

# 'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems

Alexey Gotsman

IMDEA Software Institute, Spain

Hongseok Yang

University of Oxford, UK

Carla Ferreira

NOVA LINCS, DI, FCT,  
Universidade NOVA de Lisboa, Portugal

Mahsa Najafzadeh

Sorbonne Universités, Inria,  
UPMC Univ Paris 06, France

Marc Shapiro

Sorbonne Universités, Inria,  
UPMC Univ Paris 06, France

## Abstract

Large-scale distributed systems often rely on replicated databases that allow a programmer to request different data consistency guarantees for different operations, and thereby control their performance. Using such databases is far from trivial: requesting stronger consistency in too many places may hurt performance, and requesting it in too few places may violate correctness. To help programmers in this task, we propose the first proof rule for establishing that a particular choice of consistency guarantees for various operations on a replicated database is enough to ensure the preservation of a given data integrity invariant. Our rule is modular: it allows reasoning about the behaviour of every operation separately under some assumption on the behaviour of other operations. This leads to simple reasoning, which we have automated in an SMT-based tool. We present a nontrivial proof of soundness of our rule and illustrate its use on several examples.

**Categories and Subject Descriptors** D.2.4 [Software Engineering]: Software/Program Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

**Keywords** Replication; causal consistency; integrity invariants

## 1. Introduction

To achieve availability and scalability, many modern distributed systems rely on *replicated databases*, which maintain multiple *replicas* of shared data. Clients can access the data at any of the replicas, and these replicas communicate changes to each other using message passing. For example, large-scale Internet services use data replicas in geographically distinct locations, and applications for mobile devices keep replicas locally to support offline

use. Ideally, we would like replicated databases to provide *strong consistency*, i.e., to behave as if a single centralised node handles all operations. However, achieving this ideal usually requires synchronisation among replicas, which slows down the database and even makes it unavailable if network connections between replicas fail [2, 24].

For this reason, modern replicated databases often eschew synchronisation completely; such databases are commonly dubbed *eventually consistent* [47]. In these databases, a replica performs an operation requested by a client locally without any synchronisation with other replicas and immediately returns to the client; the *effect* of the operation is propagated to the other replicas only *eventually*. This may lead to *anomalies*—behaviours deviating from strong consistency. One of them is illustrated in Figure 1(a). Here Alice makes a post while connected to a replica  $r_1$ , and Bob, also connected to  $r_1$ , sees the post and comments on it. After each of the two operations,  $r_1$  sends a message to the other replicas in the system with the update performed by the user. If the messages with the updates by Alice and Bob arrive to a replica  $r_2$  out of order, then Carol, connected to  $r_2$ , may end up seeing Bob's comment, but not Alice's post it pertains to. The *consistency model* of a replicated database restricts the anomalies that it exhibits. For example, the model of *causal consistency* [33], which we consider in this paper, disallows the anomaly in Figure 1(a), yet can be implemented without any synchronisation. The model ensures that all replicas in the system see *causally dependent* events, such as the posts by Alice and Bob, in the order in which they happened. However, causal consistency allows different replicas to see *causally independent* events as occurring in different orders. This is illustrated in Figure 1(b), where Alice and Bob concurrently make posts at  $r_1$  and  $r_2$ . Carol, connected to  $r_3$  initially sees Alice's post, but not Bob's, and Dave, connected to  $r_4$ , sees Bob's post, but not Alice's. This outcome cannot be obtained by executing the operations in any total order and, hence, deviates from strong consistency.

Such anomalies related to the ordering of actions are often acceptable for applications. What is not acceptable is to violate crucial well-formedness properties of application data, called *integrity invariants*. Consistency models that do not require any synchronisation are often too weak to ensure these. For example, consider a toy banking application where the database stores the balance of a single account that clients can make deposits to and withdrawals from. In this case, an integrity invariant may require the account balance to be always non-negative. Consider the database compu-

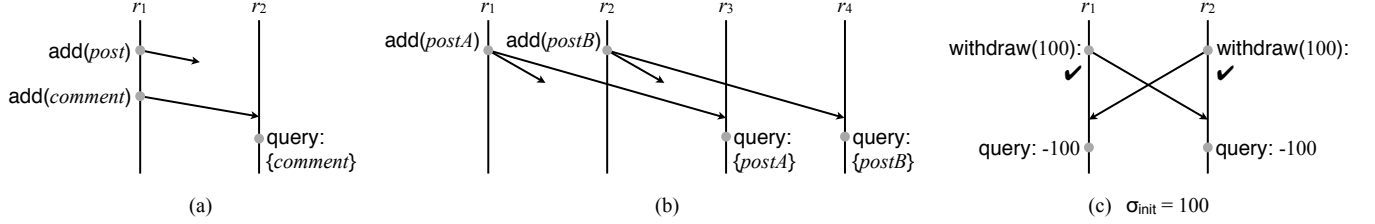
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA.

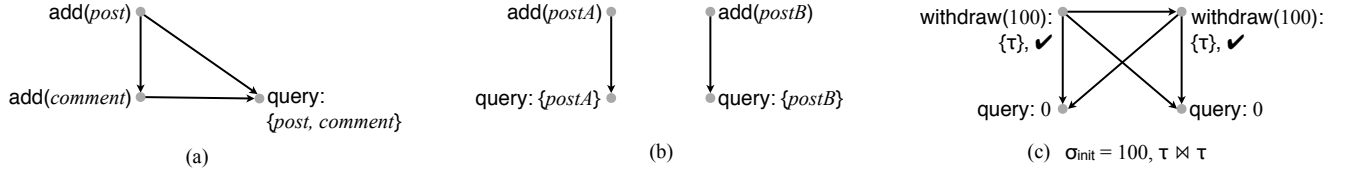
Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-3549-2/16/01...\$15.00.

<http://dx.doi.org/10.1145/2837614.2837625>



**Figure 1.** Illustrations of replicated database computations.



**Figure 2.** Examples illustrating Definition 1. We omit return values when they are  $\perp$  and token sets when they are empty.

tation in Figure 1(c), allowed by causal consistency. Initially all replicas store the same balance of 100. Alice and Bob, connected to  $r_1$  and  $r_2$ , both withdraw 100, thinking that there are sufficient funds available. Once the two replicas exchange the updates, the balance becomes  $-100$ , violating the integrity invariant. To ensure the integrity invariant in this example, we have to introduce synchronisation between replicas, and, since synchronisation is expensive, we would like to introduce it sparingly. To allow this, some research [9, 32, 42, 44] and commercial [6, 10, 35] databases now provide *hybrid* consistency models that allow the programmer to request stronger consistency for certain operations and thereby introduce synchronisation. For example, a consistency model may execute some operations under causal consistency, and some under strong consistency [32]. To preserve the integrity invariant in our banking application when using this model, only withdrawal operations need to use strong consistency, and hence, synchronise to ensure that the account is not overdrawn; deposit operations may use causal consistency and hence proceed without synchronisation. Requesting stronger consistency in hybrid models is similar to the use of fences in weak memory models of shared-memory multiprocessors and programming languages [11] (see §7 for a comparison).

Even though hybrid consistency models allow the programmer to fine-tune consistency level, using these models effectively is far from trivial. Requesting stronger consistency in too many places may hurt performance and availability, and requesting it in too few places may violate correctness. Striking the right balance requires the programmer to reason about the application behaviour on the subtle semantics of the consistency model, taking into account which anomalies are disallowed by a particular consistency strengthening and whether disallowing these anomalies is enough to ensure correctness. This difficulty is compounded by the perennial challenge of reasoning about concurrency, present even with strong consistency—having to consider the huge number of possible interactions between concurrently executing operations.

To help programmers exploit hybrid consistency models, we propose the first proof rule and tool for proving integrity invariants of applications using replicated databases with a range of hybrid models. In more detail, our first contribution is a generic hybrid consistency model (§2) that is flexible enough to encode a variety of consistency models for replicated databases proposed in the literature [9, 32, 33, 42]. It guarantees causal consistency by default and allows the programmer to additionally specify which pairs of operations may not execute without synchronisation by means of a

special *conflict relation*. For example, to ensure the non-negativity of balances in the banking application, the conflict relation may require any pair of withdrawals to synchronise, so that one of them is aware of the effect of the other. This is equivalent to executing withdrawals under strong consistency. In general, different instances of the conflict relation correspond to different interfaces for strengthening consistency proposed in the literature. Our proof rule is developed for the generic consistency model and, hence, applies to existing models that can be represented as its instantiations. We specify our consistency model formally (§3) using the approach previously proposed for specifying variants of eventual consistency [15]. In this approach, a database computation is denoted by a partial order on client operations, representing causality, and the conflict relation imposes additional constraints on this order.

Our next, and key, technical contribution is a proof rule for showing that a set of operations preserves a given integrity invariant when executed on our consistency model with a given choice of conflict relation (§4). For example, we can prove that withdrawals and deposits preserve the non-negativity of balances when executed with the conflict relation described above. To avoid explicit reasoning about all possible interactions between operations, our proof rule is *modular*: it allows us to reason about the behaviour of every operation separately under some assumption on the behaviour of other operations, which takes into account the conflict relation. In this way, our proof rule allows the programmer to reason precisely about how strengthening or weakening consistency of certain operations affects correctness.

The modular nature of our proof rule allows it to reason in terms of states of a single database copy, just like in proof rules for strongly consistent shared-memory concurrency. We have proved that this simple reasoning is sound, despite the weakness of the consistency model (§5). As part of this proof we have identified a more general *event-based* rule that reasons directly in terms of partial orders on events representing database computations, instead of database states that these events lead to. The soundness of the original *state-based* rule is proved by deriving it from the event-based one. In this way, the event-based rule explicates the reasons for the soundness of the state-based rule.

We have also developed a prototype tool that automates our proof rule by reducing checking its obligations to SMT queries (§6). Using the tool, we have verified several example applications that require strengthening consistency in nontrivial ways. These

include an extension of the above banking application, an online auction service and a course registration system. In particular, we were able to handle applications using *replicated data types* (aka CRDTs [40]), which encapsulate policies for automatically merging the effects of operations performed without synchronisation at different replicas. The fact that we can reduce checking the correctness properties of complex computations in our examples to querying off-the-shelf SMT tools demonstrates the simplicity of reasoning required by our approach.

## 2. Consistency Model, Informally

We start by presenting our generic consistency model. Even though this model is not implemented in its full generality by an existing database, it can encode a variety of models that have in fact been implemented. In this section we present the programming interface of our consistency model and describe its semantics informally, from an operational perspective. We give a formal semantics in §3.

### 2.1 Causal Consistency and Its Implementation

Our hybrid model guarantees at least *causal consistency* [33], already mentioned in §1. We therefore start by presenting informally how a typical implementation of a causally consistent database operates. Let  $\text{State}$  be the set of possible states of the data managed by the database system. We denote states by  $\sigma$  and let  $\sigma_{\text{init}}$  be a distinguished initial state. Applications define a set of operations  $\text{Op} = \{o, \dots\}$  on the data and interact with the database by issuing these operations. For simplicity, we assume that an operation always terminates and returns a single value from a set  $\text{Val}$ . We use a value  $\perp \in \text{Val}$  to model operations that return no value. We do not consider operation parameters, since these can be part of the operation name.

The database implementation consists of a set of replicas, each maintaining a complete copy of the database state; we identify replicas by  $r_1, r_2, \dots$ . For the purposes of the informal explanation, we assume that replicas never fail. A client operation is initially executed at a single replica, which we refer to as its *origin* replica. At this replica, the execution of the operation is not interleaved with that of others. This execution updates the replica state deterministically, and immediately returns a value to the client. After this, the replica sends a message to all other replicas containing the *effect* of the operation, which describes the updates done by the operation to the database state. The replicas are guaranteed to receive the message at most once. Upon receipt, the replicas apply the effect to their state.

In this paper, we abstract from a particular language in which operations may be written and assume that their semantics is given by a function

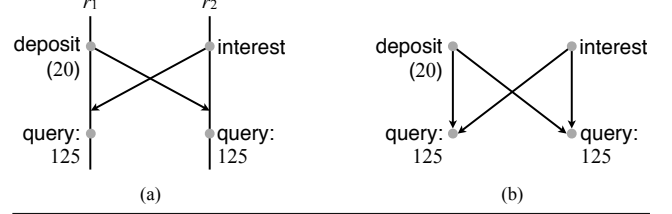
$$\mathcal{F} \in \text{Op} \rightarrow (\text{State} \rightarrow (\text{Val} \times (\text{State} \rightarrow \text{State}))). \quad (1)$$

To aid readability, for  $o \in \text{Op}$  we write  $\mathcal{F}_o$  instead of  $\mathcal{F}(o)$  and let

$$\forall o, \sigma. \mathcal{F}_o(\sigma) = (\mathcal{F}_o^{\text{val}}(\sigma), \mathcal{F}_o^{\text{eff}}(\sigma)).$$

Given a state  $\sigma$  of  $o$ 's origin replica,  $\mathcal{F}_o^{\text{val}}(\sigma) \in \text{Val}$  determines the return value of the operation and  $\mathcal{F}_o^{\text{eff}}(\sigma) \in \text{State} \rightarrow \text{State}$  its effect. The latter is a function, to be applied by every replica to its state to incorporate the operation's effect: immediately at the origin replica, and after receiving the corresponding message at all other replicas.

