Formal Modeling and Analysis of Google's Megastore in Real-Time Maude*

Jon Grov[1,2] and Peter Csaba Ölveczky[1,3]

[1] University of Oslo
[2] Bekk Consulting AS
[3] University of Illinois at Urbana-Champaign

Abstract. Cloud systems need to replicate data to ensure scalability and high availability. To enable their use for applications where consistency of the data is important, cloud systems should provide transactions. Megastore, developed and widely applied at Google, is one of very few cloud data stores that provide transactions; i.e., both data replication, fault tolerance, and data consistency. However, the only publicly available description of Megastore is short and informal. To facilitate the widespread study, adoption, and further development of Megastore's novel approach to transactions on replicated data, a much more detailed and precise description is needed. In this paper, we describe an executable formal model of Megastore in Real-Time Maude that we have developed. Our model is the result of many iterations resulting from correcting design flaws uncovered during Real-Time Maude analysis. We describe our model and explain how it can be simulated for QoS estimation and model checked to verify functional correctness.

1 Introduction

Cloud systems enable customers to deploy applications in a highly scalable and available infrastructure. Key to these features is *replication*: several copies of customer data in geographically distributed data centers allow cloud services to cope with peaks in system load, as well as with network and site failures.

Many applications require database facilities for storing valuable data. Databases provide *transactions*: for a given sequence of read and write operations on data items, the user is assured *atomicity*, which means that either no operation is completed or all operations are completed, and *serializability*, which means that the execution of concurrent transactions must provide the same result as some sequential execution. Transactions are necessary protection against inconsistency due to interleaved operations on shared data. For example, if two transactions t_1 and t_2 both read and write bank account x to deposit \$20, it is crucial to avoid both the execution $t_1 : read(x) = 10;\ \ t_2 : read(x) = 10;\ \ t_1 : x := 10 + 20;\ \ t_2 : x := 10 + 20;\ \ t_1 : write(x, 30);\ \ t_2 : write(x, 30)$, where t_1's deposit is lost, and the execution $t_1 : read(x) = 10;\ \ t_1 : x := 10 + 20;\ \ t_1 : write(x, 30);\ \ t_2 : read(x) =$

* This work was partially supported by AFOSR Grant FA8750-11-2-0084.

S. Iida, J. Meseguer, and K. Ogata (Eds.): Futatsugi Festschrift, LNCS 8373, pp. 494–519, 2014.

30; $t_2 : x := 30 + 20$; $t_2 : write(x, 50)$; $abort(t_1)$, where t_2 was allowed to read t_1's update which was later aborted.

Some applications, such as newspaper content management and social networks like Facebook, can tolerate lower degrees of consistency. Other applications have strict consistency requirements; notable examples include stock exchange systems, online auctions, banking, and medical systems: it is clear that a lost update due to concurrent transactions could have serious consequences in a system recording the medication of hospital patients.

Transactions are among the most important features of a database management system (DBMS), since a correct implementation of atomicity and serializability impose significant challenges. To quote Michael Stonebraker [20]:

> "It is possible to build your own [transaction support] on any of these systems, given enough additional code. However, the task is so difficult, we wouldn't wish it on our worst enemy. If you need [transaction support], you want to use a DBMS that provides them; it is much easier to deal with this at the DBMS level than at the application level."

Transaction management in the cloud, with geographical distribution and data replication, involves additional challenges because of:

- Performance: Concurrent access to replicas at different locations requires costly network coordination.
- Availability: The complexity of coordinating transactions across network sites increases significantly due to possible network and site failures.

Given the difficulties of transaction management on replicated data, we believe that formal methods are crucial to enable the use of cloud-based data stores also for applications where strong data consistency is required. First of all, formal analysis should be used to catch subtle "corner case" errors during design and development of the data store. Second, because of the complexity and criticality of such systems, it is necessary for application providers to be convinced that the cloud system indeed provides transaction support. Formal verification could be a major component in providing such assurance to application providers, just like formal methods can be used in Level A certification of critical avionics systems.

There are currently only a few cloud data stores with transaction support. Microsoft's SQL Azure [4] uses a master-based approach to coordination, which reduces fault-tolerance and gives worse performance for clients far from the master site. Google's Spanner [6] demands a complex infrastructure involving GPS hardware and atomic clocks, which reduces its applicability. Google's Megastore [2] provides replication and transactions through a replicated transaction log. Despite its relatively low performance, Megastore is used by Google for many well-known services such as GMail, Android Market, and Google+ [6], and is offered to customers using Google's cloud-based application platform AppEngine.

