Formal Modeling and Analysis of the Walter Transactional Data Store

Si Liu[1], Peter Csaba Ölveczky[2], Qi Wang[1], and José Meseguer[1]

[1] University of Illinois, Urbana-Champaign, USA
[2] University of Oslo, Oslo, Norway

Abstract. Walter is a distributed partially replicated data store providing Parallel Snapshot Isolation (PSI), an important consistency property that offers attractive performance while ensuring adequate guarantees for certain kinds of applications. In this work we formally model Walter's design in Maude and formally specify and verify PSI by model checking. To also analyze Walter's performance we extend the Maude specification of Walter to a probabilistic rewrite theory and perform statistical model checking analysis to evaluate Walter's throughput for a wide range of workloads. Our performance results are consistent with a previous experimental evaluation and throw new light on Walter's performance for different workloads not evaluated before.

1 Introduction

Cloud-based transaction systems provide both a challenge and an opportunity for the use of formal methods. The *challenge* has to do with the fact that the very *raison d'être* for such system is the need for a carefully chosen *compromise* between consistency guarantees and performance. Their massive use requires them to ensure scalability to large numbers of users with acceptable latency and throughput, while also guaranteeing the promised consistency properties. This is a challenge for formally-based design, because many formal methods tend to solely focus on correctness. Yet, correctness without due performance is useless for these systems. The *opportunities* are plentiful, including the following: (1) Many of these systems have never been formally specified, either at the *system specification* level or at the *property specification* level. (2) There is a need for *modularity* and *conceptual unification* in the design of these, currently quite ad-hoc and monolithic, systems. (3) There is also the prospect of using formal executable specifications for *code generation* purposes, achieving correct-by-construction systems that, by having been thoroughly analyzed in their correctness and performance aspects, can achieve very high quality.

This work is part of a long-term research effort in which we have been using Maude to both meet the challenges and exploit the opportunities described above for cloud-based transaction systems (see [4] for a survey). Specifically, we exploit the type-(1) opportunity offered by Walter [22], a well-know cloud-based transaction system that provides an important intermediate consistency guarantee, PSI. Walter is a very good type-(1) opportunity because no formal system specification exists at all; and there is no formal (or even informal) verification that it guarantees PSI. Walter is also a good stepping stone towards placing the design of cloud-based transaction systems in a formally-based modular framework. The way we are advancing this type-(2) goal is by first systematically studying system designs that cover the whole spectrum between lower-guarantees/higher-performance and higher-guarantees/lower-performance systems. We have already studied several systems in this spectrum, including RAMP [15,11], our own ROLA design [12], P-Store [19], and Megastore [9]. Walter has been a key missing design in the spectrum. The essential point is that case studies spanning the entire correctness/performance spectrum are crucial for identifying optimal decompositions of these and future systems into modular, reusable components. Finally, the fact that our Maude specification of Walter and of the other above-mention systems are *executable*, also helps us advance towards exploiting the type-(3) opportunity of achieving high-quality code generation for formal specifications. In this paper we focus on the type-(1) goal for Walter, but our sights are aimed at the type-(2)–(3) goals just as much.

Main Contributions and Outline. In Section 2 we give an overview of Walter, the PSI property and the stronger Snapshot Isolation (SI) property, and summarize the main features of Maude used in our formal modeling and analysis. Section 3 provides a detailed formal executable specification of Walter in Maude. This is a key contribution since, to the best of our knowledge, it is the first formal specification of Walter. Section 4 formalizes the SI and PSI properties and formally analyzes for the first time the whether the Walter design satisfies either of these properties. This analysis is achieved by: (i) providing a parametric method to generate all initial states for given parameters; and (ii) performing model checking analysis to verify the SI and PSI properties for all initial states for various parameter choices. Analyzing complex properties such as SI and PSI is not easy; we therefore propose a new general method for model checking such properties by adding a "monitor" to the state which records the global order of transaction starts and commits/aborts. In this way we can easily specify and model check SI and PSI; furthermore, this technique should also be applicable to analyze other consistency properties. Our analysis shows that the Walter design does indeed satisfy the PSI property for all our initial states but fails to satisfy the SI property. Section 5 makes four contributions. First, it extends the Maude model of Walter from a rewrite theory to a *probabilistic* rewrite theory by adding time and probability distributions for message delays to the original specification. Second, it carries out a systematic *statistical model checking* analysis of the key performance metric, transaction throughput, under a wide range of workloads. Third, it confirms that the performance estimates thus obtained are consistent with (i.e., exhibit similar trends as) those obtained experimentally for the Walter implementation in [22]. Fourth, it provides new insights about Walter's performance beyond the limited ranges for which such information was available by experimental evaluation in [22]. Finally, related work is discussed in Section 6, and concluding remarks are given in Section 7.

2 Preliminaries

2.1 Parallel Snapshot Isolation

To deal with huge amounts of data, cloud-based applications need to *partition* their data across distributed sites, and to provide high availability and disaster tolerance, data must be *replicated* at widely distributed sites. Such *partially replicated* data stores have to: (i) maintain some consistency of replicated data, and (ii) provide some consistency for (multi-partition) transactions that access data stored at different partitions (e.g., a transaction should not see that Don is a friend of Benny but that Benny is not a friend of Don). However, ensuring high degrees of consistency for partially replicated data stores supporting multi-partition transactions requires a lot of costly coordination, which might lead to unacceptable delays and throughput for many kinds of applications. Designers of distributed data stores therefore face a trade-off between providing good consistency properties and high performance. There are a number of consistency properties, ranging from strong consistency guarantees like serializability all the way to weak properties such as read atomicity and eventual consistency.

A popular intermediate consistency model provided by commercial database systems such as Oracle and SQL Server is *snapshot isolation* (SI) [3]. The idea is that a multi-partition transaction reads from a *snapshot* of a distributed data store that reflects a *single commit order* of transactions across sites.

In [22], the authors argue that having the same commit order across all sites is not necessary for social networks and similar applications: it does not matter much that Vlad in Moscow sees Kim's post before seeing Benny's post, whereas Don in Washington sees Benny's post before Kim's post. (Hence Benny and Kim can commit their (independent) posts without waiting for each other.) They propose a new consistency model, called *parallel snapshot isolation*, which allows different commit orders at different sites, while still guaranteeing:

- recent and "consistent" views: all operations in a transaction read the most recent version committed at the transaction execution site, as of the time when the transaction begins;
- no write-write conflicts (the write sets of committed somewhere-concurrent transactions must be disjoint); and
- preservation of *causality* across sites, which ensures that both Vlad and Benny see Kim's post before seeing Don's reply to Kim's post.

In [22] the authors specify PSI by giving some abstract pseudo-code "program" of a centralized execution that a distribution implementation must emulate.

2.2 Walter

Walter [22] is a partially replicated geo-distributed data store that supports multi-partition transactions and guarantees/implements PSI.

The key idea to ensure that all operations in a transaction read a consistent "snapshot" of the distributed data store is that *each site s* maintains a (local) *vector timestamp* $\{site_1 \mapsto k_1, \ldots, site_n \mapsto k_n\}$ representing a current snapshot of the state, as seen by site s, where $site_j \mapsto k_j$ means that the snapshot includes the first k transactions executed at site $site_i$. Each time a transactions starts executing at s, the transaction is assigned the current local snapshot/vector timestamp of site s. Remote reads can then be performed consistently according to this snapshot. Another key Walter feature is that each data item has a preferred site, so that writes at preferred sites can be committed fast (e.g., the sites that you usually use could be the preferred site for "your" data).

A transaction is executed as follows. When the "host" site s starts executing the transaction t, t is assigned the current snapshot of s. The site s then executes the read and write operations in t. For writes, Walter buffers the versions written in the transaction's write set. For reads, Walter fetches the latest appropriate version according to t's start snapshot, by checking any updates in the write set and its history of previous updates. If the associated key is not replicated locally, Walter retrieves the right version remotely from the data item's preferred site.

When the host site has finished executing the operations in the transaction, it starts committing the transaction. Read-only transactions and transactions that only write data items whose preferred site is the host site s can commit locally (*fast commit*). Walter then checks whether all versions of each data item in the history of the local site are *unmodified* since the start vector timestamp, and whether all data items are *unlocked* (i.e., not being committed by another transaction). If either check fails, Walter aborts the transaction; otherwise, Walter can commit the transaction. If a transaction cannot commit locally (*slow commit*), the executing site s uses the two-phase commit (2PC) protocol to check whether the transaction can be committed, by asking all the preferred sites of data items written by t whether t can be committed. If the data items written by t are unmodified and unlocked at such a site, the site replies with a "yes" vote and locks the corresponding data items. Otherwise, the site votes "no." If the executing site receives a "no" vote, the transaction is aborted and the other preferred sites are notified and release the appropriate locks. If all votes are "yes" votes, the transaction can be committed.

If the transaction t can be (fast or slow) committed, the site s marks t as committed, assigns it a *version* $(s, seqNo)$ (where *seqNo* is a local sequence number), updates the local history with the updates, and propagates t to other sites, which update their histories and their vector timestamps. To allow f site failures, a transaction is marked *disaster-safe durable* if its writes have been logged at $f+1$ sites. The propagation protocol first checks whether the transaction can be marked as disaster-safe durable by collecting acknowledgments from $f + 1$ sites for each data item. Upon receiving the propagation of a transaction, a site acknowledges it only after it receives all transactions that causally precede the propagated transaction (by using the transaction's start vector timestamp), and all transactions at the same executing site with a smaller sequence number. The protocol then checks whether the transaction can be marked as globally visible. This is done by committing the transaction at all sites. A transaction can be committed at a remote site when it learns that the transaction is disaster-safe durable, all transactions causally preceding the transaction have been committed locally, and all transactions at the same executing site with a smaller sequence number have been committed locally.