For example, states in the toy banking application of §1 are integers, representing the account balance:  $\text{State} = \mathbb{Z}$ . We define



**Figure 3.** (a) An illustration of a database computation; (b) the corresponding execution of Definition 1. We assume  $\sigma_{\text{init}} = 100$ .

the semantics of operations for depositing an amount  $a > 0$ , accruing a 5% interest and querying the balance:

$$\begin{aligned} \mathcal{F}_{\text{deposit}(a)}(\sigma) &= (\perp, (\lambda\sigma'. \sigma' + a)); \\ \mathcal{F}_{\text{interest}}(\sigma) &= (\perp, (\lambda\sigma'. \sigma' + 0.05 * \sigma)); \\ \mathcal{F}_{\text{query}}(\sigma) &= (\sigma, \text{skip}), \end{aligned} \quad (2)$$

where  $\text{skip} = (\lambda\sigma'. \sigma')$ . Figure 3(a) illustrates a database computation involving these operations. Note that interest first computes the interest  $0.05 * \sigma$  based on the balance  $\sigma$  at the origin replica; its effect then adds the resulting amount to the balance at each replica. In particular, in Figure 3(a) interest at  $r_2$  does not take into account the deposit made at  $r_1$ . This behaviour is the price to pay for avoiding synchronisation between replicas. The good news is that, once the replicas  $r_1$  and  $r_2$  exchange the effects of deposit and interest, they converge to the same balance, which is returned by the query operations.

Such convergence is not guaranteed for arbitrary operations. For example, we could implement interest so that its effect multiplied the balance by 1.05 at each replica where it is applied:

$$\mathcal{F}_{\text{interest}}^{\text{eff}}(\sigma) = (\lambda\sigma'. (1.05 * \sigma')). \quad (3)$$

In the scenario in Figure 3(a), this would lead the query operations to return different values, 126 at  $r_1$  and 125 at  $r_2$ . In this case, even after all messages are delivered, replicas end up in different states. This is undesirable for database users: we would like the implementation to be *convergent*, i.e., such that two replicas that see the same set of operations are in the same state. In particular, if users stop performing updates to the database, then once all outstanding messages are delivered, all replicas should reach the same state [47]. To ensure convergence, for now we require that the effects of all operations commute (we relax this condition slightly in §2.2):

$$\forall o_1, o_2, \sigma_1, \sigma_2. \mathcal{F}_{o_1}^{\text{eff}}(\sigma_1) \circ \mathcal{F}_{o_2}^{\text{eff}}(\sigma_2) = \mathcal{F}_{o_2}^{\text{eff}}(\sigma_2) \circ \mathcal{F}_{o_1}^{\text{eff}}(\sigma_1). \quad (4)$$

For example, this condition holds of the effects defined by (2). The requirement of commutativity is not very taxing: as we elaborate in §6, to satisfy (4), programmers can exploit ready-made *replicated data types* (aka CRDTs [40]). These encapsulate commutative implementations of policies for merging concurrent updates to the database.

As we explained in §1, asynchronous operation processing may lead to anomalies, and causal consistency disallows some of them. It ensures that message propagation between replicas is *causal*: if a replica sends a message containing the effect of an operation  $o_2$  after it sends or receives a message containing the effect of an operation  $o_1$ , then no replica will receive the message about  $o_2$  before it receives the one about  $o_1$ . In this case we say that the invocation of  $o_2$  *causally depends* on that of  $o_1$ . Causal propagation disallows the computation in Figure 1(a), but allows the one in Figure 1(b).

$$\begin{aligned}
\text{Token} &= \{\tau\} \\
\bowtie &= \{(\tau, \tau)\} \\
\mathcal{F}_{\text{deposit}(a)}(\sigma) &= (\perp, (\lambda\sigma'. \sigma' + a), \emptyset) \\
\mathcal{F}_{\text{interest}}(\sigma) &= (\perp, (\lambda\sigma'. \sigma' + 0.05 * \sigma), \emptyset) \\
\mathcal{F}_{\text{query}}(\sigma) &= (\sigma, \text{skip}, \emptyset) \\
\mathcal{F}_{\text{withdraw}(a)}(\sigma) &= \text{if } \sigma \geq a \text{ then } (\checkmark, (\lambda\sigma'. \sigma' - a), \{\tau\}) \\
&\quad \text{else } (\text{X}, \text{skip}, \{\tau\})
\end{aligned}$$

**Figure 4.** Operation semantics for the banking application. Note that  $a > 0$ .

## 2.2 Strengthening Consistency

The guarantees provided by causal consistency are too weak to ensure certain integrity invariants. For example, in our banking application we would like the state at each replica to satisfy the invariant

$$I = \{\sigma \mid \sigma \geq 0\}. \quad (5)$$

To ensure this, an operation for withdrawing an amount  $a > 0$  could check whether the account has sufficient funds and return  $\checkmark$  or  $\text{X}$  depending on the result:

$$\mathcal{F}_{\text{withdraw}(a)}(\sigma) = \text{if } \sigma \geq a \text{ then } (\checkmark, (\lambda\sigma'. \sigma' - a)) \text{ else } (\text{X}, \text{skip}).$$

This is enough to maintain the invariant when all operations are processed at the same replica, but not when they are processed asynchronously at different replicas. This is illustrated by the computation in Figure 1(c), already explained in §1.

The problem in this example arises because two particular operations update the database concurrently, without being aware of each other. To address this, our consistency model allows the programmer to strengthen causal consistency by specifying explicitly which operations may not be executed in this way. Namely, the model is parameterised by a *token system*  $\mathcal{T} = (\text{Token}, \bowtie)$ , consisting of a set of *tokens*  $\text{Token}$  and a symmetric *conflict relation*  $\bowtie \subseteq \text{Token} \times \text{Token}$ . Tokens are ranged over by  $\tau$  and their sets, by  $T$ . For sets  $T_1$  and  $T_2$  of tokens we let  $T_1 \bowtie T_2$  if there exists a pair of conflicting tokens coming from these sets:  $\exists \tau_1 \in T_1. \exists \tau_2 \in T_2. \tau_1 \bowtie \tau_2$ .

Each operation may acquire a set of tokens. To account for this, we redefine the type of  $\mathcal{F}$  in (1) as

$$\mathcal{F} \in \text{Op} \rightarrow (\text{State} \rightarrow (\text{Val} \times (\text{State} \rightarrow \text{State}) \times \mathcal{P}(\text{Token}))) \quad (6)$$

and let

$$\forall o, \sigma. \mathcal{F}_o(\sigma) = (\mathcal{F}_o^{\text{val}}(\sigma), \mathcal{F}_o^{\text{eff}}(\sigma), \mathcal{F}_o^{\text{tok}}(\sigma)).$$

Thus,  $\mathcal{F}_o^{\text{tok}}(\sigma) \in \mathcal{P}(\text{Token})$  gives the set of tokens acquired by the operation  $o$  when executed in the state  $\sigma$ . Informally, our consistency model guarantees that operations that acquire tokens conflicting according to  $\bowtie$  have to be causally dependent one way or another: the origin replica of one operation must have incorporated the effect of the other by the time the former operation executes. Ensuring this in implementations requires replicas to synchronise [9, 32].

In our consistency model, we can guarantee the preservation of invariant (5) in the banking application by defining operation semantics as in Figure 4. Thus, *withdraw* acquires a token  $\tau$  conflicting with itself, and all other operations do not acquire any tokens. Then the scenario in Figure 1(c) cannot happen: one withdrawal would have to be aware of the other and would therefore fail. However, deposits and interest accruals can be causally independent with all operations, and replicas can therefore execute them without any synchronisation [9, 32]. In this example, the token  $\tau$  is

analogous to a mutual exclusion lock in shared-memory concurrency. Our proof method (§4) establishes that this use of the token is indeed sufficient to preserve the integrity invariant (5).

Since operations acquiring conflicting tokens have to be causally dependent, causal message propagation (§2.1) ensures that all replicas see such operations in the same order. This allows us to weaken (4) to require commutativity only for operations that do not acquire conflicting tokens:

$$\begin{aligned}
&\forall o_1, o_2, \sigma_1, \sigma_2. (\mathcal{F}_{o_1}^{\text{tok}}(\sigma_1) \bowtie \mathcal{F}_{o_2}^{\text{tok}}(\sigma_2)) \vee \\
&(\mathcal{F}_{o_1}^{\text{eff}}(\sigma_1) \circ \mathcal{F}_{o_2}^{\text{eff}}(\sigma_2) = \mathcal{F}_{o_2}^{\text{eff}}(\sigma_2) \circ \mathcal{F}_{o_1}^{\text{eff}}(\sigma_1)). \quad (7)
\end{aligned}$$

As we show in §3, this is sufficient to ensure the property of convergence that we introduced in §2.1. For example, the operations in Figure 4 satisfy (7). Furthermore, if all operations except query acquired the token  $\tau$ , then we would be able to implement interest by the effect given by (3) without compromising convergence.

## 3. Formal Semantics

We now formally define the semantics of our consistency model, i.e., the set of all client-database interactions it allows. To keep the presentation as simple as possible, we define the semantics declaratively: our formalism does not refer to implementation-level concepts, such as replicas or messages, even though we do use these concepts in informal explanations. We build on an approach previously used to specify forms of eventual consistency [15]. Namely, our denotations of database computations consist of a set of events, representing operation invocations by clients, and a relation on events, describing abstractly how the database processes the corresponding operations.

Assume a countably infinite set  $\text{Event}$  of *events*, ranged over by  $e, f, g$ . A relation is a *strict partial order* if it is transitive and irreflexive. For a relation  $R$  we write  $(e, f) \in R$  and  $e \xrightarrow{R} f$  interchangeably.

**DEFINITION 1.** Given a token system  $\mathcal{T} = (\text{Token}, \bowtie)$ , an *execution* is a tuple  $X = (E, \text{oper}, \text{rval}, \text{tok}, \text{hb})$ , where:

- $E$  is a finite subset of  $\text{Event}$ ;
- $\text{oper} : E \rightarrow \text{Op}$  gives the operation whose invocation a given event denotes;
- $\text{rval} : E \rightarrow \text{Val}$  gives the return value of the operation;
- $\text{tok} : E \rightarrow \mathcal{P}(\text{Token})$  gives the set of tokens acquired by the operation;
- $\text{hb} \subseteq E \times E$ , called **happens-before**, is a strict partial order such that

$$\forall e, f \in E. \text{tok}(e) \bowtie \text{tok}(f) \implies (e \xrightarrow{\text{hb}} f \vee f \xrightarrow{\text{hb}} e). \quad (8)$$

Operationally, each event represents an invocation of an operation at its origin replica. The applications of the operation's effect at other replicas are not recorded in an execution explicitly. Instead, the happens-before relation records causal dependencies between operations arising from such applications:  $e \xrightarrow{\text{hb}} f$  means that either the operations denoted by  $e$  and  $f$  were executed at the same replica in this order, or they were executed at different replicas and the message containing the effect of  $e$  had been delivered to the replica performing  $f$  before  $f$  was executed. Hence, if we have  $e \xrightarrow{\text{hb}} f$ , then the effect of  $e$  is incorporated into the state to which  $f$  is applied and may influence its return value. We give examples of executions in Figures 2 and 3(b). The ones in Figures 2(b) and 3(b) model the computations of the database informally illustrated in Figures 1(b) and 3(a), respectively.

The transitivity of  $\text{hb}$  in Definition 1 reflects the guarantee of causal message propagation in implementations explained in

§2.1 [15]. For example, in the execution of Figure 2(a), the transitivity of hb mandates the edge between the addition of a post and the query (cf. Figure 1(a)). The condition (8) formalises the stronger consistency guarantee provided by tokens: operations acquiring conflicting tokens have to be causally dependent. For example, since the two withdraw operations in Figure 2(c) acquire a token  $\tau$  with  $\tau \triangleright \tau$ , they have to be related by happens-before. Finally, we require executions to contain only finitely many events, because in this paper we are only concerned with safety properties of applications.

We write  $\text{Exec}(\mathcal{T})$  for the set of all executions over the token system  $\mathcal{T}$ . In the following, we denote components of  $X$  and similar structures as in  $X.E$ . We let  $X_{\text{init}}$  be the unique execution with  $X_{\text{init}}.E = \emptyset$ .