In this paper, we use the rewriting-logic-based Real-Time Maude language and tool [17] to formally model, simulate, and model check Megastore. The design of Megastore is informally described in the paper [2]. However, designing a complete

fault-tolerant protocol requires much more detail than publicly available. Our contributions are:

1. We provide a precise, formal model of Megastore, which includes many details and aspects not even described informally in [2]. Because of the ambiguity and the lack of detail in the informal specification, we had to make a number of assumptions and design choices in our formalization. Our model is the result of several modifications resulting from extensive model checking during this formalization process.
2. We show how Megastore can be model checked and probabilistically simulated using Maude and Real-Time Maude.
3. We provide a general method for analyzing serializability in distributed transactional systems with replicated data.

Our formal model should facilitate further research on the Megastore approach. In particular, we are working on combining Megastore with the FLACS approach [8] to provide serializable transactions also across partitions.

The rest of the paper is organized as follows: Section 2 gives some background on Maude and Real-Time Maude. Section 3 presents an overview of Megastore and its approach to fault-tolerance. Section 4 describes our formal model of Megastore. Section 5 explains how we have formally analyzed our model. Finally, Section 6 discusses related work and gives some concluding remarks.

2 Maude and Real-Time Maude

Real-Time Maude [13] is a language and tool that extends Maude [5] to support the formal specification and analysis of real-time systems. The specification formalism emphasizes *ease* and *generality* of specification, and is particularly suitable for modeling distributed real-time systems in an object-oriented style. Real-Time Maude specifications are executable, and the tool provides a variety of formal analysis methods, including simulation, reachability analysis, and LTL and timed CTL model checking.

2.1 Maude

Maude [5] is a rewriting-logic-based formal language and high-performance simulation and model checking tool. A Maude module specifies a *rewrite theory* [10,3] $(\Sigma, E \cup A, R)$, where:

- Σ is an algebraic *signature*; that is, a set of declarations of *sorts*, *subsorts*, and *function symbols*.
- $(\Sigma, E \cup A)$ is a *membership equational logic theory* [11], 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 state space as an algebraic data type.

– R is a collection of *labeled conditional rewrite rules* specifying the system's local transitions, each of which has the form[1] $[l] : t \longrightarrow t'$ **if** $\bigwedge_{j=1}^{m} cond_j$, where each $cond_j$ in the condition is either an equality $u_j = v_j$ (u_j and v_j have the same normal form) or a rewrite $t_j \longrightarrow t'_j$ (t_j rewrites to t'_j in zero or more rewrite steps), and l is a *label*. Such a rule specifies a *one-step transition* from a substitution instance of t to the corresponding substitution instance of t', *provided* the condition holds. The rules are universally quantified by the variables appearing in the Σ-terms t, t', u_j, v_j, t_j, and t'_j, and are applied *modulo* the equations $E \cup A$.[2]

We briefly summarize the syntax of Maude and refer to [5] for more details. Operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s. They can have user-definable syntax, with underbars '`_`' marking the argument positions. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are used by the Maude engine to match terms *modulo* the declared axioms. An operator can also be declared to be a constructor (`ctor`) that defines the carrier of a sort. 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 be introduced on the fly in a statement without being declared previously, in which case they have the form $var{:}sort$. An equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "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.

In object-oriented Maude modules, a *class* declaration

`class` C `|` att_1 `:` s_1`,` `...` `,` att_n `:` s_n `.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C in a given state is represented as a term `<` O `:` C `|` $att_1 : val_1, ..., att_n : val_n$ `>` of sort `Object`, 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`, where the declaration `msg` m `:` s_1 `...` s_n `->` `Msg` defines the syntax of the message (`m`) and the sorts (s_1 `...` s_n) of its parameters.

The state is a term of the sort `Configuration` in a concurrent object-oriented system, and has the structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative, so that rewriting is *multiset rewriting* supported directly in Maude. Since a class attribute may have sort `Configuration`, we can have *hierarchical* objects which contain a subconfiguration of other (possibly hierarchical) objects and messages.

[1] An equational condition $u_i = w_i$ can also be a *matching equation*, written $u_i := w_i$, which instantiates the variables in u_i to the values that make $u_i = w_i$ hold, if any.