The paper [22] briefly discusses failure handling, but does not give much detail. The authors have implemented Walter in about 30K lines of code, and have implemented Facebook- and Twitter-like applications on top of Walter using the Amazon EC2 cloud platform to experiment with and evaluate Walter's performance in isolation, and as a backend for social networking, in a distributed setting (with nodes in US, Ireland, and Singapore). They use their distributed deployment to estimate the transaction latency and throughput (committed transactions per second) for read-only, write-only, and 90% read workloads.

The authors to do not prove or justify that Walter actually guarantees PSI.

2.3 Rewriting Logic and Maude

In rewriting logic [17] a concurrent system is specified a as *rewrite theory* $(\Sigma, E \cup A, R)$, where $(\Sigma, E \cup A)$ is a *membership equational logic theory* [6], with Σ an algebraic signature declaring sorts, subsorts, and function symbols, E a set of conditional equations, and A a set of equational axioms. It specifies the system's state space as an algebraic data type. R is a set of *labeled conditional rewrite rules*, specifying the system's local transitions, of the form $[l] : t \longrightarrow t'$ **if** *cond*, where *cond* is a condition and l is a label. Such a rule specifies a transition from an instance of t to the corresponding instance of t', provided the condition holds.

Maude [6] is a language and tool for specifying, simulating, and model checking rewrite theories. The distributed state of an object-oriented system is formalized as a *multiset* of objects and messages. A class C with attributes att_1 to att_n of sorts s_1 to s_n is declared `class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n. An object of class C is modeled as a term `<` o `:` C `|` att_1 `:` v_1`, ... ,` att_n `:` v_n `>`, with o its object identifier, and where the attributes att_1 to att_n have the current values v_1 to v_n, respectively. Upon receiving a message, an object can change its state and/or send messages to other objects. For example, the rewrite rule

```
rl [l] :  m(O,z)   < O : C | a1 : x, a2 : O' >
      =>           < O : C | a1 : x + z, a2 : O' >  m'(O',x + z) .
```

defines a transition where an incoming message `m`, with parameters `O` and `z`, is consumed by the target object `O` of class `C`, the attribute `a1` is updated to `x + z`, and an outgoing message `m'(O',x + z)` is generated.

2.4 Statistical Model Checking and PVeStA

Probabilistic distributed systems can be modeled as *probabilistic rewrite theories* [1] with rules of the form

$$[l] : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \ \ \textbf{if} \ \ cond(\overrightarrow{x}) \ \ \textit{with probability} \ \ \overrightarrow{y} := \pi(\overrightarrow{x})$$

where the term t' has new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. The concrete values of the new variables $\overrightarrow{y}$ in $t'(\overrightarrow{x}, \overrightarrow{y})$ are chosen probabilistically according to the probability distribution $\pi(\overrightarrow{x})$.

Statistical model checking [20,23] is an attractive formal approach to analyzing (purely) probabilistic systems. Instead of offering a yes/no answer, it can verify a property up to a user-specified level of confidence by running Monte-Carlo simulations of the system model. We then use PVeStA [2], a parallelization of the tool VeStA [21], to statistically model check purely probabilistic systems against properties expressed as QuaTEx expressions [1]. The expected value of a QuaTEx expression is iteratively evaluated w.r.t. two parameters α and δ by sampling, until we obtain a value v so that with $(1-\alpha)100\%$ statistical confidence, the expected value is in the interval $[v - \frac{\delta}{2}, v + \frac{\delta}{2}]$.

3 A Formal Model of Walter in Maude

This section defines a formal executable model of Walter in Maude. The whole model is available at `https://sites.google.com/site/siliunobi/walter`.

3.1 Data Types, Classes, and Messages

We formalize Walter in an object-oriented style, where the state consists of a number of *replica* (or *site*) objects, each modeling a local database, and a number of messages traveling between the objects. A *transaction* is formalized as an object which resides inside the replica object that executes the transaction.

Some Data Types. A *version* is a pair `version(oid,sqn)` consisting of a site `oid` where the transaction is executed, and a sequence number `sqn` local to that site. A vector timestamp is a map from site identifiers to sequence numbers:

```
pr MAP{Oid,Nat} * (sort Map{Oid,Nat} to VectorTimestamp) .
```

The sort `OperationList` represents lists of read and write operations as terms such as (x `:= read` $k1$) (y `:= read` $k2$) `write(`$k1, x+y$`)`, where `LocalVar` denotes the "local variable" that stores the value of the key read by the operation, and `Expression` is an expression involving the transaction's local variables:

```
op write : Key Expression -> Operation [ctor] .
op _:=read_ : LocalVar Key -> Operation [ctor] .
op waitRemote : Key LocalVar -> Operation [ctor] .
pr LIST{Operation} * (sort List{Operation} to OperationList) .
```

`waitRemote(`k, x`)` means that the transaction execution is awaiting the value of the key (or data item) k from a remote site to be assigned to the local variable x.

Classes. A *transaction* is modeled as an object of the following class Txn:

```
class Txn | operations : OperationList,  readSet : ReadSet,
            writeSet : WriteSet,         localVars : LocalVars,
            startVTS : VectorTimestamp,  txnSQN : Nat .
```

The `operations` attribute denotes the transaction's remaining operations. The `readSet` attribute denotes the versions of data items read by the transaction as a ','-separated set of pairs `versionRead(`k`,`*version*`)`. `writeSet` denotes the write set of the transaction as a map `(`k_1 `|->` val_1`), ..., (`k_n `|->` val_n`)`. `localVars` maps the transaction's local variables to their current values. `startVTS` refers to the vector timestamp assigned to the transaction when it starts to execute, and `txnSQN` is the transaction's sequence number given upon commit.

A *replica*, or *site*, stores parts of the database, and executes the transactions for which it is the host/server. A replica is formalized as an object instance of the following class `Replica`:

```
class Replica | history : Datastore,  sqn : Nat,  gotTxns : ObjectList,
                executing : ObjectList,  committed : ObjectList,
                aborted : ObjectList,  committedVTS : VectorTimestamp,
                gotVTS : VectorTimestamp,  locked : Locks,
                votes : Vote,  voteSites : TxnSites,  abortSites : TxnSites,
                dsSites : PropagateSites,  vsbSites : TxnSites,
                dsTxns : OidSet,  gvTxns : OidSet,
                recPropTxns : PropagatedTxns,  recDurableTxns : DurableTxns .
```

The `history` attribute represents the site's local database, as well as propagated updates also on data items not stored at the replica, as a map from keys to lists of updates `<` *value* `,` *version* `>`:

```
op <_,_> : Value Version -> ValueVersion [ctor] .

pr LIST{ValueVersion} * (sort List{ValueVersion} to ValueVersionList) .
pr MAP{Key,ValueVersionList} * (sort Map{Key,ValueVersionList} to Datastore) .
```

The `sqn` attribute denotes the replica's current local sequence number. The attributes `gotTxns`, `executing`, `committed` and `aborted` denote the transaction (objects) which are, respectively, waiting to be executed, executing, committed, and aborted. A site executes transactions sequentially. Concurrent transactions can be modeled by transactions executed at different sites. The attributes `committedVTS` and `gotVTS` indicate for each site how many transactions of that site have been committed at, respectively, received by, this site. The `locked` attribute denotes the locked keys and their associated transactions at this site:

```
op lock : Oid Key -> Lock .  --- Txn Oid locks Key
pr SET{Lock} * (sort Set{Lock} to Locks) .
```

The `votes` attribute denotes a collection of votes in the two-phase commit:

```
sort Vote .
op noVote : -> Vote [ctor] .
op vote : Oid Oid Bool -> Vote [ctor] .  --- Txn, Participant, vote
op _;_ : Vote Vote -> Vote [ctor assoc comm id: noVote] .
```

The `voteSites` attribute refers to, for each transaction, the remaining replicas from which the coordinator is awaiting votes:

```
sort TxnSites .
op noTS : -> TxnSites [ctor] .
op txnSites : Oid OidSet -> TxnSites [ctor] .
op _;_ : TxnSites TxnSites -> TxnSites [ctor assoc comm id: noTS] .
```

Similarly, the attribute `abortSites` denotes for each transaction the remaining sites from which the coordinator is awaiting the acknowledgments to abort the transaction. (The coordinator first notifies the corresponding sites to abort a transaction, and it will abort it locally after it gets the replies from those sites.)

The remaining attributes refer to the transaction replication. The attributes `dsSites` and `vsbSites` denote the remaining sites from which each transaction is awaiting acknowledgments to mark itself as disaster-safe durable or globally visible transaction, respectively. The sort `PropagateSites` contains the keys in each transaction's write set, because for a transaction to be disaster-safe durable each key must be replicated:

```
sort PropagateSites .
op noPS : -> PropagateSites [ctor] .
op propagateSites : Oid Key OidSet -> PropagateSites [ctor] .
op _;_ : PropagateSites PropagateSites ->
          PropagateSites [ctor assoc comm id: noPS] .
```

The attributes `dsTxns` and `gvTxns` denote the set (of sort `OidSet`) of disaster-safe durable and globally visible transactions, respectively. The last two attributes `recPropTxns` and `recDurableTxns` buffer the received propagation and disaster-safe durable messages from the coordinator.

