \mathtt{deq} \triangleright 2 \rangle$. [4] Note that such a justification is not required by *WCC* to be consistent with the return value of $a : \mathtt{deq} \triangleright 2$. Similarly, the return value of $b : \mathtt{deq} \triangleright 1$ can be justified by the serialization $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ a : \mathtt{deq} \triangleright \_\ b : \mathtt{deq} \triangleright 1 \rangle$.

C. Causal Memory

Causal memory (*CM*) was originally defined by Ahamad et al. [14] on read/write registers. Recently, Perrin et al. [15] extended it to arbitrary replicated data types. *CM* is stronger than *WCC* in that when justifying the return value $\mathtt{rval}(e)$ of each event e, *CM* takes into account not only the *operation invocations* of the set $\mathtt{vis}^{-1}(e)$ of events visible to e as in *WCC* but also *the return values* of the set $\mathtt{so}^{-1}(e)$ of events that precede e in the same session. In other words, compared to *WCC*, *CM* requires that each session is consistent with respect to the previous return values provided [18].

Definition 15 (Causal Memory)**.**

$$CM \triangleq (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar}) \land (V(e) = \mathtt{so}^{-1}(e)) \land \text{RVAL}.$$

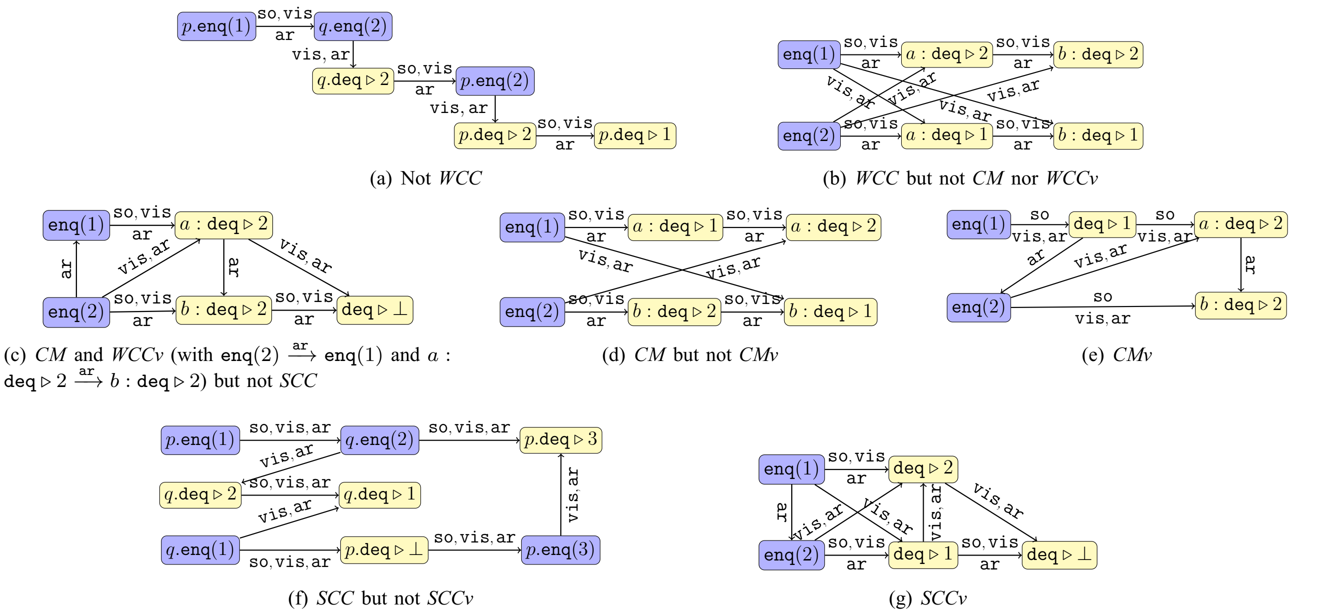
Example 7 (Causal Memory)**.** Although the history of Fig. 3(b) satisfies *WCC*, it does not satisfy *CM*. Specifically, being aware of the return value of $a : \mathtt{deq} \triangleright 2$, $b : \mathtt{deq} \triangleright 2$ is unjustifiable. That is, it is impossible to construct a valid serialization consisting of $\mathtt{enq}(2)$, $a : \mathtt{deq} \triangleright 2$, $\mathtt{enq}(1)$, and $b : \mathtt{deq} \triangleright 2$ subject to $\mathtt{enq}(1) \xrightarrow{\mathtt{vis}} a : \mathtt{deq} \triangleright 2 \xrightarrow{\mathtt{vis}} b : \mathtt{deq} \triangleright 2$.

[4]For clarity, we include the event (i.e., b) to be justified in the serialization.

TABLE II

CAUSAL CONSISTENCY VARIANTS SPECIFIED IN THE (vis, ar, V) SPECIFICATION FRAMEWORK. (NEW VARIANTS ARE MARKED WITH [$*$].)