[2] Operationally, a term is reduced to its E-normal form modulo A before any rewrite rule is applied in Real-Time Maude.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
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 parameterized family of transitions (one for each substitution instance) in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`, the attribute `a1` of the object `O` is changed to `x + w`, and a new message `m'(O',x)` is generated. The message `m(O,w)` is *removed* from the state by the rule, since it does *not* occur in the right-hand side of the rule. Likewise, the message `m'(O',x)` is *generated* by the rule, since it *only* occurs in the right-hand side of the rule. By convention, attributes whose values do not change and do not affect the next state of other attributes or messages, such as `a3`, need not be mentioned in a rule. Similarly, attributes whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, such as `a2`, can be omitted from right-hand sides of rules.

A *subclass* inherits all the attributes and rules of its superclasses.

Formal Analysis in Maude. A Maude module is executable under some conditions, such as the equations being confluent and terminating, possibly modulo some structural axioms, and the theory being coherent [5].

Maude's *rewrite command* simulates *one* of the many possible system behaviors from the initial state by rewriting the initial state. Maude's *search* command uses a breadth-first strategy to search for states that are reachable from the initial state, match the *search pattern*, and satisfy the *search condition*.

Maude's *linear temporal logic model checker* analyzes whether each behavior satisfies a temporal logic formula. *State propositions*, possibly parametrized, are operators of sort `Prop`, and their semantics is defined by equations of the form

`ceq` *statePattern* `|=` *prop* `=` *b* `if` *cond*

for *b* a term of sort `Bool`, which defines the state proposition *prop* to hold in all states *t* such that *t* `|=` *prop* evaluates to `true`. A temporal logic *formula* is constructed by state propositions and temporal logic operators such as `True`, `False`, `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `W` ("weak until"). The command

`(red modelCheck(`*t*`,` *formula*`) .)`

then checks whether the temporal logic formula *formula* holds in all behaviors starting from the initial state *t*. Such model checking terminates if the state space reachable from the initial state *t* is finite.

2.2 Real-Time Maude

A Real-Time Maude [17] *timed module* specifies a *real-time rewrite theory* [16], that is, a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R)$, such that:

1. $(\Sigma, E \cup A)$, contains a specification of a sort `Time` defining the (discrete or dense) time domain.
2. The rules in R are decomposed into:
   - "ordinary" rewrite rules that model *instantaneous* change that is assumed to take zero time, and
   - *tick (rewrite) rules* of the form

     `crl [`l`] : {`t`} => {`t'`} in time` u `if` *cond*

     that model the elapse of time in a system, where `{_}` is a constructor of a new sort `GlobalSystem` and u is a term of sort `Time` denoting the *duration* of the rewrite.

The initial state of a system must be equationally reducible to a term $\{t_0\}$. The form of the tick rules then ensures uniform time elapse in all parts of a system.

Real-Time Maude extends Maude's analysis features to the real-time setting. Real-Time Maude's *timed fair rewrite* command simulates *one* behavior of the system *up to a certain duration*. It is written with syntax

`(tfrew` t `in time <=` *timeLimit* `.)`

where t is the term to be rewritten ("the initial state"), and *timeLimit* is a ground term of sort `Time`. Real-Time Maude extends Maude's *search* command to search for states that can be reached within a given time interval from the initial state.

Real-Time Maude provides both *unbounded* and *time-bounded* LTL model checking. The unbounded model checking command

`(mc` t `|=u` *formula* `.)`

checks whether the temporal logic formula *formula* holds in all behaviors starting from the initial state t. When the reachable state space is infinite, *time-bounded* LTL model checking, in which each behavior starting in t is only analyzed up to a certain time bound, can be used to ensure termination of the model checking.

3 Overview of Megastore

A *data store* is a system providing functionality to write and access persistent data. Data stores are used to offload the complexity of data management from individual applications by providing transaction support, access control, and/or fault recovery. A data store often uses *replication* to ensure high availability in the presence of site and/or network failures: several copies of the same data are stored at different locations.

Megastore [2] is a data store offering very high availability and transaction support. It is deployed within Google's own cloud infrastructure. In addition

to being widely used internally at Google, Megastore is also used by Google's customers through the cloud-based application platform AppEngine. Megastore handles more than three billion write and 20 billion read transactions daily and stores nearly a petabyte of data across many global data centers [2].

