Automatic Analysis of Consistency Properties of Distributed Transaction Systems in Maude

Si Liu[1], Peter Csaba Ölveczky[2], Min Zhang[3], Qi Wang[1], and José Meseguer[1]

[1] University of Illinois, Urbana-Champaign, USA
[2] University of Oslo, Oslo, Norway
[3] Shanghai Key Laboratory of Trustworthy Computing, China

Abstract. Many transaction systems distribute, partition, and replicate their data for scalability, availability, and fault tolerance. However, observing and maintaining strong consistency of distributed and partially replicated data leads to high transaction latencies. Since different applications require different consistency guarantees, there is a plethora of consistency properties—from weak ones such as read atomicity through various forms of snapshot isolation to stronger serializability properties—and distributed transaction systems (DTSs) guaranteeing such properties. This paper presents a general framework for formally specifying a DTS in Maude, and formalizes in Maude nine common consistency properties for DTSs so defined. Furthermore, we provide a fully automated method for analyzing whether the DTS satisfies the desired property for all initial states up to given bounds on system parameters. This is based on automatically recording relevant history during a Maude run and defining the consistency properties on such histories. To the best of our knowledge, this is the first time that model checking of all these properties in a unified, systematic manner is investigated. We have implemented a tool that automates our method, and use it to model check state-of-the-art DTSs such as P-Store, RAMP, Walter, Jessy, and ROLA.

1 Introduction

Applications handling large amounts of data need to partition their data for scalability and elasticity, and need to replicate their data across widely distributed sites for high availability and fault and disaster tolerance. However, guaranteeing strong consistency properties for transactions over partially replicated distributed data requires lot of costly coordination that results in long transaction delays. Different applications require different consistency guarantees, and balancing well the trade-off between performance and consistency guarantees is key to designing distributed transaction systems (DTSs). There is therefore a plethora of consistency properties for DTSs over partially replicated data—from weak properties such as read atomicity through various forms of snapshot isolation to strong serializability guarantees—and DTSs providing such guarantees.

DTSs and their consistency guarantees are typically specified informally and validated only by testing; there is very little work on their automated formal

analysis (see Section 8). We have previously formally modeled and analyzed single state-of-the-art industrial and academic DTSs, such as Google's Megastore, Apache Cassandra, Walter, P-Store, Jessy, ROLA, and RAMP, in Maude [14].

In this paper we present a *generic* framework for formalizing both DTSs and their consistency properties in Maude. The modeling framework is very general and should allow us to naturally model most DTSs. We formalize nine popular consistency models in this framework and provide a fully automated method—and a tool which automates this method—for analyzing whether a DTS specified in our framework satisfies the desired consistency property for all initial states with the user-given number of transactions, data items, sites, and so on.

In particular, we show how one can automatically add a monitoring mechanism which records relevant history during a run of a DTS specified in our framework, and we define the consistency properties on such histories so that the DTS can be directly model checked in Maude. We have implemented a tool that uses Maude's meta-programming features to automatically add the monitoring mechanism, that automatically generates all the desired initial states, and that performs the Maude model checking. We have applied our tool to model check state-of-the-art DTSs such as variants of RAMP, P-Store, ROLA, Walter, and Jessy. To the best of our knowledge, this is the first time that model checking of all these properties in a unified, systematic manner is investigated.

This paper is organized as follows. Section 2 provides background on rewriting and Maude. Section 3 gives an overview of the consistency properties that we formalize. Section 4 presents our framework for modeling DTSs in Maude, and Section 5 explains how to record the history in such models. Section 6 formally defines consistency models as Maude functions on such recorded histories. Section 7 briefly introduces our tool which automates the entire process. Finally, Section 8 discusses related work and Section 9 gives some concluding remarks.

2 Rewriting Logic and Maude

Maude [14] is a rewriting-logic-based executable formal specification language and high-performance analysis tool for object-based distributed systems.

A Maude module specifies a *rewrite theory* $(\Sigma, E \cup A, R)$, where:

- Σ is an algebraic *signature*; i.e., a set of *sorts*, *subsorts*, and *function symbols*.
- $(\Sigma, E \cup A)$ is a *membership equational logic theory* [14], with E a set of possibly conditional equations and membership axioms, and A a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms A. The theory $(\Sigma, E \cup A)$ specifies the system's states as members of an algebraic data type.
- R is a collection of *labeled conditional rewrite rules* $[l] : t \longrightarrow t'$ **if** *cond*, specifying the system's local transitions.

Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`. The mathematical variables in such statements are declared with the keywords `var` and `vars`, or can have

the form *var*:*sort* and be introduced on the fly. An equation $f(t_1, \ldots, t_n) = t$ with the `owise` ("otherwise") attribute can be applied to a subterm $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied. Maude also provides standard parameterized data types (sets, maps, etc.) that can be instantiated (and renamed); for example, `pr SET{Nat} * (sort Set{Nat} to Nats)` defines a sort `Nats` of *sets* of natural numbers.

A *class* declaration `class` C `|` att_1 `:` s_1`, ...,` att_n `:` s_n declares a class C of objects with attributes att_1 to att_n of sorts s_1 to s_n. An *object instance* of class C is represented as a term `<` O `:` C `|` $att_1 : val_1, \ldots, att_n : val_n$ `>`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`. A system state is modeled as a term of the sort `Configuration`, and has the structure of a *multiset* made up of objects and messages.

The dynamic behavior of a system is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule (with label `l`)

```
rl [l] :  m(O,w)
          < O : C | a1 : x, a2 : O', a3 : z >
       =>
          < O : C | a1 : x + w, a2 : O', a3 : z >
          m'(O',x) .
```

defines a family of transitions in which a message `m(O, w)` is read and consumed by an object `O` of class `C`, whose attribute `a1` is changed to `x + w`, and a new message `m'(O',x)` is generated. Attributes whose values do not change and do not affect the next state, such as `a3` and `a2`, need not be mentioned in a rule.

Maude also supports *metaprogramming* in the sense that a Maude specification M can be represented as a *term* $\overline{M}$ (of sort `Module`), so that a module transformation can be defined as a Maude function f : `Module` $\rightarrow$ `Module`.

Reachability Analysis in Maude. Maude provides a number of analysis methods, including rewriting for simulation purposes, reachability analysis, and linear temporal logic (LTL) model checking. In this paper, we use reachability analysis. Given an initial state *init*, a state pattern *pattern* and an (optional) condition *cond*, Maude's `search` command searches the reachable state space from *init* in a breadth-first manner for states that match *pattern* such that *cond* holds:

`search [`*bound*`]` *init* `=>!` *pattern* `such that` *cond* `.`

where *bound* is an upper bound on the number of solutions to look for. The arrow `=>!` means that Maude only searches for *final* states (i.e., states that cannot be further rewritten) that match *pattern* and satisfies *cond*. If the arrow is instead `=>*` then Maude searches for all reachable states satisfying the search condition.

