Combining state- and event-based semantics to verify highly available programs

Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter

University of Kaiserslautern, Germany
{p_zeller,bieniusa,poetzsch}@cs.uni-kl.de

Abstract. Replicated databases are attractive for managing the data of distributed applications that require high availability, low latency, and high throughput. However, these benefits entail weak consistency which comes at a price: it becomes harder to reason about application correctness. We address this difficulty with a verification technique for highly available programs. We augment an existing sequential programming language with primitives for interacting concurrently with a highly available database and extend the state-based operational semantics of that language accordingly. To this end we make use of existing event-based database semantics.
We then present a reduction of the extended semantics to a simpler one, which is again sequential and therefore easier to handle in verification tools. Our verification tool *Repliss* uses this technique and demonstrates its feasibility.

1 Introduction

Replication is an essential mechanism for implementing highly available and scalable information systems like social networks, e-government, or e-commerce applications. One popular approach for building highly available apps is to delegate all synchronization aspects to a distributed database while the application processes are typically stateless. To achieve high availability even in the presence of network failures, the underlying databases have to weaken their consistency guarantees. This trade-off has been formalized in the CAP [9] and PACELC [1] theorems.

When prioritizing high availability over strong consistency, reasoning about the correctness of an application that utilizes a replicated distributed database becomes significantly more difficult as developers have to consider the consequences of concurrent updates of shared data and its implication for the guarantees when reading data. This complexity extends also to the verification effort to prove the correctness of such programs. Concurrent requests on the database give rise not only to interleaved executions within a replica, but also temporal divergence of replicas. These aspects are hard to tackle in a direct way such as induction over all possible executions. Another challenge is in the different approaches to formalize database semantics and programming languages. Consistency models of highly available databases are often formalized axiomatically using event graphs [7].

Programming language semantics are on the other hand often formalized using operational semantics.

In this paper, we present a novel semantics and verification technique for programs utilizing weakly-consistent databases. We combine axiomatic consistency models with operational semantics, by extending a sequential programming language with constructs for concurrent requests and primitives for accessing the database. This allows us to reduce the verification problem for distribution and replication to a sequential program with nondeterministic steps for simulating the possible effects of concurrent requests (Section 3). In Section 4, we present our soundness proof for this reduction which we have formally proven using Isabelle/HOL [19]. Our database semantics targets recent trends emerging in highly available databases, like convergent and commutative replicated data types (CRDTs) [20] and transactional causal$^+$ consistency (TCC$^+$) [16,2].

Finally, we have built a verification tool named *Repliss* (Section 5) which employs this reduction for partially automating the verification of highly available programs. The tool takes a program and a specification of functional properties as input. The functional properties may use history invariants, which can describe causal relations between several requests or the effects that requests have on the database state. Further, the user must also provide additional invariants to guide the prover. The tool then uses random testing to find counterexamples and symbolic execution to verify the absence of errors. As a case study, we demonstrate how Repliss can be used to verify a highly available chat application.

Before we present the details of our work, we show why existing verification approaches do not solve our problem.

Related Work. The challenge of weak consistency in verification is well known and has been approached with a number of different techniques. Weak memory models have been studied in depth in the context of concurrent programming for multi-core machines [8]. However, the techniques in this area usually target linearizability as a correctness criterion and employ hardware-supported synchronization mechanisms such as memory fences or CAS-operations. In distributed systems, it is neither feasible to consider linearizability as consistency notion nor to implement the same concurrency control mechanisms as in weak memory system. This precludes the direct applicability of these techniques to our scenario. In the following, we therefore focus on related work that shares our application domain.

Composite Replicated Data Types [10] allow to compose basic data types into application-specific data representations that are synchronized atomically. Their area of application is similar to our setting, though our approach is more widely applicable as we model procedures involving several transactions on arbitrary combinations of objects. More importantly, their approach is axiomatic and based on a denotational semantics, which is more difficult to adapt in a tool implementing the technique.

CISE [11,18] is a tool, which can automatically determine the procedures in an application, which require stronger consistency guarantees for correctness. This line of work focuses on combining weak consistency with strong synchronization

for some operations, whereas our work only considers weak consistency. CISE does not consider features like transactions or replicated data types directly. Instead, application procedures are assumed to have a single atomic effect which is applied on every replica asynchronously. This is similar to the implementation technique of operation-based CRDTs, where effects have to be commutative to ensure convergence. Soteria [17] is a similar tool which is based on state-based implementations of CRDTs instead. In contrast, our model handles data types as components with a high-level (axiomatic) specification and not their concrete implementation.

QUELEA [21] is another tool supporting the development of applications on top of weakly consistent databases. Unlike our approach and the previously discussed approaches, the specifications in QUELEA are not given as invariants. Instead, the user specifies constraints on the order between operations and the tool automatically chooses the necessary consistency level.

Q9 [12] is a symbolic execution engine for finding bugs in programs written on top of weakly consistent databases. The tool only supports bounded verification, where the number of concurrent effects is limited, so unlike Repliss, it cannot be used to prove the absence of errors in the general case. Weak consistency is modeled using commutative effects, which works well for symbolic execution, but is less suitable when working with invariants as we do.

Chapar [14] is a framework for verifying causally consistent, replicated databases and applications employing such databases. The development is formally verified using Coq and the goal of verifying application is similar to ours. Their approach is different, though. They have implemented a model checker for applications, thus providing automation. However, the kind of applications which can be analyzed is restricted, since the model checker can check all possible reorderings of one concrete execution where all parameters have fixed values.

None of the work discussed so far handles the integration of transactions into a technique to reason about programs. This aspect has been tackled in work in different contexts, for example in a program logic for handling Java Card's transaction mechanism [4]. Transactions in Java Card provide atomicity, but do not handle concurrency. We are not aware of other work integrating weakly consistent transactions into a verification technique.

2 A Formal Semantics for Highly Available Programs

In this section, we extend the operational semantics of an imperative core calculus with primitives for concurrent procedure invocations and database interactions. Procedures define the external interface (API) of a program. Clients invoke the procedures of a program which are then processed concurrently.

In highly available systems, operations must be able to progress even if only the local replica is available. This is reflected in our system model, where database operations are executed locally first and propagated to other replicas asynchronously.