We now define the semantics of our consistency model as the set of all executions  $X \in \text{Exec}(\mathcal{T})$  over a token system  $\mathcal{T}$  whose return values  $X.\text{rval}$  and token sets  $X.\text{tok}$  are computed using  $\mathcal{F}$  as informally described in §2. To define this set, we first let the *context* of an event  $e$  in an execution  $X$  be

$$\text{ctxt}(e, X) = (E, (X.\text{oper})|_E, (X.\text{rval})|_E, (X.\text{tok})|_E, (X.\text{hb})|_E),$$

where  $E = (X.\text{hb})^{-1}(e)$  and  $\cdot|_E$  is the restriction to events in  $E$ . Operationally-speaking, the context consists of those events whose effects have been incorporated into the state of the replica where the operation  $X.\text{oper}(e)$  executes; it is these events that influence the outcomes of  $e$ —the return value  $X.\text{rval}(e)$  and the token set  $X.\text{tok}(e)$ . For example, the context of each of the query events in Figure 3(b) consists of the deposit and interest events. This reflects the events that the corresponding replica has seen before executing query in Figure 3(a).

It is technically convenient for us to initially formulate definitions without assuming effect commutativity (7). In this case,  $X.\text{rval}(e)$  and  $X.\text{tok}(e)$  are not determined by  $\text{ctxt}(e, X)$  uniquely. In operational terms, this is because the state that a replica will be in after seeing the events in  $\text{ctxt}(e, X)$  depends on the order in which the replica finds out about these events: although causal message propagation ensures that messages about causally dependent events in  $\text{ctxt}(e, X)$  will be delivered to the replica in the order consistent with  $X.\text{hb}$ , messages about causally independent events may be delivered in arbitrary order. We therefore first define a function

$$\text{eval}_{\mathcal{F}}^{\dagger} : \text{Exec}(\mathcal{T}) \rightarrow \mathcal{P}(\text{State})$$

that yields the *set* of all possible states that a replica may end up in after seeing the events in a given execution, such as  $\text{ctxt}(e, X)$ . For an execution  $Y$ , we define  $\text{eval}_{\mathcal{F}}^{\dagger}(Y)$  inductively on the size of  $Y.E$ . If  $Y.E = \emptyset$ , then  $\text{eval}_{\mathcal{F}}^{\dagger}(Y) = \{\sigma_{\text{init}}\}$ . Otherwise,

$$\text{eval}_{\mathcal{F}}^{\dagger}(Y) = \{\mathcal{F}_{Y.\text{oper}(e)}^{\text{eff}}(\sigma')(\sigma) \mid e \in \max(Y) \wedge \sigma \in \text{eval}_{\mathcal{F}}^{\dagger}(Y|_{Y.E - \{e\}}) \wedge \sigma' \in \text{eval}_{\mathcal{F}}^{\dagger}(\text{ctxt}(e, Y))\},$$

where

$$\max(Y) = \{e \in Y.E \mid \neg \exists f \in Y.E. (e, f) \in Y.\text{hb}\}. \quad (9)$$

Thus, to compute  $\text{eval}_{\mathcal{F}}^{\dagger}(Y)$  for a non-empty  $Y$ , we choose an hb-maximal event  $e$  in  $Y$ . Operationally, this is the event whose effect is incorporated last by the replica  $r$  whose state we are determining. We then pick a state  $\sigma$  that  $r$  could be in right before incorporating the effect of  $e$ . The set of such states is obtained by invoking  $\text{eval}_{\mathcal{F}}^{\dagger}$  on the execution  $Y|_{Y.E - \{e\}}$ , describing the events  $r$  knew about when it incorporated  $e$ . To determine the effect of  $e$ 's operation, we pick a state  $\sigma'$  that the replica  $r'$  that generated  $e$  could be in at the time of this generation. The set of such states is computed by invoking  $\text{eval}_{\mathcal{F}}^{\dagger}$  on the execution  $\text{ctxt}(e, Y)$ , describing the events that replica  $r'$  knew about when it generated  $e$ . Then the effect of  $e$ 's operation is  $\mathcal{F}_{Y.\text{oper}(e)}^{\text{eff}}(\sigma')$ , and we determine the

state of the replica  $r$  after  $e$  by applying this effect to the state  $\sigma$ :  $\mathcal{F}_{Y.\text{oper}(e)}^{\text{eff}}(\sigma')(\sigma)$ .

To illustrate  $\text{eval}_{\mathcal{F}}^{\dagger}$ , consider the execution  $Y$  consisting of the deposit and interest events in Figure 3(b) and the operation semantics  $\mathcal{F}$  in Figure 4. Recall that in this case  $\sigma_{\text{init}} = 100$ . We can evaluate  $Y$  in two ways, corresponding to the orders in which replicas  $r_1$ , respectively  $r_2$ , apply the effects of the events in the computation in Figure 3(a):

$$\begin{aligned} \text{eval}_{\mathcal{F}}^{\dagger}(Y) &= \{\mathcal{F}_{\text{interest}}^{\text{eff}}(\sigma_{\text{init}})(\mathcal{F}_{\text{deposit}(20)}^{\text{eff}}(\sigma_{\text{init}})(\sigma_{\text{init}})), \\ &\quad \mathcal{F}_{\text{deposit}(20)}^{\text{eff}}(\sigma_{\text{init}})(\mathcal{F}_{\text{interest}}^{\text{eff}}(\sigma_{\text{init}})(\sigma_{\text{init}}))\} \\ &= \{100 + 20 + 5, 100 + 5 + 20\} = \{125\}. \end{aligned}$$

Both ways of evaluation lead to the same outcome. This would not be the case if we used a function  $\mathcal{F}'$  identical to  $\mathcal{F}$ , but with the effect of interest defined by (3), which violates (7). In this case,

$$\text{eval}_{\mathcal{F}'}^{\dagger}(Y) = \{100 + 20 + 6, 100 + 5 + 20\} = \{126, 125\},$$

which corresponds to the diverging database computation we explained in §2.1.

We note that, for notational convenience,  $\text{eval}_{\mathcal{F}}^{\dagger}$  takes as a parameter a whole execution including return values ( $\text{rval}$ ) and token sets ( $\text{tok}$ ) associated with its events. However, the function as we defined it does not depend on these: the state is determined solely based on the operations performed ( $\text{oper}$ ) and happens-before relationships among them ( $\text{hb}$ ).

**DEFINITION 2.** An execution  $X \in \text{Exec}(\mathcal{T})$  is **consistent** with  $\mathcal{T}$  and  $\mathcal{F}$ , denoted  $X \models \mathcal{T}, \mathcal{F}$ , if

$$\forall e \in X.E. \exists \sigma \in \text{eval}_{\mathcal{F}}^{\dagger}(\text{ctxt}(e, X)).$$

$$(X.\text{rval}(e) = \mathcal{F}_{X.\text{oper}(e)}^{\text{val}}(\sigma)) \wedge (X.\text{tok}(e) = \mathcal{F}_{X.\text{oper}(e)}^{\text{tok}}(\sigma)).$$

We let  $\text{Exec}(\mathcal{T}, \mathcal{F}) = \{X \mid X \models \mathcal{T}, \mathcal{F}\}$  be the set of executions allowed by our consistency model.

**PROPOSITION 3.**

$$\forall X \in \text{Exec}(\mathcal{T}, \mathcal{F}). \forall e \in X.E. (\text{ctxt}(e, X) \in \text{Exec}(\mathcal{T}, \mathcal{F})).$$

Operationally,  $X \models \mathcal{T}, \mathcal{F}$  means that the outcomes in  $X$  can be produced by the database implementation sketched in §2 with some order of message delivery. The executions in Figures 2 and 3(b) are consistent with the parameters in Figure 4 or the expected semantics of operations on posts and comments. In particular, the execution in Figure 2(c) is consistent because the context of the right-hand-side withdraw includes the left-hand-side withdraw. Evaluating this context yields a zero balance, which causes the right-hand-side withdraw to generate skip as its effect.

**LEMMA 4.** If  $X \models \mathcal{T}, \mathcal{F}$ , then  $\text{eval}_{\mathcal{F}}^{\dagger}(X)$  is a singleton set. Furthermore, so is  $\text{eval}_{\mathcal{F}}^{\dagger}(\text{ctxt}(e, X))$  for any  $e \in X.E$ .

The lemma shows that in Definition 2 it does not matter how we choose the order of evaluation in  $\text{eval}_{\mathcal{F}}^{\dagger}$ . When viewed operationally, this independence implies the convergence property from §2.1: two replicas that see the same events will end up in the same state. The proof of Lemma 4, given in [26, §A], exploits properties (7) and (8). This proof is subtle because (7) does not require commutativity for the effects of pairs of operations that acquire conflicting tokens.

Motivated by Lemma 4, we define the evaluation of consistent executions

$$\text{eval}_{\mathcal{F}} : \text{Exec}(\mathcal{T}, \mathcal{F}) \rightarrow \text{State}$$

as follows:  $\text{eval}_{\mathcal{F}}(X)$  is the unique  $\sigma$  such that  $\text{eval}_{\mathcal{F}}^{\dagger}(X) = \{\sigma\}$ .

To illustrate the flexibility of our consistency model, we show how it can represent some of the existing models; we provide more instantiations in §6.

**Causal consistency** [16, 33] is the baseline model we obtain without using any tokens:  $\text{Token} = \emptyset$  and  $\forall o, \sigma. \mathcal{F}_o^{\text{tok}}(\sigma) = \emptyset$ . Then (8) is a tautology and (7) is equivalent to (4), so that all effects have to commute.

**Sequential consistency** [29] is a form of strong consistency and the strongest consistency model we can obtain from ours. It requires every operation to acquire a mutual exclusion token:

$$\text{Token} = \{\tau\}; \quad \bowtie = \{(\tau, \tau)\}; \quad \forall o, \sigma. \mathcal{F}_o^{\text{tok}}(\sigma) = \{\tau\}.$$

Then in any execution  $X \in \text{Exec}((\text{Token}, \bowtie), \mathcal{F})$ , the happens-before  $X.\text{hb}$  is total, and each event in  $X$  is aware of the effects of all events preceding it in  $X.\text{hb}$ .

**RedBlue consistency** [32] is a hybrid consistency model that classifies operations as either *red* or *blue*:  $\text{Op} = \text{Op}_r \uplus \text{Op}_b$ . Red operations are guaranteed sequential consistency, and blue operations, only causal consistency. To express this in our model, we again use a mutual exclusion token:  $\text{Token} = \{\tau\}$  and  $\bowtie = \{(\tau, \tau)\}$ . Red operations acquire  $\tau$ , and blue operations acquire no tokens:

$$(\forall o \in \text{Op}_r. \forall \sigma. \mathcal{F}_o^{\text{tok}}(\sigma) = \{\tau\}) \wedge (\forall o \in \text{Op}_b. \forall \sigma. \mathcal{F}_o^{\text{tok}}(\sigma) = \emptyset).$$

Then red operations are totally ordered by happens-before, and blue ones are ordered only partially. The token assignment in our banking application (Figure 4) is an instance of the RedBlue consistency, where withdraw operations are red, and all others are blue.

Our framework cannot express some of common consistency models, such as prefix consistency [43], which is stronger than causal consistency. However, the framework could be adjusted to assume prefix consistency as a baseline following [17].

#### 4. State-based Proof Rule

We consider the following verification problem: given a token system  $\mathcal{T} = (\text{Token}, \bowtie)$ , prove that operations  $\mathcal{F}$  maintain an integrity invariant  $I \subseteq \text{State}$  over database states. Formally, we establish that any execution consistent with  $\mathcal{T}$  and  $\mathcal{F}$  evaluates to a state satisfying  $I$ :

$$\text{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \text{eval}_{\mathcal{F}}^{-1}(I).$$

By Proposition 3 this implies that the return value of every event in an execution  $X \in \text{Exec}(\mathcal{T}, \mathcal{F})$  can be obtained by applying its operation to a state satisfying  $I$ :

$$\forall e \in X.E. \exists \sigma \in I. (X.\text{rval}(e) = \mathcal{F}_{X.\text{oper}(e)}^{\text{val}}(\sigma)).$$

For example, we show that any execution consistent with Figure 4 evaluates to a state satisfying the invariant (5). Hence, a query operation will always return a non-negative balance.