The state also contains an object mapping each key to the sites storing the key (these sites are also called the *replicas* of the key):

```
class Table | table : ReplicaTable .
```

Elements of sort `ReplicaTable` are '`;;`'-separated sets of terms `sites(`k_i`,`$replicas_i$`)`, where the list $replicas_i$ denotes the sites replicating the key k_i. The first element in such a list is the *preferred site* of the corresponding key:

```
sort KeyReplicas .
op [_] : KeyReplicas -> ReplicaTable [ctor] .
op eptTable : -> KeyReplicas [ctor] .
op sites : Key OidList -> KeyReplicas [ctor] .
op _;;_ : KeyReplicas KeyReplicas -> KeyReplicas [ctor assoc comm id: eptTable] .
```

Initial state. The following shows an initial state (with some parts replaced by '...') with three replicas, r1, r2, and r3, where r1 and r2 are the coordinators for, respectively, transactions t1, and t2 and t3. Key x is replicated at r1 and r2, key y at r2 and r3, and key z at r3 and r1, with r1, r2 and r3 the respective preferred sites. Transaction t1 is the read-only transaction (xl :=read x) (yl :=read y), transaction t2 is a write-only transaction write(y, 3) write(z, 8), while transaction t3 is a read-write transaction on key x. Initially, the value of each key is [0], and its version is version(0,0):

```
eq init =
< tb : Table | table : [sites(x,r1 r2) ;; sites(y,r2 r3) ;; sites(z,r3 r1)] >
< r1 : Replica |
        gotTxns : < t1 : Txn | operations : ((xl :=read x) (yl :=read y)),
                               readSet : empty, writeSet : empty,
                               localVars : (xl |-> [0], yl |-> [0]),
                               startVTS : empty, txnSQN : 0 >,
        history : (x |-> (< [0],version(0,0) >),
                   z |-> (< [0],version(0,0) >)), sqn : 0,  ... >
< r2 : Replica |
        gotTxns : < t2 : Txn | operations : (write(y,3) write(z,8)), ... >
                  < t3 : Txn | operations : ((xl := read x)
                               write(x, xl plus 1)),  ... > ... >
< r3 : Replica | history : (y |-> (< [0],version(0,0) >),
                            z |-> (< [0],version(0,0) >)), ... > .
```

Messages between sites have the form msg *content* from *sender* to *receiver*. The message content (or simply message) request(*key*, *txn*, *vts*) sends a read request for transaction txn to *key*'s preferred site to retrieve its state from the snapshot determined by vector timestamp *vts*. The preferred site replies with a message reply(*txn*, *key*, *value_version*), where *value_version* is chosen based on the incoming vector timestamp. The message prepare(*txn*,*keys*, *vts*) sends the key(s) *keys* in transaction *txn* to their preferred sites with the transaction's start vector timestamp *vts*. Those preferred sites reply with a message prepare-reply(*txn*, *vote*). The messages abort(*txn*) and aborted(*txn*) are sent out when the coordinator distributes the "abort" decision to the participants, and when the participants acknowledge the decision. The message propagate(*txn*, *sqn*, *vts*, *ws*) sends a transaction *txn*'s sequence number *sqn*, vector timestamp *vts*, and write set *ws* to all sites. The sites reply with a message propagate-ack(txn) to acknowledge that the transaction *txn* has been propagated successfully. The message ds-durable(*txn*) is sent to all sites once the transaction *txn* has been marked as disaster-safe durable. The sites then reply with a message visible(*txn*) to acknowledge the notification.

3.2 Formalizing Walter's Behavior

This section formalizes the dynamic behavior of Walter using rewrite rules.

Starting a transaction. A replica starts executing a transaction by moving the first transaction TID in gotTxns to executing, and assigns its committed vector timestamp VTS to the transaction's start vector timestamp:[3]

```
rl [start-txn] :
   < RID : Replica | gotTxns : (< TID : Txn | startVTS : empty > ;; TXNS),
                     executing : emptyTxnList, committedVTS : VTS >
 =>
   < RID : Replica | gotTxns : TXNS,
                     executing : < TID : Txn | startVTS : VTS > > .
```

Executing a transaction. We can now execute the operations of the transaction, and start with a read operation X :=read K. There are three cases to consider: (i) the transaction has already written to key K (buffered in the write set); (ii) there is no preceding write in the transaction but the executing site replicates K; or (iii) neither (i) nor (ii) holds.

In case (i), the local variable X is given the value V buffered in the write set:

```
rl [execute-read-own-write] :
   < RID : Replica | executing :
         < TID : Txn | operations : ((X :=read K) OPS),
                       writeSet : (K |-> V, WS), localVars : VARS > >
 =>
   < RID : Replica | executing :
         < TID : Txn | operations : OPS,
                       writeSet : (K |-> V, WS),
                       localVars : insert(X,V,VARS) > > .
```

In case (ii) (the site RID replicates K: localReplica(K,RID,REPLICA-TABLE)), the replica chooses the last update < V,VERSION > in its local history DS that is visible to the transaction's start snapshot VTS:

```
crl [execute-read-local] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | executing :
          < TID : Txn | operations : ((X :=read K) OPS), writeSet : WS,
                        readSet : RS, localVars : VARS, startVTS : VTS >,
                      history : DS >
  =>
    < TABLE : Table | >
    < RID : Replica | executing :
          < TID : Txn | operations : OPS,  localVars : insert(X,V,VARS),
                        readSet : (versionRead(K,VERSION),RS) > >
  if (not $hasMapping(WS,K)) /\ localReplica(K,RID,REPLICA-TABLE) /\
     < V,VERSION > := choose(VTS,DS[K]) .
```

[3] We do not give variable declarations, but follow the convention that variables are written in (all) capital letters.

In case (iii), the site sends a `request` message (with the transaction's start vector timestamp `VTS`, since the remote site must choose the version consistent with the snapshot) to `K`'s preferred site (`preferredSite(...)`) to fetch the version. The "next operation" of the transaction changes to `waitRemote(K,X)`:

```
crl [execute-read-remote] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | executing :
          < TID : Txn | operations : ((X :=read K) OPS), writeSet : WS,
                        startVTS : VTS > >
  =>
    < TABLE : Table | >
    < RID : Replica | executing :
          < TID : Txn | operations : (waitRemote(K,X) OPS) > >
    (msg request(K,TID,VTS) from RID to preferredSite(K,REPLICA-TABLE))
  if (not $hasMapping(WS,K)) /\ (not localReplica(K,RID,REPLICA-TABLE)) .
```

The remote (preferred) site responds to such a request by sending the snapshot-consistent value and version (`choose(VTS, DS[K])`) of the requested key:

```
rl [receive-remote-request] :
   (msg request(K, TID, VTS) from RID' to RID)
   < RID : Replica | history : DS >
 =>
   < RID : Replica | >
   (msg reply(TID, K, choose(VTS, DS[K])) from RID to RID') .
```

The executing site then merges the fetched value and version in the local history, and updates the read set and local variables:

```
rl [receive-remote-reply] :
   (msg reply(TID, K, < V,VERSION >) from RID' to RID)
   < RID : Replica | history : DS, executing :
         < TID : Txn | operations : (waitRemote(K, X) OPS), readSet : RS,
                       localVars : VARS > >
 =>
   < RID : Replica | executing :
         < TID : Txn | operations : OPS,
                       readSet : (versionRead(K, VERSION), RS),
                       localVars : insert(X, V, VARS) >,
                     history : merge(K, < V,VERSION >, DS) > .
```

When the next transaction operation is a write operation `write(K, EXPR)`, the expression `EXPR` to be written is evaluated w.r.t. the current values of the local variables, and the resulting value is added to the write set:

```
rl [execute-write] :
   < RID : Replica | executing :
         < TID : Txn | operations : (write(K, EXPR) OPS),
                       localVars : VARS, writeSet : WS > >
```

```
 =>
   < RID : Replica | executing :
         < TID : Txn | operations : OPS,
                       writeSet : insert(K,eval(EXPR,VARS),WS) > > .
```

Commit a Transaction. When all the currently executing transaction's operations have been performed, the site starts to commit the transaction. A read-only transaction (`writeSet` is `empty`) is committed locally:

```
rl [commit-read-only-txn] :
   < RID : Replica | committed : TXNS', executing :
         < TID : Txn | operations : nil, writeSet : empty > >
 =>
   < RID : Replica | committed : (TXNS' ;; < TID : Txn | >),
                     executing : emptyTxnList > .
```

There are two cases for committing a write transaction: *fast commit* if the executing site is the preferred site of all keys written by the transaction; and *slow commit* if the transaction's write sets contains keys with non-local preferred sites.