3 Transactional Consistency

Different applications require different consistency guarantees. There are therefore many consistency properties for DTSs on partially replicated distributed

data stores. This paper focuses on the following nine, which span a spectrum from weak consistency such as read committed to strong consistency like serializability:

- *Read committed* (*RC*) [6] disallows a transaction[4] from seeing any uncommitted or aborted data.
- *Cursor stability* (*CS*) [16], widely implemented by commercial SQL systems (e.g., IBM DB2 [1]) and academic prototypes (e.g., MDCC [22]), guarantees *RC* and in addition prevents the *lost update* anomaly.
- *Read atomicity* (*RA*) [5] guarantees that either *all* or *none* of a (distributed) transaction's updates are visible to other transactions. For example, if Alice and Bob become friends on social media, then Charlie should not see that Alice is a friend of Bob's, and that Bob is not a friend of Alice's.
- *Update atomicity* (*UA*) [12,27] guarantees read atomicity and prevents the lost update anomaly.
- *Snapshot isolation* (*SI*) [6] requires a multi-partition transaction to read from a snapshot of a distributed data store that reflects a single commit order of transactions across sites, even if they are independent of each other: Alice sees Charlie's post before seeing David's post if and only if Bob sees the two posts in the same order. Charlie and David must therefore coordinate the order of committing their posts even if they do not know each other.
- *Parallel snapshot isolation* (*PSI*) [38] weakens *SI* by allowing different commit orders at different sites, while guaranteeing that a transaction reads the most recent version committed at the transaction execution site, as of the time when the transaction begins. For example, Alice may see Charlie's post before seeing David's post, even though Bob sees David's post before Charlie's post, as long as the two posts are independent of each other. Charlie and David can therefore commit their posts without waiting for each other.
- *Non-monotonic snapshot isolation* (*NMSI*) [4] weakens *PSI* by allowing a transaction to read a version committed after the transaction begins: Alice may see Bob's post that committed after her transaction started executing.
- *Serializability* (*SER*) [35] ensures that the execution of concurrent transactions is equivalent to one where the transactions are run one at a time.
- Strict Serializability (*SSER*) strengthens *SER* by enforcing the serial order to follow real time.

4 Modeling Distributed Transaction Systems in Maude

This section presents a framework for modeling in Maude DTSs that satisfy the following general assumptions:

[4] A transaction is a user application request, typically consisting of a sequence of read and/or write operations on data items, that is submitted to a (distributed) database.

- We can identify and record "when"[5] a transaction starts executing at its server/proxy and "when" the transaction is committed and aborted at the different sites involved in its validation.
- The transactions record their read and write sets.

If a such a DTS is modeled in this framework, our tool can automatically model check whether it satisfies the above consistency properties, as long as it can detect the read and write sets and the above events: start of transaction execution, and abort/commit of a transaction at a certain site. This section explains how the system should be modeled so that our tool automatically discovers these events.

We make the following additional assumptions about the DTSs we target:

- The database is distributed across of a number of *sites*, or *servers* or *replicas*, that communicate by asynchronous *message passing*. Data are *partially replicated* across these sites: a data item may be replicated/stored at more than one site. The sites replicating a data item are called that item's *replicas*.
- Systems evolve by message passing or local computations. Servers communicate by asynchronous message passing with arbitrary but finite delays.
- A client forwards a transaction to be executed to some server (called the transaction's *executing server* or *proxy*), which executes the transaction.
- Transaction execution should terminate in commit or abort.

4.1 Modeling DTSs in Maude

A DTS is modeled in an object-oriented style, where the state consists of a number of *replica* objects, each modeling a local database/server/site, and a number of messages traveling between the replica objects. A transaction is modeled as an object which resides inside the replica object executing the transaction.

Basic Data Types. There are user-defined sorts `Key` for data items (or keys) and `Version` for versions of data items, with a partial order `<` on versions, with v `<` v' denoting that v' is a later version of v in `<`:

```
op _<_ : Version Version -> Bool.
```

We then define key-version pairs `<`*key*`,`*version*`>` and sets of such pairs, that model a transaction's read and write sets, as follows:

```
sorts Key Version KeyVersion .
op <_,_> : Key Version -> KeyVersion .
pr SET{KeyVersion} * (sort Set{KeyVersion} to KeyVersions) .
```

Example 1. A version in our Maude model [28] of the Walter transactional data store [38] consists of the executing server's identifier, and a sequence number local to that server:

[5] Since we do not necessarily deal with real-time systems, this "when" may not denote the real time, but when the event takes place *relative* to other events.

```
op <<_,_>> : Oid Nat -> Version .
```

where the sort `Oid` refers to the object identifier, and `Nat` models sequence numbers as natural numbers.

To track the status of a transaction (on non-proxies, or remote servers) we define a sort `TxnStatus` consisting of some transaction's identifier and its status; this is used to indicate whether a remote transaction (one executed on another server) is committed on this server:

```
op [_,_] : Oid Bool -> TxnStatus [ctor] .
pr SET{TxnStatus} * (sort Set{TxnStatus} to TxnStatusSet) .
```

Modeling Replicas. A *replica* (or *site*) stores parts of the database, executes the transactions for which it is the proxy, helps validating other transactions, and is formalized as an object instance of a subclass of the following class `Replica`:

```
class Replica | executing : Configuration,   committed : Configuration,
                aborted : Configuration,     decided : TxnStatusSet .
```

The attributes `executing`, `committed`, and `aborted` contain, respectively, transactions that are being executed, and have been committed or aborted on the executing server; `decided` is the status of transactions executed on other servers.

To model a system-specific replica a user should specify it as an object instance of a subclass of the class `Replica` with new attributes.

Example 2. A replica in our Maude model of Walter [28] is modeled as an object instance of the following subclass `Walter-Replica` of class `Replica` that adds 14 new attributes (only 4 shown below):

```
class Walter-Replica | store : Datastore,     sqn : Nat,
                       locked : Locks,        votes : Vote, ...
subclass Walter-Replica < Replica .
```

Modeling Transactions. A *transaction* should be modeled as an object of a subclass of the following class `Txn`:

```
class Txn | readSet : KeyVersions, writeSet : KeyVersions .
```

where `readSet` and `writeSet` denote the key/version pairs read and written by the transaction, respectively.

Example 3. Walter transactions can be modeled as object instances of the subclass `Walter-Txn` with four new attributes:

```
class Walter-Txn | operations : OperationList,  localVars : LocalVars,
                   startVTS : VectorTimestamp,  txnSQN : Nat .
subclass Walter-Txn < Txn .
```

Modeling System Dynamics. We describe how the rewrite rules defining the start of a transaction execution and aborts and commits at different sites should be defined so that our tool can detect these events.

– The start of a transaction execution must be modeled by a rewrite rule where the transaction object appears in the proxy server's `executing` attribute in the right-hand side, but not in the left-hand side, of the rewrite rule.

  Example 4. A Walter replica starts executing a transaction by moving the transaction `TID` in `gotTxns` (buffering transactions from clients) to `executing`:[6]

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

– When a transaction is *committed* on the executing server, the transaction object must appear in the `committed` attribute in the right-hand side—but not in the left-hand side—of the rewrite rule. Furthermore, the `readSet` and `writeSet` attributes must be explicitly given in the transaction object.

  Example 5. In Walter, when all operations of an executing read-only transaction have been performed, the proxy commits the transaction directly:

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

– When a transaction is aborted by the executing server, the transaction object must appear in the `aborted` attribute in the right-hand side, but not in the left-hand side, of a rewrite rule. Again, the transaction should present its attributes `writeSet` and `readSet` (to be able to record relevant history).

  Example 6. If either of the two conflict checks for fast commit fails, the executing Walter replica aborts the transaction:

```
crl [fast-commit-failed] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica |
        executing : < TID : Txn | operations : nil, writeSet : WS,
                                  readSet : RS, startVTS : VTS > TRANSES,
        aborted : TRANSES',  history : DS,  locked : LOCKS >
```

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

```
  =>
     < TABLE : Table | >
     < RID : Walter-Replica | executing : TRANSES,
                              aborted : TRANSES' < TID : Txn | > >
     if WS =/= empty /\ allLocalPreferred(WS,RID,REPLICA-TABLE)
        /\ (modified(WS,VTS,DS) or locked(WS,LOCKS)) .
```

– A rewrite rule that models when a transaction's status is decided remotely (i.e., not on the executing server) must contain in the right-hand side (only) the transaction's identifier and its status in the replica's `decided` attribute.