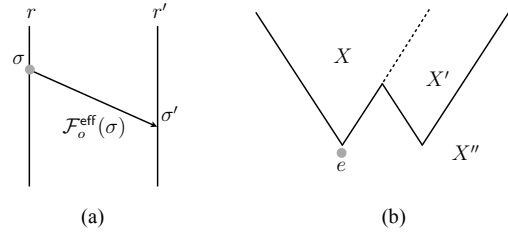
The key challenge of the above verification problem is the need to consider infinitely many executions consistent with  $\mathcal{T}$  and  $\mathcal{F}$ . Our main technical contribution is the proof rule for solving this problem that avoids considering all such executions explicitly. Instead, the proof rule is *modular* in that it allows us to reason about the behaviour of every operation separately. Our proof rule is also *state-based* in that it reasons in terms of states obtained by evaluating parts of executions or, from the operational perspective, in terms of replica states.

We give our proof rule in Figure 5 and explain it from the operational perspective. The rule assumes that the invariant  $I$  holds of the initial database state  $\sigma_{\text{init}}$  (condition S1). Consider a computation of the database implementation from §2 and a state  $\sigma$  of a replica  $r$  at some point in this computation. The proof rule assumes that  $\sigma \in I$  and aims to establish that executing any operation  $o$  at  $r$  will preserve the invariant  $I$ . This is easy if we only consider how

$\exists G_0 \in \mathcal{P}(\text{State} \times \text{State}), G \in \text{Token} \rightarrow \mathcal{P}(\text{State} \times \text{State})$  such that

$$\begin{array}{l} \text{S1. } \sigma_{\text{init}} \in I \\ \text{S2. } G_0(I) \subseteq I \wedge \forall \tau. G(\tau)(I) \subseteq I \\ \text{S3. } \forall o, \sigma, \sigma'. (\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\text{tok}}(\sigma))^\perp))^*) \\ \quad \implies (\sigma', \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma')) \in G_0 \cup G(\mathcal{F}_o^{\text{tok}}(\sigma)) \end{array} \quad \frac{}{\text{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \text{eval}_{\mathcal{F}}^{-1}(I)}$$

**Figure 5.** State-based proof rule for a token system  $\mathcal{T} = (\text{Token}, \bowtie)$ . For  $T \subseteq \text{Token}$  we let  $G(T) = \bigcup_{\tau \in T} G(\tau)$  and  $T^\perp = \{\tau \mid \tau \in \text{Token} \wedge \neg \exists \tau' \in T. \tau \bowtie \tau'\}$ . We denote by  $R^*$  the reflexive and transitive closure of a relation  $R$ . For a relation  $R \in \mathcal{P}(A \times B)$  and a predicate  $P \in \mathcal{P}(A)$ , the expression  $R(P)$  denotes the image of  $P$  under  $R$ .



**Figure 6.** Graphical illustrations of (a) the state-based rule; and (b) the event-based rule.

$o$ 's effect changes the state of  $r$ , since this effect is applied to the state  $\sigma$  where it was generated:

$$\forall \sigma. (\sigma \in I \implies \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma) \in I). \quad (10)$$

The difficulty comes from the need to consider how  $o$ 's effect changes the state of any other replica  $r'$  that receives it; see Figure 6(a). At the time of the receipt,  $r'$  may be in a different state  $\sigma'$ , due to operations executed at  $r'$  concurrently with  $o$ . We can show that it is sound to assume that this state  $\sigma'$  also satisfies the invariant. Thus, to check that the operation  $o$  preserves the invariant when applied at any replica, it is sufficient to ensure

$$\forall \sigma, \sigma'. (\sigma, \sigma' \in I \implies \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma') \in I). \quad (11)$$

However, establishing this without knowing anything about the relationship between  $\sigma$  and  $\sigma'$  is a tall order. In the bank account example, both  $\sigma = 100$  and  $\sigma' = 0$  satisfy the integrity invariant (5). Then  $\mathcal{F}_{\text{withdraw}(100)}^{\text{eff}}(\sigma)(\sigma') = -100$ , which violates the invariant. Condition (11) fails in this case because it does not take into account the tokens acquired by withdraw.

The proof rule in Figure 5 addresses the weakness of (11) by allowing us to assume a certain relationship between the state where an operation is generated ( $\sigma$ ) and where its effect is applied ( $\sigma'$ ), which takes into account the tokens acquired by the operation. To express this assumption, the rule uses a form of rely-guarantee reasoning [27]. Namely, it requires us to associate each token  $\tau$  with a *guarantee* relation  $G(\tau)$ , describing all possible state changes that an operation acquiring  $\tau$  can cause. Crucially, this includes not only the changes that the operation can cause on the state of its origin replica, but also any change that its effect causes at any other replica it is propagated to. We also have a guarantee relation  $G_0$ , describing the changes that can be performed by an operation without acquiring any tokens. Condition S2 requires the guarantees to preserve the invariant.

Like (11), condition S3 considers an arbitrary state  $\sigma$  of  $o$ 's origin replica  $r$ , assumed to satisfy the invariant  $I$ . The condition then considers any state  $\sigma'$  of another replica  $r'$  to which the effect of  $o$  is propagated. The conclusion of S3 requires us to prove that applying the effect  $\mathcal{F}_o^{\text{eff}}(\sigma)$  of the operation  $o$  to the state  $\sigma'$  satisfies the union of  $G_0$  and the guarantees associated with the tokens  $\mathcal{F}_o^{\text{tok}}(\sigma)$  that the operation  $o$  acquires. By S2, this implies that the effect of the operation preserves the invariant. Condition S3 further allows us to assume that the state  $\sigma'$  of  $r'$  can be obtained from the state  $\sigma$  of  $r$  by applying a finite number of changes allowed by  $G_0$  or the guarantees for those tokens that do not conflict with any of the tokens acquired by the operation  $o$ , i.e.,  $G_0 \cup G((\mathcal{F}_o^{\text{tok}}(\sigma))^\perp)$ . Informally, acquiring a token denies other replicas permissions to concurrently perform changes that require conflicting tokens.

We now use our proof rule to show that the operations in the banking application (Figure 4) preserve the integrity invariant (5). We assume that the initial state  $\sigma_{\text{init}}$  satisfies the invariant. The guarantees are as follows:

$$\begin{aligned} G(\tau) &= \{(\sigma, \sigma') \mid 0 \leq \sigma' < \sigma\}; \\ G_0 &= \{(\sigma, \sigma') \mid 0 \leq \sigma \leq \sigma'\}. \end{aligned} \quad (12)$$

Since withdrawals acquire the token  $\tau$ , the guarantee  $G(\tau)$  for this token allows decreasing the balance without turning it negative; the guarantee  $G_0$  allows increasing a non-negative balance. Then condition S2 is satisfied. We show how to check the condition S3 in the most interesting case of  $o = \text{withdraw}(a)$ . Consider  $\sigma$  and  $\sigma'$  satisfying the premiss of S3:

$$\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\text{tok}}(\sigma))^\perp))^*.$$

Since  $\mathcal{F}_o^{\text{tok}}(\sigma) = \{\tau\}$ , we have  $(\mathcal{F}_o^{\text{tok}}(\sigma))^\perp = \emptyset$ . Thus,  $(\sigma, \sigma') \in G_0^*$ . This and  $\sigma \in I$  imply

$$0 \leq \sigma \leq \sigma'. \quad (13)$$

If  $\sigma < a$ , then  $\mathcal{F}_o^{\text{eff}}(\sigma)(\sigma') = \sigma'$ . Furthermore,  $\sigma' \geq 0$  by (13). Thus,  $(\sigma', \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma')) = (\sigma', \sigma') \in G_0$ , which implies the conclusion of S3.

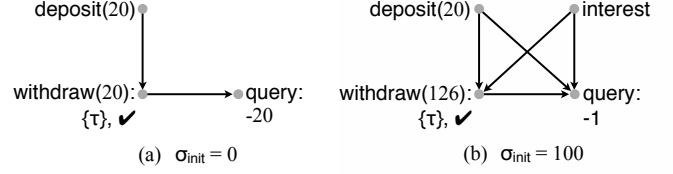
If  $\sigma \geq a$ , then  $\mathcal{F}_o^{\text{eff}}(\sigma)(\sigma') = \sigma' - a$ . Since  $\sigma \leq \sigma'$  by (13), we have  $\sigma' \geq a$ . Thus,  $(\sigma', \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma')) = (\sigma', \sigma' - a) \in G(\{\tau\})$ , which implies the conclusion of S3. Operationally, in this case our proof rule establishes that, if there was enough money in the account at the replica where the withdrawal was made, then there will be enough money at any replica the withdrawal is delivered to. This completes the proof of our example.

In a banking application with multiple accounts, we could ensure non-negativity of balances by associating every account  $c$  with a token  $\tau_c$  such that  $\tau_c \bowtie \tau_c$ , but  $\tau_c \not\bowtie \tau_{c'}$  for another account  $c'$ . Thus, withdrawals from the same account would have to synchronise, while withdrawals from different accounts could proceed without synchronisation. Our proof rule easily deals with this generalisation by associating every token  $\tau_c$  with a guarantee describing the changes to the corresponding account. As we elaborate in §6, the banking application we verify with the aid of our tool allows multiple accounts. There we also provide more complex examples of using our proof rule. For now, it is instructive to see how the proof rule is specialised for some of the simpler instantiations of our consistency model from §3.

**Sequential consistency.** Recall that for sequential consistency,  $\bowtie = \{(\tau, \tau)\}$  and we always have  $\mathcal{F}_o^{\text{tok}}(\sigma) = \{\tau\}$ , so that  $(\mathcal{F}_o^{\text{tok}}(\sigma))^\perp = \emptyset$ . Let  $G_0 = \emptyset$ , so that we always have  $\sigma = \sigma'$  in S3. Then S2 and S3 require us to find  $G(\tau)$  such that

$$G(\tau)(I) \subseteq I \wedge \forall o, \sigma. (\sigma \in I \implies (\sigma, \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma)) \in G(\tau)).$$

It is easy to show that we can find such a  $G(\tau)$  if and only if (10) holds for all  $o$ . Thus, in this case it is sufficient to check that the



**Figure 7.** Executions illustrating the unsoundness of the state-based proof rule on weaker consistency models.

effect of an operation preserves the invariant when applied to the same state where it was generated.

**Causal consistency.** We have  $\text{Token} = \emptyset$  and the conditions S2 and S3 become equivalent to

$$\begin{aligned} G_0(I) &\subseteq I \wedge (\forall o, \sigma, \sigma'. (\sigma \in I \wedge (\sigma, \sigma') \in G_0^*) \\ &\implies (\sigma', \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma')) \in G_0). \end{aligned}$$

In this case the effects of all operations are described by a single guarantee relation  $G_0$ . We need to show that every operation satisfies this guarantee while assuming that concurrently executing operations at other replicas do. Note that (11), for all  $o$ , is a special case of the above obligation for  $G_0 = I \times I$ . Thus, (11) is an invariant-based version of the above rely-guarantee proof rule.

As we elaborate in §7, our proof rule bears a lot of similarity to proof rules for strongly consistent shared-memory concurrency [21, 27, 36]. The reasons for the soundness of our proof in the setting of weak consistency are subtle. Its soundness relies crucially on the fact that our consistency model guarantees at least causal consistency and on the commutativity of operation effects (7). For example, some consistency models do not guarantee the transitivity of happens-before [8, 47] and thus allow the execution in Figure 7(a), which uses the operations in Figure 4. Here a withdrawal hb-follows a deposit; a query sees only the withdrawal, thus violating the integrity invariant (5). Since we have proved these operations to preserve the invariant using our proof rule, this rule is unsound over a consistency model allowing the execution in Figure 7(a). We note that the obligation (11), for all  $o$ , establishes the invariant  $I$  even for a consistency model where hb is only acyclic, but not necessarily transitive.