Fast Commit. To fast commit a transaction, two checks for conflicts are performed at the site: one check for any modified key, and another check for any locked key, i.e., a key being committed concurrently by another transaction. `modified(WS,VTS,DS)` checks whether there is a key in the write set `WS` and a version of that key in the history `DS` that is not visible to the snapshot `VTS`, and `locked(WS,LOCKS)` checks whether there is a key in `WS` that also appears in `LOCKS`. The following rule shows the case when both checks pass:

```
crl [fast-commit-success] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | executing :
          < TID : Txn | operations : nil, writeSet : WS,
                        startVTS : VTS, txnSQN : TXNSQN >,
                      committed : TXNS', history : DS, locked : LOCKS,
                      sqn : SQN, committedVTS : VTS', dsSites : PSTS >
  =>
    < TABLE : Table | >
    < RID : Replica | executing : emptyTxnList,
                      committed : (TXNS' ;; < TID : Txn | txnSQN : SQN' >),
                      history : update(WS,version(RID,SQN'),DS),
                      sqn : SQN',  committedVTS : insert(RID,SQN',VTS'),
                      dsSites : PSTS ; txnPropagateSites(TID,WS) >
    propagateTxn(TID,SQN',VTS,WS,allSites(REPLICA-TABLE),RID)
    if WS =/= empty /\ allLocalPreferred(WS,RID,REPLICA-TABLE) /\
       (not modified(WS,VTS,DS)) /\ (not locked(WS,LOCKS)) /\
       SQN' := SQN + 1 .
```

The site commits the transaction by assigning a new local sequence number `SQN'`, and updating the local history (`update(...)`). The site then propagates the transaction to remote sites. This is done by generating propagation messages using `propagateTxn`, which produces one propagation message for each site. The site then keeps track of the sites that have acknowledged the propagation (`txnPropagateSites(...)`).

If either check fails, the transaction is aborted:

```
crl [fast-commit-failed] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | executing :
          < TID : Txn | operations : nil, writeSet : WS, startVTS : VTS >,
                      aborted : TXNS', history : DS, locked : LOCKS >
 =>
    < TABLE : Table | >
    < RID : Replica | executing : emptyTxnList,
                      aborted : (TXNS' ;; < TID : Txn | >) >
    if WS =/= empty /\ allLocalPreferred(WS,RID,REPLICA-TABLE) /\
       (modified(WS,VTS,DS) or locked(WS,LOCKS)) .
```

Slow Commit. Slow commit uses two-phase commit among the preferred sites of the keys in the transaction's write set. The executing site distributes the `prepare` messages to those preferred sites (`allPreferredSites(...)`), asking the participants to vote based on whether the corresponding keys are unmodified and unlocked. The `prepare` messages are produced by the function `prepareTxn`:

```
crl [slow-commit-prepare] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | voteSites : VSTS, executing :
          < TID : Txn | operations : nil, writeSet : WS, startVTS : VTS > >
  =>
    < TABLE : Table | >
    < RID : Replica | voteSites : (VSTS ; voteSites(TID,RIDS)),
                      executing : < TID : Txn | > >
    prepareTxn(TID,keys(WS),VTS,RIDS,REPLICA-TABLE,RID)
    if WS =/= empty /\ (not allLocalPreferred(WS,RID,REPLICA-TABLE)) /\
       RIDS := allPreferredSites(WS,REPLICA-TABLE) /\ (not (TID in VSTS)) .
```

The receiver of a `prepare` message performs the two checks as in fast commit: if either check fails, a `false` vote is sent back; otherwise, the participant locks the key(s) and sends back a `true` vote:

```
rl [slow-commit-receive-prepare] :
   (msg prepare(TID,KS,VTS) from RID' to RID)
   < RID : Replica | locked : LOCKS, history : DS >
 =>
   if (not locked(KS,LOCKS)) and (not modified(KS,VTS,DS))
   then < RID : Replica | locked : (addLock(KS,TID),LOCKS) >
        (msg prepare-reply(TID,true) from RID to RID')
   else < RID : Replica | >
        (msg prepare-reply(TID,false) from RID to RID') fi .
```

When the executing replica receives a vote, it first checks whether all votes have been collected (VSTS'[TID] == empty), and then checks whether all votes associated to the transaction are true votes (allYes(TID, VOTES')). If so, the coordinator decides to propagate the transaction as in the fast commit; otherwise, the coordinator aborts the transaction, and notifies the participants that voted true to release the locks. This is done by producing "abort" messages for the corresponding participants RIDS (propagateAbort(TID,RIDS,RID)). The following rule shows the "aborted" branch:

```
crl [slow-commit-receive-vote-abort] :
    (msg prepare-reply(TID, FLAG) from RID' to RID)
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | votes : VOTES, voteSites : VSTS,
                      abortSites : ABORTS >
  =>
    < TABLE : Table | >
    < RID : Replica | votes : VOTES', voteSites : VSTS',
                      abortSites : ABORTS ; voteSites(TID,RIDS) >
    propagateAbort(TID, RIDS, RID)
    if VSTS' := remove(TID, RID', VSTS) /\
       VOTES' := VOTES ; vote(TID, RID', FLAG) /\
       VSTS'[TID] == empty /\ (not allYes(TID, VOTES')) /\
       RIDS := yesSites(TID, VOTES') .
```

The abort procedure is straightforward: the participant releases the lock(s) held by the transaction TID, and the executing site aborts the transaction:

```
rl [slow-commit-receive-abort] :
   (msg abort(TID) from RID' to RID)
   < RID : Replica | locked : LOCKS >
 =>
   < RID : Replica | locked : release(TID,LOCKS) >
   (msg aborted(TID) from RID to RID') .

crl [slow-commit-receive-aborted] :
    (msg aborted(TID) from RID' to RID)
    < RID : Replica | executing : < TID : Txn | >, aborted : TXNS',
                      abortSites : ABORTS >
  =>
    (if ABORTS'[TID] == empty  --- all acks received; abort the txn locally
     then < RID : Replica | executing : emptyTxnList,
                            aborted : (TXNS' ;; < TID : Txn | >),
                            abortSites : ABORTS' >
     else < RID : Replica | abortSites : ABORTS' > fi)
   if ABORTS' := remove(TID,RID',ABORTS) .
```

Transaction Propagation. After a transaction commits, the executing site propagates it to other sites by invoking the propagation protocol. Upon receiving a propagation message for transaction TID, the receiving site performs two checks: (i) whether it has gotten all transactions that causally precede transaction TID, and (ii) all transactions from TID's executing site with a smaller sequence number. (i) is indicated by VTS' gt VTS, meaning that the latest snapshot the site got is greater than the incoming snapshot VTS, and (ii) by s(VTS'[RID']) == SQN, meaning that the corresponding latest sequence number the site got is exactly one smaller than the incoming sequence number SQN. If either check fails, the site buffers the propagated information regarding the transaction (nonPropagatedTxns), and waits until the "missing" transactions are propagated to it; otherwise, the transaction is considered to be propagated successfully (propagatedTxns), and the site updates its local history (if the site is not the coordinator itself), and then sends back the acknowledgment:

```
crl [receive-propagate] :
    (msg propagate(TID,SQN,VTS,WS) from RID' to RID)
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | gotVTS : VTS', history : DS, recPropTxns : PTXNS >
  =>
    < TABLE : Table | >
    (if s(VTS'[RID']) == SQN and (VTS' gt VTS)
     then if RID =/= RID'
          then < RID : Replica | gotVTS : VTS'', history : DS',
                                 recPropTxns : PTXNS' >
               (msg propagate-ack(TID) from RID to RID')
          else < RID : Replica | gotVTS : VTS'', recPropTxns : PTXNS' >
               (msg propagate-ack(TID) from RID to RID')
          fi
     else < RID : Replica | recPropTxns : PTXNS'' >
     fi)
     if PTXNS' := propagatedTxns(TID,SQN,VTS) ; PTXNS /\
        PTXNS'' := nonPropagatedTxns(TID,SQN,VTS,WS,RID') ; PTXNS /\
        VTS'' := insert(RID',SQN,VTS') /\
        DS' := update(locRepWS(WS,RID,REPLICA-TABLE),version(RID',SQN),DS) .
```

A failed propagated transaction (nonPropagatedTxns) is acknowledged whenever those two checks pass. The site transforms nonPropagatedTxns to propagatedTxns, and sends back the acknowledgment:

```
crl [later-propagate-ack] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | gotVTS : VTS', history : DS, recPropTxns :
                      (nonPropagatedTxns(TID, SQN, VTS, WS, RID') ; PTXNS) >
  =>
    < TABLE : Table | >
    (if RID =/= RID'
     then < RID : Replica | gotVTS : VTS'', history : DS', recPropTxns :
                            (propagatedTxns(TID, SQN, VTS) ; PTXNS) >
          (msg propagate-ack(TID) from RID to RID')
     else < RID : Replica | gotVTS : VTS'', history : DS, recPropTxns :
                            (propagatedTxns(TID, SQN, VTS) ; PTXNS) >
          (msg propagate-ack(TID) from RID to RID')
     fi)
    if s(VTS'[RID']) == SQN /\ VTS' gt VTS /\
       VTS'' := insert(RID',SQN,VTS') /\
       DS' := update(locRepWS(WS,RID,REPLICA-TABLE),version(RID',SQN),DS) .
```