Example 7. Upon receiving the "disaster-safe durable" message, the remote Walter replica "commits" the transaction `TID` by setting its status to `true`:

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

These requirements are not very strict. The Maude models of the DTSs RAMP [31], Faster [26], Walter [28], ROLA [27], Jessy [30], and P-Store [34] can all be seen as instantiations of our modeling framework, with very small syntactic changes, such as defining transaction and replica objects as subclasses of `Txn` and `Replica`, changing the names of the attributes and sorts, etc. The Apache Cassandra NoSQL key-value store can be seen as a transaction system where each transaction is a single operation; the Maude model of Cassandra in [32] can also be easily modified to fit within our modeling framework.

5 Adding Execution Logs

To formalize and analyze consistency properties of distributed transaction systems we add an "execution log" that records the *history* of relevant events during a system execution. This section explains how this history recording can be added *automatically* to a model of a DTS that is specified as explained in Section 4.

5.1 Execution Log

To capture the total order of relevant events in a run, we use a "logical global clock" to order all key events (i.e., transaction starts, commits, and aborts). This clock is incremented by one each time such an event takes place.

A transaction in a replicated DTS is typically committed both locally (at its executing server) and remotely at different times. To capture this, we define a "time vector" using Maude's map data type that maps replica identifiers (of sort `Oid`) to (typically "logical") clock values (of sort `Time`, which here are the natural numbers: `subsort Nat < Time`):

```
pr MAP{Oid,Time} * (sort Map{Oid,Time} to VectorTime) .
```

where each element in the mapping has the form *`replica-id |-> time`*.

An execution log (of sort `Log`) maps each transaction (identifier) to a record <*proxy*, *issueTime*, *finishTime*, *committed*, *reads*, *writes*>, with *proxy* its proxy server, *issueTime* the starting time at its proxy server, *finishTime* the commit/abort times at each relevant server, *committed* a flag indicating whether the transaction is committed at its proxy, *reads* the key-version pairs read by the transaction, and *writes* the key-version pairs written:

```
sort Record .
op <_,_,_,_,_,_> : Oid Time VectorTime Bool KeyVersions KeyVersions -> Record .
pr MAP{Oid,Record} * (sort Map{Oid,Record} to Log) .
```

5.2 Logging Execution History

We show how the relevant history of an execution can be recorded during a run of our Maude model by transforming the original Maude model into one which also records this history.

First, we add to the state a `Monitor` object that stores the current logical global time in the `clock` attribute and the current log in the `log` attribute:

```
< M : Monitor | clock : Time, log : Log >.
```

The log is updated each time an interesting event (see Section 4.1) happens. Our tool identifies those events and *automatically* transforms the corresponding rewrite rules by adding and updating the monitor object.

EXECUTING. A transaction starts executing when the transaction object appears in a `Replica`'s `executing` attribute in the right-hand side, but not in the left-hand side, of a rewrite rule. The monitor then adds a record for this transaction, with the proxy and start time, to the log, and increments the logical global clock.

Example 8. The rewrite rule in Example 4 where a Walter replica is served a transaction is modified by adding and updating the monitor object (in blue):

```
rl [start-txn] :
   < O@M : Monitor | clock : GT@M, log : LOG@M >
   < RID : Walter-Replica | executing : TRANSES,   committedVTS : VTS,
               gotTxns :  < TID : Txn | startVTS : empty > ;; TXNS >
 =>
   < O@M : Monitor | clock : GT@M + 1 , log : LOG@M,
                     (TID |-> < RID, GT@M, empty, false, empty, empty >) >
   < RID : Walter-Replica | gotTxns : TXNS,
               executing : TRANSES < TID : Txn | startVTS : VTS > > .
```

where the monitor O@M adds a new record for the transaction TID in the log, with starting time (i.e., the current logical global time) GT@M at its executing server RID, finish time (empty), flag (false), read set (empty), and write set (empty). The monitor also increments the global clock by one.

Commit. A transaction commits at its proxy when the transaction object appears in the proxy's committed attribute in the right-hand side, but not in the left-hand side, of a rewrite rule. The record for that transaction is updated with commit status, versions read and written, and commit time, and the global logical clock is incremented.

Example 9. The monitor object is added to the rewrite rule in Example 5 for committing a read-only transaction:

```
rl [commit-read-only-txn] :
   < O@M : Monitor | clock : GT@M, log : LOG@M ,
             (TID |-> < RID, T@M, VTS@M, FLAG@M, READS@M, WRITES@M)) >
   < RID : Walter-Replica | committed : TRANSES',
                            executing : TRANSES
        < TID : Txn | operations : nil, writeSet : empty, readSet : RS > >
 =>
   < O@M : Monitor | clock : GT@M + 1 , log : LOG@M ,
             (TID |-> < RID, T@M, insert(RID,GT@M,VTS@M), true, RS, empty >)
   < RID : Walter-Replica | committed : (TRANSES' < TID : Txn | >),
                            executing : TRANSES > .
```

The monitor updates the log for the transaction TID by setting its finish time at the executing server RID to GT@M (insert(RID,GT@M,VTS@M)), setting the committed flag to true, setting the read set to RS and write set to empty (this is a read-only transaction), and increments the global clock.

Abort. Abort is treated as commit, but the commit flag remains false.

Example 10. The monitor object is added to the rewrite rule in Example 6 for aborting a transaction:

```
crl [fast-commit-failed] :
    < O@M : Monitor | clock : GT@M, log : LOG@M ,
              (TID |-> < RID, T@M, VTS@M, FLAG@M, READS@M, WRITES@M)) >
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica |
        executing : < TID : Txn | operations : nil, writeSet : WS,
            readSet : RS, startVTS : VTS > TRANSES,
        aborted : TRANSES',  history : DS,  locked : LOCKS >
  =>
    < O@M : Monitor | clock : GT@M + 1 , log : LOG@M ,
              (TID |-> < RID, T@M, insert(RID,GT@M,VTS@M), false, RS, WS >)
    < TABLE : Table | >
    < RID : Walter-Replica | executing : TRANSES,
```

```
                              aborted : TRANSES' < TID : Txn | > >
  if WS =/= empty /\ allLocalPreferred(WS,RID,REPLICA-TABLE)
     /\ (modified(WS,VTS,DS) or locked(WS,LOCKS)) .
```

DECIDED. When a transaction's status is decided remotely, the record for that transaction's decision time at the remote replica is updated with the current global time.

Example 11. The rewrite rule from Example 7 for committing a transaction remotely is transformed into the following rewrite rule:

```
crl [receive-ds-durable-visible] :
   < O@M : Monitor | clock : GT@M, log : LOG@M ,
       TID |-> < VTS1@M,VTS2@M,FLAG@M,READS@M,WRITES@M > >
    msg ds-durable(TID) from RID' to RID
    < RID : Walter-Replica |
            recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
            committedVTS : VTS', locked : LOCKS, decided : TSS >
  =>
    < O@M : Monitor | clock : GT@M + 1 , log : LOG@M ,
       TID |-> < VTS1@M, insert(RID,GT@M,VTS2@M) , FLAG@M,READS@M,WRITES@M > >
    < RID : Walter-Replica |
            committedVTS : insert(RID',SQN,VTS'),
            locked : release(TID,LOCKS),  decided : (TSS, [TID,true]) >
    msg visible(TID) from RID to RID'
    if VTS' gt VTS /\ s(VTS'[RID']) == SQN .
```

where the monitor `O@M` only needs to add the commit time for the replica `RID`, besides advancing the global time.