Figure 1 shows the definitions regarding system state that we use in our formalization. In the initial state, all fields are empty maps or sets. We grouped the fields by the different aspects of our semantics, which we explain in detail below. To support database operations, we model the database state using event graphs including information about all database calls, the happens-before relation between them and the respective transactions. For procedure invocations and the sequential semantics we keep the local state and the currently active procedure per invocation. Furthermore, we record the history of procedure invocations to make them available in specifications. Finally, we add direct support for generating and using unique identifiers. Besides these aspects, we also explain how we handle partial failures and invariants when we explain the rules below (these do not require fields in the system state).

Database operations:
- $call : callId \hookrightarrow callInfo$
- $happensBefore : (callId \times callId)\ set$
- $visibleCalls : invocId \hookrightarrow callId\ set$
- $currentTransaction : invocId \hookrightarrow txid$
- $callOrigin : callId \hookrightarrow txid$
- $txStatus : txid \hookrightarrow txStatus$
- $transactionOrigin : txid \hookrightarrow invocId$

Procedure invocations:
- $localState : invocId \hookrightarrow localState$
- $currentProc : invocId \hookrightarrow procedureImpl$

History Recording:
- $invocationOp : invocId \hookrightarrow (procName \times any\ list)$
- $invocationRes : invocId \hookrightarrow any$

Unique Identifiers:
- $generatedIds : any \hookrightarrow invocId$
- $knownIds : any\ set$

Programs:
- $querySpec : (operationContext \times operation \times any\ list \times res) \to bool$
- $procedure : (procName \times any\ list) \hookrightarrow (localState \times procedureImpl)$
- $invariant : invariantContext \to bool$

Fig. 1. Fields of the system state. We use $\times$ for product types, $\hookrightarrow$ for map types (functions returning an option type), $\tau\ set$ for sets containing elements of type τ, and $\tau\ list$ for lists. The type *any* is the type we use for arbitrary values used in the program. Other relevant types are explained in the text.

The rules of our semantics are shown in Figure 2. We write $S \xrightarrow{i,a} S'$ to denote that the system makes a step from state S to S' by executing action a in procedure invocation i. In every step a different invocation can progress, resulting in a fine-grained interleaving semantics. $S \xrightarrow{tr}{}^{*} S'$ denotes the reflexive, transitive closure with the trace tr. A trace is a sequence of $(invocId, action)$ pairs.

Each rule describes the complete effect of a single action which includes some orthogonal aspects of our semantics. In the following we therefore describe the different aspects and how they manifest in the rules.

$$\frac{}{S \xrightarrow{\epsilon}{}^* S}\ (steps\text{-}empty) \qquad \frac{S_1 \xrightarrow{tr}{}^* S_2 \qquad S_2 \xrightarrow{i,a} S_3}{S_1 \xrightarrow{tr\cdot(i,a)}{}^* S_3}\ (steps)$$

$$\frac{\begin{array}{c} procedure_{prog}(procName, args) \triangleq (initialState, impl) \\ uniqueIdsInList(args) \subseteq knownIds(S) \qquad invocationOp(S,i) = \bot \end{array}}{S \xrightarrow{i,invoc(procName,args)} S \begin{bmatrix} currentProc(S,i) := impl \\ localState(S,i) := initialState \\ invocationOp(S,i) := (procName, args) \end{bmatrix}}\ (invocation)$$

$$\frac{\begin{array}{c} localState(S,i) \triangleq ls \\ currentProc(S,i) \triangleq f \qquad f(ls) = return(res) \qquad currentTransaction(S,i) = \bot \end{array}}{S \xrightarrow{i,return(res)} S \begin{bmatrix} localState(S,i) := \bot \\ invocationRes(S,i) := res \\ knownIds(S) := knownIds(S) \cup uniqueIds(res) \end{bmatrix}}\ (return)$$

$$\frac{localState(S,i) \triangleq ls \qquad currentProc(S,i) \triangleq f \qquad f(ls) = localStep(ls')}{S \xrightarrow{i,local} S\left[localState(S,i) := ls'\right]}\ (local)$$

$$\frac{\begin{array}{c} localState(S,i) \triangleq ls \\ currentProc(S,i) \triangleq f \qquad f(ls) = beginAtomic(ls') \qquad currentTransaction(S,i) = \bot \\ txStatus(S,t) = \bot \qquad visibleCalls(S,i) \triangleq vis \qquad newTxns \subseteq committedTransactions(S) \\ newCalls = callsInTransaction(S, newTxns) \downarrow_{happensBefore(S)} \qquad snapshot = vis \cup newCalls \end{array}}{S \xrightarrow{i,beginAtomic(t,newTxns)} S \begin{bmatrix} localState(S,i) := ls' \\ currentTransaction(S,i) := t \\ txStatus(S,t) := uncommitted \\ transactionOrigin(S,t) := i \\ visibleCalls(S,i) := snapshot \end{bmatrix}}\ (atomic)$$

$$\frac{\begin{array}{c} localState(S,i) \triangleq ls \\ currentProc(S,i) \triangleq f \qquad f(ls) = endAtomic(ls') \qquad currentTransaction(S,i) \triangleq t \end{array}}{S \xrightarrow{i,endAtomic} S \begin{bmatrix} localState(S,i) := ls' \\ currentTransaction(S,i) := \bot \\ txStatus(S,t) := committed \end{bmatrix}}\ (commit)$$

$$\frac{\begin{array}{c} localState(S,i) \triangleq ls \\ currentProc(S,i) \triangleq f \qquad f(ls) = dbOperation(ls', op, args) \qquad currentTransaction(S,i) \triangleq t \\ call(S,c) = \bot \qquad querySpec_{prog}(operationContext(S,i), op, args, res) \qquad visibleCalls(S,i) \triangleq vis \end{array}}{S \xrightarrow{i,dbOp(c,op,args,res)} S \begin{bmatrix} localState(S,i) := ls'(res) \\ call(S,c) := (op, args, res) \\ callOrigin(S,c) := t \\ visibleCalls(S,i) := vis \cup \{c\} \\ happensBefore(S) := happensBefore(S) \cup (vis \times \{c\}) \end{bmatrix}}\ (DB\text{-}operation)$$

$$\frac{\begin{array}{c} localState(S,i) \triangleq ls \qquad currentProc(S,i) \triangleq f \\ f(ls) = newId(ls') \qquad generatedIds(S,uid) = \bot \qquad uniqueIds(uid) = \{uid\} \qquad ls'(uid) \triangleq ls'' \end{array}}{S \xrightarrow{i,newId(uid)} S \begin{bmatrix} localState(S,i) := ls'' \\ generatedIds(S,uid) := i \end{bmatrix}}\ (new\text{-}id)$$

$$\frac{}{S \xrightarrow{i,crash} S\left[localState(S,i) := \bot\right]}\ (crash) \qquad \frac{res = invariant_{prog}(invContext(S))}{S \xrightarrow{i,invCheck(txns,\ res)} S}\ (inv)$$

Fig. 2. Interleaving semantics. We use $x \triangleq y$ as an abbreviation for $x = Some(y)$ for option types.