When the executing site has collected propagation acknowledgments from $f + 1$ sites, it marks the transaction as disaster-safe durable. This is done by the function `dsDurable`, which counts the number of received acks in `dsSites`. The site also distributes the decision to all sites by using function `dsDurableTxn` to produce a `ds-durable` message to each site, and records that information in `vsbSites`. If there is no need to distribute the decision, the transaction is marked as globally visible directly (by adding it to `gvTxns`):

```
crl [receive-propagate-ack] :
    (msg propagate-ack(TID) from RID' to RID)
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Replica | dsSites : PSTS, vsbSites : VSBS,
                      committed : (TXNS ;; < TID : Txn | writeSet : WS,
                          startVTS : VTS, txnSQN : SQN > ;; TXNS'),
                      dsTxns : DSTXNS, gvTxns : GVTXNS >
  =>
    < TABLE : Table | >
    (if dsDurable(TID,PSTS')
     then if RIDS =/= empty
          then < RID : Replica | dsSites : PSTS', vsbSites : VSBS',
                                 dsTxns : (TID, DSTXNS) >
               dsDurableTxn(TID,RIDS,RID)
          else < RID : Replica | dsSites : PSTS', vsbSites : VSBS',
                                 dsTxns : (TID, DSTXNS),
                                 gvTxns : (TID, GVTXNS) >
          fi
     else < RID : Replica | dsSites : PSTS' >
     fi)
    if PSTS' := add(TID,keys(WS),RID',REPLICA-TABLE,PSTS) /\
       (not TID in DSTXNS) /\ RIDS := allServers(REPLICA-TABLE) \ RID /\
       VSBS' := VSBS ; voteSites(TID,RIDS) .
```

A propagation acknowledgment that arrives after the transaction has been marked as disaster-safe durable is ignored:

```
rl [receive-propagate-ack-after-ds-durable-mark] :
   (msg propagate-ack(TID) from RID' to RID)
   < RID : Replica | dsTxns : TID , DSTXNS >
 =>
   < RID : Replica | > .
```

Upon receiving the “disaster-safe durable” decision, the site tries to commit the transaction locally:

```
crl [receive-ds-durable-visible] :
    (msg ds-durable(TID) from RID' to RID)
    < RID : Replica | recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
                      recDurableTxns : DTXNS, committedVTS : VTS',
                      locked : LOCKS >
  =>
    < RID : Replica | recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
                      recDurableTxns : (durableTxns(TID) ; DTXNS),
                      committedVTS : insert(RID',SQN,VTS'),
                      locked : release(TID,LOCKS) >
    (msg visible(TID) from RID to RID')
    if VTS' gt VTS /\ s(VTS'[RID']) == SQN .
```

To commit transaction `TID`, the site must pass three checks: (i) the propagation message has been received and acknowledged (`propagatedTxns(TID,SQN,VTS)` shown in `recPropTxns`), (ii) `VTS'` is greater than `VTS`, and (iii) all transactions from `TID`'s executing site with a smaller sequence number have been received (`s(VTS'[RID']) == SQN`). A `visible` message is then sent back, and all corresponding locks are released.

The site fails to commit the transaction immediately after receiving the decision if any check fails. The following rule shows the case when the site has not yet acknowledged the propagation:

```
rl [receive-ds-durable-not-visible-not-ack-propagated] :
   (msg ds-durable(TID) from RID' to RID)
   < RID : Replica | recPropTxns : (nonPropagatedTxns(TID,SQN,VTS,WS,RID')
                     ; PTXNS), recDurableTxns : DTXNS >
 =>
   < RID : Replica | recPropTxns : (nonPropagatedTxns(TID,SQN,VTS,WS,RID')
                     ; PTXNS), recDurableTxns : (nonDurableTxns(TID,RID')
                     ; DTXNS) > .
```

The site commits any failed committed transaction (`nonDurableTxns`) whenever those checks pass, by changing `nonDurableTxns` to `durableTxns`. It also sends back a `visible` message, updates the committed vector timestamp, and releases all corresponding locks:

```
crl [later-visible] :
    < RID : Replica | recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
                      recDurableTxns : (nonDurableTxns(TID,RID') ; DTXNS),
                      committedVTS : VTS', locked : LOCKS >
  =>
    < RID : Replica | recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
                      recDurableTxns : (durableTxns(TID) ; DTXNS),
                      committedVTS : insert(RID',SQN,VTS'),
                      locked : release(TID,LOCKS) >
    (msg visible(TID) from RID to RID')
    if VTS' gt VTS /\ s(VTS'[RID']) == SQN .
```

Finally, after receiving `visible` messages from all sites, the executing site marks the transaction as globally visible:

```
crl [receive-visible] :
    (msg visible(TID) from RID' to RID)
    < RID : Replica | vsbSites : VSBS, gvTxns : GVTXNS >
  =>
    (if VSBS'[TID] == empty
     then < RID : Replica | vsbSites : VSBS', gvTxns : (TID , GVTXNS) >
     else < RID : Replica | vsbSites : VSBS' >
     fi)
    if VSBS' := remove(TID,RID',VSBS) .
```

4 Correctness Analysis

In this section we use reachability analysis—from all initial system configurations up to given bounds on the number of transactions, sites, etc.—to analyze whether Walter satisfies PSI and SI. To analyze these complex properties, we use a novel technique which adds a "logical global clock" to record the global order of transaction starts and commits/aborts.

4.1 Parametric Generation of Initial States

To analyze all possible all initial configurations we introduce a new operator `init` so there is a one-step rewrite `init(`*parameters*`)` $\longrightarrow c_0$ for each possible initial configuration c_0, and declare a sort for *sets* of configurations:

```
sort ConfigSet .  subsort Configuration < ConfigSet .
op empty : -> ConfigSet .
op _;_ : ConfigSet ConfigSet -> ConfigSet [assoc comm id: empty] .
```

We define a function

```
op initAux : s1 ... sn -> ConfigSet .
```

such that `initAux(`*params*`,`*params′*`)` generates all possible initial states for such parameters, and add the following rewrite rule to our model:

```
var C : Configuration .  var CS : ConfigSet .
crl [init] : init(params) => C if C ; CS := initAux(params,params') .
```

`init`'s parameters are the number of read-only transactions, the number of write-only transactions, the number of read-write transactions, the number of sites, the number of keys, and the replication factor. Each transaction has two operations.

We start with generating the replica table and key-variable pairs. `keyVars` consists of "`;`"-separated key-variable pairs $< k_1, var_1 >$; ... ; $< k_n, var_n >$, each of which has a key k and the associated local variable var. The function `kvars` extracts `KEYS` key-variable pairs from `keyVars`:

```
--- generate table and key-var pairs:
crl initAux(RTX,WTX,RWTX,SITES,KEYS,RF,none)
 => $initAux(RTX,WTX,RWTX,SITES,KVARS,genKeyVarSet(KVARS),RF,
    < 0 : Table | table : initTable(KVARS) >)
 if KVARS := kvars(KEYS,keyVars) .
```

The function `initTable` initializes the replica table for each key with its replicas of `nil`:

```
--- initialize table with generated keys:
op initTable : KeyVars -> ReplicaTable .
op $initTable : KeyVars ReplicaTable -> ReplicaTable .
eq initTable(KVARS) = $initTable(KVARS,[emptyTable]) .
eq $initTable((< K,VAR > ; KVARS),[KEYREPLICAS]) =
     $initTable(KVARS,[sites(K,nil) ;; KEYREPLICAS]) .
eq $initTable(noKeyVar,[KEYREPLICAS]) = [KEYREPLICAS] .
```

We now generate replicas, assign the keys to them, and update the replica table accordingly:

```
--- generate replicas:
rl $initAux(RTX,WTX,RWTX,s PARS,KVARS,KS,RF,C)
=> $initAux(RTX,WTX,RWTX,PARS,KVARS,KS,RF,C
     < s PARS : Replica | gotTxns : emptyTxnList, history : empty, sqn : 0,
       executing : emptyTxnList, committed : emptyTxnList,
       aborted : emptyTxnList, committedVTS : empty, gotVTS : empty,
       locked : empty, dsSites : noPS, vsbSites : noVS, dsTxns : empty,
       gvTxns : empty, recPropTxns : noPT, recDurableTxns : noDT,
       votes : noVote, voteSites : noVS, abortSites : noVS >) .

--- assign keys to replicas and update table accordingly:
crl $initAux(RTX,WTX,RWTX,0,(< K,VAR > ; KVARS),KS,s RF,
    < RID : Replica | history : VS >
    < 0 : Table | table : [sites(K,RIDS) ;; KEYREPLICAS] > C)
 => $initAux(RTX,WTX,RWTX,0,(< K,VAR > ; KVARS),KS,RF,
    < RID : Replica | history : (VS,K |-> (< [0],version(0,0) >)) >
    < 0 : Table | table : [sites(K,RIDS RID) ;; KEYREPLICAS] > C)
    if not $hasMapping(VS,K) .
```

Note that, to assign a key to a replica, we *nondeterministically* add a replica `RID` to key `K`'s replicating sites. Once the key has been assigned to *replication factor* replicas, we continue to the next key by resetting the replication factor to `rf`:

```
--- next key
rl $initAux(RTX,WTX,RWTX,0,(< K,VAR > ; KVARS),KS,0,C)
=> $initAux(RTX,WTX,RWTX,0,KVARS,KS,rf,C) .
```

We are now ready to generate transactions. We only illustrate how to generate read-write transactions; generating read/write-only transactions is similar, and is given at `https://sites.google.com/site/siliunobi/walter`.

```
--- generate rw-txns
rl $initAux(RTX,WTX,s RWTX,0,noKeyVar,(< K,VAR >,KS),RF,
   < RID : Replica | gotTxns : emptyTxnList > C)
=> $initAux(RTX,WTX,RWTX,0,noKeyVar,(< K,VAR >,KS),RF,
   < RID : Replica | gotTxns :
     < s RWTX : Txn | operations : ((VAR :=read K) write(K,s RWTX)),
       readSet : empty, writeSet : empty, localVars : (VAR |-> [0]),
       startVTS : empty, txnSQN : 0 > > C) .
```

Note that VAR in the key-variable pair < K,VAR > is used to initialize the local variable VAR |-> [0].