To illustrate that our rule becomes unsound if we drop the requirement of effect commutativity (7), consider the operations in Figure 4, but with the effect of interest defined by (3). It is easy to show that the premiss of the rule holds for the invariant (5) even with this change. At the same time, the execution in Figure 7(b) violates the invariant, yet is consistent with the operations in Figure 4 according to Definition 2. This is because the evaluation determining the effect of  $\text{withdraw}(126)$  can order  $\text{deposit}(20)$  before interest, whereas the evaluation determining the outcome of query can order these operations the other way round, resulting in a smaller balance. Again, the obligation (11) establishes the invariant even without (7): it ensures

$$\forall X \in \text{Exec}(\mathcal{T}, \mathcal{F}). \text{eval}_{\mathcal{F}}^{\dagger}(X) \subseteq I.$$

## 5. Event-based Proof Rule and Soundness

We now prove the soundness of the state-based proof rule. To this end, we present an *event-based* proof rule (Figure 8), from which the state-based one is derived. This event-based rule highlights the reasons for the soundness of the state-based one. Instead of reasoning about replica states, the event-based rule reasons about executions describing the events that replicas know about; the evaluation of the corresponding effects yields the replica states in the

$$\begin{array}{l}
\exists \mathbb{G} \in \mathcal{P}(\text{Exec}(\mathcal{T}) \times \text{Exec}(\mathcal{T})) \text{ such that} \\
\text{E1. } X_{\text{init}} \in \mathbb{I} \\
\text{E2. } \mathbb{G}(\mathbb{I}) \subseteq \mathbb{I} \\
\text{E3. } \forall X, X', X''. \forall e \in X''.E. \\
\quad (X \in \mathbb{I} \wedge X' = X''|_{X''.E - \{e\}} \wedge X'' \in \text{Exec}(\mathcal{T}, \mathcal{F}) \wedge \\
\quad e \in \max(X'') \wedge X = \text{ctxt}(e, X'') \wedge (X, X') \in \mathbb{G}^*) \\
\quad \implies (X', X'') \in \mathbb{G} \\
\hline
\text{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}
\end{array}$$

**Figure 8.** Event-based proof rule.

state-based rule. In particular, we specify the desired integrity invariant as a predicate on executions:  $\mathbb{I} \subseteq \text{Exec}(\mathcal{T})$ . The event-based rule establishes that any execution consistent with given  $\mathcal{T} = (\text{Token}, \bowtie)$  and  $\mathcal{F}$  belongs to  $\mathbb{I}$ :  $\text{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$ .

As before, we explain the event-based rule from the operational perspective. The rule again uses rely-guarantee reasoning, but with the guarantee  $\mathbb{G}$  represented by a relation on executions. The guarantee describes the change to a replica's knowledge brought on by the replica executing a new operation or receiving the effect of an operation originally executed elsewhere.

Conditions E1 and E2 are similar to S1 and S2: E1 requires the invariant  $\mathbb{I}$  to allow an empty execution  $X_{\text{init}}$  (§3), which evaluates to the initial database state  $\sigma_{\text{init}}$ ; E2 requires the guarantee to preserve the invariant. Condition E3 is graphically illustrated in Figure 6(b). Similarly to S3, the condition E3 considers any operation, denoted by an event  $e$ , and checks that the change to the database state made by the operation satisfies the guarantee. This check is done not only at the origin replica  $r$  of  $e$ , but also at any other replica  $r'$  that receives its effect. The execution  $X$  can be thought of as describing the events known to the replica  $r$  when it executed the operation denoted by  $e$ . We assume that the execution  $X$  satisfies the invariant  $\mathbb{I}$ . The execution  $X'$  describes the events known to the replica  $r'$  just before it receives the effect of  $e$ ;  $X''$  describes the events known to  $r'$  after this, so that  $X' = X''|_{X''.E - \{e\}}$ . The execution  $X''$  is consistent with  $\mathcal{T}$  and  $\mathcal{F}$ ; the conditions in the proof rule imply that so are  $X$  and  $X'$ . The condition  $e \in \max(X'')$  (see (9)) reflects the fact that  $e$  is the latest event received by  $r'$ . The condition  $X = \text{ctxt}(e, X'')$  ensures that  $X$  is a part of  $X' = X''|_{X''.E - \{e\}}$ . This reflects the guarantee of causal message propagation: when  $r'$  receives the effect of  $e$ , this replica is guaranteed to know about all the events that the replica  $r$  knew about when it executed  $e$ .

Even though the rule allows us to assume that  $X$  is part of  $X'$ , the latter may contain additional events that the replica  $r'$  found out about by the time it received the effect of  $e$ . The rule allows us to assume that the changes in the knowledge of  $r'$  brought on by adding these events satisfy the guarantee:  $(X, X') \in \mathbb{G}^*$ . In exchange, the rule requires us to ensure that adding the event  $e$  to the knowledge of replica  $r'$  will also satisfy the guarantee:  $(X', X'') \in \mathbb{G}$ .

In the following, we use the fact that the premiss of the implication in E3 entails that all events in  $X'.E - X.E$  are causally independent with  $e$ .

**PROPOSITION 5.** *For all  $X, X', X''$  and  $e \in X''.E$ ,*

$$\begin{array}{l}
(X' = X''|_{X''.E - \{e\}} \wedge e \in \max(X'') \wedge X = \text{ctxt}(e, X'')) \\
\implies \neg \exists f \in (X'.E - X.E). (e \xrightarrow{X''.\text{hb}} f \vee f \xrightarrow{X''.\text{hb}} e).
\end{array}$$

**PROOF.** Consider  $f \in (X'.E - X.E)$ . Since  $e \in \max(X'')$ , we cannot have  $e \xrightarrow{X''.\text{hb}} f$ . If  $f \xrightarrow{X''.\text{hb}} e$ , then  $f \in X.E$  due to  $X = \text{ctxt}(e, X'')$ . But this contradicts  $f \in (X'.E - X.E)$ .  $\square$

We now give the proof of soundness of the event-based rule and sketch the derivation of the state-based one (we give a full proof of the latter in [26, §A]).

Let  $\sqsubseteq$  be the following partial order on executions:

$$X \sqsubseteq X' \iff (X = X'|_{X.E} \wedge ((X'.\text{hb})^{-1})(X.E) \subseteq X.E). \quad (14)$$

When  $X \sqsubseteq X'$ , we say that  $X$  is a *causal cut* of  $X'$ ; any event is included into  $X$  together with its causal dependencies in  $X'$ . Operationally,  $X \sqsubseteq X'$  means that  $X$  and  $X'$  can describe the knowledge of a replica at different points in the same database computation.

**PROPOSITION 6.**

$$\forall X \in \text{Exec}(\mathcal{T}, \mathcal{F}). \forall Y. (Y \sqsubseteq X \implies Y \in \text{Exec}(\mathcal{T}, \mathcal{F})).$$

**THEOREM 7.** *The event-based proof rule in Figure 8 is sound.*

**PROOF.** Assume E1-E3 hold. We prove that

$$\forall X'' \in \text{Exec}(\mathcal{T}, \mathcal{F}). \forall Y. (Y \sqsubseteq X'' \implies (Y, X'') \in \mathbb{G}^*), \quad (15)$$

i.e., that the guarantee  $\mathbb{G}$  allows us to transition into a consistent execution  $X''$  from any of its causal cuts  $Y$ . The desired conclusion  $\text{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$  follows from (15): it implies  $(X_{\text{init}}, X'') \in \mathbb{G}^*$ , but  $X_{\text{init}} \in \mathbb{I}$  (E1) and  $\mathbb{G}$  preserves  $\mathbb{I}$  (E2).

The proof of (15) is done by induction on the size of  $X''$ . In the base case, we must have  $Y = X'' = X_{\text{init}}$ , which implies  $(Y, X'') \in \mathbb{G}^*$ . In the induction step, we consider  $X'' \in \text{Exec}(\mathcal{T}, \mathcal{F})$  and  $Y \sqsubseteq X''$  such that  $Y \neq X''$ . We pick an event  $e \in (X''.E - Y.E)$  such that  $e \in \max(X'')$  and define  $X$  and  $X'$  as in E3:

$$X = \text{ctxt}(e, X'') \wedge X' = X''|_{X''.E - \{e\}}.$$

Then

$$Y \sqsubseteq X' \wedge X \sqsubseteq X'. \quad (16)$$

By Proposition 6 we have  $X, X' \in \text{Exec}(\mathcal{T}, \mathcal{F})$ . Thus, we can apply the induction hypothesis to  $X'$  and its causal cuts  $X$  and  $Y$ , as well as to  $X$  and its causal cut  $X_{\text{init}}$ , getting:

$$(Y, X') \in \mathbb{G}^* \wedge (X, X') \in \mathbb{G}^* \wedge (X_{\text{init}}, X) \in \mathbb{G}^*.$$

By E1 and E2,  $(X_{\text{init}}, X) \in \mathbb{G}^*$  implies  $X \in \mathbb{I}$ . Together with  $(X, X') \in \mathbb{G}^*$ , this allows us to apply E3 and obtain  $(X', X'') \in \mathbb{G}$ . This and  $(Y, X') \in \mathbb{G}^*$  imply  $(Y, X'') \in \mathbb{G}^*$ , as required.  $\square$

In operational terms, the statement (15) established in the proof ensures that any sequence of changes in the knowledge of a replica during a database computation is described by  $\mathbb{G}^*$ . The above proof relies crucially on the fact that our consistency model guarantees at least causal consistency. For example, in (16) we can deduce  $X \sqsubseteq X'$  from  $X = \text{ctxt}(e, X'')$  because happens-before is transitive.

**COROLLARY 8.** *The state-based proof rule in Figure 5 is sound.*

**PROOF SKETCH.** Assume a state-based invariant  $I \subseteq \text{State}$ . We construct the corresponding event-based invariant  $\mathbb{I}$  as the set of all executions that evaluate to a state in  $I$ :  $\mathbb{I} = \text{eval}_{\mathcal{F}}^{-1}(I)$ . Then the conclusion  $\text{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$  of the event-based rule implies the conclusion  $\text{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \text{eval}_{\mathcal{F}}^{-1}(I)$  of the state-based rule.

We now show that the premiss of the state-based rule implies that of the event-based rule. Assume state-based guarantees  $G_0$  and  $G$  that satisfy S1-S3. We construct the corresponding event-based guarantee  $\mathbb{G}$  by describing the change to the knowledge of a replica



brought on by incorporating the effect of an operation satisfying the state-based guarantees  $G_0$  and  $G$ :

$$\mathbb{G} = \{(X, Y) \mid \exists e. (Y.E - X.E) = \{e\} \wedge X \sqsubseteq Y \wedge (\text{eval}_{\mathcal{F}}(X), \text{eval}_{\mathcal{F}}(Y)) \in G_0 \cup G(Y.\text{tok}(e))\}. \quad (17)$$

Thus, the guarantee  $\mathbb{G}$  consists of pairs  $(X, Y)$ , where  $Y$  extends  $X$  by a single event  $e$  representing the operation, and the two executions evaluate to a pair of states in  $G_0$  or  $G(\tau)$  for some token  $\tau$  acquired by  $e$ .

It remains to prove that the event-based guarantee  $\mathbb{G}$  satisfies conditions E1-E3. Conditions E1 and E2 trivially follow from conditions S1 and S2; we thus only need to show that S3 implies E3. Assume that for some  $X, X', X''$  and  $e \in X''.E$ , the premiss of E3 holds:

$$X \in \mathbb{I} \wedge X' = X''|_{X''.E - \{e\}} \wedge X'' \in \text{Exec}(\mathcal{T}, \mathcal{F}) \wedge e \in \max(X'') \wedge X = \text{ctxt}(e, X'') \wedge (X, X') \in \mathbb{G}^*. \quad (18)$$

Let  $\sigma = \text{eval}_{\mathcal{F}}(X)$ ,  $\sigma' = \text{eval}_{\mathcal{F}}(X')$  and  $\sigma'' = X''.\text{oper}(e)$ . We now show that the premiss of S3 holds:

$$\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\text{tok}}(\sigma))^{\perp}))^*. \quad (19)$$

First of all,  $\sigma \in I$  follows from  $X \in \mathbb{I}$  by the definition of  $\mathbb{I}$ . Furthermore, by Proposition 5, all events in  $(X'.E - X.E)$  are unrelated to  $e$  in  $(X''.\text{hb} \cup (X''.\text{hb})^{-1})$ . But then by (8), they cannot acquire tokens that conflict with the ones acquired by  $e$ :

$$\forall f \in (X'.E - X.E). \neg(X''.\text{tok}(e) \bowtie X''.\text{tok}(f)).$$

Using this fact,  $(X, X') \in \mathbb{G}^*$  given by (18) and the definition of  $\mathbb{G}$  given by (17), we can show that

$$(\sigma, \sigma') = (\text{eval}_{\mathcal{F}}(X), \text{eval}_{\mathcal{F}}(X')) \in (G_0 \cup G((X''.\text{tok}(e))^{\perp}))^* = (G_0 \cup G((\mathcal{F}_o^{\text{tok}}(\sigma))^{\perp}))^*,$$

thus establishing (19). Then the conclusion of S3 yields  $(\sigma', \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma')) \in G_0 \cup G(\mathcal{F}_o^{\text{tok}}(\sigma))$ , so that

$$\begin{aligned} (\text{eval}_{\mathcal{F}}(X'), \text{eval}_{\mathcal{F}}(X'')) &= (\sigma', \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma')) \\ &\in G_0 \cup G(\mathcal{F}_o^{\text{tok}}(\sigma)) \\ &= G_0 \cup G(X''.\text{tok}(e)). \end{aligned} \quad (20)$$

This implies the conclusion of E3:  $(X', X'') \in \mathbb{G}$ .  $\square$

The above proof relies crucially on Lemma 4, which allows us to define  $\text{eval}_{\mathcal{F}}$ . The lemma guarantees that, when evaluating executions, choosing different orders for causally independent events does not affect the resulting state. In (20) this allows us to choose a particular convenient order of evaluating  $X''$  that applies the operation  $o$  last. Lemma 4 holds due to the commutativity condition (7), and this illustrates the importance of this condition for the soundness of the state-based rule.