5.3 Implementing Monitoring Mechanism

We have formalized/implemented the transformation from a Maude specification of a DTS into one with a monitor as a meta-level function `monitorRules` in Maude. Specifically, the transformation takes as input Maude modules including the rewrite rules specifying the system dynamics, instruments the "interesting" rewrites rules at the meta-level according to the monitoring mechanism, and outputs a new *flattened* model of the system and the monitor.

The function `monitorRules` takes as argument a rewrite rule and returns a new one that is possibly equipped with the monitor and its behavior. The following two equations are defined to formalize the monitoring mechanism:

```
ceq monitorRules(rl T => T' [ATR] .) = (rl T1 => T2 [ATR] .)
    if '__[T1,T2] := monitorTerms(T,T',false) .

ceq monitorRules(crl T => T' if COND [ATR] .) =
    (crl T1 => T2 if COND [ATR] .)
 if '__[T1,T2] := monitorTerms(T,T',false) .
```

The first equation handles an unconditional rewrite rule, while the second one handles a conditional one.

The meta-level function `monitorTerms` takes terms `T` and `T'` from both sides of a rule, and returns, if monitor needed, two new ones `T1` and `T2`, forming the new rule with the monitor's behavior. The monitor's behavior depends on the input rewrite rule in terms of either the execution, local commit, remote commit, or abort of the transaction. In the cases where the monitor should be added, the term `T1` is a concatenation of the term `T` and the monitor *pattern* (e.g., `< O@M : Monitor | clock: GT@M, log: (TID |-> < VTS1@M, VTS2@M, FLAG@M, READS@M, WRITES@M), LOG@M)>`), while the term `T2` is a concatenation of the term `T'`, and the resulting monitor pattern determined by the status change of the transaction encoded by the input rewrite rule.

Example 12. The definition of `monitorTerms` in the case of COMMIT is defined by the following equation:

```
--- case 2: locally committed
ceq monitorTerms(T,T') =
    '__['__[newMonitor(getTID(T1)),T],
        '__[resultMonitor(getTID(T), getReplicaID(T),'true.Bool,
    getAttr('readSet‘:_,T1),getAttr('writeSet‘:_,T1)),T']]
if  isCommitted(T,T') /\  T1 := getAttr('executing‘:_,T) .
```

where the function `newMonitor` is used to construct a new monitor pattern with a given transaction's identifier, and `resultMonitor` constructs the resulting pattern with the transaction's identifier, replica's identifier, a Boolean value indicating whether the transaction is successfully committed or not, the set of versions read, and the set of versions written by the transaction. The function `isCommitted` returns true if the transition from `T` to `T'` is caused by the local commit of the transition `T1`, which is determined by whether the transaction appears in the attribute `committed` in `T'`, but not in that in `T`.

6 Formalizing Consistency Models in Maude

This section formalizes the consistency properties in Section 3 as functions on the "history log" of a *completed* run. We can then use these functions to automatically analyze in Maude whether a system with a given initial state satisfies a desired consistency property by searching for a *final* state (all transactions have finished) where the desired property does *not* hold on the history log. The entire Maude specification of these functions is available at `https://github.com/siliunobi/cat`.

Read Committed (RC). (A transaction cannot read any writes by uncommitted transactions.) Note that standard definitions for single-version databases disallow reading versions that are not committed at the time of the read. We follow the definition for multi-versioned systems by Adya, summarized by Bailis et al. [5], that defines the *RC* property as follows: (i) a committed transaction

cannot read a version that was written by an aborted transaction; and (ii) a transaction cannot read *intermediate values*: that is, if T writes two versions `<X,V>` and `<X,V'>` with `V < V'`, then no $T' \neq T$ can read `<X,V>`.

The first equation defining the function `rc`, specifying when RC holds, checks whether some (committed) transaction `TID1` read version `V` of key `X` (i.e., `<X,V>` is in `TID`'s read set `<X,V>,RS`, where `RS` matches the rest of `TID`'s read set), and this version `V` was written by some transaction `TID2` that was never committed (i.e., `TID2`'s commit flag is `false`, and its write set is `<X,V>,WS'`). The second equation checks whether there was an *intermediate* read of a version `<X,V>` that was overwritten by the same transaction `TID2` that wrote the version:[7]

```
op rc : Log -> Bool .
eq rc(TID1 |-> <O,T,VT,true,(<X,V>,RS),WS>,
      TID2 |-> <O',T',VT',false,RS',(<X,V>,WS')>, LOG) = false .
eq rc(TID1 |-> <O,T,VT,true,(<X,V>,RS),WS>,
      TID2 |-> <O',T',VT',true,RS',(<X,V>,<X,V'>,WS')>,
      LOG)  = false if V < V' .
eq rc(LOG) = true [owise] .
```

Read Atomicity (RA). A system guarantees RA if it prevents fractured reads and prevents transactions from reading uncommitted or aborted data. A transaction T_j exhibits *fractured reads* if transaction T_i writes versions x_m and y_n, T_j reads version x_m and version y_k, and $k < n$ [5]. The function `fracRead` checks whether there are fractured reads in the log. There is a fractured read if a transaction `TID2` reads `X` and `Y`, transaction `TID1` writes `X` and `Y`, `TID2` reads the version `VX` of `X` written by `TID1`, and reads a version `VY'` of `Y` written *before* `VY` (`VY' < VY`):

```
 op fracRead : Log -> Bool .
ceq fracRead(TID1 |-> <O,T,VT,true,(<X,VX>,<Y,VY'>,RS),WS>,
             TID2 |-> <O',T',VT',true,RS',(<X,VX>,<Y,VY>,WS')>, LOG)
    = true if VY' < VY .
eq fracRead(LOG) = false [owise] .
```

We define RA as the combination of RC and no fractured reads:

```
op ra : Log -> Bool .
eq ra(LOG) = rc(LOG) and not fracRead(LOG) .
```

Cursor Stability (CS) [16] strengthens RC by also preventing lost updates (LU). LU can only happen with multiple *conditional writes* (i.e., a transaction first fetches some value of an data item, and then updates it with a new value) fetching the same data. Once one of those transactions commits its writes, the others must be aborted. The function `lu` captures the case when there are two committed transactions `TID1` and `TID2`, both of which read the same version

[7] The configuration union and the union operator '`,`' for maps and sets are declared *associative* and *commutative*. The first equation therefore matches *any* log where some committed transaction read a key-version pair written by some aborted transaction.

V of data item X (i.e., <X,V> is in the read sets of both TID1 and TID2), and commit on the same key (i.e., the two transactions wrote <X,VX> and <X,VX'>, respectively):

```
op lu: Log -> Bool .
eq lu(TID1 |-> <O,T,VT,true,(<X,V>,RS),(<X,VX>,WS)>,
      TID2 |-> <O',T',VT',true,(<X,V>,RS'),(<X,VX'>,WS')>,LOG) = true .
eq lu(LOG) = false [owise] .
```

CS can then be specified as conjunction of *RC* and no lost updates:

```
op cs : Log -> Bool .
eq cs(LOG) = rc(LOG) and not lu(LOG) .
```

Update Atomicity [12,27] provides read atomicity and prevents lost updates:

```
op ua : Log -> Bool .
eq ua(LOG) = ra(LOG) and not lu(LOG) .
```

Snapshot Isolation (SI) is defined by two properties in [38]:

- 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[8] transactions must be disjoint.