Data are replicated among sites (data centers), and Megastore can tolerate failure of up to $n-1$ replicas, with n the total number of replicas. A *transaction* is a sequence of read and write operations on entities, followed by a commit request. Clients can issue transaction requests from any site replicating the relevant data, and updates are propagated to the other replicas before the transaction commits.

In Megastore, data are stored as *entities*, each entity being a set of key-value pairs. Entities are organized into *entity groups*. Transactional serializability is only guaranteed for operations within the same entity group.

Initially, all operations in a transaction are executed locally at the receiving site. When a commit request is issued, a coordination procedure between the sites is used to decide whether or not the transaction is valid and can be committed. If not, usually due to some concurrent update of the same data, the entire transaction is aborted and must be restarted from the beginning.

Megastore uses the Paxos protocol [9] for coordinating updates. This allows most transactions to complete even in the presence of site and/or network failures. Section 3.1 explains the behavior of Megastore in more detail, and Section 3.2 explains how Megastore deals with recovery from faults.

3.1 Transaction Execution in Megastore

Any Megastore site S may receive transaction requests for entities replicated at S. Entities are versioned, and Megastore provides reads with different levels of consistency. We focus here on *current reads*, which give the most recent version written. Any transaction updating an entity must perform a current read before performing the update.

Each site has a *coordinator*, which is always informed about whether the local replica is up-to-date. When a current read is issued, it is executed locally if and only if granted by the local coordinator. Otherwise, a majority read is required, as explained in Section 3.2.

During the execution of a transaction t, read operations are completed immediately, while write operations are buffered. When receiving the commit request, the site receiving t, denoted the *originating site* of t, initiates the coordination procedure. Megastore's approach to combine availability with serializability is to partition data into relatively small units (entity groups), and maintain a separate transaction log for each entity group. This log is replicated, and serializability within the entity group is ensured, since, at any given time, only one transaction is allowed to update the log.

A transaction t accessing an entity group eg reads entities in eg from a given log position lp. t's updates are buffered during transaction execution. When all operations of t are completed and t is ready for commit, the originating site of t prepares a log entry for eg containing t's updates and runs *Paxos* [9] to request that this log entry becomes entry $lp + 1$ in the replicated log.

Paxos is a generic consensus protocol for distributed systems which consists of the following phases:

1. Agree on a leader.
2. The leader then proposes a value to the participating processes.
3. Once the proposed value is acknowledged by at least a majority of the processes, the leader informs all participants about the decision.

In the presence of failures, this may be insufficient to reach consensus, in which case a new round is initiated where another process becomes the leader.

Megastore optimizes Paxos by including in each log entry the Paxos leader for the *next* log entry. Phase 1 is therefore replaced by a request from the originating site directly to the leader. In the case of conflict, i.e., if multiple sites request different log entries for the same log position, Paxos ensures that only one is elected, and the others are aborted.

After a successful Paxos round, each site replicating eg then appends the chosen log entry for position $lp + 1$ to the local copy of the transaction log for eg, and subsequently updates the local data store.

3.2 Fault Tolerance and Failure Recovery

Failures may cause some processes to stop responding and/or may block network messages from being delivered. Fault tolerance implies that a transaction execution must be able to proceed even if some replicating sites are unable to apply the update. This means that a previously failed site may have missed updates on some entity groups.

To provide fault tolerance, Megastore requires that even if a site is unable to apply an update for some entity group, the site's coordinator must be informed and then mark the entity group as invalid. This is part of the Paxos coordination procedure, and means that the coordinator of a failed site must be reachable. Otherwise the update is blocked until the entire site is confirmed to be down by Megastore's underlying failure detection mechanism.

If a site, upon executing a current read, sees that the entity group in question is invalid, it performs a *majority read* and a *catchup* before proceeding. During the majority read, the local site s_l requests from each other replicating site s_r the most recent log position known to be valid by s_r. When s_l has received a reply from a majority of the replicating sites, it performs catchup as follows: any log position missing at s_l is requested from some updated site. When the catchup is complete, the local coordinator marks the replica as valid, and the current read operation can proceed.

4 Formalizing Megastore in Real-Time Maude

This section explains how we have formalized Megastore in Real-Time Maude. Our model contains 56 rewrite rules, of which we only present 15 in this paper. The entire executable formal specification is available at

`http://folk.uio.no/jongr/megastore/maude.html`. Section 4.1 lists our system assumptions, Section 4.2 presents our model of Megastore in the absence of failures, and Section 4.3 shows our model in the presence of failures.