Procedure invocations. In our semantics, a procedure invocation is triggered by an application request from some client. Clients may invoke procedures concurrently, but each single invocation executes sequentially.

Programs are modeled with a partial function (field *procedure*) that takes the procedure name and the arguments of the invocation and, if the procedure is defined for the given arguments, returns the initial local state of the procedure and its implementation. The implementation (*procedureImpl*) is given by a function which calculates the next action based on the current invocation state. Possible actions are local evaluations (*local*), generating unique identifiers (*newId*), database related actions (*beginAtomic*, *endAtomic*, *dbOp*) and returning from an invocation (*return*). In the system state, we use the fields *currentProc* to store the implementation and the field *localState* to store the local invocation state.

The rule *invocation* describes the start of a procedure invocation. The precondition of the rule enforces that the procedure is defined for the given arguments. The remaining aspects of this rule are related to tracking the history and handling of unique identifiers (see below). The *return* rule is similar.

For local actions in a procedure invocation we have the rule (*local*), which subsumes the standard sequential semantics of the core calculus.

Database operations. Instead of modeling the database state explicitly and thus assuming a concrete implementation of the database, we represent the current state using event graphs of the database calls [6]. This is a common practice for specifying the semantics of highly available databases and of replicated data types [10,7,23].

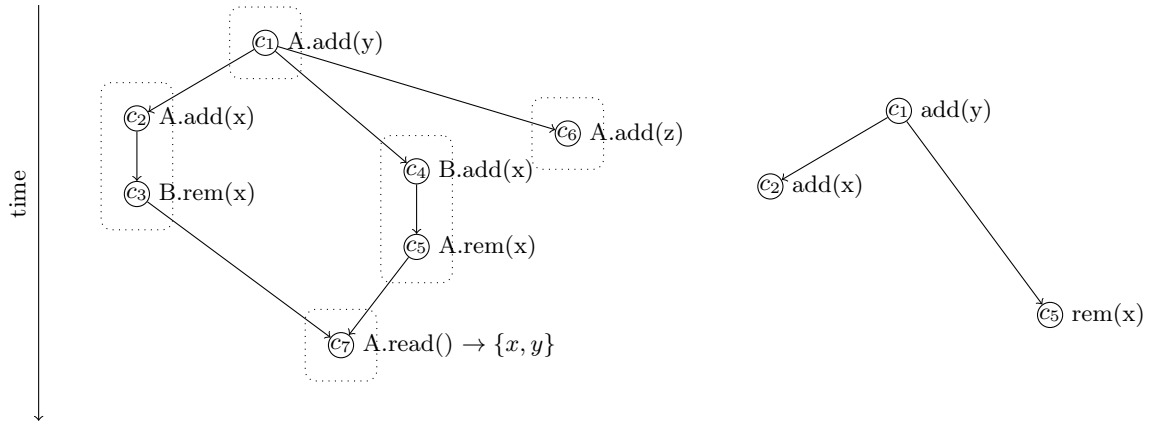


Fig. 3. Illustration of an event graph (left) and the extracted operation context for query c_7 (right).

Figure 3 shows an event graph of an execution involving two replicated sets A and B. Each call to the database is represented by a node. Several calls can

be bundled together in a transaction (boxes with dashed line). We add an edge from a call c_1 to a call c_2 if call c_1 happened before call c_2. Calls that are not reachable from each other, such as c_2 and c_5, are concurrent. The result of a read operation, like c_7 in the example, depends only on the calls that happened before. Also, each call operates only on one object. The subgraph of calls that happened before an operation is called the operation context. For query c_7, the operation context is visualized on the right part of Figure 3. Given an operation context and the parameters of the database operation, the specification (*querySpec* in the formal model) of the corresponding data type yields the result of the operation. For example, the specification of an add-wins replicated data type is defined as:

$$\begin{aligned} &spec_{\mathrm{CrdtAWSet}}(call, happensBefore, read()) = \\ &\quad \big\{x.\ \exists a \in call.\ a.op = add(x) \\ &\qquad \wedge \neg(\exists r \in call.\ r.op = rem(y) \wedge (a, r) \in happensBefore)\big\} \end{aligned}$$

Since no remove happened after the $add(x)$ in c_2, the read operation c_7 from our example returns a set containing x. Though we do not model replicas explicitly, they are represented as concurrent events in the event graph.

In the formal model, each database call is identified by a *callId*; the partial function *call* in the system state stores information about each call. The *callInfo* consists of the arguments to the operation and its return value. The *happensBefore* relation records the partial order between calls and *callOrigin* stores the transaction each call originated from. Transactions are identified by a *txid*, and its *txStatus* can be "running" or "committed". The procedure invocation that started a transaction is stored in *transactionOrigin*. Together, this information represents the history of database calls. Additionally, the field *visibleCalls* keeps the set of database calls that are visible at an procedure invocation and *currentTransaction* the currently running transaction (if any) for an invocation.

The rule *atomic* describes what happens on the database when starting a new transaction. Recall that we here consider databases in which transactions work on causally consistent snapshots, modeled as a set of visible updates on the database. To obtain the snapshot at the start of a transaction, we choose an arbitrary set of committed transactions *newTxns* which are to become visible. We derive the new set of visible calls by first taking all calls to update operations from these transactions: $callsInTransaction(S, newTxns) = \{c.\ \exists t \in newTxns.\ callOrigin(S, t) \triangleq t\}$. Next, we calculate the downwards closure of this set with respect to the happens-before relation on calls. The downwards closure of a set S and relation R is defined as $S \downarrow_R = S \cup \{x.\ \exists y.\ (x, y) \in R \wedge y \in S\}$. In our model, the happens-before relation is by construction transitive, so that this definition is sufficient to include all causal dependencies in the snapshot, therefore providing causal consistency (we have verified this in Isabelle and omit the proofs here).