The function notSnapshotRead holds when SI-1 is violated. The first conditional equation handles the case when a transaction TID1 reads another transaction TID2's version written (<X,V>), while the most recent committed version from TID1's perspective is <X,V'> written by TID3 (TID3's commit time T' at its proxy RID3 is between TID2's commit time T at its proxy RID2 and TID1's start time T1). The second conditional equation checks if TID1 read some version that was committed after it started (T1 < T):

```
 op notSnapshotRead : Log -> Bool .
ceq notSnapshotRead(
       TID1 |-> <RID1,T1,VT1,true,(<X,V>,RS1),WS1>,
       TID2 |-> <RID2,T2,(RID2 |-> T,VT2),true,RS2,(<X,V>,WS2)>,
       TID3 |-> <RID3,T3,(RID3 |-> T',VT3),true,RS3,(<X,V'>,WS3)>,
       LOG) = true if V =/= V' /\ T' < T1 /\ T' > T .
ceq notSnapshotRead(
       TID1 |-> <RID1,T1,VT1,true,(<X,V>,RS1),WS1>,
       TID2 |-> <RID2,T2,(RID2 |-> T,VT2),true,RS2,(<X,V>,WS2)>,
       LOG) = true if V =/= V' /\ T1 < T .
eq notSnapshotRead(LOG) = false [owise] .
```

[8] Two committed transactions are *concurrent* if one of them has a commit timestamp (at its proxy) between the start and the commit timestamp of the other.

The function `wwConflict` captures write-write conflicts: there are two transactions `TID1` and `TID2`, both writing key `X`, and `TID2` is committed at its proxy at time `T`, which comes after the start time `T1` of `TID1` but before its commit time `T2` at its proxy. The committed (flag `true`) transactions `TID1` and `TID2` are therefore concurrent and write the same key, and hence we have a write-write conflict:

```
 op wwConflict: Log -> Bool .
ceq wwConflict(
       TID1 |-> < RID, T1, (RID |-> T2 , VT2), true, RS, (< X,V > , WS) >,
       TID2 |-> < RID', T3, (RID' |-> T , VT4), true, RS', (< X,V' > , WS') >,
       LOG) = true if T > T1 /\ T < T2 .
 eq wwConflict(LOG) = false [owise] .
```

SI holds when there is no violation of snapshot read and no write-write conflicts:

```
op si : Log -> Bool .
eq si(LOG) = not notSnapshotRead(LOG) and not wwConflict(LOG) .
```

Parallel snapshot isolation (PSI) is given by three properties [38]:

- 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*[9] transactions must be disjoint.
- PSI-3 (commit causality across sites): If a transaction T_1 commits at a site S before a transaction T_2 starts at site S, then T_1 cannot commit after T_2 at any site.

The function `notSiteSnapshotRead` checks whether the system log satisfies PSI-1 by returning `true` if there is a transaction that did not read the most recent committed version at its executing site when it began:

```
 op notSiteSnapshotRead : Log -> Bool .
ceq notSiteSnapshotRead(
       TID1 |-> < RID1, T, VT1, true, (< X,V > , RS1), WS1 >,
       TID2 |-> < RID2, T', (RID1 |-> T2 , VT2), true, RS2, (< X,V > , WS2) >,
       TID3 |-> < RID3, T'', (RID1 |-> T3 , VT3), true, RS3, (< X,V' > , WS3) >,
       LOG) = true if V =/= V' /\ T3 < T /\ T3 > T2 .
ceq notSiteSnapshotRead(
       TID1 |-> < RID1, T, VT1, true, (< X,V > , RS1), WS1 >,
       TID2 |-> < RID2, T', (RID1 |-> T2 , VT2), true, RS2, (< X,V > , WS2) >,
       LOG) = true if T < T2 .
 eq notSiteSnapshotRead(LOG) = false [owise] .
```

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

In the first equation, the transaction TID1, hosted at site RID1, has in its read set a version < X,V > written by TID2. Some transaction TID3 wrote version < X,V' > and was committed at RID1 after TID2 was committed at RID1 (T3 > T2) and before TID1 started executing (T3 < T). Hence, the version read by TID1 was stale. The second equation checks if TID1 read some version that was committed at RID1 after TID1 started (T < T2).

The function someWhereConflict checks whether PSI-2 holds by looking for a write-write conflict between any pair of committed *somewhere-concurrent transactions* in the system log:

```
 op someWhereConflict : Log -> Bool .
ceq someWhereConflict(
       TID1 |-> < RID1, T, (RID1 |-> T1 , VT1), true, RS, (< X,V > , WS) >,
       TID2 |-> < RID2, T', (RID1 |-> T2 , VT2), true, RS', (< X,V' > , WS') >,
       LOG) = true if T2 > T /\ T2 < T1 .
 eq someWhereConflict(LOG) = false [owise] .
```

In contrast with the function wwConflict, the above function checks whether the transactions with the write conflict are concurrent at the transaction TID1's proxy RID1. Here, TID2 commits at RID1 at time T2, which is between TID1's start time T and its commit time T1 at RID1.

The function notCausality analyzes PSI-3 by checking whether there was a "bad situation" in which a transaction TID1 committed at site RID2 *before* a transaction TID2 started at site RID2 (T1 < T2), while TID1 committed at site RID *after* TID2 committed at site RID (T3 > T4):

```
 op notCausality : Log -> Bool .
ceq notCausality(
        TID1 |-> < RID1, T, (RID2 |-> T1 , RID |-> T3 , VT2), true, RS, WS >,
        TID2 |-> < RID2, T2, (RID |-> T4 , VT4), true, RS', WS' >,
        LOG) = true if T1 < T2 /\ T3 > T4 .
 eq notCausality(LOG) = false [owise] .
```

PSI can then be defined by combining the above three properties:

```
op psi : Log -> Bool .
eq psi(LOG) = not notSiteSnapshotRead(LOG) and
              not someWhereConflict(LOG) and not notCausality(LOG) .
```

Non-monotonic snapshot isolation (NMSI) is the same as *PSI* except that a transaction may read a version committed even after the transaction begins [3]. *NMSI* can therefore be defined as the conjunction of PSI-2 and PSI-3:

```
op nmsi : Log -> Bool .
eq nmsi(LOG) = not someWhereConflict(LOG) and not notCausality(LOG) .
```

Serializability (SER) means that the concurrent execution of transactions is equivalent to executing them in some (non-overlapping in time) sequence [35].

A formal definition of *SER* is based on *direct serialization graphs* (DSGs): an execution is serializable if and only if the corresponding DSG is acyclic. Each node in a DSG corresponds to a committed transaction, and directed edges in a DSG correspond to the following types of direct dependencies [2]:

- Read dependency: Transaction T_j *directly read-depends* on transaction T_i if T_i writes some version x_i and T_j reads that version x_i.
- Write dependency: Transaction T_j *directly write-depends* on transaction T_i if T_i writes some version x_i and T_j writes x's next version after x_i in the version order.
- Antidependency: Transaction T_j *directly antidepends* on transaction T_i if T_i reads some version x_k and T_j writes x's next version after x_k.

There is a directed edge from a node T_i to another node T_j if transaction T_j directly read-/write-/antidepends on transaction T_i.

The dependencies/edges can easily be extracted from the our log as follows:

- If there is a key-version pair `< X , V >` both in `T2`'s read set and in `T1`'s write set, then `T2` read-depends on `T1`.
- If `T1` writes `< X, V1 >` and `T2` writes `< X, V2 >`, and `V1 < V2`, and there *no* version `< X, V >` with `V1 < V < V2`, then `T2` write-depends on `T1`.
- `T2` antidepends on `T1` if `< X, V1 >` is in `T1`'s read set, `< X, V2 >` is in `T2`'s write set with `V1 < V2` and there is no version `< X, V >` such that `V1 < V < V2`.