## 6. Examples and Automation

We have developed a prototype tool that automates the state-based proof rule by reducing its obligations to SMT queries. Using the tool, we have verified three applications: an extended version of the banking application in Figure 4, an auction service and a course registration system. Our results are summarised in Figure 9. In the following, we first show more sophisticated uses of our proof rule using fragments of the auction and courseware applications, as well as the consistency model of parallel snapshot isolation [38, 42]. We then present our automation approach and the complete applications that we verified.

### 6.1 Auction Service

Figure 10 shows a fragment of an auction application. An auction can be either open or closed. While the auction is open, a client can

Application	# ops	# tokens	# invariants	time (ms)
<i>Banking</i>	5	1	1	385
<i>Auction</i>	14	9	12	5297
<i>Courseware</i>	5	5	2	512

**Figure 9.** Characteristics of the applications verified and the time taken by the tool. The numbers of operations are given ignoring operation parameters. The numbers of tokens are similarly given without taking into account tokens associated with different instances of the same object, such as different bank accounts. The tool was run on a Mac Mini, 3 GHz Intel Core i7.

State = $\mathcal{P}(\mathbb{N}) \times (\mathbb{N} \cup \{\perp\})$
$\sigma_{\text{init}} = (\emptyset, \perp)$
$I = \{(B, w) \mid w \neq \perp \implies B \neq \emptyset \wedge w = \max(B)\}$
Token = $\{\tau_c, \tau_p\}$
$\bowtie = \{(\tau_c, \tau_c), (\tau_c, \tau_p), (\tau_p, \tau_c)\}$
$\mathcal{F}_{\text{place}(b)}((B, w)) = \text{if } w \neq \perp \text{ then } (\mathbf{X}, \text{skip}, \{\tau_p\})$
$\quad \quad \quad \text{else } (\checkmark, (\lambda(B', w'). (B' \cup \{b\}, w')), \{\tau_p\})$
$\mathcal{F}_{\text{close}}((B, w)) = \text{if } (w \neq \perp \vee B = \emptyset) \text{ then } (\mathbf{X}, \text{skip}, \{\tau_c\})$
$\quad \quad \quad \text{else } (\checkmark, (\lambda(B', w'). (B', \max(B))), \{\tau_c\})$
$\mathcal{F}_{\text{query}}((B, w)) = ((B, w), \text{skip}, \emptyset)$

**Figure 10.** A fragment of an auction application.

place a bid with the amount  $b$  using the  $\text{place}(b)$  operation. A client can also close the auction at any time using the  $\text{close}$  operation, which declares the winner. Finally, clients can query the database state using  $\text{query}$ .

The database state is of the form  $(B, w)$ . Here  $B$  contains the amounts of the bids placed; for simplicity, we do not distinguish two bids with the same amount. The component  $w$  is either  $\perp$ , signifying that the auction is still open, or the winning bid. A successful  $\text{place}(b)$  operation has the effect of adding  $b$  to  $B$ . The  $\text{close}$  operation writes the winning bid into  $w$ . Note that the effects of two  $\text{close}$  operations do not commute. To satisfy (7), and to ensure that clients can only close the auction once, we let  $\text{close}$  operations acquire a token  $\tau_c$  such that  $\tau_c \bowtie \tau_c$ .

The integrity invariant  $I$  we would like to maintain in the auction application is that, if the auction is closed, then the winning bid is the maximal of all the bids placed. Without using any other tokens than  $\tau_c$ , this invariant can be violated: Alice can close the auction and declare the winner, e.g., 100, without being aware of a higher bid 105 placed concurrently by Bob. A query aware of both operations will return the bid set containing 105 and 100 but mark 100 as the winning bid in the set.

To preserve the invariant in the RedBlue consistency model (§3), we would have to use strong consistency for both  $\text{place}$  and  $\text{close}$  operations, i.e., let them acquire the mutually exclusive token  $\tau_c$ . To address this inefficiency, Bales et al. [9] proposed a hybrid model where consistency can be strengthened using *multi-level locks*, analogous to readers-writer locks from shared memory. In our example, we represent such a lock by a pair of tokens:  $\tau_c$ , introduced before, and  $\tau_p$ . Each  $\text{close}$  operation acquires  $\tau_c$ , and each  $\text{place}$  operation,  $\tau_p$ . We have  $\tau_c \bowtie \tau_p$ . Hence, for every pair of  $\text{close}$  and  $\text{place}(b)$  operations, either  $\text{close}$  is aware of the bid  $b$  and takes it into account when computing the winner, or  $\text{place}(b)$  is aware that the auction has been closed and, hence, does not place the bid. However, we do not have  $\tau_p \bowtie \tau_p$  and, hence, bid placements can be causally independent. In our analogy with a readers-writer lock, bid placements play the role of readers and closing the auction, the role of a writer.

$\text{State} = \mathcal{P}(\text{Student}) \times \text{RWset}(\text{Course}) \times \mathcal{P}(\text{Student} \times \text{Course})$   
 $\sigma_{\text{init}} = (\emptyset, \emptyset_{\text{RWset}}, \emptyset)$   
 $I = \{(S, C, E) \mid E \subseteq \mathcal{P}(S \times \text{contents}(C))\}$   
 $\text{Token} = \{\tau_{e(c)}, \tau_{r(c)} \mid c \in \text{Course}\}$   
 $\bowtie = \{(\tau_{e(c)}, \tau_{r(c)}), (\tau_{r(c)}, \tau_{e(c)}) \mid c \in \text{Course}\}$   
 $\mathcal{F}_{\text{register}(s)}((S, C, E)) =$   
 $(\perp, (\lambda(S', C', E'). (S' \cup \{s\}, C', E')), \emptyset)$   
 $\mathcal{F}_{\text{addCourse}(c)}((S, C, E)) =$   
 $(\perp, (\lambda(S', C', E'). (S', \text{add}(c, C'), E')), \emptyset)$   
 $\mathcal{F}_{\text{enrol}(s, c)}((S, C, E)) =$   
 if  $(s \notin S \vee c \notin \text{contents}(C))$  then  $(\mathbf{X}, \text{skip}, \{\tau_{e(c)}\})$   
 else  $(\checkmark, (\lambda(S', C', E'). (S', C', E' \cup \{(s, c)\})), \{\tau_{e(c)}\})$   
 $\mathcal{F}_{\text{remCourse}(c)}((S, C, E)) =$   
 if  $(c \notin \text{contents}(C) \vee \exists s. (s, c) \in E)$  then  $(\mathbf{X}, \text{skip}, \{\tau_{r(c)}\})$   
 else  $(\checkmark, (\lambda(S', C', E'). (S', \text{remove}(c, C'), E')), \{\tau_{r(c)}\})$   
 $\mathcal{F}_{\text{query}}((S, C, E)) = ((S, \text{contents}(C), E), \text{skip}, \emptyset)$   
 $\text{RWset}(\text{Course}) = \mathcal{P}(\text{Course}) \times \mathcal{P}(\text{Course})$   
 $\emptyset_{\text{RWset}} = (\emptyset, \emptyset)$   
 $\text{add}(c, (A, T)) = (A \cup \{c\}, T)$   
 $\text{remove}(c, (A, T)) = (A, T \cup \{c\})$   
 $\text{contents}((A, T)) = A - T$

**Figure 11.** A fragment of a courseware application.

Balegas et al. [9] show how to implement multi-level locks so that a replica can place a bid without any synchronisation; only an operation closing the auction has to synchronise with other replicas to make sure that no bids are placed concurrently. Thus, the most frequent operation of bid placement is the least expensive.

We now use our proof rule to show that the above consistency choice is indeed sufficient to preserve the invariant  $I$ . Let

$$\begin{aligned}
 G_0 &= \{((B, w), (B, w)) \mid (B, w) \in I\}; \\
 G(\tau_p) &= \{((B, \perp), (B', \perp)) \mid B \subseteq B'\}; \\
 G(\tau_c) &= \{((B, \perp), (B, \max(B))) \mid B \neq \emptyset\}.
 \end{aligned}$$

Then the condition S2 in Figure 5 is satisfied. We show how to check the condition S3 in the most interesting case of  $o = \text{place}(b)$ .

Consider  $\sigma = (B, w)$  and  $\sigma' = (B', w')$  satisfying the premiss of S3. Then  $\sigma \in I$ . Also, since

$$(\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\text{tok}}(\sigma))^\perp))^*$$

and  $(\mathcal{F}_o^{\text{tok}}(\sigma))^\perp = \{\tau_p\}$ , we get

$$w' = w \wedge B \subseteq B' \wedge (w \neq \perp \implies B' = B). \quad (21)$$

If  $w \neq \perp$ , then  $\mathcal{F}_o^{\text{eff}}(\sigma)(\sigma') = \sigma'$  and, by (21),  $\sigma = \sigma'$ . Since  $\sigma \in I$ , we have

$$(\sigma', \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma')) = (\sigma', \sigma') = (\sigma, \sigma) \in G_0.$$

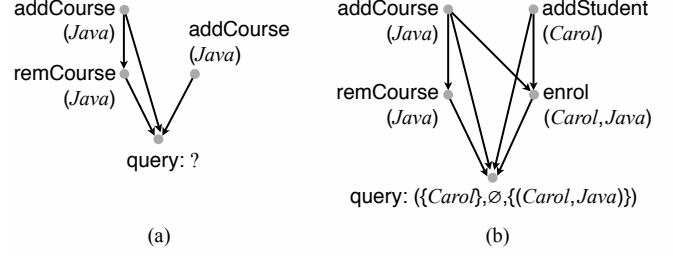
This implies the conclusion of S3.

If  $w = \perp$ , then  $\mathcal{F}_o^{\text{eff}}(\sigma)(\sigma') = (B' \cup \{b\}, w')$ . In this case (21) implies  $w' = w = \perp$ . We then get

$$(\sigma', \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma')) = ((B', w'), (B' \cup \{b\}, w')) \in (G_0 \cup G(\tau_p)),$$

the desired conclusion of S3. Operationally, our proof rule establishes that, if the auction was open at the replica where the bid was placed, then it will be open at any replica the bid is delivered to.

Similarly to our banking application (§4), we can deal with multiple auctions by using a pair of tokens  $(\tau_c, \tau_p)$  for every auction. The above proof generalises straightforwardly to this case.



**Figure 12.** Executions illustrating the need for (a) replicated data types and (b) tokens in the courseware application.

## 6.2 Courseware

Our next example illustrates a different kind of an integrity invariant and the use of replicated data types [40] to construct commutative operations. Figure 11 shows a fragment of a courseware application. We assume sets of courses  $\text{Course}$  and students  $\text{Student}$ . A client can add a course  $c$  using `addCourse(c)` and register a student  $s$  using `register(s)`. A registered student  $s$  can be enrolled into a course  $c$  using `enrol(s, c)`. In the application fragment we consider, student registrations and enrolments cannot be cancelled. However, a course  $c$  that has not secured any student enrolment can be removed using `remCourse(c)`. As usual, we also have a query operation.

A database state  $(S, C, E)$  consists of the set of students  $S$ , the set of courses  $C$  and the enrolment relation  $E$  between students and courses. The set of courses is actually not just an ordinary set, but a *replicated remove-wins set*  $\text{RWset}(\text{Course})$ , explained in the following. The effects of operations are mostly as expected, with courses accessed using special functions `add`, `remove` and `contents` on the replicated set. Note that the operation `enrol(c, s)` takes effect only if the student  $s$  is registered and the course  $c$  exists. The operation `remCourse(c)` removes the course  $c$  only when it exists and has no students enrolled into it.

Using a replicated data type for the set of courses is needed to satisfy (7), because additions to and removals from a usual set do not commute. To illustrate, consider the execution in Figure 12(a). There Alice adds a course on Java and then changes her mind and removes the course; concurrently, Bob adds the same Java course. If we maintained the information about courses using a usual set, then the outcome of the query in the figure would depend on the order in which we evaluate the effects of the causally independent operations `addCourse(Java)` by Bob and `remCourse(Java)` by Alice: the query would return  $\emptyset$  if the addition was evaluated before removal, and  $\{\text{Java}\}$  otherwise (see Definition 2). In an actual database, implementing the operations using ordinary sets would violate the replica convergence property (§2.1).

Replicated data types [40] provide implementations of operations on data structures with commutative effects. They differ in the way in which they resolve conflicting updates to the data structure, such as those in Figure 12(a): when using an *add-wins* set, the query in the figure will return  $\{\text{Java}\}$ , and when using a *remove-wins* set,  $\emptyset$  [39]. The decision which data type to use ultimately depends on application requirements. To keep presentation manageable, in our example we use one of the simplest set data types, which provides a rudimentary version of the remove-wins semantics.