Finally, we add the visible calls from the current invocation to the snapshot. We also set the transaction status of the new transaction id t to *uncommitted* and store the current transaction for the invocation.

When ending the current transaction (rule *commit*), its *txStatus* is set to *committed*. This allows it to be included in new snapshots, which eventually makes the database calls in the transaction visible to others. As the atomic rule can only pick committed transactions and no new calls can be added to it after committing, transactions are atomic.

When executing a database operation (rule *DB-operation*), we extract the *operationContext* from the current state. As stated earlier, the operation context consists of the currently visible calls (formally: $\lambda x.$ **if** $x \in \mathit{visibleCalls}(S, i)$ **then** $\mathit{call}(S, x)$ **else** $\bot$) and the happens-before relation restricted to the visible calls (formally: $\mathit{happensBefore}(S) \cap (\mathit{visibleCalls}(S, i) \times \mathit{visibleCalls}(S, i))$). We then nondeterministically pick a result *res*, which satisfies the query specification of the program in the operation context. The database call is then recorded in the state by adding the operation with its arguments and result to the existing calls. We also record the current transaction as the originating transaction for the new call. The happens-before relation is also updated by making the new call causally depend on all currently visible calls. The new call is then added to the set of visible calls, such that following operations depend on it.

History Recording. As for the database, we also store a history of invocations of API-procedures including the respective arguments (*invocationOp*) and the result for completed invocations (*invocationRes*). By using these in specifications, we can relate different procedure invocations and link procedure invocation with their corresponding changes in the database state. The rules *invocation* and *return* update this information accordingly.

Unique Identifiers. In practice, unique identifiers are often generated using UUIDs or using a replica-specific identifier together with a locally unique identifier. Since identifiers for database entries appear in most applications, we include a builtin action, which lets applications generate globally unique identifiers (see rule *new-id*). With this extension, we avoid proving the correctness of an identifier generator for every application. Moreover, it allows us to handle generated identifiers as special values, which cannot be forged by clients.

To model unique identifiers, we require that the type *any* comes with a function $\mathit{uniqueIds} : \mathit{any} \to \mathit{any\ set}$ extracting the unique identifiers of a value. The *new-id* rule ensures that the generated value includes exactly one unique identifier and that the generated value is in the domain of the *ls'* function. This allows us to include a kind of type-check in the action to generate a unique identifier of a specific form.

To describe the semantics, we keep track of all generated unique identifiers in *generatedIds*. The set *knownIds* represents the identifiers which could be known to clients, i.e. identifiers which have been returned from an invocation of the application API (see rule *return*). In the *invocation* rule, we enforce that clients can only invoke the API with known identifiers.

Partial Failures. Since we are considering a distributed application, it is necessary to handle partial failures. This is captured in the semantics with the rule *crash*,

which models a crash of a single procedure invocation and loses all locally stored information. Afterwards, the invocation cannot continue, since there is no local state.

Invariants. We use invariants to specify the application. Invariants can refer to the database state as well as the history of procedure invocations, which makes the specification language more expressive than related work where invariants can only refer to the database state.

Formally, rule *inv* is always enabled so the invariant can be checked (and therefore must hold) at all times. However, the invariant cannot involve arbitrary aspects of the current state. The function *invContext* provides a special view on the state for checking invariants, which only contains part of the system state. All information local to a particular procedure invocation is not included (i.e. *txStatus*, *generatedIds*, *localState*, *currentProc*, *currentTransaction*, and *visibleCalls*). Moreover, the function *invContext* only includes database calls from committed transactions. These restrictions are essential for simplifying the verification efforts, which we discuss in the next section.

Using the invariant checks in our transition relation $S \xrightarrow{tr}{}^{*} S'$ we can define program correctness as follows:

$$\begin{aligned} \mathit{traces}(\mathit{program}) &:= \{tr \mid \exists S'.\ \mathit{initialState}(\mathit{program}) \xrightarrow{tr}{}^{*} S'\} \\ \mathit{traceCorrect}(\mathit{trace}) &:= \forall i.\ (i, \mathit{invCheck}(\mathit{false})) \notin \mathit{trace} \\ \mathit{programCorrect}(\mathit{program}) &:= \forall \mathit{trace} \in \mathit{traces}(\mathit{program}).\ \mathit{traceCorrect}(\mathit{trace}) \end{aligned}$$

The set *traces* includes all traces admitted by a program; the predicate *traceCorrect* defines that a trace is correct if it does not contain an invariant check on any procedure invocation i that evaluates to *False*; and the last definition states that a program is correct iff all its traces are correct.

Using these definitions to reason about correctness is however not very practical. We have to consider all possible traces, which includes the interleavings of several concurrent procedure invocations that are allowed by the transition relation.

3 Single-invocation Semantics

To address the challenge of handling concurrency in our setting, we have developed a proof technique, which reduces the formal verification problem to a simpler problem, where we can reason about only one procedure invocation at a time.

Our proof technique is based on invariants and reduces the proof obligations to checking that the initial system state satisfies the invariant and that each procedure invocation maintains the invariant. When verifying a single procedure invocation, the effects of other invocations only need to be considered at specific program points, namely at the procedure invocation and before the start of transactions. We use the invariant and generic properties of executions to reason about possible state changes at these program points. For the procedure to be

verified, we must then guarantee that the invariant is maintained at the end of transactions, right after the start of a procedure invocation and after returning from a procedure invocation. The latter two are necessary because at these program points the information stored in the history of procedure invocations is updated.

Technically, we formalize the reduction using a second operational semantics, the *single-invocation semantics*, which we present in this section. In Section 4, we then prove that reduction from the interleaving semantics to the single-invocation semantics is sound: If a program is correct with respect to the single-invocation semantics, it is also correct in the interleaving semantics.

The main difference between the two semantics is that the single-invocation semantics only allows steps in a single invocation. Effects from different invocations are treated with nondeterministic steps in the rules for starting a procedure invocation and beginning a transaction. In these cases, the rules of the single-invocation semantics assume an arbitrary state change, assuming that the invariant is maintained, the new state is well-formed, and the history of the new state is an extension of the former history.