We have defined in Maude a data type `Dsg` for DSGs:

```
sorts Dsg Edge .    subsort Edge < Dsg .

op <_;_> : Oid Oid -> Edge [ctor] .
eq E ; E = E .

op emptyDsg : -> Dsg [ctor] .
op _;_ : Dsg Dsg -> Dsg [ctor assoc comm id: emptyDsg] .
```

We have defined a function `dsg` that constructs the DSG from a log by iteratively adding edges between *committed* transactions (aborted transactions are dropped from the log):

```
op dsg : Log -> Dsg .
op dsg : Log Log Dsg -> Dsg .

eq dsg(LOG) = dsg(LOG,LOG,emptyDsg) .
eq dsg((TID |-> < O,T,VT,false,RS,WS >,LOG'),LOG,DSG) = dsg(LOG',LOG,DSG) .
```

For each transaction in the log, the construction builds up dependency edges with the read set and write set in order; within the read/write set, the construction checks dependencies for each version read/written.

First, we find read dependencies for versions read in the read set:

```
ceq dsg((TID |-> < O,T,VT,true,(KVER,RS),WS >,LOG'),
       (TID' |-> < O',T',VT',true,RS',(KVER,WS') >,LOG),DSG)
  = dsg((TID |-> < O,T,VT,true,RS,WS >,LOG'),
       (TID' |-> < O',T',VT',true,RS',(KVER,WS') >,LOG),
       (DSG ; < TID' ; TID >)) if TID =/= TID' .
```

where the transaction `TID` reads `KVER` written by another transaction `TID'`, and thus a new edge `< TID' ; TID >` is added to the DSG.

If `VS'` written by the transaction `TID'` is the *next* version of `VS` read by the transaction `TID`, we add an antidependency edge `< TID ; TID' >` to the DSG:

```
ceq dsg((TID |-> < O,T,VT,true,(< X,VS >,RS),WS >,LOG'),
        (TID' |-> < O',T',VT',true,RS',(< X,VS' >,WS') >,LOG),DSG)
  = dsg((TID |-> < O,T,VT,true,RS,WS >,LOG'),
        (TID' |-> < O',T',VT',true,RS',(< X,VS' >,WS') >,LOG),
        (DSG ; < TID ; TID' >))
    if VS < VS' /\ TID =/= TID' /\ not committedBetween(X,VS,VS',LOG) .
```

where the function `committedBetween` returns true if there is a version committed between the two versions in the version order. It is defined as:

```
 op committedBetween : Key Version Version Log -> Bool .
ceq committedBetween(X,VS,VS',
                     (TID |-> < O,T,VT,true,RS, (< X,VS'' >,WS) >,LOG))
   = true if VS'' < VS' /\ VS < VS'' .
 eq committedBetween(X,VS,VS',LOG) = false [owise] .
```

If there is no more edge to add for the current version read, we move to the next version read:

```
eq dsg((TID |-> < O,T,VT,true,(< X,VS >,RS),WS >,LOG'),LOG,DSG)
 = dsg((TID |-> < O,T,VT,true,RS,WS >,LOG'),LOG,DSG) [owise] .
```

Once all versions read in the read set are handled, we continue to build up the DSG by investigating the write set.

If the transaction `TID'` writes the *next* version `VS'` of `VS` written by another transaction `TID`, an write-dependency edge `<TID ; TID'>` is added to the DSG:

```
ceq dsg((TID |-> < O,T,VT,true,empty,(< X,VS >,WS) >,LOG'),
       (TID' |-> < O',T',VT',true,RS',(< X,VS' >,WS') >,LOG),DSG)
  = dsg((TID |-> < O,T,VT,true,empty,WS >,LOG'),
       (TID' |-> < O',T',VT',true,RS',(< X,VS' >,WS') >,LOG),
          (DSG ; < TID ; TID' >))
    if VS < VS' /\ TID =/= TID' /\ not committedBetween(X,VS,VS',LOG) .
```

If `VS` written by the transaction `TID` is the *next* version of `VS'` read by the transaction `TID'`, we add an antidependency edge `< TID' ; TID >` to the DSG:

```
ceq dsg((TID |-> < O,T,VT,true,empty,(< X,VS >,WS) >,LOG'),
        (TID' |-> < O',T',VT',true,(< X,VS' >,RS'),WS' >,LOG),DSG)
  = dsg((TID |-> < O,T,VT,true,empty,WS >,LOG'),
        (TID' |-> < O',T',VT',true,(< X,VS' >,RS'),WS' >,LOG),
           (DSG ; < TID' ; TID >))
    if VS' < VS /\ TID =/= TID' /\ not committedBetween(X,VS',VS,LOG) .
```

If there is no more edge to add for the current version written, we move to the next version written:

```
eq dsg((TID |-> < O,T,VT,true,empty,(< X,VS >,WS) >,LOG'),LOG,DSG)
 = dsg((TID |-> < O,T,VT,true,empty,WS >,LOG'),LOG,DSG) [owise] .
```

When all dependency edges are handled for a transaction (indicated by `empty` for both the read set and write set), we move to the next (committed) transaction:

```
eq dsg((TID |-> < O,T,VT,true,empty,empty >,LOG'),LOG,DSG)
 = dsg(LOG',LOG,DSG) .
```

Finally, we get the resulting DSG out of the execution history:

```
eq dsg(empty,LOG,DSG) = DSG .
```

Based on the constructed DSG, we define a function `cycle : Dsg -> Bool` that checks whether the DSG has cycles:

```
op cycle : Dsg -> Bool .
eq cycle(DSG) = cycle(txnIds(DSG),DSG,empty) .

 op cycle : OidSet Dsg OidSet -> Bool .
ceq cycle((TID,TIDS),DSG,TIDS') = true if TID in TIDS' .
ceq cycle((TID,TIDS),DSG,TIDS') =
      cycle(destNodes(TID,DSG),DSG,(TIDS',TID))
        or cycle(TIDS,DSG,TIDS') if not (TID in TIDS') .
 eq cycle(empty,DSG,TIDS') = false .
```

where the function `txnIds` returns a set of identifiers of all transactions in the DSG; the function `destNodes` computes for some node in the DSG all nodes it directly points to by the dependency edges:

```
op txnIds : Dsg -> OidSet .
eq txnIds(DSG ; < TID ; TID' >) = TID ; TID' ; txnIds(DSG) .
eq txnIds(emptyDsg) = empty .

op destNodes : Oid Dsg -> OidSet .
eq destNodes(TID,(< TID ; TID' > ; DSG)) = TID' ; destNodes(TID,DSG) .
eq destNodes(TID,DSG) = empty [owise] .
```

SER then holds if there is no cycle in the constructed DSG:

```
op ser : Log -> Bool .
eq ser(LOG) = not cycle(dsg(LOG)) .
```

Strict Serializability (SSER) guarantees that all transactions can be serialized in an order that also respects the real time order. For example, under *SSER*, once an update transaction commits its writes, all later transactions (where "later" is defined by wall-clock time modeled by our logical global clock) should return the version of that transaction or the version of a later update transaction.

Thus we define the following three functions to check if the read, write, or anti- dependency does not respect the real-time order, respectively.