The data type represents the replicated set of courses using a pair of sets  $(A, T)$ . The function `add(c, ·)` puts  $c$  into the set of  $A$ , and the function `remove(c, ·)` puts  $c$  into the set  $T$ , called the *tombstone* set. To get the contents of the replicated set, we just take the difference between  $A$  and  $T$ . The functions `add(c, ·)` and `remove(c, ·)` commute: even if the removal is evaluated first, it will

still cancel the subsequent addition<sup>1</sup>. This ensures that the effects of all operations in Figure 11 commute and thus satisfy (7).

The integrity invariant  $I$  we would like to maintain in this application is that the enrolment relation refers to existing courses and students only. This property is an instance of *referential integrity*, which requires an object referenced in one part of the database to exist in another. Without using tokens, the operations in our application can break the invariant. This is illustrated by the execution in Figure 12(b). There a Java course initially has no students enrolled. Then Alice removes the course and concurrently Bob enrolls Carol into it, thinking that the course is still available. This results in Carol being enrolled into a non-existent course.

To ensure that such situations do not happen, we use a pair of conflicting tokens for each course  $c \in \text{Course}$ :  $\tau_{e(c)}$  and  $\tau_{r(c)}$ . The operation  $\text{enrol}(s, c)$  acquires  $\tau_{e(c)}$ , and the operation  $\text{remCourse}(c)$  acquires  $\tau_{r(c)}$ . Then for every pair of operations  $\text{enrol}(s, c)$  and  $\text{remCourse}(c)$ , either the enrolment operation is aware that the course has been removed, or the removal is aware that there are still students enrolled into the course; in either case the corresponding operation takes no effect. However, other pairs of operations can be causally independent and, hence, do not have to synchronise. This includes pairs of operations enrolling students into courses and pairs of operations manipulating courses, such as those in Figure 12(a). The above use of tokens is equivalent to associating every course with a multi-level lock [9] that can be in one of two modes, one of which allows enrolling students into a course ( $\tau_{e(c)}$ ) and the other removing the course ( $\tau_{r(c)}$ ). Unlike in the auction application above, neither of the tokens  $\tau_{e(c)}$  or  $\tau_{r(c)}$  conflicts with itself, and thus, neither of the above lock modes is exclusive.

Our proof rule can establish that the above consistency choice is sufficient to preserve the integrity invariant. To this end, we use the following guarantees, associating changes with tokens as expected:

$$\begin{aligned} G_0 &= (I \times I) \cap \{((S, C, E), (S', C', E)) \mid \\ &\quad S \subseteq S' \wedge \text{contents}(C) \subseteq \text{contents}(C')\}; \\ G(\tau_{e(c)}) &= (I \times I) \cap \{((S, C, E), (S, C, E')) \mid \\ &\quad \exists s. E' = E \uplus \{(s, c)\}\}; \\ G(\tau_{r(c)}) &= (I \times I) \cap \{((S, C, E), (S, C', E)) \mid \\ &\quad \text{contents}(C) = \text{contents}(C') \uplus \{c\}\}. \end{aligned}$$

The actual proof is similar to that of the auction application above and is omitted.

### 6.3 Parallel Snapshot Isolation

We show that our generic consistency model (§3) can be instantiated to capture *parallel snapshot isolation (PSI)*, a consistency model recently proposed for replicated databases [38, 42], which strengthens causal consistency in a way different from the models we have considered so far. We then give a proof rule specific to PSI.

We assume that the database consists of a finite set  $\text{Obj}$  of objects, ranged over by  $x, y$ . The objects store values from a set  $\text{Val}$ , so that we let  $\text{State} = \text{Obj} \rightarrow \text{Val}$ . Operations in PSI perform computations that read and write objects, and any two operations writing to the same object acquire conflicting tokens. Thus, by (8) updates to the same object can never be concurrent, and the programmer does not have to merge them explicitly. However, PSI does not provide strong consistency, since it allows updates to *different* objects to be concurrent. For example, PSI allows the outcome in Figure 2(b) when the operations  $\text{add}(\text{postA})$  and  $\text{add}(\text{postB})$  write to different objects, e.g., representing the feeds of different users.

<sup>1</sup> In fact, once an element was removed, it can never be successfully added again, which may not be a desirable behaviour. There are replicated sets that provide a more sophisticated semantics [39].

$\exists G \in \mathcal{P}(\text{State} \times \text{State})$  such that

$$\begin{array}{l} \text{PSI1. } \sigma_{\text{init}} \in I \\ \text{PSI2. } G(I) \subseteq I \\ \text{PSI3. } \forall o, \sigma, \sigma'. (\sigma \in I \wedge (\sigma, \sigma') \in (G \cap (=_{\text{dom}(\mathcal{S}_o^{\text{updates}}(\sigma))})^*)) \\ \quad \implies (\sigma', \mathcal{F}_{S,o}^{\text{eff}}(\sigma)) \in G \\ \hline \text{PSIExec}(\mathcal{S}) \subseteq \text{eval}_{\mathcal{F}_S}^{-1}(I) \end{array}$$

**Figure 13.** Proof rule for PSI. For a set of objects  $\Omega$ , we define the relation  $(=_{\Omega})$  of type  $\text{State} \times \text{State}$  as follows: for all states  $\sigma, \sigma'$ ,  $\sigma =_{\Omega} \sigma'$  iff  $\forall x \in \Omega. \sigma(x) = \sigma'(x)$ .

To represent PSI in our framework, we consider a token system  $\mathcal{T}_{\text{PSI}} = (\text{Token}, \bowtie)$ , that associates every object  $x$  with a mutual exclusion token  $\tau_x$ :

$$\text{Token} = \{\tau_x \mid x \in \text{Obj}\}; \quad \tau_x \bowtie \tau_y \iff x = y.$$

To define the semantics of operations  $o \in \text{Op}$ , we assume a function

$$S \in \text{Op} \rightarrow (\text{State} \rightarrow \text{Val} \times (\text{Obj} \rightarrow \text{Val}))$$

and let

$$\forall o, \sigma. \mathcal{S}_o(\sigma) = (\mathcal{S}_o^{\text{val}}(\sigma), \mathcal{S}_o^{\text{updates}}(\sigma)).$$

Thus,  $\mathcal{S}_o^{\text{val}}(\sigma)$  gives the return value of an operation  $o$  executed in a state  $\sigma$ , and  $\mathcal{S}_o^{\text{updates}}(\sigma)$  gives the values that  $o$  writes to objects. We lift  $S$  to a function  $\mathcal{F}_S$  of the type (6) as follows:

$$\begin{aligned} \mathcal{F}_{S,o}(\sigma) &= (\mathcal{S}_o^{\text{val}}(\sigma), \\ &\quad (\lambda \sigma'. \lambda x. \text{if } x \in \text{dom}(S) \text{ then } S(x) \text{ else } \sigma'(x)), \\ &\quad \{\tau_x \mid x \in \text{dom}(S)\}), \end{aligned}$$

where  $S = \mathcal{S}_o^{\text{updates}}(\sigma)$ . Thus, the effect of an operation  $o$  is limited to writing the values specified by  $\mathcal{S}_o$ ; this is unlike the general case of our consistency model, which allows arbitrary effects. The acquired tokens are those for the objects written according to  $\mathcal{S}_o$ . Note that the effect specified by  $\mathcal{F}_{S,o}(\sigma)$  changes only the values of the objects written to by the operation, but the converse is not true: an operation can write to an object the same value it originally stored. This will still trigger token acquisitions and, hence, create causality relationships with other operations writing to the same object.

**PROPOSITION 9.** *For any  $\mathcal{S}$ , the function  $\mathcal{F}_S$  satisfies (7).*

The proof is given in [26, §A]. Note that  $\mathcal{F}_S$  does not always satisfy (4): the flexibility allowed by (7) is crucial to represent PSI as an instance of our generic consistency model.

We write  $\text{PSIExec}(\mathcal{S}) = \{X \mid X \models \mathcal{T}_{\text{PSI}}, \mathcal{F}_S\}$  for the set of all PSI executions with operation semantics given by  $\mathcal{S}$ . It is easy to show that this definition is equivalent to a recently-proposed declarative definition of PSI [17].

Figure 13 gives a proof rule for checking that operations  $\mathcal{S}$  preserve an integrity invariant  $I$  when executed on PSI. The rule requires us to specify the changes performed by all operations using a single guarantee  $G$ , which has to preserve  $I$  (condition PSI2). Condition PSI3 then requires us to check that the effect of an operation  $o$  generated in a state  $\sigma \in I$  satisfies the guarantee when applied to another state  $\sigma'$ . This state  $\sigma'$  can be assumed to result from  $\sigma$  by a finite number of changes allowed by the guarantee  $G$  that *do not modify the objects written by the operation  $o$* . Intuitively, the latter constraint comes from the fact that such operations acquire tokens conflicting with those of  $o$ .

**THEOREM 10.** *The rule in Figure 13 is sound.*

$\exists G_0 \in \mathcal{P}(\text{State} \times \text{State}), G \in \text{Token} \rightarrow \mathcal{P}(\text{State} \times \text{State})$   
such that

- T1.  $\sigma_{\text{init}} \in I$
  - T2.  $G_0(I) \subseteq I \wedge \forall \tau. G(\tau)(I) \subseteq I$
  - T3.  $\forall o. \exists T. \exists P_1, \dots, P_n, Q_1, \dots, Q_n \in \mathcal{P}(\text{State}).$
  - T3a.  $T = \bigcap \{ \mathcal{F}_o^{\text{tok}}(\sigma) \mid \sigma \in I \} \wedge$
  - T3b.  $I \subseteq \bigcup_{i=1}^n P_i \wedge$
  - T3c.  $\forall i = 1..n. P_i \subseteq Q_i \wedge$
  - T3d.  $(G_0 \cup G(T^\perp))(Q_i) \subseteq Q_i \wedge$
  - T3e.  $(Q_i \times (\mathcal{F}_o^{\text{eff}}(P_i)(Q_i))) \subseteq (G_0 \cup G(T))$
- 
- $\text{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \text{eval}_{\mathcal{F}}^{-1}(I)$

**Figure 14.** Proof rule used by our tool. We assume a token system  $\mathcal{T} = (\text{Token}, \bowtie)$  and use the same notation as in Figure 5. We let  $\mathcal{F}_o^{\text{eff}}(P_i)(Q_i) = \{ \mathcal{F}_o^{\text{eff}}(\sigma)(\sigma') \mid \sigma \in P_i \wedge \sigma' \in Q_i \}$ .

The proof, given in [26, §A], derives the rule for PSI directly from the event-based rule in Figure 8. We could derive the rule for PSI from a generalisation of the state-based rule in Figure 5 that associates guarantees with *sets* of tokens. However, to simplify presentation we opted for the simpler version of the state-based rule at the expense of a more complex derivation of the rule for PSI.

#### 6.4 Automation

Our tool uses the proof rule in Figure 14, which is derived from the one in Figure 5 and is more amenable to automation. The premisses T1 and T2 are identical to S1 and S2; T3 changes S3 in two ways. A minor change is motivated by the fact that our tool currently handles only operations that acquire the same set of tokens regardless of the state they are executed in. Hence, T3 precomputes the set of tokens  $T$  acquired by an operation  $o$  (T3a). The key way in which T3 changes S3 is that it eliminates the transitive closure of the guarantees, which is hard to automate. Whereas S3 quantifies over states  $\sigma$  where the effect of an operation  $o$  is generated and  $\sigma'$  where it is applied, T3 considers properties of these states, respectively denoted by predicates  $P_i$  and  $Q_i$ ,  $i = 1..n$ . T3b requires the predicates  $P_i$  to cover all possible states in which  $o$  can be executed. T3c requires  $Q_i$  to cover  $P_i$ , reflecting the fact that the effect of  $o$  can be applied in a state different from the one where it was generated. T3d requires  $Q_i$  to be *stable* under the changes allowed by the guarantees [27]. Finally, T3e checks that, if the effect of  $o$  is generated in a state satisfying  $P_i$ , then applying this effect to a state satisfying  $Q_i$  is consistent with the guarantees. Note that the constraints T3c and T3d have the same effect as relating the states  $\sigma$  and  $\sigma'$  in S3 by a transitive closure of guarantees.

For example, consider the operation  $o = \text{withdraw}(a)$  from the banking application in Figure 4. We let  $T = \{\tau\}$  and use the guarantees (12). We use two predicates:

$$P_1 = \{\sigma \mid \sigma \geq a\}; \quad P_2 = \{\sigma \mid 0 \leq \sigma < a\}.$$

These are motivated by the condition of the if-then-else in  $\mathcal{F}_o^{\text{eff}}$ , as well as the invariant  $I$ . We then let  $Q_1 = P_1$  and  $Q_2 = I$ . It is easy to check that the obligations in T3 are fulfilled.