Moreover, the single-invocation semantics does not include a dedicated step to check the invariant. The invariant has to be checked in the following three steps: directly after a procedure invocation (*S-invocation*), after the end of a transaction (*S-commit*), and after a procedure invocation returns to the client (*S-return*).

Correspondingly, we adapted the transition relation to be $S \xrightarrow{i,(a,v)} S'$ for a single step. The value v is *true* if the step fulfills the necessary invariant checks. A program is *correct* with respect to the single-invocation semantics if for all possible executions the trace contains only actions for which v is true.

Figure 4 shows the rules of the single-invocation semantics that differ from the interleaving semantics. The rule *S-steps* enforces that only steps on one invocation can be taken in a trace. The rules *local*, *DB-operation*, and *new-id* are not included in Figure 4, because they do not involve the invariant – the parameter v is always *true* in these transitions and the rules are otherwise equivalent to the interleaving semantics. The interesting aspects are in the handling of transactions and invocations. Here, we add the nondeterministic state changes and check the invariant.

Rule S-atomic At the beginning of a transaction a new snapshot is determined, which means that this is a place where changes from concurrent invocations might become visible to the current invocation. We model this with a nondeterministic state change from the current state S to a new state S'. The predicate $\mathit{growing}(i, S, S')$ requires that S' must contain at least the invocations and database operations from S. However, it might have grown with further events:

$$\mathit{growing}(i, S, S') :=$$
$$\mathit{wellformed}(S) \wedge \Big(\exists tr.\ S \xrightarrow{tr}{}^{*} S' \wedge (\forall (i', a) \in tr.\ i' \neq i) \wedge (\forall i'.\ (i', \mathit{crash}) \notin tr)\Big)$$

$$\frac{}{S \xrightarrow{i,\epsilon}{}^{*} S}\ \textit{(S-steps-empty)} \qquad \frac{S_1 \xrightarrow{i,tr}{}^{*} S_2 \qquad S_2 \xrightarrow{i,t} S_3}{S_1 \xrightarrow{i,tr\cdot t}{}^{*} S_3}\ \textit{(S-steps)}$$

$$\frac{\begin{array}{c} \textit{invocationOp}(S,i) = \bot \\ \textit{procedure}_{prog}(\textit{procName}, \textit{args}) \triangleq (\textit{initialState}, \textit{impl}) \qquad \textit{uniqueIdsInList}(\textit{args}) \subseteq \textit{knownIds}(S') \\ \textit{wellformed}(S') \qquad \forall tx.\ \textit{txStatus}(S', tx) \neq \textit{uncommitted} \\ \textit{invariant\_all}(S') \qquad \textit{invocationOp}(S', i) = \bot \qquad v = \textit{invariant\_all}(S'') \\ \forall tx.\ \textit{transactionOrigin}(S'', tx) \neq i \qquad S'' = S' \begin{bmatrix} \textit{visibleCalls}(S,i) := \emptyset \\ \textit{currentProc}(S,i) := \textit{impl} \\ \textit{localState}(S,i) := \textit{initialState} \\ \textit{invocationOp}(S,i) := (\textit{procName}, \textit{args}) \end{bmatrix} \end{array}}{S \xrightarrow{i,(\textit{invoc}(\textit{procName},\textit{args}),v)} S''}\ \textit{(S-invocation)}$$

$$\frac{\begin{array}{c} \textit{localState}(S,i) \triangleq ls \\ \textit{currentProc}(S,i) \triangleq f \qquad f(ls) = \textit{return}(\textit{res}) \qquad \textit{currentTransaction}(S,i) = \bot \\ v = \textit{invariant\_all}(S') \qquad S' = S \begin{bmatrix} \textit{visibleCalls}(S,i) := \bot \\ \textit{currentProc}(S,i) := \bot \\ \textit{localState}(S,i) := \bot \\ \textit{invocationRes}(S,i) := \textit{res} \\ \textit{knownIds}(S) := \textit{knownIds}(S) \cup \textit{uniqueIds}(\textit{res}) \end{bmatrix} \end{array}}{S \xrightarrow{i,(\textit{return}(\textit{res}),v)} S'}\ \textit{(S-return)}$$

$$\frac{\begin{array}{c} \textit{localState}(S,i) \triangleq ls \qquad \textit{currentProc}(S,i) \triangleq f \qquad f(ls) = \textit{beginAtomic}(ls') \\ \textit{currentTransaction}(S,i) = \bot \qquad \textit{txStatus}(S,t) = \bot \qquad \textit{growing}(i,S,S') \\ \forall t.\ \textit{transactionOrigin}(S,t) \triangleq i \leftrightarrow \textit{transactionOrigin}(S',t) \triangleq i \qquad \textit{invariant\_all}(S') \\ \forall tx.\ \textit{txStatus}(S', tx) \neq \textit{uncommitted} \qquad \textit{wellformed}(S') \qquad \textit{wellformed}(S'') \\ \textit{localState}(S',i) \triangleq ls \qquad \textit{currentProc}(S',i) \triangleq f \qquad \textit{currentTransaction}(S',i) = \bot \\ \textit{visibleCalls}(S,i) = \textit{vis} \qquad \textit{visibleCalls}(S',i) = \textit{vis} \qquad \textit{newTxns} \subseteq \textit{committedTransactions}(S) \\ \textit{newCalls} = \textit{callsInTransaction}(S, \textit{newTxns}) \downarrow_{\textit{happensBefore}(S)} \qquad \textit{vis}' = \textit{vis} \cup \textit{newCalls} \\ \textit{consistentSnapshot}(S', \textit{vis}') \qquad \textit{txStatus}(S', t) = \bot \qquad \forall c.\ \textit{callOrigin}(S', c) \neq t \\ \textit{transactionOrigin}(S', t) = \bot \qquad S'' = S' \begin{bmatrix} \textit{txStatus}(t) := \textit{uncommitted} \\ \textit{transactionOrigin}(t) := i \\ \textit{currentTransaction}(i) := t \\ \textit{localState}(i) := ls' \\ \textit{visibleCalls}(i) := \textit{vis}' \end{bmatrix} \end{array}}{S \xrightarrow{i,(\textit{beginAtomic}(t,\textit{newTxns}),\textit{true})} S''}\ \textit{(S-atomic)}$$

$$\frac{\begin{array}{c} \textit{localState}(S,i) \triangleq ls \\ \textit{currentProc}(S,i) \triangleq f \qquad f(ls) = \textit{endAtomic}(ls') \qquad \textit{currentTransaction}(S,i) \triangleq t \\ v = \textit{invariant\_all}(S') \qquad \textit{wellformed}(S') \qquad S' = S \begin{bmatrix} \textit{localState}(S,i) := ls' \\ \textit{currentTransaction}(S,i) := \bot \\ \textit{txStatus}(S,t) := \textit{committed} \end{bmatrix} \end{array}}{S \xrightarrow{i,(\textit{endAtomic},v)} S'}\ \textit{(S-commit)}$$

Fig. 4. Single invocation semantics.