If a transaction reads some stale data, the read dependency violates the real-time order:

```
 op notRtReadDep : Log -> Bool .
ceq notRtReadDep((TID1 |-> < RID,T,VT,true,(< X,VS >,RS),WS >,
      TID2 |-> < RID',T1,(RID' |-> T2,VT'),true,RS',
        (< X,VS >,WS') >,LOG)) = true
    if T2 < T /\ rtCommittedBetween(X,T2,T,LOG) .
```

where the transaction `TID1` reads the stale data `< X,VS >` written by `TID2`. The data is stale because there is some version committed between `TID2`'s commit and `TID1`'s start in real time. This is determined by the function `rtCommittedBetween`:

```
 op rtCommittedBetween : Key Time Time Log -> Bool .
ceq rtCommittedBetween(X,T1,T2,
     (TID |-> < RID,T,(RID |-> T',VT),true,RS,(< X,V >,WS) >,LOG))
   = true if T1 < T' /\ T' < T2 .
 eq rtCommittedBetween(X,T1,T2) = false [owise] .
```

The write dependency violates the real-time order if one version is the next version of the other in the version order, but there is some version committed in-between in real time:

```
 op notRtWriteDep : Log -> Bool .
ceq notRtWriteDep((
      TID1 |-> < RID,T1,(RID |-> T1',VT),true,RS,(< X,VS >,WS) >,
      TID2 |-> < RID',T2,(RID' |-> T2',VT'),true,RS',
      (< X,VS' >,WS') >,LOG)) = true
    if VS < VS' /\ not committedBetween(X,VS,VS',LOG) /\
       T1' < T2' /\ rtCommittedBetween(X,T1',T2',LOG)  .
```

where the version `VS'` is the next version of `VS`, but there is some version committed between the respective commit times `T1'` and `T2'`.

The antidependency violates the real-time order if the version written is the next version of the version read in the version order with some version committed between the two versions in real time:

```
 op notRtAntiDep : Log -> Bool .
ceq notRtAntiDep((
      TID1 |-> < RID1,T1,(RID1 |-> T1',VT1),true,(< X,VS >,RS1),WS1 >,
      TID2 |-> < RID2,T2,(RID2 |-> T2',VT2),true,RS2,(< X,VS' >,WS2) >,
      LOG)) = true
    if VS < VS' /\ not committedBetween(X,VS,VS',LOG) /\
       T1' < T2' /\ rtCommittedBetween(X,T1',T2',LOG) .
```

Finally, by combining the three checks we have the definition for *SSER*:

```
op sser : Log -> Bool .
eq sser(LOG) = not notRtReadDep(LOG) and
               not notRtWriteDep(LOG) and not notRtAntiDep(LOG) .
```

7 Formal Analysis of Consistency Properties of DTSs

We have implemented the *Consistency Analysis Tool* (CAT) that automates the method in this paper. CAT takes as input:

- A Maude model of the DTS specified as explained in Section 4.
- The *number* of each of the following parameters: read-only, write-only, and read-write transactions; operations for each type of transaction; clients; servers; keys; and replicas per key. The tool analyzes the desired property for *all* initial states with the number of each of these parameters.
- The consistency property to be analyzed.

Given these inputs, CAT performs the following steps:

1. adds the monitoring mechanism to the user-provided system model;
2. generates all possible initial states with the user-provided number of the different parameters; and
3. executes the following command to search, from all generated initial states, for *one* reachable *final* state where the consistency property does *not* hold:

   ```
   search [1] init =>! C:Configuration
       < M:Oid : Monitor | log: LOG:Log  clock: N:Nat >
           such that not consistency-property(LOG:Log) .
   ```

   where the underlined functions are parametric, and are instantiated by the user inputs; e.g., *consistency-property* is replaced by the corresponding function `rc`, `psi`, `nmsi`, ..., or `ser`, depending on which property to analyze.

CAT outputs either "`No solution`," meaning that all runs from all the given initial states satisfy the desired consistency property, or a counterexample (in Maude at the moment) showing a behavior that violates the property.

We have applied our tool to 15 Maude models of state-of-the-art DTSs (different variants of RAMP and Walter, ROLA, Jessy, P-Store, and Cassandra) against all nine properties. Table 1 shows our experience with CAT: all model checking results are as expected. It is worth remarking that our automatic analysis found all the violations of properties that the respective systems should violate. There are also some cases where model checking is not applicable ("-" in Table 1): (i) some system models do not include a mechanism for committing a transaction on remote servers (i.e., no commit time on any remote server is recorded by the monitor). Thus, model checking *NMSI* or *PSI* is not applicable; and (ii) Cassandra is a NoSQL key-value store that concerns only

non-transactional consistency models. Thus, model checking transactional consistency properties is not applicable.

We have performed our analysis with different initial states, with up to 4 transactions, 4 operations per transaction, 2 clients, 2 servers, 2 keys, and 2 replicas per key. Each analysis command took about 10 minutes (worst case) to execute on a 2.9 GHz Intel 4-Core i7-3520M CPU with 3.6 GB memory.

Table 1. Model Checking Results w.r.t. Consistency Properties. "✓", "×", and "-" refer to satisfying and violating the property, and "not applicable", respectively.

Maude Model	LOC	Consistency Property								
		RC	RA	CS	UA	NMSI	PSI	SI	SER	SSER
RAMP-F [31]	330	✓	✓	×	×	-	-	×	×	×
RAMP-F+1PW [26]	302	✓	✓	×	×	-	-	×	×	×
RAMP-F+FC [26]	305	✓	✓	×	×	-	-	×	×	×
RAMP-F¬2PC [26]	320	✓	×	×	×	-	-	×	×	×
RAMP-S [31]	255	✓	✓	×	×	-	-	×	×	×
RAMP-S+1PW [26]	237	✓	✓	×	×	-	-	×	×	×
RAMP-S¬2PC [26]	248	✓	×	×	×	-	-	×	×	×
Faster [26]	300	✓	×	×	×	-	-	×	×	×
Faster+FC [26]	220	✓	×	×	×	-	-	×	×	×
ROLA [27]	411	✓	✓	✓	✓	-	-	×	×	×
Jessy [30]	413	✓	✓	✓	✓	✓	×	×	×	×
Walter¬Repl [27]	800	✓	✓	✓	✓	✓	✓	×	×	×
Walter [28]	830	✓	✓	✓	✓	✓	✓	×	×	×
P-Store [34]	438	✓	✓	✓	✓	✓	✓	✓	✓	×
Cassandra [29]	252	-	-	-	-	-	-	-	-	-

8 Related Work

Formalizing Consistency Properties in a Single Framework. Adya [2] uses dependencies between reads and writes to define different isolation models in database systems. Bailis et al. [5] adopts this model to define read atomicity. Burckhardt et al. [10] and Cerone et al. [12] propose axiomatic specifications of consistency models for transaction systems using visibility and arbitration relationships. Shapiro et al. [37] propose a classification along three dimensions (total order, visibility, and transaction composition) for transactional consistency models. Crooks et al. [15] formalizes transactional consistency properties in terms of observable states from a client's perspective. On the non-transactional side, Burckhardt [9] focuses on session and eventual consistency models. Viotti *et al.* [40] expands his work by covering more than 50 non-transactional consistency properties. Szekeres *et al.* [39] propose a unified model based on result visibility to formalize both transactional and non-transactional consistency properties.

All of these studies propose semantic models of consistency properties suitable for theoretical analysis. In contrast, we aim at algorithmic methods for automatically verifying consistency properties based on executable specifications of both the systems and their consistency models. Furthermore, none of the studies covered all of the transactional consistency models considered in this paper.