Consistency Models		Alternative Names	vis	ar	$V(e)$
Causal Consistency Variants	*WCC* (Def. 8 [15])	CC (Def. 4.2 [18])	$\mathtt{hb} \subseteq \mathtt{vis}$	$\mathtt{vis} \subseteq \mathtt{ar}$	$V(e) = \emptyset$
	CM ([14], [18])	CC (Def. 9 [15])			$V(e) = \mathtt{so}^{-1}(e)$
	SCC	[$*$]			$V(e) = \mathtt{vis}^{-1}(e)$
	WCCv	CCv (Def. 4.5 [18], Def. 12 [15]) CAUSALCONSISTENCY (Def. 5.1 [2]) CAUSALITY (Def. 26 [11])		$\mathtt{vis} \subseteq \mathtt{ar} \wedge to(\mathtt{ar}, E)$	$V(e) = \emptyset$
	CMv	[$*$]			$V(e) = \mathtt{so}^{-1}(e)$
	SCCv	[$*$]			$V(e) = \mathtt{vis}^{-1}(e)$



(a) Not *WCC* (b) *WCC* but not *CM* nor *WCCv*

(c) *CM* and *WCCv* (with $\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$ and $a : \mathtt{deq} \triangleright 2 \xrightarrow{\mathtt{ar}} b : \mathtt{deq} \triangleright 2$) but not *SCC* (d) *CM* but not *CMv* (e) *CMv*

(f) *SCC* but not *SCCv* (g) *SCCv*

Fig. 3. Examples for causal consistency variants on objects of FIFO queue fq. Both p and q are of type fq in 3(a) and 3(f). The queue q in other subfigures is implicitly assumed. In each history, events are grouped into sessions which are horizontally laid out. The arrows for relations implied by transitivity are not drawn. We use labels, such as a and b, to make events unique.

The history of Fig. 3(d) satisfies *CM*. For example, the return values of $a : \mathtt{deq} \triangleright 2$ and $b : \mathtt{deq} \triangleright 1$ can be justified by serializations $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ a : \mathtt{deq} \triangleright 1\ a : \mathtt{deq} \triangleright 2 \rangle$ and $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ b : \mathtt{deq} \triangleright 2\ b : \mathtt{deq} \triangleright 1 \rangle$, respectively.

D. Strong Causal Consistency

We strengthen *CM* to *SCC* (Strong Causal Consistency) by further requiring each session to be consistent with respect to the return values provided by other sessions. Formally, we have $V(e) = \mathtt{vis}^{-1}(e)$.

Definition 16 (Strong Causal Consistency)**.**

$$SCC \triangleq (\mathtt{hb} \subseteq \mathtt{vis}) \wedge (\mathtt{vis} \subseteq \mathtt{ar}) \wedge (V(e) = \mathtt{vis}^{-1}(e)) \wedge \mathrm{RV{\small AL}}.$$

Example 8 (Strong Causal Consistency)**.** The history of Fig. 3(f) satisfies *SCC*. The return values of $p.\mathtt{deq} \triangleright \bot$, $q.\mathtt{deq} \triangleright 2$, $q.\mathtt{deq} \triangleright 1$, and $p.\mathtt{deq} \triangleright 3$ can be justified by the serializations $\langle q.\mathtt{enq}(1)\ p.\mathtt{deq} \triangleright \bot \rangle$, $\langle p.\mathtt{enq}(1)\ q.\mathtt{enq}(2)\ q.\mathtt{deq} \triangleright 2 \rangle$, $\langle p.\mathtt{enq}(1)\ \ q.\mathtt{enq}(2)\ \ q.\mathtt{deq} \triangleright 2\ \ q.\mathtt{enq}(1)\ \ q.\mathtt{deq} \triangleright 1 \rangle$, and $\langle q.\mathtt{enq}(1)\ p.\mathtt{deq} \triangleright \bot\ p.\mathtt{enq}(3)\ p.\mathtt{enq}(1)\ q.\mathtt{enq}(2)\ p.\mathtt{deq} \triangleright 3 \rangle$, respectively.

The history of Fig. 3(c) does not satisfy *SCC*. (For now, ignore $\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$ and $a \xrightarrow{\mathtt{ar}} b$ which are for *WCCv*.) Specifically, being aware of the return values of $a : \mathtt{deq} \triangleright 2$ and $b : \mathtt{deq} \triangleright 2$, $\mathtt{deq} \triangleright \bot$ is unjustifiable: it is impossible to construct a valid serialization consisting of all the events, since 2 is dequeued twice.

E. Weak Causal Convergence

WCCv (Weak Causal Convergence) [2], [15] is the convergent counterpart of *WCC*. It strengthens *WCC* by imposing a total order over all events in an execution, which provides all sessions with a uniform way of resolving conflicts caused by concurrent events. Consequently, the return value $\mathtt{rval}(e)$ of each event e is evaluated on the set $\mathtt{vis}^{-1}(e)$ of events visible to e, ordered by this common total order $\mathtt{ar}$, while ignoring all of their return values.

Definition 17 (Weak Causal Convergence)**.**

$$\begin{aligned} \textit{WCCv} \triangleq \quad & (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar} \land \textit{to}(\mathtt{ar}, E)) \\ & \land (V(e) = \emptyset) \land \text{RVal}. \end{aligned}$$

Example 9 (Weak Causal Convergence)**.** Although the history of Fig. 3(b) satisfies *WCC*, it does not satisfy *WCCv*. Specifically, the justification for the return value of $b : \mathtt{deq} \triangleright 2$ requires $\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} \mathtt{enq}(2)$, while the one for $b : \mathtt{deq} \triangleright 1$ requires $\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$.

The history of Fig. 3(c) satisfies *WCCv*. The serialization $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ a : \mathtt{deq} \triangleright 2 \rangle$ for justifying $a : \mathtt{deq} \triangleright 2$, the one $\langle \mathtt{enq}(2)\ b : \mathtt{deq} \triangleright 2 \rangle$ for $b : \mathtt{deq} \triangleright 2$, and the one $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ a : \mathtt{deq} \triangleright \_\ b : \mathtt{deq} \triangleright \_\ \mathtt{deq} \triangleright \perp \rangle$ for $\mathtt{deq} \triangleright \perp$ agree with a common total order $\mathtt{ar}$, e.g., $\langle \mathtt{enq}(2)\ \mathtt{enq}(1)\ a : \mathtt{deq} \triangleright 2\ b : \mathtt{deq} \triangleright 2\ \mathtt{deq} \triangleright \perp \rangle$.