This definition allows us to use any general property we can prove about steps taken in other invocations. Additionally, the rule allows us to assume the invariant for the new state S'.

The remaining aspects of the rule describe the other state changes and are equivalent to the interleaving semantics.

Rule S-commit When a transaction is committed, we check the invariant in the state after the commit and record the result of the invariant check in the trace. This ensures that an execution is considered incorrect, if a transaction breaks the invariant.

Rule S-invocation. An invocation can only be executed at the beginning of the trace, since the rule demands that the invocation i is not yet used in the current state S. The rule then nondeterministically chooses a state S' which satisfies the invariant and starts the procedure invocation, which yields state S''. The rule also allows us to assume that there are no uncommitted transactions at the start of an invocation and that we start from a well-formed state. A state is defined to be well-formed if it is reachable from the initial state.

We then check whether the invariant holds in S'' and record the result in the trace. This is necessary, because invariants can refer to unfinished procedure invocations, so starting an invocation can cause an invariant violation.

Rule S-return. For a return statement, we check the invariant in the state after completing the invocation.

4 Soundness of the Reduction

We now show that it is in fact sufficient to prove a program correct with respect to the single-invocation semantics in order to ensure correctness in all possible concurrent executions according to the interleaving semantics. Figure 5 illustrates the main steps in our proof with an example. The full proof is formalized in Isabelle/HOL[1]. Below we give the corresponding definitions and lemmas.

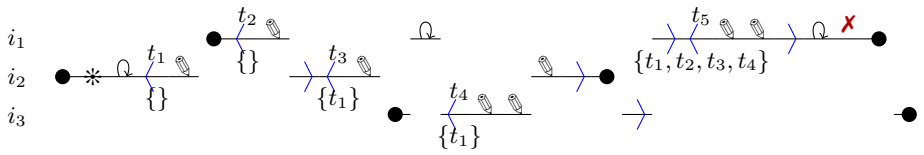
Definition 4.1 (Packed traces)**.** A trace tr is **packed** for a procedure invocation i if it only switches to invocation i at the start of a procedure invocation (*invoc*) or at the start of a transaction (*beginAtomic*) action:
$\forall i', a.\ i' \neq i \wedge [(i', \_), (i, a)] \in tr \rightarrow (\textit{is-invoc}(a) \vee \textit{is-beginAtomic}(a))$. We say a trace is **packed** if it is packed for all procedure invocations.

Lemma 4.2 (Reduction to packed traces)**.** A program is correct if all its traces that are packed and do not contain *crash*-steps are correct.

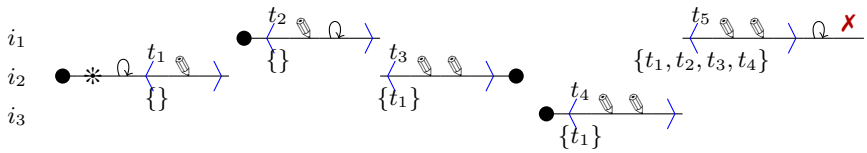
[1] The Isabelle proofs are available at
`https://github.com/peterzeller/repliss-isabelle/`

Trace actions: begin a procedure invocation, return from an invocation, start a transaction, commit a transaction, database operation, create a new unique identifier, local steps, and a failing invariant. We annotate each start of a transaction with a transaction id and the set of transactions that are visible to this transaction.

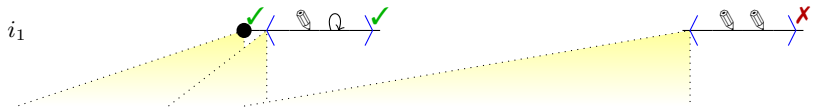
Step 1: Assume we have a failing trace in the interleaving semantics, for example:



Step 2: In Lemma 4.2, we show that in this case there is an equivalent packed trace that is also failing. We construct this packed trace by moving actions to the front. This reordering does not invalidate any snapshots and since snapshots are fixed in the trace, the effect of the trace is preserved. For the example above, we can construct the following packed trace which ends with a failing invariant:



Step 3: In Lemma 4.3, we show that there is a corresponding trace in the single invocation semantics, where we only consider procedure invocation i_1 while actions from other invocations are summarized using invariants (visualized by lightcones). The proof obligation for the single invocation semantics is to check the invariant after an invocation return and after each transaction commit. In the picture below, the proof obligation is always to check that the actions between a light source and a check mark preserve the invariant.



Thus, if we prove that no invariant check ever fails in the single invocation semantics, we have shown the correctness of the application.

Fig. 5. Overview of our soundness proof applied to an exemplary trace.

Proof sketch. The proof is essentially a reduction argument [15] which uses commutativity of actions performed on different invocations.

Let tr be a failing trace of the program, i.e. a trace containing a failing invariant check. We show that we can then construct a packed trace which is also failing. We consider the prefix of tr up to the first failing invariant check. We then can reorder the actions in this prefix to get a packed trace.

We prove this by induction over the number of procedure invocations, which are not yet packed. To show that a single invocation i can be packed without unpacking any other invocation, we use another induction over the minimal index k with a not-allowed invocation switch. This cannot be the first action for invocation i, since *invoc* is an allowed invocation switch. Thus, let k' be the last action on invocation i before k. We then split the trace into $tr = tr_{[..k'-1]} \cdot tr_{k'} \cdot tr_{[k'+1..k-1]} \cdot tr_k \cdot tr_{[k+1..]}$ and reorder it by moving tr_k to the front to get $tr' = tr_{[..k'-1]} \cdot tr_{k'} \cdot tr_k \cdot tr_{[k'+1..k-1]} \cdot tr_{[k+1..]}$. By changing the order of the trace, we have eliminated all unwanted invocation switches up to index k. We can show that the action of tr_k commutes with the actions it is swapped with, which are all from a different invocation.