Model Checking Distributed Transaction Systems. There is very little work on model checking state-of-the-art DTSs, maybe because the complexity of these systems requires expressive formalisms. Engineers at Amazon Web Services successfully used TLA+ to model check key algorithms in Amazon's Simple Storage Systems and DynamoDB database [33]; however, they do not state which consistency properties, if any, were model checked. The designers of the TAPIR transaction protocol have specified and model checked correctness properties of their design using TLA+ [43]. The IronFleet framework [21] combines TLA+ analysis and Floyd-Hoare-style imperative verification to reason about protocol-level concurrency and implementation complexities, respectively. Their methodology requires "considerable assistance from the developer" to perform the proofs. Cai [11] proposes some basic patterns for modeling real-time transactions, and uses Timed Computation Tree Logic (TCTL) to specify the timeliness and ACID properties. Li [25] models three multi-version concurrency control mechanisms using the patterns in [11], and verifies transaction timeliness and *SER* in UPPAAL. These case studies are based on timed automata, and none of them checks state-of-the-art DTSs.

Distributed model checkers [23,42] are used to model check *implementations* of distributed systems such as Cassandra, ZooKeeper, the BerkeleyDB database and a replication protocol implementation.

Our previous work [19,20,32,29,34,26,27,28,30] specifies and model checks *single* DTSs and consistency properties in different ways, as opposed to in a single framework that, furthermore, automates the "monitoring" and analysis process.

Other Formal Reasoning about Distributed Database Systems. Cerone et al. [13] develop a new characterization of *SI* and apply it to the static analysis of DTSs. Bernardi et al. [7] propose criteria for checking the robustness of transactional programs against consistency models. Bouajjani et al. [8] propose a formal definition of eventual consistency, and reduce the problem of checking eventual consistency to reachability and model checking problems. Gotsman *et al.* [18] propose a proof rule for reasoning about non-transactional consistency choices.

There is also work [41,24,36] that focuses on specifying, implementing and verifying distributed systems using the Coq proof assistant. Their executable Coq "implementations" can be seen as executable high-level formal specifications, but the theorem proving requires nontrivial user interaction.

Finally, the authors in [17] apply both model checking and meta-programming techniques in Maude to a distributed snapshot algorithm and its reachability property, but they do not consider neither transaction systems nor consistency properties.

9 Concluding Remarks

In this paper we have provided an object-based framework for formally modeling distributed transaction systems (DTSs) in Maude, have explained how such models can be automatically instrumented to record relevant events during a run, and have formally defined a wide range of consistency properties on such histories of events. We have implemented a tool which automates the entire instrumentation and model checking process. Our framework is very general: we could easily adapt previous Maude models of state-of-the-art DTSs such as Apache Cassandra, P-Store, RAMP, Walter, Jessy, and ROLA to our framework.

We then model checked the DTSs w.r.t. all the consistency properties for all initial states with 4 transactions, 2 sites, and so on. This analysis was sufficient to differentiate the DTSs according to which consistency properties they satisfy.

In future work we should formally relate our definitions of the consistency properties to other (non-executable) formalizations of consistency properties. We should also extend our work to formalizing and model checking non-transactional consistency properties for key-value stores such as Cassandra.

References

1. IBM DB2, `https://www.ibm.com/analytics/us/en/db2/`
2. Adya, A.: Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions. MIT (1999)
3. Ardekani, M.S., Sutra, P., Preguiça, N.M., Shapiro, M.: Non-monotonic snapshot isolation. CoRR abs/1306.3906 (2013), `http://arxiv.org/abs/1306.3906`
4. Ardekani, M.S., Sutra, P., Shapiro, M.: Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In: SRDS. pp. 163–172 (2013)
5. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable atomic visibility with RAMP transactions. ACM Trans. Database Syst. 41(3), 15:1–15:45 (2016)
6. 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. pp. 1–10. ACM (1995)
7. Bernardi, G., Gotsman, A.: Robustness against consistency models with atomic visibility. In: CONCUR. LIPIcs, vol. 59, pp. 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
8. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: POPL. pp. 285–296. ACM (2014)
9. Burckhardt, S.: Principles of Eventual Consistency, Foundations and Trends in Programming Languages, vol. 1. Now Publishers (2014)
10. Burckhardt, S., Leijen, D., Fähndrich, M., Sagiv, M.: Eventually consistent transactions. In: ESOP. LNCS, vol. 7211, pp. 67–86. Springer (2012)
11. Cai, S.: Modeling real-time transactions in UPPAAL. Tech. rep., Mälardalen University (2015), http://www.es.mdh.se/publications/3911
12. Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: CONCUR. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)

13. Cerone, A., Gotsman, A.: Analysing snapshot isolation. In: PODC. pp. 55–64. ACM (2016)
14. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
15. Crooks, N., Pu, Y., Alvisi, L., Clement, A.: Seeing is believing: A client-centric specification of database isolation. In: PODC. pp. 73–82. ACM (2017)
16. Date, C.: An Introduction to Database Systems. Addison-Wesley, 5 edn. (1990)
17. Doan, H.T.T., Ogata, K., Bonnet, F.: Specifying a distributed snapshot algorithm as a meta-program and model checking it at meta-level. In: ICDCS. IEEE Computer Society (2017)
18. Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In: POPL. pp. 371–384. ACM (2016)
19. 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)
20. 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)
21. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: SOSP. ACM (2015)
22. Kraska, T., Pang, G., Franklin, M.J., Madden, S., Fekete, A.: MDCC: multi-data center consistency. In: EuroSys. pp. 113–126. ACM (2013)
23. Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J.F., Gunawi, H.S.: SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: OSDI. USENIX Association (2014)
24. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: Certified causally consistent distributed key-value stores. In: POPL. pp. 357–370. ACM (2016)
25. Li, J.: Model checking transaction properties for concurrent real-time transactions in UPPAAL. Master's thesis, Mälardalen University (2016)
26. Liu, S., Ölveczky, P.C., Ganhotra, J., Gupta, I., Meseguer, J.: Exploring design alternatives for RAMP transactions through statistical model checking. In: ICFEM. LNCS, Springer (2017)
27. 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. LNCS, Springer (2018)
28. Liu, S., Ölveczky, P., Wang, Q., Meseguer, J.: Formal modeling and analysis of the Walter transactional data store. In: WRLA. LNCS, Springer (2018)
29. 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)
30. Liu, S., Ölveczky, P., Wang, Q., Gupta, I., Meseguer, J.: Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. Tech. rep., University of Illinois at Urbana-Champaign (2018), http://hdl.handle.net/2142/101836
31. Liu, S., Ölveczky, P.C., Rahman, M.R., Ganhotra, J., Gupta, I., Meseguer, J.: Formal modeling and analysis of RAMP transaction systems. In: SAC. ACM (2016)
32. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: ICFEM. LNCS, vol. 8829. Springer (2014)

33. 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 (April 2015)
34. Ölveczky, P.C.: Formalizing and validating the P-Store replicated data store in Maude. In: WADT'16. LNCS, vol. 10644. Springer (2017)
35. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (Oct 1979)
36. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. Proc. ACM Program. Lang. 2(POPL), 28:1–28:30 (Dec 2017)
37. Shapiro, M., Ardekani, M.S., Petri, G.: Consistency in 3d. In: CONCUR. LIPIcs, vol. 59, pp. 3:1–3:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
38. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP. ACM (2011)
39. Szekeres, A., Zhang, I.: Making consistency more consistent: A unified model for coherence, consistency and isolation. In: PaPoC. ACM (2018)
40. Viotti, P., Vukolić, M.: Consistency in non-transactional distributed storage systems. ACM Comput. Surv. 49(1), 19:1–19:34 (2016)
41. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: PLDI. pp. 357–368. ACM (2015)
42. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: transparent model checking of unmodified distributed systems. In: NSDI. pp. 213–228. USENIX Association (2009)
43. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. In: SOSP 2015. pp. 263–278. ACM (2015)