One of 768 initial states generated by init(1,1,1,2,2,2) is

```
< 0 : Table | table :[replicatingSites(k1,1 2);;replicatingSites(k2,2 1)]>
< 1 : Replica | gotTxns : (< 3 : Txn | localVars :(k1l |->[0], k2l |->[0]),
                operations :((k2l :=read k2) (k1l :=read k1)), ... >),
                history :(k1 |-> <[0],version(0,0)>,
                          k2 |-> <[0],version(0,0)>), ... >
< 2 : Replica | gotTxns :( < 2 : Txn | localVars :(k1l |->[0], k2l |->[0]),
                operations :(write(k2,1) write(k1,2)), ... > ;;
                           < 1 : Txn | localVars : k2l |->[0],
                operations :((k2l :=read k2) write(k2,1)), ... >),
                history :(k1 |-> <[0],version(0,0)>,
                          k2 |-> <[0],version(0,0)>), ... >
```

where both k1 and k2 are replicated at sites 1 and 2, and have preferred sites 1 and 2, respectively. Site 1 has one read-only transaction to execute, and site 2 has one write-only and one read-write transaction to execute.

4.2 Analyzing the Correctness Properties

This section formalizes SI and PSI as reachability properties and analyzes them using Maude. To analyze these properties we add to the state an object

```
< m : Monitor | clock : clock, log : log >
```

which stores crucial information about the whole execution. The *clock* is a kind of "logical global clock" that totally orders transaction starts and commits/aborts. This logical global clock is incremented by one every time a transaction starts executing, and every time a transaction is committed or aborted somewhere. The *log* maps each transaction to a record record(*rid*, *issueTime*, *finishTime*, *committed*, *reads*, *writes*), with *rid* the transaction's host site, *issueTime* its issue "time" according to the logical global clock, *finishTime* its commit/abort "times" at each site, *committed* a flag that is true if the transaction is committed, *reads* its key/versions read, and *writes* its write set.

We modify our rewrite rules to update the Monitor whenever a transaction starts or is committed/aborted somewhere. For example, when a site commits a propagated transaction, the monitor records the commit time GT for that transaction at site RID and increments the logical global time by one:

```
crl [receive-ds-durable-visible] :
    (msg ds-durable(TID) from RID' to RID)
    < M : Monitor | clock : GT,
                    log : (TID |-> record(RID',T1,VTS1,true,READS,WRITES)
                           , LOG) >
    < RID : Replica | recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
                      recDurableTxns : DTXNS, committedVTS : VTS',
                      locked : LOCKS >
  =>
    < M : Monitor | clock : GT + 1 ,
                    log : (TID |-> record(RID',T1,insert(RID,GT,VTS1),
                                          true,READS,WRITES) , LOG) >
    < RID : Replica | recDurableTxns : (durableTxns(TID) ; DTXNS),
                      committedVTS : insert(RID',SQN,VTS'),
                      locked : release(TID,LOCKS) >
    (msg visible(TID) from RID to RID')
    if VTS' gt VTS /\ s(VTS'[RID']) == SQN .
```

Since Walter is terminating if a finite number of transactions are issued, we check the consistency properties by inspecting this monitor object in the final states, when all transactions have finished.

Formalizing Snapshot Isolation. SI is defined by two properties in [22]:

- SI-1 (Snapshot Read): All operations in a transaction read the most recent committed version as of time when the transaction began.
- SI-2 (No Write-Write Conflicts): The write sets of each pair of committed concurrent[4] transactions must be disjoint.

We analyze SI-1 (and all other properties) by searching for a reachable *final* state whose system log shows that the execution did not satisfy the property. The following function `p1-si` returns `true` if and only the system log at the end of the execution shows that property SI-1 is satisfied:

```
op p1-si : Log -> Bool .

ceq p1-si(TID1 |-> record(RID1,TS1,VTS1,true,(v(X,V),RS),WS) ,
          TID2 |-> record(RID2,TS2,(RID |-> TC,VTS2),true,RS',
                          (v(X,V),WS')) ,
          TID3 |-> record(RID3,TS3,(RID' |-> TC',VTS3),true,RS'',
                          (v(X,V'),WS'')) , LOG) = false
    if V =/= V' /\ TC' < TS1 /\ TC' > TC .

ceq p1-si(TID1 |-> record(RID1,TS1,VTS1,true,
                          (v(X,version(0,0)),RS),WS) ,
          TID2 |-> record(RID2,TS2,(RID |-> TC,VTS2),true,RS',
                          (v(X,V),WS')) , LOG) = false
```

[4] Two committed transactions are *concurrent* if one of them has a commit timestamp between the start and the commit timestamp of the other.

```
   if TC < TS1 .

eq p1-si(LOG) = true [owise] .
```

The first equation handles the case when a transaction `TID1` reads another transaction `TID2`'s versions written (matched by `v(X,V)`), while the most recent committed version from `TID1`'s perspective is in fact `v(X,V')` (denoted by the condition, where `TID3`'s commit time `TC'` is between `TID1`'s start time `TS1` and `TID2`'s commit time `TC`). The second equation handles the case when a transaction reads the initial version, in which case any other version committed before the transaction began serves as the most recent committed version.

Likewise, the function `p2-si` inspects the system log and returns `true` if and only if the recorded execution satisfies property SI-2:

```
op p2-si : Log -> Bool .

ceq p2-si(TID1 |-> record(RID1,TS1,VTS1,true,RS,(v(X,V),WS)),
          TID2 |-> record(RID2,TS2,(RID |-> TC,VTS2),true,RS',
                           (v(X,V'),WS')), LOG) = false
   if TC > TS1 /\ TC < max(VTS1) .

eq p2-si(LOG) = true [owise] .
```

The first equation characterizes the case when SI-2 does *not* hold: There are two transactions `TID1` and `TID2` that both wrote key/data item `X` (since `v(X, V)` and `v(X, V')` are in the respective write sets). Furthermore, some site `RID` committed `TID2` at logical time `TC`, which comes after the start time `TS1` of `TID1` but before the latest commit time (`max(VTS1)`) of `TID1`. The committed (flags are `true`) transactions `TID1` and `TID2` were therefore concurrent and wrote the same key, and hence we have a write-write conflict.

Formalizing Parallel Snapshot Isolation. As mentioned in Section 2.1, PSI is given by three properties [22]:

- PSI-1 (Site Snapshot Read): All operations read the most recent committed version at the transaction's site as of time when the transaction began.
- PSI-2 (No Write-Write Conflicts): The write sets of each pair of committed *somewhere-concurrent*[5] transactions must be disjoint.
- PSI-3 (Commit Causality Across Sites): If a transaction T_1 commits at a site A before a transaction T_2 starts at site A, then T_1 cannot commit after T_2 at any site.

The following function `p1-psi` checks whether or not the execution recorded in the system log satisfied PSI-1, by returning `false` if there was a transaction that did not read the most recent committed version at its site when it began:

[5] Two transactions are *somewhere-concurrent* if they are concurrent at either of their sites.

```
op p1-psi : Log -> Bool .

ceq p1-psi(TID1 |-> record(RID1,TS1,VTS1,true,(v(X,V),RS),WS),
           TID2 |-> record(RID2,TS2,(RID1 |-> TC,VTS2),true,RS',
                            (v(X,V),WS')) ,
           TID3 |-> record(RID3,TS3,(RID1 |-> TC',VTS3),true,RS'',
                            (v(X,V'),WS'')) , LOG) = false
   if V =/= V' /\ TC' < TS1 /\ TC' > TC .

ceq p1-psi(TID1 |-> record(RID1,TS1,VTS1,true,
                            (v(X,version(0,0)),RS),WS) ,
           TID2 |-> record(RID2,TS2,(RID1 |-> TC,VTS2),true,RS',
                            (v(X,V),WS')) , LOG) = false
    if TC < TS1 .

eq p1-psi(LOG) = true [owise] .
```

In the first equation, the transaction TID1, hosted at site RID1, has a version v(X, V) in its read set. This version was written by transaction TID2. However, there is a transaction TID3 that wrote version v(X, V') and was committed *at RID1 after* TID2 was committed at RID1 (TC' > TC) and *before* TID1 started executing (TC' < TS1). Hence, the version v(X, V) read by TID1 was too old.

The second equation above defines the bad case when a transaction TID1 hosted at RID1 reads the initial version v(X, version(0,0)) even though there was a version v(X,V) written by a transaction TID2 that committed at RID1 before TID1 began executing (TC < TS1).