F. Causal Memory Convergence

CMv (Causal Memory Convergence) is the convergent counterpart of *CM*, which requires $\mathtt{ar}$ to be a total order.

Definition 18 (Causal Memory Convergence)**.**

$$\begin{aligned} \textit{CMv} \triangleq \quad & (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar} \land \textit{to}(\mathtt{ar}, E)) \\ & \land (V(e) = \mathtt{so}^{-1}(e)) \land \text{RVal}. \end{aligned}$$

Example 10 (Causal Memory Convergence)**.** Although the history of Fig. 3(d) satisfies *CM*, it does not satisfy *CMv* which enforces a total order $\mathtt{ar}$ over all events. As shown in Example 7, the justification for the return value of $a : \mathtt{deq} \triangleright 2$ requires $\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} \mathtt{enq}(2)$, while the justification for the return value of $b : \mathtt{deq} \triangleright 1$ requires $\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} \mathtt{enq}(1)$.

The history of Fig. 3(e) satisfies *CMv*: the serialization $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1 \rangle$ for justifying the return value of $\mathtt{deq} \triangleright 1$, the one $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\ \mathtt{enq}(2)\ a : \mathtt{deq} \triangleright 2 \rangle$ for $a : \mathtt{deq} \triangleright 2$, and the one $\langle \mathtt{enq}(2)\ b : \mathtt{deq} \triangleright 2 \rangle$ for $b : \mathtt{deq} \triangleright 2$ agree with a common total order $\mathtt{ar}$, e.g., $\langle \mathtt{enq}(1)\ \mathtt{deq} \triangleright 1\ \mathtt{enq}(2)\ a : \mathtt{deq} \triangleright 2\ b : \mathtt{deq} \triangleright 2 \rangle$.

G. Strong Causal Convergence

SCCv (Strong Causal Convergence) is the convergent counterpart of *SCC*, which further requires $\mathtt{ar}$ to be a total order.

Definition 19 (Strong Causal Convergence)**.**

$$\begin{aligned} \textit{SCCv} \triangleq \quad & (\mathtt{hb} \subseteq \mathtt{vis}) \land (\mathtt{vis} \subseteq \mathtt{ar} \land \textit{to}(\mathtt{ar}, E)) \\ & \land (V(e) = \mathtt{vis}^{-1}(e)) \land \text{RVal}. \end{aligned}$$

Example 11 (Strong Causal Convergence)**.** Although the history of Fig. 3(f) satisfies *SCC*, it does not satisfy *SCCv*. Specifically, the justification for the return value of $q.\mathtt{deq} \triangleright 1$ requires $q.\mathtt{enq}(2) \xrightarrow{\mathtt{ar}} q.\mathtt{enq}(1)$, while the one for $p.\mathtt{deq} \triangleright 3$ requires $q.\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} q.\mathtt{enq}(2)$.

The history of Fig. 3(g) satisfies *SCCv*. For $\mathtt{deq} \triangleright \perp$ to return $\perp$, it must be aware of the event $\mathtt{deq} \triangleright 2$ and by transitivity all other events. It can be justified by the serialization $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright 1\ \mathtt{deq} \triangleright 2\ \mathtt{deq} \triangleright \perp \rangle$. So, with $\mathtt{deq} \triangleright 1 \xrightarrow{\mathtt{vis}} \mathtt{deq} \triangleright 2$ and $\mathtt{enq}(1) \xrightarrow{\mathtt{ar}} \mathtt{enq}(2)$, $\mathtt{deq} \triangleright 1$ and $\mathtt{deq} \triangleright 2$ can be justified by the serializations $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright 1 \rangle$ and $\langle \mathtt{enq}(1)\ \mathtt{enq}(2)\ \mathtt{deq} \triangleright 1\ \mathtt{deq} \triangleright 2 \rangle$, respectively.

TABLE III
NOTATIONS USED IN *MONGODB-CC*.

Notations	Description
$(sec, counter)$	hybrid logical clock (HLC)
ct_c	the greatest cluster time known by client c
ot_c	the last operation time at client c
ct_s	cluster time at server s
aot_s	the last applied operation time at server s
clock_s	current physical clock at server s
oplog_s	operation log at server s
CLUSTER(s)	the cluster that s belongs to
PRIMARY(s)	the primary node in CLUSTER(s)
ISPRIMARY(s)	whether s is the primary node in CLUSTER(s)
ISSECONDARY(s)	whether s is a secondary node in CLUSTER(s)
k	key
v	value
ct	cluster time
ot	operation time
aot	applied operation time
oplog	oplog

V. CASE STUDY: MONGODB

To demonstrate the usefulness of new consistency models discovered in our (vis, ar, V) framework, we prove that the causal consistency protocol of MongoDB [20] satisfies *CMv*.

MongoDB is a distributed database supporting data replication and sharding [20]. It is *document*-oriented, where a document is an ordered set of key-value pairs. Thus, for simplicity, we model MongoDB as a typical key-value store, which provides GET(k) and PUT(k, v) operations to clients. We focus on the causal consistency protocol, called *MongoDB-CC*, of MongoDB in the *failure-free sharded cluster* deployment, where each shard is replicated in a cluster consisting of a primary node and several secondary nodes. Only primary nodes can accept PUT operations from clients. Table III provides a summary of notations used in the protocol.