The commutativity proof involves a case distinction over all possible actions in the system. An interesting case here is moving an *endAtomic* action to the front. In principle, this could affect other transactions, but since we do not change the transaction snapshots when reordering the trace, we are guaranteed to get the same results.

□

Lemma 4.3 (Simulation)**.** Let tr be a packed trace of an execution starting in state S and ending in state S' where S is well-formed (i.e. reachable from the initial state) and satisfies the invariant and S' does not satisfy the invariant. Moreover, assume that tr is packed and does not contain any crashes. Then, there is an execution in the single-invocation semantics starting with state S and the trace of this execution is not correct.

Proof sketch. We show that the single-invocation semantics can simulate the distributed (interleaving) semantics. To this end, we define a coupling relation between a state S_d of the distributed execution and a state S_i of a single-invocation execution with invocation i. The coupling invariant distinguishes two cases: 1) When the last step in the distributed execution was in invocation i, then the states must be equal. 2) When the last step was on an invocation different from i, it must hold that S_i is greater than S_d with respect to the *growing* relation and that the local state of invocation i is equivalent in S_i and S_d.

In the simulation proof, the case of switching between invocations can only occur at the beginning of invocations or at the start of a transaction (because we assumed the trace is packed). In both cases, the single-invocation semantics allows nondeterministic state-transitions, which allow the single-invocation execution to catch up with the distributed invocation. The remaining cases are straightforward.

□

Theorem 4.4 (Soundness of verification technique)**.** When a program is correct with respect to the single-invocation semantics and the initial state satisfies the invariant, then the program is correct with respect to the distributed (interleaving) semantics.

Proof sketch. We show that all executions are correct. Because of Lemma 4.2, it is sufficient to consider executions with packed traces without crashes. Let tr be a trace for such an execution.

For the sake of a contradiction assume tr is not a correct trace, i.e. there is a failing invariant check in the trace. We now consider the first time when an invariant-violating state is reached and the prefix of tr leading to this state. As the state does not satisfy the invariant, though the initial state does (by assumption), we can apply Lemma 4.3 and obtain a failing trace in the single-invocation semantics. However, this is a contradiction to the assumption that the program is correct in the single-invocation semantics.

□

5 The Repliss Tool

We have developed the Repliss verification tool, which uses our technique to partially automate the verification of highly available applications. In principle, our reduction to the sequential single-invocation semantics could be combined with any technique for verifying sequential programs. For Repliss we chose to implement a symbolic execution engine with the CVC4 [3] automated theorem solver as a backend.

Let us illustrate how Repliss can be used to verify a chat application. This example is inspired by an experience report from Discord [22], who migrated their chat service from a single centralized database to the replicated and weakly consistent database Cassandra [13]. Although the code had been well tested prior to deployment, when the new solution was first used in production, some messages ended up with missing metadata. This problem occurred when a user edited a message while another user concurrently deleted the message. When modeling the chat application in Repliss, a small counter example is generated illustrating this problematic case. Below, we present a model where the Bug is fixed, which can then be verified using Repliss.

5.1 Implementation

An essential aspect of developing a highly available program is to find a suitable data model for storing the persistent state. To this end, Repliss provides a library of built-in replicated data types (CRDTs [20]). There are different variants of the same data type, which differ in how concurrent updates are handled. It is important to choose the appropriate variant to get the desired application behavior.

Our implementation of the chat application is shown in Figure 6. The data model for the chat application is defined in lines 1-4 of the code. Repliss uses

a suffix, such as rw, to denote how conflicts between concurrent updates are resolved. In the example, we use a set named chat of type Set_rw to store the set of messages that belong to the chat. We further use a Map_dw to store the data for each message. Each message has an author and a message content. For the author, we use a simple register with last-writer-wins semantics, since it is only written when creating a new message. The message content is stored in a multi-value register, which keeps multiple versions in case of concurrent assignments.

The choice of data types determines the *querySpec* in the formal semantics. Repliss includes the specifications for all supported CRDTs and knows how to compose the semantics in the case of nested CRDTs (e.g. a CRDT-set used as the value in a CRDT-map). In the example, we use the delete-wins (dw) variant of maps. This is important for the message map to ensure that a message is deleted in the case of concurrent invocations of deleteMessage and editMessage. This choice fixes the bug in the original application. For the set of messages we use the remove-wins variant (rw) to have semantics compatible with the delete-wins behavior of the map. Choosing a different semantics would make it hard to maintain consistency between the set of messages and the related information stored in the message map.

Starting from line 6 in Figure 6 the procedures of the program are implemented. The **atomic**-blocks correspond to the *beginAtomic* and *endAtomic* actions in the formal semantics, the references to the chat- and message-CRDTs correspond to a database call, and the expression **new** MessageId triggers the generation of a new unique identifier.

5.2 Specification

To prove the correctness of our application, we next specify its invariants. To address the database state, we use queries (as in Property 1 below). Further, invariants can address a history of procedure invocations and database calls. This enables us to express some temporal properties and relate effects of procedure invocations, as demonstrated in Property 2.

Property 1. For the chat application, referential integrity is important: every MessageId occurring in the message set should have a corresponding entry in the message map. Formally, this relation can be expressed as a first order logical formula using the queries defined on the data types:

```
invariant forall m: MessageId :: chat_contains(m) ==> message_exists(m)
```

When queries are used in an invariant, we have to define in which operation context the query should be evaluated. Remember that invariants cannot access the current database snapshot of a specific procedure invocation (field *visibleCalls*). We therefore specify that this invariant has to hold in all valid database snapshots. Repliss automatically adds this quantification to the invariant if free queries are used.

```
1  crdt chat: Set_rw[MessageId]
2  crdt message: Map_dw[MessageId, {
3      author: Register[UserId],
4      content: MultiValueRegister[String]}]
5
6  def sendMessage(from: UserId, content: String): MessageId {
7      var m: MessageId
8      atomic {
9          m = new MessageId
10         call message_author_assign(m, from)
11         call message_content_assign(m, content)
12         call chat_add(m) }
13     return m }
14 def editMessage(id: MessageId, newContent: String) {
15     atomic {
16         if (message_exists(id)) {
17             call message_content_assign(id, newContent) }}}
18 def deleteMessage(message_id: MessageId) {
19     atomic {
20         if (message_exists(message_id)) {
21             call chat_remove(message_id)
22             call message_delete(message_id) }}}
23 def getMessage(m: MessageId): getMessageResult {
24     atomic {
25         if (message_exists(m)) {
26             return found(message_author_get(m), message_content_getFirst(m))
27         } else {
28             return notFound() }}}
```

Fig. 6. Model of Chat application in Repliss.