The function p2-psi checks whether PSI-2 holds in the execution reflected in the system log, by checking whether there is a write-write conflict between any pair of committed *somewhere-concurrent transactions* in the system log:

```
op p2-psi : Log -> Bool .

ceq p2-psi(TID1 |-> record(RID1, TS1, (RID1 |-> TC , VTS1), true, RS,
                            (v(X,V), WS)) ,
           TID2 |-> record(RID2, TS2, (RID1 |-> TC' , VTS2), true, RS',
                            (v(X,V') , WS')) , LOG) = false
    if TC' > TS1 and TC' < TC .

eq p2-psi(LOG) = true [owise] .
```

This is similar to the equation for p2-si. The difference is that we check whether the transactions with the write conflict are concurrent *at the transaction TID1's site RID1*. Here, TID2 commits *at RID1* at time TC', which is between TID1's start time TS1 and its commit time TC at RID1.

Finally, we define a function p3-psi that analyzes PSI-3 by checking whether there was "bad situation" in which a transaction TID1 committed at site RID2 *before* a transaction TID2 started at site RID2 (TC1 < TS2), while TID1 committed at site RID *after* TID2 committed at site RID (TC1 > TC2):

```
op p3-psi : Log -> Bool .

ceq p3-psi((TID1 |-> record(RID1,TS1,(RID2 |-> TC ,RID |-> TC1 ,VTS1),
                            true,RS,WS),
            TID2 |-> record(RID2,TS2,(RID1 |-> TC' ,RID |-> TC2 ,VTS2),
                            true,RS',WS') , LOG)) = false
    if TC < TS2 /\ TC1 > TC2 .

eq p3-psi(LOG) = true [owise] .
```

Analysis Results. We have analyzed Walter from all initial states with up to 3 transactions, 2 sites, 2 keys, and 2 replicas per key. The following command searches for a reachable final state where the log shows that SI-1 is violated:

```
Maude> (search [1] init(1,0,2,2,2,2) =>!
          < M:Oid : Monitor | log : LOG:Log >  C:Configuration
             such that not p1-si(LOG:Log) .)

Solution 1
...
LOG:Log --> 1 |-> record(2,1,(1 |-> 6, 2 |-> 3),true,
                         v(k1,version(0,0)),v(k1,version(2,1))),
            2 |-> record(1,0,1 |-> 2,false,empty,empty),
            3 |-> record(1,4,1 |-> 5,true,
                         (v(k1,version(0,0)), v(k2,version(0,0))),empty) ...
```

The counterexample shows a *long fork* anomaly that violates SI-1: transaction 3 read version v(k1,version(0,0)), while it should have read version v(k1,version(2,1)) to satisfy SI. The reason is that when transaction 3 begins (at time 4), transaction 1 has already committed on site 2 (at time 3), and therefore the most recent committed version for transaction 3 to read is version(2,1). The anomaly happens because transaction 1 committed on site 1 at time 6, which is after transaction 3 established the snapshot at time 4.

Table 1. Results from model checking Walter against SI and PSI with up to 3 transactions, 2 sites, 2 keys and 2 replicas per key; × indicates that a counterexample was found, and ✓ shows that no counterexample was found.

	SI-1	SI-2	PSI-1	PSI-2	PSI-3
1 read-only, 2 read-write	×	×	✓	✓	✓
1 read-only, 1 write-only, 1 read-write	×	×	✓	✓	✓
3 read-write	×	×	✓	✓	✓
2 read-only, 1 read-write	×	✓	✓	✓	✓
2 read-only, 1 write-only	×	✓	✓	✓	✓

We have also performed our analysis with other combinations of 3 transactions. Table 1 summarizes our analysis results, showing that PSI is satisfied in all our cases. Each `search` command took about 2 hours (worst-case) to execute on a 3.4 GHz × 8 Intel Core i7-2600 CPU with 11.7 GB memory.

5 Performance Estimation by Statistical Model Checking

In this section we use PVeStA statistical model checking to estimate the performance of Walter. Our evaluations are consistent with those measured experimentally in [22]. In addition, we evaluate Walter in a wider range of settings than in [22], thereby providing further insight about Walter. For example, the experiments with fast commit in [22] assume full replication, whereas we also experiment with a partially replicated setting (which necessitates remote reads, etc.), and with workloads involving both slow and fast commits.

5.1 Probabilistic Modeling of Walter

For statistical model checking in PVeStA we need to eliminate nondeterminism in the untimed nondeterministic model in Section 3, and for performance estimation we need to add time and probabilities. All of this can be achieved by following the techniques in [8] and *probabilistically* assign to each message a *delay*. The idea is that if each rewrite rule is triggered by the arrival of a message (either directly, or indirectly by becoming enabled as a result of applying a rule that is triggered by the arrival of a message) and the delay is sampled probabilistically from a dense/continuous time interval, then the probability that two messages have the same delay is 0, and hence no two actions are enabled simultaneously, eliminating nondeterminism and introducing time.

In more detail, nodes send messages of the form `[`Δ`,`*rcvr* `<-` *msg*`]`, where Δ is the message delay, *rcvr* is the recipient, and *msg* is the message content. When time Δ has elapsed, this message becomes a *ripe* message `{`T`,`*rcvr* `<-` *msg*`}`, where T is the "current global time" (used for analysis purposes only). Such a ripe message must then be consumed by the receiver *rcvr* before time advances.

We exemplify with the rule `[receive-remote-request]` how we have transformed the untimed non-probabilistic rewrite rules to the timed and probabilistic setting. In the probabilistic rule below, the incoming message `request` is equipped with the current global time `T`, and the outgoing message `reply` is equipped with a delay `D` sampled from the probability distribution `distr(...)`:

```
rl [receive-remote-request-prob] :
   {T, RID <- request(K,TID,VTS,RID')}
   < RID : Replica | history : DS >
 =>
   < RID : Replica | >
   [D, RID' <- reply(TID,K,choose(VTS,DS[K]),RID)]
  with probability D := distr(...) .
```

5.2 Extracting Performance Measures from Executions

This time we add to the state the monitor object

`<` *m* `: Monitor | log:` *log* `>`.

The `clock` is no longer needed, since now "real" time is given by the message arrival times. Furthermore, since we now analyze transaction throughput and delay, the log is simpler: a list of records `record(`*tid*, *issueTime*, *finishTime*, *committed*`)`, with *tid* the transaction identifier, *issueTime* its issue time, *finishTime* its commit/abort time, and *committed* a flag that is `true` if *tid* is committed.

We define a number of functions on (states with) such a monitor that extract different performance metrics from this "execution log." The function `throughput` computes the number of committed transactions per time unit:

```
op throughput : Config -> Float [frozen] .
eq throughput(< M : Monitor | log: LOG >  REST)
 = committedNumber(LOG) / totalRunTime(LOG) .
```

where `committedNumber` gives the number of committed transactions in `LOG`:

```
op committedNumber : Record -> Float .
op $committedNumber : Record Float -> Float .
eq committedNumber(RECORD) = $committedNumber(RECORD,0.0) .
eq $committedNumber((record(TID,T1,T2,true) ; RECORD),NUMBER) =
     $committedNumber(RECORD,NUMBER + 1.0) .
eq $committedNumber((record(TID,T1,T2,false) ; RECORD),NUMBER) =
     $committedNumber(RECORD,NUMBER) .
eq $committedNumber(noRecord,NUMBER) = NUMBER .
```

and `totalRunTime` returns the time when all transactions are finished (i.e., the largest *finishTime* in `LOG`):

```
op totalRunTime : Record -> Float .
op trt : Record Float -> Float .
eq totalRunTime(RECORD) = trt(RECORD,0.0) .
eq trt((record(TID,T1,T2,FLAG) ; RECORD),T) =
     if T2 > T then trt(RECORD,T2) else trt(RECORD,T) fi .
eq trt(noRecord,T) = T .
```

5.3 Experimental Setup

We performed our experiments with 100 (read-only and/or write-only) transactions, 1 or 5 operations per transaction, 100 keys, and up to 4 sites. The number of sites and the transaction size are the same as in the experiments in [22]. We used lognormal message delay distributions with parameters $\mu = 3$ and $\sigma = 1$ for local delays, and $\mu = 1$ and $\sigma = 2$ for remote delays.

Generating initial states. Statistical model checking verifies a property up to a user-specified level of confidence by running Monte-Carlo simulations from a given initial state. We use an operator `probInit` to *probabilistically* generate initial states: `probInit(`*rtx*`,` *wtx*`,` *rwtx*`,` *sites*`,` *keys*`,` *rf*`,` *rops*`,` *wops*`,` *rwops*`,` *distr*`)` generates an initial state with *rtx* read-only transactions, *wtx* write-only transactions, *rwtx* read-write transactions, *sites* sites, *keys* keys, *rf* replication level, *rops* operations per read-only transaction, *wops* operations per write-only transaction, *rwops* operations per read-write transactions, and *distr* the key access distribution (the probability that an operation accesses a certain key). To capture the fact that some keys may be accessed more frequently than others, we also use Zipfian distributions in our experiments.