A. States

1) Logical Clocks and Cluster Time: MongoDB-CC uses hybrid logical clocks (HLC) [21]. An HLC is a pair $(sec, counter)$ of physical time (in seconds) and a counter to distinguish operations that occurred within the same second. Hybrid logical clocks are compared lexicographically (Algorithm 4).

The cluster time is the time value of a node's logical clock [20]. It ticks (Line 1 of Algorithm 4) only when a PUT operation is applied in the primary node of a cluster (Line 13 of Algorithm 2). Nodes maintain and distribute their greatest known cluster time via messages.

2) Client States: Each client c keeps track of the greatest known cluster time ct_c. It also maintains the timestamp of its last operation, denoted ot_c, capturing the client's causal dependencies.

3) Server States: Each server s keeps track of the greatest known cluster time ct_s. It uses an append-only operation log oplog_s to record the operations, as well as their timestamps,

Algorithm 1 Client operations at client c.

```
1: function GET(k)
2:     s ← a server storing key k
3:     ⟨v, ct, ot⟩ ← s.GET-REQUEST(k, ct_c, ot_c)
4:     ct_c ← max(ct_c, ct)
5:     ot_c ← ot        ▷ ts(GET) ← (ot_c, s), dt(GET) ← ot_c
6:     return v
7: function PUT(k, v)
8:     s ← the primary node storing key k
9:     ⟨ct, ot⟩ ← s.PUT-REQUEST(k, v, ct_c)
10:    ct_c ← ct
11:    ot_c ← ot        ▷ ts(PUT) ← (ot_c, s), dt(PUT) ← ot_c
12:    return ok
```

Algorithm 2 Server operations at server s.

```
    store_s[k] ← 0 for each key k                 ▷ Initialization
1: function GET-REQUEST(k, ct, ot)
2:     ct_s ← max(ct_s, ct)
3:     if aot_s < ot then
4:         PRIMARY(s).NOOP(ct_s, ot)              ▷ for liveness
5:         if ISSECONDARY(s) then
6:             repeat
7:                 s.REPLICATE()
8:             until aot_s ≥ ot
9:     v ← store_s[k]
10:    return ⟨v, ct_s, aot_s⟩
11: function PUT-REQUEST(k, v, ct)
12:    ct_s ← max(ct_s, ct)
13:    ct_s ← TICK()
14:    aot_s ← ct_s
15:    store_s[k] ← v
16:    oplog_s ← oplog_s ∘ ⟨k, v, aot_s⟩
17:    return ⟨ct_s, aot_s⟩                        ▷ ct_s = aot_s
```

Algorithm 3 Replication at server s.

```
1: function REPLICATE()                           ▷ Run periodically
2:     if ISSECONDARY(s) then
3:         ⟨oplog, ct⟩ ← PRIMARY(s).PULL-OPLOG(ct_s, aot_s)
4:         ct_s ← max(ct_s, ct)
5:         for ⟨k, v, ot⟩ ∈ oplog do              ▷ in ot order
6:             store_s[k] ← v
7:             aot_s ← ot
8:         oplog_s ← oplog_s ∘ oplog
9: function PULL-OPLOG(ct, aot)
10:    ct_s ← max(ct_s, ct)
11:    oplog ← oplog entries after aot in oplog_s
12:    return ⟨oplog, ct_s⟩
```

Algorithm 4 Clock management at server s.

Hybrid logical clocks are compared lexicographically:

$$hlc_1 = hlc_2 \iff (hlc_1.sec = hlc_2.sec) \land (hlc_1.counter = hlc_2.counter).$$

$$hlc_1 < hlc_2 \iff (hlc_1.sec < hlc_2.sec) \lor (hlc_1.sec = hlc_2.sec \land hlc_1.counter < hlc_2.counter).$$

```
1: function TICK()
2:     if ct_s.sec ≥ clock_s then
3:         return ⟨ct_s.sec, ct_s.counter + 1⟩
4:     else
5:         return ⟨clock_s, 0⟩
6: function NOOP(ct, ot)
7:     ct_s ← max(ct_s, ct)
8:     while aot_s < ot do
9:         ct_s ← TICK()
10:        aot_s ← ct_s
11:        oplog_s ← oplog_s ∘ ⟨NO-OP, aot_s⟩
```

applied at s. Additionally, it maintains in aot_s the timestamp of the last operation applied at s.

B. Protocol

Algorithms 1 and 2 show the core of *MongoDB-CC*, handling GET and PUT operations at the client and server side, respectively. The pseudocode for replication and clock management are shown in Algorithms 3 and 4, respectively.

1) GET(k): A client c sends a GET request, containing the key k, its greatest known cluster time ct_c, and its last operation time ot_c, to a server s which stores key k. The server s first updates its cluster time ct_s (Line 2). To guarantee causality, it then checks whether the causal dependencies specified by ot_c have been applied locally, by comparing its last applied operation time aot_s with $ot \leftarrow \mathsf{ot}_c$ (Line 3). If $\mathsf{aot}_s < ot$ and s is a secondary node (Line 5), the server keeps replicating oplog from its primary node until $\mathsf{aot}_s \geq ot$ (Line 8). To ensure liveness, we allow the primary node to catch up by keeping applying NO-OP (Line 6 of Algorithm 4) in case $\mathsf{aot}_s < ot$ holds in the primary node (Line 4). (Note that if $\mathsf{aot}_s \geq ot$ and s is the primary node, Line 4 does nothing.) Once $\mathsf{aot}_s \geq ot$ holds, the server s retrieves the value v of key k in local store_s (Line 9). Finally, the value v, as well as ct_s and aot_s are returned to the client (Line 10). Upon receiving the reply, the client updates its ct_c and ot_c accordingly.