Property 2. We use the history of procedure invocations to express properties at the external interface of the application. To this end, we use the *invocationOp* and *invocationRes* maps, which contain the operation (input) and result of each procedure invocation. In the invariant below, we relate the results of the getMessage procedure with invocations of sendMessage: If getMessage returns a certain user u as part of a message, then there must be an invocation of sendMessage with that user.

```
invariant forall g: invocationId, m: MessageId, u: UserId, c: String ::
  g.info == getMessage(m) && g.result == getMessage_res(found(u, c))
    ==> (exists s: invocationId, c2: String :: s.info == sendMessage(u, c2))
```

The original implementation of the chat application did not satisfy this property – it returned null as the author value, even though the user null never sent a message.

5.3 Correctness

Repliss can verify the referential integrity constraint from Property 1 automatically. For the verification to succeed it is important that we used transactions for sendMessage and deleteMessage such that no client can observe a database state where the chat-set is updated and the message-map is not. Moreover, it was necessary that we chose data types with compatible semantics such that concurrent updates are merged into concurrent states.

```
1  // For every author assignment, there is a corresponding invocation of sendMessage:
2  invariant forall c: callId, m: MessageId, u: UserId ::
3    c.op == message_author_assign(m, u)
4    ==> (exists i: invocationId, s: String :: i.info == sendMessage(u, s))
5  // For assignments of the content field, there is a prior assignment to the author field:
6  invariant forall c1: callId, m: MessageId, s: String ::
7    c1.op == message_content_assign(m, s)
8    ==> (exists c2: callId, u: UserId ::
9           c2.op == message_author_assign(m, u) && c2 happened before c1)
10 // There is no update after a delete:
11 invariant !(exists write: callId, delete: callId, m: MessageId ::
12       ((exists u: UserId :: write.op == message_author_assign(m, u))
13        || (exists s: String :: write.op == message_content_assign(m, s)))
14   && delete.op == message_delete(m) && delete happened before write)
```

Fig. 7. Further invariants for Chat application.

For the history invariant in Property 2, the verification is more involved. As explained before, our proof approach is based on an invariant and fully compositional, i.e. each procedure is checked individually. Therefore, the invariant needs to be strong enough, such that assuming the invariant in a pre-state gives us enough information to prove the invariant in the post-state. We thus define additional invariants (Fig.7) which enable Repliss to verify Property 2. The verification by Repliss takes approx. 40 seconds when verifying both properties together.

6 Conclusion and Future Work

We have presented a novel proof technique for verifying applications built on top of weakly consistent databases. Our proof technique is designed to be used with partially automated tools like the Repliss verification tool. The soundness of our technique is formally verified using Isabelle/HOL with a proof based on a formal small-step semantics. The semantics comprises only eight different steps to facilitate the validation of the assumptions made for the soundness proof.

While we restricted our consistency model to causally consistent transactions, we are confident that the central ideas of our approach can be transferred to other consistency models. To support stronger consistency models, we could extend the predicate *growing* used in the *atomic* rule in Figure 4. Here, a model based on tokens as used by CISE [11] could restrict the changes that may be done by concurrent invocations. For weaker consistency models, it suffices to drop assumptions from our proof rules. None of these changes requires adaptations of the proof technique. Essential for our approach is simply that transactions are isolated (i.e. concurrent transactions cannot see each others updates) and that application replicas only communicate via the replicated database.

In this paper, we have demonstrated the applicability of our proof technique with the chat application. In current work, we are improving the automation of the approach to facilitate the invariant preservation proofs. Application on larger case studies is work-in-progress, but we expect it to scale well in the

number of procedures, as each procedure can be verified individually against the invariants. In that sense, our approach is composable. However, the number or size of invariants is likely to grow with the number of procedures, which could restrict the scalability of the technique. We expect that further techniques to partition invariants according to components will be necessary to handle more complex applications.

Acknowledgement. This research is supported in part by the EU H2020 project "LightKone" (732505) `https://www.lightkone.eu/`.

References

1. Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.
2. Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
3. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.
4. Bernhard Beckert and Wojciech Mostowski. A program logic for handling JAVA card's transaction mechanism. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2621 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2003.
5. Rastislav Bodík and Rupak Majumdar, editors. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 2016.
6. Sebastian Burckhardt. Principles of eventual consistency. *Foundations and Trends in Programming Languages*, 1(1-2):1–150, 2014.
7. Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: specification, verification, optimality. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 271–284. ACM, 2014.
8. Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19:1–19:43, 2015.
9. Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.
10. Alexey Gotsman and Hongseok Yang. Composite replicated data types. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 585–609. Springer, 2015.

11. Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In Bodík and Majumdar [5], pages 371–384.
12. Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. In *32nd ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2018.
13. Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
14. Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In Bodík and Majumdar [5], pages 357–370.
15. Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
16. Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 401–416. ACM, 2011.
17. Sreeja Nair, Gustavo Petri, and Marc Shapiro. Invariant safety for distributed applications. *CoRR*, abs/1903.02759, 2019.
18. Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE tool: proving weakly-consistent applications correct. In Peter Alvaro and Alysson Bessani, editors, *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016*, pages 2:1–2:3. ACM, 2016.
19. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
20. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011.
21. K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 413–424. ACM, 2015.
22. Stanislav Vishnevskiy. How discord stores billions of messages. `https://blog.discordapp.com/how-discord-stores-billions-of-messages-7fa6ec7ee4c7`. Accessed: 2018-11-16.
23. Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of crdts. In Erika Ábrahám and Catuscia Palamidessi, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings*, volume 8461 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014.