Each PVeStA simulation starts from `probInit`, which rewrites to a *different* initial state in each simulation. The reason is that this expression involves generating certain values—such as the transactions—*probabilistically.*

5.4 Statistical Model Checking Results

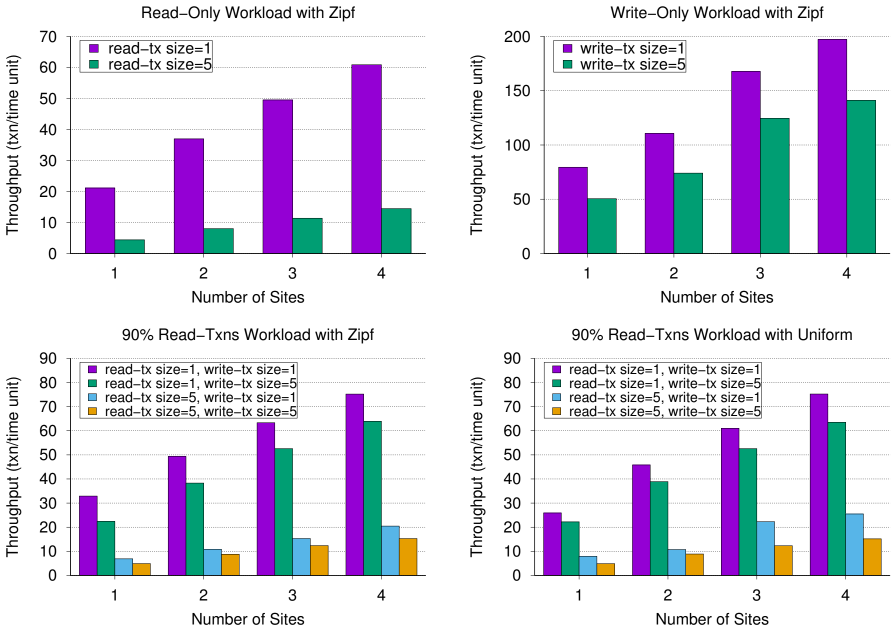

Fig. 1. Throughput with fast commit under different workloads.

The plots in Fig. 1 show the *throughput* with only fast commit as a function of the number of sites, with read-only, write-only or 90% reads workload, and with uniform and Zipfian distributions. The plots show that read throughput scales nearly linearly with the number of sites; write throughput also grows with the number of sites, but not linearly. With a mixed workload, throughput is mostly determined by the transaction size. Our results are consistent with those in [22]. For uniform distribution we only plot the results with a mixed workload.

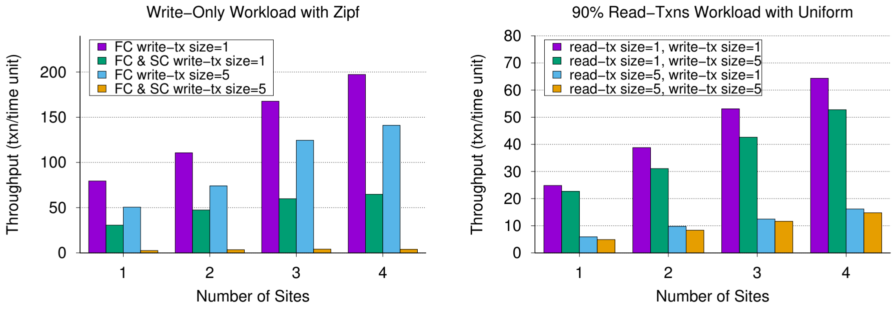

Fig. 2. Throughput with fast commit (FC) and slow commit (SC).

The plots in Fig. 2 show the throughput with mixed (both fast and slow) commit protocols under the same experimental settings as in Fig. 1. As shown in the left plot, throughput is mostly determined by the transaction size in the mixed workload; the trends of, and the differences among, various transaction sizes are consistent with those in Fig. 1. We only plot the results with Zipfian distribution, which are consistent with those with uniform distribution.

Our Maude model—including the infrastructure for statistical model checking—is around 1.8K LOC. Computing the probabilities took a couple of minutes on 30 servers, each with a 64-bit Intel Quad Core Xeon E5530 CPU with 12 GB memory. Each point in the plots represents the average of 3 statistical model checking results. The confidence level for all our statistical experiments is 95%.

6 Related Work

Maude and PVeStA have been used to model and analyze the correctness and performance of a number of distributed data stores: the Cassandra key-value store [13,14,16], RAMP [11,15], Google's Megastore [9,10], and P-Store [19]. In contrast to these papers, our paper formalizes a different state-of-the-art algorithm, Walter, and, in particular, shows how the *snapshot isolation* and *parallel snapshot isolation* consistency models can be formalized and analyzed in Maude. In [12] we use PVeStA to compare the performance of our own new ROLA design with that of Walter. However, that paper focused on ROLA, and did not present the formalization of Walter or the SI and PSI properties.

In other applications of formal methods for distributed data stores, engineers at Amazon have used TLA+ and its model checker TLC to model and analyze the correctness of key parts of Amazon's celebrated cloud computing infrastructure [18]. In contrast to our work, they only use formal methods for correctness analysis. The designers of the TAPIR transaction protocol for distributed storage systems have also specified and model checked correctness (but not performance) properties of their design using TLA+ [24].

The papers [5,7] formalize a number of consistency models, including SI and PSI, but do not show how to analyze these properties.

7 Conclusions

We have formally analyzed and verified in Maude the design of Walter [22], a partially replicated distributed data store providing multi-partition transactions and guaranteeing parallel snapshot isolation (PSI), an important consistency property that offers attractive performance while providing adequate guarantees for certain kinds of applications. No formal specification of Walter existed before this work. Furthermore, PSI was only informally described by pseudo-code in [22] and no formal verification existed. This work has used model checking and systematic generation of initial states to verify that Walter satisfies PSI for all such states. We have also extended the Maude specification of Walter to model time and probabilistic communication delays as a probabilistic rewrite theory, and have then used statistical model checking analysis to study Walter's latency and throughput performance for a wide range of workloads. The results of the statistical model checking analysis are consistent with the experimental results in [22] but offer also new insights about Walter's performance for a wider range of workloads than those evaluated experimentally in [22].

We view this work as a stepping stone towards two substantially more ambitions goals: (1) Walter's design is an important data point in the consistency/performance spectrum of cloud-based data storage systems. In the near future we plan to use the experience gained in modeling and analyzing Walter and other designs in this spectrum to build a library of formally specified components supporting the modular design of new cloud-based data storage systems. (2) We also plan to use Walter's executable specification in Maude for code generation purposes to obtain high-quality implementations directly from formally analyzed designs. Our experience strongly suggests that high levels of system quality and reliability could be achieved by systematically deriving cloud-based system implementations from thoroughly analyzed formal designs.

References

1. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. Electr. Notes Theor. Comput. Sci. 153(2) (2006)
2. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: CALCO'11. LNCS, vol. 6859. Springer (2011)

3. Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: SIGMOD 1995. pp. 1–10. ACM (1995)
4. Bobba, R., Grov, J., Gupta, I., Liu, S., Meseguer, J., Ölveczky, P.C., Skeirik, S.: Design, formal modeling, and validation of cloud storage systems using Maude. Tech. rep., University of Illinois at Urbana-Champaign (2017), http://hdl.handle.net/2142/96274
5. Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: CONCUR. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
7. Crooks, N., Pu, Y., Alvisi, L., Clement, A.: Seeing is believing: A client-centric specification of database isolation. In: PODC 2017. pp. 73–82. ACM (2017)
8. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Statistical model checking for composite actor systems. In: WADT'12. LNCS, vol. 7841. Springer (2013)
9. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Specification, Algebra, and Software. LNCS, vol. 8373. Springer (2014)
10. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: SEFM. LNCS, vol. 8702. Springer (2014)
11. Liu, S., Ölveczky, P.C., Ganhotra, J., Gupta, I., Meseguer, J.: Exploring design alternatives for RAMP transactions through statistical model checking. In: Proc. ICFEM'17. LNCS, vol. 10610. Springer (2017)
12. Liu, S., Ölveczky, P.C., Santhanam, K., Wang, Q., Gupta, I., Meseguer, J.: ROLA: A new distributed transaction protocol and its formal analysis. In: FASE 2018. LNCS, Springer (2018), to appear. Extended version available at `https://sites.google.com/site/fase18submission/`
13. Liu, S., Ganhotra, J., Rahman, M., Nguyen, S., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. Leibniz Transactions on Embedded Systems 4(1), 03:1–03:26 (2017)
14. Liu, S., Nguyen, S., Ganhotra, J., Rahman, M.R., Gupta, I., Mesegue, J.: Quantitative analysis of consistency in NoSQL key-value stores. In: QEST 2015. pp. 228–243 (2015)
15. Liu, S., Ölveczky, P.C., Rahman, M.R., Ganhotra, J., Gupta, I., Meseguer, J.: Formal modeling and analysis of RAMP transaction systems. In: SAC'16. ACM (2016)
16. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: ICFEM'14. LNCS, vol. 8829. Springer (2014)
17. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
18. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Communications of the ACM 58(4), 66–73 (2015)
19. Ölveczky, P.C.: Formalizing and validating the P-Store replicated data store in Maude. In: WADT 2016. LNCS, vol. 10644. Springer (2017)
20. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: CAV'05. LNCS, vol. 3576. Springer (2005)
21. Sen, K., Viswanathan, M., Agha, G.A.: VESTA: A statistical model-checker and analyzer for probabilistic systems. In: QEST'05. IEEE Computer Society (2005)

22. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP 2011. ACM (2011)
23. Younes, H.L.S., Simmons, R.G.: Statistical probabilistic model checking with a focus on time-bounded properties. Inf. Comput. 204(9), 1368–1409 (2006)
24. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. In: Proc. Symposium on Operating Systems Principles, (SOSP'15). ACM (2015)