2) PUT(k, v): A client sends a PUT request, containing the key k, the value v, and its greatest known cluster time ct_c, to the server s which is the primary node storing key k. The server s first updates its cluster time ct_s (Line 12). Then it ticks its ct_s (Line 13) and advances its last applied operation time aot_s to the new ct_s (Line 14). After being applied in local store_s (Line 15), the PUT operation, as well as its operation time aot_s, is appended to oplog_s (Line 16). Finally, both ct_s and aot_s are returned to the client (Line 17). Upon receiving the reply, the client updates its ct_c and ot_c accordingly.

3) Replication: In a cluster, each secondary node s periodically pulls the oplog entries with greater operation time than aot_s from the primary node (Line 3 of Algorithm 3).

The retrieved entries in *oplog* are appended to local oplog_s (Line 8), and the operations in it are applied in local store_s in increasing order of their operation times (Line 5). In the end, aot_s refers to the operation time of the last entry in its current oplog_s (Line 7). Additionally, secondary nodes and the primary node also distribute and keep track of their greatest known cluster time during replication.

C. Correctness Proof

We prove that *MongoDB-CC* satisfies *CMv* (Definition 18) by showing that every history $H = (E, \mathtt{op}, \mathtt{rval}, \mathtt{so})$ of *MongoDB-CC* satisfies *CMv* with respect to kvs (Definition 4). The key is to extract appropriate visibility and arbitration relations from H such that $(H, \mathtt{vis}, \mathtt{ar}) \models \mathit{CMv}$.

In the following, we use G, P, P_k and G_k to denote the set of GET events, the set of PUT events, the set of PUT events on key k, and the set of GET events on key k, respectively. For a $\text{PUT}(k, v)$ event e, we write $e \triangleright \langle k, v, ot\rangle$ to emphasize that it is associated with an operation time on Line 16 of Algorithm 2. For a $\text{GET}(k)$ event e, we write $e \triangleright \langle k, v, ot\rangle$ to denote that it retrieves the value v with operation time ot (Line 9).

1) Timestamps: We first define timestamp $\mathtt{ts}(e)$ for each event e as follows.

Definition 20 (Timestamps)**.** For an event e issued by client c, $\mathtt{ts}(e) = (\mathsf{ot}_c, s)$, where ot_c is the last operation time of client c on Line 5 for GET and Line 11 for PUT of Algorithm 1, and s is the identifier of the server processing e (Lines 2 and 8).

Timestamps are compared lexicographically (we assume a total order over the set of identifiers of servers). For notational convenience, we further define the dependency time $\mathtt{dt}(e) \triangleq \mathtt{ts}(e).\mathsf{ot}_c$ for each event e. Note that for a PUT event e, $\mathtt{dt}(e)$ is its operation time assigned on Line 16 of Algorithm 2.

2) Visibility: The visibility relation $\mathtt{vis}$ is based on the following *read-from* relation $\mathtt{rf}$.

Definition 21 (Read-from Relation)**.** $(e, f) \in \mathtt{rf}$ if and only if $e = \text{PUT}(k, v) \triangleright \langle k, v, ot\rangle$ and $f = \text{GET}(k) \triangleright \langle k, v, ot\rangle$ for some key k.

Definition 22 (Visibility)**.** The visibility relation $\mathtt{vis}$ is defined to be the transitive closure of the union of session order $\mathtt{so}$ and read-from relation $\mathtt{rf}$. Formally, $\mathtt{vis} = (\mathtt{so} \cup \mathtt{rf})^+$.

By induction on the structure of $\mathtt{vis}$, we can show that $\mathtt{vis}$ is reflected in $\mathtt{dt}$. Formally,

Lemma 1. *Let e_1 and e_2 be two events of history H. We have*

$$e_1 \xrightarrow{\mathtt{vis}} e_2 \implies \mathtt{dt}(e_1) \leq \mathtt{dt}(e_2).$$

Furthermore,

$$e_1 \xrightarrow{\mathtt{vis}} e_2 \land e_2 \in P \implies \mathtt{dt}(e_1) < \mathtt{dt}(e_2).$$

Proof. By induction on the structure of $\mathtt{vis}$.

- CASE I: $e_1 \xrightarrow{\mathtt{so}_c} e_2$ for some client c.
  - ot_c is monotonically increasing. So $\mathtt{dt}(e_1) \leq \mathtt{dt}(e_2)$.
  - If e_2 is a PUT, ot_c is increased due to the ticking on Line 13 of Algorithm 2. So $\mathtt{dt}(e_1) < \mathtt{dt}(e_2)$.
- CASE II: $e_1 \xrightarrow{\mathtt{rf}} e_2$.
  - According to the waiting condition on Line 3 of Algorithm 2, $\mathsf{ot}_c \leq ot$ always holds on Line 5 of Algorithm 1. So $\mathtt{dt}(e_1) \leq \mathtt{dt}(e_2)$.
  - e_2 is a GET.
- CASE III: There is some event e' such that $e_1 \xrightarrow{\mathtt{vis}} e' \xrightarrow{\mathtt{vis}} e_2$.
  - By induction, we have $\mathtt{dt}(e_1) \leq \mathtt{dt}(e') \leq \mathtt{dt}(e_2)$.
  - By induction, if e_2 is a PUT, $\mathtt{dt}(e_1) \leq \mathtt{dt}(e') < \mathtt{dt}(e_2)$.