Our tool accepts as input a token system  $\mathcal{T}$ , the semantics of operations  $\mathcal{F}$  and an integrity invariant  $I$ , the latter two in the SMT-LIB format (we leave a programming language for writing operations as future work). The tool generates predicates  $P_i$  from preconditions of branches in  $\mathcal{F}$ . As  $Q_i$ , the tool takes either  $P_i$  or the invariant  $I$ . Finally, the tool generates guarantees  $G_0$  and

$G$  by intersecting the semantics of operations  $\mathcal{F}$  with the invariant  $I$ . The required obligations are then discharged using the Z3 SMT solver [1]. Currently our tool assumes that the condition (7) of operation commutativity is fulfilled: checking commutativity automatically is nontrivial [28].

**Applications verified.** The applications verified using our tool (Figure 9) are more realistic versions of the examples we discussed before (Figures 4, 10 and 11).

The banking application extends the one in Figure 4 by considering multiple accounts and allowing clients to transfer money between accounts. We preserve the non-negativity of all balances by associating a mutual exclusion token with each account, as described in §4.

The auction application extends the one in Figure 10 by additionally maintaining information about buyers, sellers and products, and by allowing clients to sell multiple product items in a single auction. Buyers and sellers can register and unregister. Registered buyers can bid in open auctions, and registered sellers can add products, create auctions consisting of these and close auctions. The complex data model of this application requires multiple integrity invariants, including referential integrity constraints spanning multiple parts of the database. This makes it nontrivial to see if enough synchronisation has been added to the application to preserve these invariants, and our tool copes with this task.

The courseware application extends the one in Figure 11 by allowing clients to cancel student registrations and enrolments. It also imposes an additional integrity invariant limiting the number of students that can register for a course; maintaining this invariant requires extra synchronisation.

The above case studies demonstrate the feasibility of applying our proof rule to realistic applications.

## 7. Related Work

**Reasoning in strongly consistent shared memory.** Our state-based proof rule interprets tokens as permissions to perform certain state changes. Such interpretations have been used in various logics for strongly consistent shared memory [20, 21, 36]. For example, such a logic could allow threads to modify the memory in a particular way only when holding a mutual exclusion lock, similar to our use of a token in the banking application (§4).

This similarity suggests that existing work in shared memory may be helpful in exploring the novel area of replicated databases. However, the distributed and weakly consistent setting in which our proof rule is applied makes the reasons for its soundness subtle. In this setting, we do not have an illusion of a single copy of the database state and a global notion of time this copy would evolve with: as Figure 1(b) illustrates, different processes can see events as occurring in different orders. The usual justification for the soundness of the proof rules for strong consistency relies on the concepts of global time and state: when considering a thread holding a mutex lock, such proof rules reason that no other thread can hold the lock *at the same time* and, hence, modify *the* memory state in the way associated with the lock. In this setting, locks constrain the global order on events. In contrast, tokens in our consistency model provide a more subtle guarantee (8), only constraining the partial happens-before relation.

**Reasoning about consistency in distributed systems and databases.** Several papers have considered reasoning about correctness properties on weak consistency models of replicated and centralised databases.

Bailis et al. [7] have proposed a criterion for checking when an integrity invariant is preserved by running operations without using any synchronisation at all. But they do not provide guidelines on how to introduce synchronisation if the invariant is violated.

Li et al. [31, 32] have proposed a static analysis that uses the proof rule (11) to check if executing operations on causal consistency preserves a given integrity invariant. In case when (11) fails for some operation  $o$ , the analysis suggests to execute  $o$  under strong consistency in the RedBlue consistency model (§3). However, the analysis does not check that the result will indeed validate the invariant, and our proof rule fills this gap.

Sivaramakrishnan et al. [41] have proposed a static analysis that automatically chooses consistency levels in a replicated database given programmer-supplied contracts. However, these contracts are more low-level than our invariants, since they typically constrain the happens-before relation. For example, in the banking application (Figure 4) their contract requires happens-before to totally order all withdrawal operations. The static analysis then ensures that the contract is followed, but not that it ensures the integrity invariant (5).

Lu et al. [34] proposed proof rules for establishing correctness properties of transactions running on non-hybrid weak consistency models of classical relational databases, such as snapshot isolation [12]. In contrast, we concentrate on hybrid consistency models of modern replicated databases, which are more sophisticated.

Fekete [22] considered a hybrid consistency model for relational databases where some transactions execute under snapshot isolation [12] and some under serialisability, a form of strong consistency. He proposed conditions determining which transactions in an application need to execute under serialisability for the whole application to be *robust*, i.e., produce only behaviours that would be obtained by executing *all* transactions under serialisability. In contrast, our proof rule only checks integrity invariants while allowing the application to produce weakly consistent behaviours and, hence, benefit from the resulting performance gains.

**Weak memory models.** Strong consistency is forgone not only by modern databases, but also by shared-memory multiprocessors and programming languages, which provide *weak memory models*. All such models used in practice are hybrid, in that they allow the programmer to strengthen consistency on demand, e.g., using memory fences. However, weak memory models usually provide only a limited number of operations on data, such as reads, writes and compare-and-swaps on single memory cells. Concurrent writes to the same memory cell result in one value being overwritten by the other. In contrast, we deal with arbitrary operations (6) that merge concurrent updates in a user-defined way.

That said, in the future there may be a fruitful exchange of ideas between program logics for applications using weakly consistent databases and those running on weak memory models. In particular, there have been recent proposals of program logics for the “release-acquire” fragment of the C/C++ memory model [45, 46]. This fragment is analogous to causal consistency, with the above caveats about the operations allowed. However, the published logics do not meaningfully handle operations requesting the stronger “sequentially consistent” level of C/C++. Reasoning about on-demand requests of stronger-than-causal consistency is precisely the goal of the present paper.

Several papers [3–5, 13, 18, 19] have verified application correctness on weak memory models using model checkers and abstract interpreters. These papers thus explore verification approaches different from the one considered in this paper. Additionally, most of the papers have focused on models similar to TSO [3, 13, 18, 19], which is stronger than the causal consistency model we consider as a baseline. As the target correctness property, papers on weak memory models have often considered robustness (see above), which is too strong a requirement for our setting. On the other hand, some of the papers [3, 4, 13, 19] automatically inferred fences required to satisfy a correctness property. We do not address the inference of consistency choices, although in the future

our state-based proof rule can serve as a basis for this. In particular, the proof in Figure 6 can be used to refine a given conflict relation: when the stability check T3d fails, the relation can be extended to so as to shrink the set  $G(T^\perp)$  and make the check pass.

**Consistency models.** Our conflict relations are similar to those used by Pedone and Schiper [37] to specify constraints on message delivery in a broadcast algorithm. We use the conflict relations to define a high-level consistency model, which abstracts from a message-based database implementation.

In a position paper, Li et al. [30] independently proposed an idea of a hybrid consistency model similar to ours. Their model does not have a formal semantics and is less flexible than ours, since their analogue of the conflict relation is defined directly on operations, instead of indirectly using tokens. This does not allow the synchronisation mandated for an operation to depend on the state it is executed in and, hence, does not allow expressing parallel snapshot isolation (§6).

**Specifying consistency models.** The formal specification of our consistency model (§3) builds on a framework previously proposed to specify forms of eventual consistency [15]. Despite this similarity, we take a somewhat different approach to specifying the semantics of operations. Previous work [15] specified the return value of an event by an arbitrary function of its context in the execution (§3). In contrast, our Definition 2 uses a particular function  $\text{eval}_{\mathcal{F}}^1$ , itself constructed from more primitive functions  $\mathcal{F}_o^{\text{eff}}$ , operating on states. This choice allows us to define the semantics of operations in terms of states, as opposed to events, which can then be used in our state-based proof rule. The use of states also allows to use off-the-shelf SMT solvers to discharge the required verification conditions. However, it is likely that our event-based rule may be adapted to the operation specifications used in [15].

## 8. Conclusion and Future Work

We presented the first proof rule establishing that a given consistency choice in a replicated database is sufficient to preserve a given integrity invariant. Our proof rule is modular and simple to use. We demonstrated this by small but nontrivial examples, and by reducing the verification conditions of the proof rule to SMT checks. Despite this simplicity, the soundness of our proof rule is nontrivial: the rule fully exploits the guarantees provided by our consistency model while correctly accounting for anomalies it allows.

Our results represent only an initial step in building an infrastructure of reasoning methods for applications using modern replicated databases. They open several avenues for future work. First, our generic consistency model is not implemented by any database in its full generality; we use it only as a means to compactly represent a selection of more specific models in existing implementations. However, in the future the generic model can serve as a basis for exploring the space of possible hybrid consistency models. One could also consider a database that implements our model in its general form.

Second, the soundness of our proof rule relies on the fact that our consistency model guarantees at least causal consistency (§4). Even though causal consistency can be implemented without any synchronisation between replicas, this model has its cost [14]. In the future, we plan to propose proof rules for weaker models where causality preservation is not guaranteed for all operations. We also hope to generalise our methods to more expressive correctness properties than integrity invariants.

Finally, in this paper we used the event-based proof rule just to structure the proof of soundness of the state-based one. However, the event-based rule is also interesting in its own right. In the future it can be used in cases when, to prove a correctness property,

we need to maintain information about the computation history. For example, this is often necessary when reasoning about shared-memory concurrency [23, 25].

**Acknowledgements.** We thank Valter Balegas for helpful discussions and Vincent Gramoli for comments on an earlier draft of this paper. Gotsman was supported by an EU FET project ADVENT. Ferreira, Najafzadeh, and Shapiro were supported in part by an EU project SyncFree (FP7 609 551, 2013–2016). Yang was supported by EPSRC and an Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP, No. R0190-15-2011).

## References

- [1] <https://github.com/Z3Prover/z3>.
- [2] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.
- [3] P. A. Abdulla, M. F. Atig, and N. T. Phong. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP*, 2015.
- [4] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, 2014.
- [5] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, 2013.
- [6] Amazon. Supported operations in DynamoDB. <http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/APISummary.html>, 2015.
- [7] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 2015.
- [8] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.
- [9] V. Balegas, N. Preguiça, R. Rodrigues, S. Duarte, C. Ferreira, M. Najafzadeh, and M. Shapiro. Putting the consistency back into eventual consistency. In *EuroSys*, 2015.
- [10] Basho Inc. Using strong consistency in Riak. <http://docs.basho.com/riak/latest/dev/advanced/strong-consistency/>, 2015.
- [11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.
- [12] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.
- [13] A. Bouajjani, E. Derevenet, and R. Meyer. Checking and enforcing robustness against TSO. In *ESOP*, 2013.
- [14] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. E. T. Rodrigues. On the use of clocks to enforce consistency in the cloud. *IEEE Data Eng. Bull.*, 38(1), 2015.
- [15] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *POPL*, 2014.
- [16] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.
- [17] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR*, 2015.
- [18] A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Predicate abstraction for relaxed memory models. In *SAS*, 2013.
- [19] A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI*, 2015.
- [20] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.
- [21] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.
- [22] A. Fekete. Allocating isolation levels to transactions. In *PODS*, 2005.
- [23] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, 2010.
- [24] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.
- [25] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, 2013.
- [26] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems (extended version). Available from <http://software.imdea.org/~gotsman/>.
- [27] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*. North-Holland, 1983.
- [28] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *PLDI*, 2011.
- [29] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), 1979.
- [30] C. Li, J. Leitão, A. Clement, N. Preguiça, and R. Rodrigues. Minimizing coordination in replicated systems. In *Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*, 2015.
- [31] C. Li, J. Leitão, A. Clement, N. M. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX ATC*, 2014.
- [32] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *OSDI*, 2012.
- [33] 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 *SOSP*, 2011.
- [34] S. Lu, A. J. Bernstein, and P. M. Lewis. Correct execution of transactions at different isolation levels. *IEEE Trans. Knowl. Data Eng.*, 16(9), 2004.
- [35] Microsoft. Consistency levels in DocumentDB. <http://azure.microsoft.com/en-us/documentation/articles/documentdb-consistency-levels/>, 2015.
- [36] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3), 2007.
- [37] F. Pedone and A. Schiper. Generic broadcast. In *DISC*, 1999.
- [38] M. Saeida Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, 2013.
- [39] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.
- [40] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.
- [41] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI*, 2015.
- [42] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.
- [43] D. Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12), 2013.
- [44] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.
- [45] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, 2014.
- [46] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, 2013.
- [47] W. Vogels. Eventually consistent. *CACM*, 52(1), 2009.