□

Theorem 1. *The visibility relation $\mathtt{vis}$ is a partial order.*

Proof. It suffices to prove that $\mathtt{vis}$ is acyclic (thus, irreflexive). Suppose for a contradiction that there is a cycle $C : e_1 \xrightarrow{\mathtt{vis}} e_2 \xrightarrow{\mathtt{vis}} \cdots \xrightarrow{\mathtt{vis}} o_i \xrightarrow{\mathtt{vis}} e_1$. By Lemma 1, all events in C are GET and they have the same $\mathtt{dt}$. Furthermore, all of them cannot occur at the same client, as this would imply a cycle in session order. Assume e and e' in C are on different clients. Since $e \xrightarrow{\mathtt{vis}} e'$ and both of them are GET, there must be some PUT e'' such that $e \xrightarrow{\mathtt{vis}} e'' \xrightarrow{\mathtt{vis}} e'$. By Lemma 1, $\mathtt{dt}(e) < \mathtt{dt}(e'') \leq \mathtt{dt}(e')$, implying $\mathtt{dt}(e) \neq \mathtt{dt}(e')$, a contradiction. □

Theorem 2. $\mathtt{hb} \subseteq \mathtt{vis}$.

Proof. By definitions of $\mathtt{hb}$ and $\mathtt{vis}$, we have

$$\mathtt{hb} \triangleq (\mathtt{so} \cup \mathtt{vis})^+ = \mathtt{vis}^+ = \mathtt{vis}.$$

Clearly, $\mathtt{hb} \subseteq \mathtt{vis}$. □

3) Arbitration: We construct the arbitration relation $\mathtt{ar}$ which is a total order over all events as follows.

Definition 23 (Arbitration)**.** Order the PUT events in $\mathtt{ar}$ by their timestamps (Definition 20). For each client c, we insert its GET events one by one in session order: each GET event g on client c is placed immediately after the later (in $\mathtt{ar}$) of (1) the previous (in $\mathtt{so}$) event of g on client c, if any; and (2) the PUT event p (in $\mathtt{ar}$) such that $p \xrightarrow{\mathtt{rf}} g$.

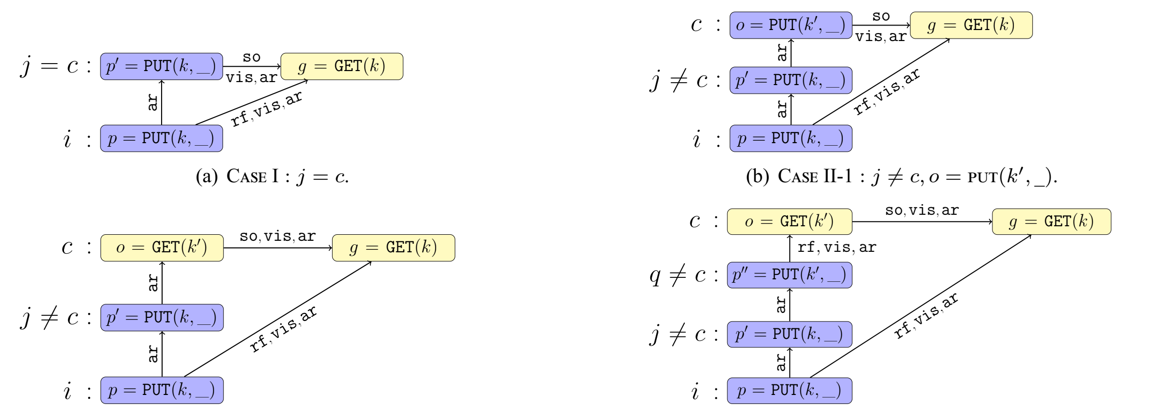
Theorem 3.

$$\mathtt{vis} \subseteq \mathtt{ar}.$$

Proof. Let e_1 and e_2 be two events of history H. We need to show that if $e_1 \xrightarrow{\mathtt{vis}} e_2$, then $e_1 \xrightarrow{\mathtt{ar}} e_2$. By induction on the structure of $\mathtt{vis}$.

- CASE I : $e_1 \xrightarrow{\mathtt{so}} e_2$ on the same client. By Condition 1 of $\mathtt{ar}$, $e_1 \xrightarrow{\mathtt{ar}} e_2$.
- CASE II : $e_1 \xrightarrow{\mathtt{rf}} e_2$. By Condition 2 of $\mathtt{ar}$, $e_1 \xrightarrow{\mathtt{ar}} e_2$.
- CASE III : There is some event e' such that $e_1 \xrightarrow{\mathtt{vis}} e' \xrightarrow{\mathtt{vis}} e_2$. By induction and the transitivity of $\mathtt{vis}$ and $\mathtt{ar}$, we have $e_1 \xrightarrow{\mathtt{ar}} e' \xrightarrow{\mathtt{ar}} e_2$.

□



(a) CASE I : $j = c$.

(b) CASE II-1 : $j \neq c, o = \text{PUT}(k', \_)$.

(c) CASE II-2-A : $j \neq c, o = \text{GET}(k')$. There are no events between p' and o in $S_{\preceq e}$.

(d) CASE II-2-B : $j \neq c, o = \text{GET}(k')$. There is a set of events between p' and o in $S_{\preceq e}$.

Fig. 4. Illustration of the proof of Theorem 4.

4) Return Values: We need to prove that $(H, \mathtt{vis}, \mathtt{ar}) \models \text{RVAL}(\mathsf{kvs}, V)$, where $V(e) = \mathtt{so}^{-1}(e)$ for each event e. The key is to show, for each GET event e, that

$$\forall e \in G : \mathtt{rval}(e) \in \mathtt{eval}_{\mathsf{kvs}}(\mathtt{ctxt}_A(e, V), \mathtt{op}(e)),$$

where $\mathtt{ctxt}_A(e, V) = A|_{\mathtt{vis}^{-1}(e),\mathtt{op},\mathtt{rval}|_{\mathtt{so}^{-1}(e)},\mathtt{vis},\mathtt{ar}}$. Since $V(e) = \mathtt{so}^{-1}(e)$ in *CMv*, we only need to construct a *valid serialization* for e, denoted S_e, consisting of all previous (in $\mathtt{so}$) events before e and all PUT events in $\mathtt{vis}^{-1}(e)$. Suppose e is on client c. We denote the set of events in S_e by cP, namely, $cP \triangleq \mathtt{so}^{-1}(e) \cup (\mathtt{vis}^{-1}(e) \cap P)$.

Theorem 4. S_e is a valid serialization of cP.

Proof. Let $S_{\preceq e} \triangleq S_e \circ \langle e \rangle$, where '$\circ$' denotes concatenation. We need to show that each GET(k) event in $S_{\preceq e}$ returns the value written by the most recently preceding PUT event on key k. Consider $g = \text{GET}(k)$ on client c in $S_{\preceq e}$. Suppose $p \xrightarrow{\mathtt{rf}} g$ and p is on client i. We must show that no other PUT$(k, _)$ events place in between p and g in $S_{\preceq e}$. Suppose by contradiction that event $p' = \text{PUT}(k, \_)$ on client j does. We distinguish between $j = c$ and $j \neq c$.

- CASE I : $j = c$ (Fig. 4(a)). Since $p \xrightarrow{\mathtt{ar}} p'$, $\mathtt{ts}(p) < \mathtt{ts}(p')$. Since p and p' are applied on the same primary node, $\mathtt{dt}(p) < \mathtt{dt}(p')$. Since $p' \xrightarrow{\mathtt{ar}} g$ and $j = c$, $p' \xrightarrow{\mathtt{so}} g$. So, when g is issued by client c, $\mathtt{ot}_c \geq \mathtt{dt}(p') > \mathtt{dt}(p)$ on Line 3 of Algorithm 1. By Line 8 of Algorithm 2, it is impossible for g to read from p on Line 9 of Algorithm 2.
- CASE II : $j \neq c$. Since $p' \xrightarrow{\mathtt{ar}} g$, there is some event o on client c such that $p' \xrightarrow{\mathtt{ar}} o \xrightarrow{\mathtt{ar}} g$. Let o be the first such event. We perform a case analysis according to whether o is a PUT or a GET.
  - CASE II-1 : $o \in P_{k'}$ (Fig. 4(b)). Since $p \xrightarrow{\mathtt{ar}} p' \xrightarrow{\mathtt{ar}} o$, $\mathtt{ts}(p) < \mathtt{ts}(p') < \mathtt{ts}(o)$. Since both p and p' are applied on the same primary node, $\mathtt{dt}(p) < \mathtt{dt}(p') \leq \mathtt{dt}(o)$. Since $o \xrightarrow{\mathtt{so}} g$, when g is issued by client c, $\mathtt{ot}_c \geq \mathtt{dt}(o) > \mathtt{dt}(p)$ on Line 3 of Algorithm 1. By Line 8 of Algorithm 2, it is impossible for g to read from p on Line 9 of Algorithm 2.
  - CASE II-2 : $o \in G_{k'}$. We consider two cases.
    - CASE II-2-A : There are no events between p' and o in $S_{\preceq e}$ (Fig. 4(c)). By construction of S_e, $k' = k$ and $p' \xrightarrow{\mathtt{rf}} o$. Therefore, $\mathtt{dt}(p) < \mathtt{dt}(p') \leq \mathtt{dt}(o)$. By a similar argument in CASE II-1, it is impossible for g to read from p.
    - CASE II-2-B : There is a set, denoted B, of events between p' and o in $S_{\preceq e}$ (Fig. 4(d)). By the choice of o, B contains no events on client c. Therefore, all events in B are PUT. Moreover, there exists some event $p'' \in B$ such that $p'' \xrightarrow{\mathtt{rf}} o$; otherwise, o should be placed before p' in $\mathtt{ar}$. Then, $\mathtt{dt}(p) < \mathtt{dt}(p') \leq \mathtt{dt}(p'') \leq \mathtt{dt}(o)$. By a similar argument in CASE II-1, it is impossible for g to read from p.

□

VI. RELATED WORK

The (vis, ar) specification framework for arbitrary eventually consistent replicated data types is recently proposed by Burckhardt et al. [1], [2]. It introduces the visibility and arbitrary relations, which have been widely adopted in the literature. Using the (vis, ar) framework, Viotti et al. [11] provide a comprehensive overview of more than 50 different consistency models in distributed systems. Emmi et al. [22] develop a fine-grained consistency specification methodology for software API via visibility relaxation. Perrin et al. [23] introduce update consistency as a convergent version of PRAM consistency [13]. However, none of them has identified or overcome the two limitations of the (vis, ar) framework in Section I. In this paper, we extend the (vis, ar) framework into a more generic one called (vis, ar, V) for weakly consistent replicated data types.

Several works are devoted to developing uniform frameworks for consistency models in shared-memory multiprocessor systems. Steinke et al. [24] present a unified theory of shared-memory consistency models based on four consistency properties. Enumerating all combinations of these four properties produces a lattice of consistency models. Alglave [25] provides a generic framework for weak consistency models in modern multiprocessor architectures. It uses the global time model and addresses the store atomicity relaxation.

Causal consistency has been widely used in distributed systems [14], [19], [26]. There are three known causal consistency variants in the literature, namely *WCC* (Weak Causal Consistency) [15], [18], *CM* (Causal Memory) [14], [15], [18], and *WCCv* (Weak Causal Convergence) [2], [15], [18]. All of them can be formally specified in our (vis, ar, V) framework. Moreover, by following the recipes, we discover three new causal consistency variants, namely *SCC* (Strong Causal Consistency), *CMv* (Causal Memory Convergence), and *SCCv* (Strong Causal Convergence). As a case study, we show that the causal consistency protocol of MongoDB [20] satisfies *CMv*. MongoDB has been claimed to be one of the first commercial databases that provide causal consistency [20]. As far as we know, this is the first correctness proof for MongoDB protocols against formal specifications.

VII. Conclusion and Future Work

We extend the (vis, ar) specification framework for eventually consistent replicated data types into a more generic one called (vis, ar, V) for weakly consistent replicated data types. It covers both non-convergent consistency models and the consistency models in which each event is required to be aware of the return values of some or all events that are visible to it. Moreover, it helps to discover new consistency models. As a case study, we show that the causal consistency protocol of MongoDB satisfies *CMv* (Causal Memory Convergence), a new causal consistency variant discovered in our framework.

In this paper, we take causal consistency variants as examples to demonstrate the (vis, ar, V) framework. In the future, we will explore more consistency models, including PRAM consistency variants [13], [23] and sequential consistency variants [17], and their uses in practical systems. We also plan to implement our specification framework in Coq [27], which would facilitate formal reasoning about consistency protocols.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (No. 61932021, 61702253, and 61772258). Hengfeng Wei is the corresponding author.

References

[1] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, “Replicated data types: Specification, verification, optimality,” in *Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL’2014)*, 2014, pp. 271–284.

[2] S. Burckhardt, “Principles of eventual consistency,” *Found. Trends Program. Lang.*, vol. 1, no. 1-2, pp. 1–150, Oct. 2014.

[3] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services,” *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.

[4] E. A. Brewer, “Towards robust distributed systems (abstract),” in *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC’2000)*, 2000, pp. 7–7.

[5] D. Abadi, “Consistency tradeoffs in modern distributed database system design: Cap is only part of the story,” *IEEE Computer*, vol. 45, no. 2, pp. 37–42, Feb. 2012.

[6] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, “Managing update conflicts in bayou, a weakly connected replicated storage system,” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP’1995)*, 1995, pp. 172–182.

[7] W. Vogels, “Eventually consistent,” *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009.

[8] C. A. Ellis and S. J. Gibbs, “Concurrency control in groupware systems,” in *Proceedings of the 1989 International Conference on Management of Data (SIGMOD’1989)*, 1989, pp. 399–407.

[9] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping, “High-latency, low-bandwidth windowing in the jupiter collaboration system,” in *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology (UIST’1995)*, 1995, pp. 111–120.

[10] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski, “Specification and complexity of collaborative text editing,” in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC’2016)*, 2016, pp. 259–268.

[11] P. Viotti and M. Vukolić, “Consistency in non-transactional distributed storage systems,” *ACM Comput. Surv.*, vol. 49, no. 1, pp. 19:1–19:34, Jun. 2016.

[12] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[13] R. J. Lipton and J. S. Sandberg, *PRAM: A scalable shared memory*. Princeton University, Department of Computer Science, 1988.

[14] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, “Causal memory: Definitions, implementation, and programming,” *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[15] M. Perrin, A. Mostefaoui, and C. Jard, “Causal consistency: Beyond memory,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’2016)*, 2016, pp. 26:1–26:12.

[16] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS’2011)*, 2011, pp. 386–400.

[17] A. Gotsman and S. Burckhardt, “Consistency models with global operation sequencing and their composition,” in *Proceedings of the 31st International Symposium on Distributed Computing (DISC’2017)*, 2017, pp. 23:1–23:16.

[18] A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza, “On verifying causal consistency,” in *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL’2017)*, 2017, p. 626–638.

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, “Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops,” in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP’2011)*, 2011, pp. 401–416.

[20] M. Tyulenev, A. Schwerin, A. Kamsky, R. Tan, A. Cabral, and J. Mulrow, “Implementation of cluster-wide logical clock and causal consistency in mongodb,” in *Proceedings of the 2019 International Conference on Management of Data (SIGMOD’2019)*, 2019, pp. 636–650.

[21] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, “Logical physical clocks,” in *International Conference on Principles of Distributed Systems (OPODIS’2014)*, 2014, pp. 17–32.

[22] M. Emmi and C. Enea, “Weak-consistency specification via visibility relaxation,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[23] M. Perrin, A. Mostefaoui, and C. Jard, “Update consistency for wait-free concurrent objects,” in *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS’2015)*, 2015, pp. 219–228.

[24] R. C. Steinke and G. J. Nutt, “A unified theory of shared memory consistency,” *J. ACM*, vol. 51, no. 5, pp. 800–849, Sep. 2004.

[25] J. Alglave, “A shared memory poetics,” Ph.D. dissertation, L’université Paris Denis Diderot, 2010.

[26] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, “Cure: Strong semantics meets high availability and low latency,” in *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS’2016)*, 2016, pp. 405–414.

[27] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.