4.1 System Assumptions

Based on the description in [2], we make the following system assumptions:

- Megastore is deployed across geographically distant sites connected by a wide-area network. The network delays between two nodes can therefore vary significantly, and we do not assume FIFO delivery between the same pair of nodes.
- A site always knows all the other replicating sites for an entity group.
- Sites can fail and recover spontaneously, and messages can be dropped due to site or network failures.
- Coordinators are supposed to be very stable. Furthermore, Megastore requires that the coordinator of each running site is accessible; otherwise update transactions are blocked until the given replica is confirmed down and can be excluded. We therefore assume that coordinators are always available.
- Small time differences caused by clock skews of the local clocks are ignored.

4.2 The Model without Failure Handling

We model Megastore in an object-oriented way, where the global state consists of a multiset of site objects and messages traveling between them. Each site is modeled as an object instance of the following class:

```
class Site |
   entityGroups : Configuration,
   localTransactions : Configuration,
   coordinator : EntGroupLogPosPairSet .
```

The attribute `entityGroups` contains one `EntityGroup` object for each entity group replicated at the site, and the attribute `localTransactions` contains one `Transaction` object for each active transaction originating at the site. The attribute `coordinator` denotes the local coordinator state for each entity group, and is a `;`-separated set of terms *eg* `upToDateAt` *lp*, denoting that the entity group *eg* is up-to-date at log position *lp*, and terms *eg* `invalidAt` *lp*, denoting that the local replica of *eg* may be missing some log entries at or before *lp*.

Entity Groups. Each local entity group copy is modeled as an object instance of the following class:

```
class EntityGroup |
   entitiesState : EntitySet,
   transactionLog : LogEntryList,
   replicas : EntityGroupReplicaSet,
   proposals : PaxosProposalSet,
   pendingWrites : PendingWriteList .
```

The attribute `entitiesState` describes the available versions of each entity in the entity group. Each such record is a term `entity(`eg`,`i`) |-> (lpos(`p_1`),`v_1`) :: ... :: (lpos(`p_k`),`v_k`)`, where `entity(`eg`,`i`)` denotes the ith entity of the entity group eg, and `(lpos(`p_j`),`v_j`)` is an entity version containing the value v_j, created at log position p_j.

The attribute `transactionLog` denotes the local copy of the replicated transaction log which is the core of Megastore's replication protocol. Each log entry belongs to a given *log position*. A log entry `(`t lp s ol`)` contains the identity t of the originating transaction, the log position lp, the leader site s for the *next* log entry, and the list ol of write operations executed by t.

The attribute `replicas` denotes the set of sites replicating this entity group. The attribute `proposals` denotes the local state in ongoing Paxos processes involving this entity group. It contains two types of values: `proposal(`s`,`t`,`lp`,`pn`)`, which represents a request from site s to become the leader for log position lp on behalf of transaction t, and `accepted(`s`,`le`,`pn`)`, which states that this site has accepted Paxos proposal number pn containing the log entry le from site s.

Megastore executes write operations in two steps: (i) write to the log, which occurs immediately when the chosen log entry is committed; and (ii) updating the actual data in the `entityState`. The attribute `pendingWrites` maintains a list of write operations waiting to be applied to the `entityState`.

Transactions. A transaction request is a `::`-separated list of current read operations `cr(`e`)` and write operations `w(`e`,`v`)`. Transactions being executed are modeled as object instances of the following class:

```
class Transaction |
   operations : OperationList,
   reads : EntitySet,
   writes : OperationList,
   status : TransStatus,
   readState : ReadStateSet,
   paxosState : PaxosStateSet .
```

The attribute `operations` initially contains the transaction request. During execution, operations are removed from this list. For a read operation the resulting entity is stored in the attribute `reads`. The attribute `writes` is used to buffer write operations. `status` denotes the overall transaction status: `idle`, `executing(`lp`,`t`)` (the transaction is executing at log position lp and will continue executing for time t), and `in-paxos`, which is used during the commit process. The attributes `readState` and `paxosState` store transient data for each entity group accessed by the transaction execution.

Modeling Communication. We assume that the communication delay is nondeterministic. The *set* of possible delays depends on the sender and receiver, and is given by `possibleMsgDelays(`s_1`,` s_2`)` as a '`;`'-separated set of time values:

```
sort TimeSet .    subsort Time < TimeSet .
op emptyTimeSet : -> TimeSet .
```

```
op _;_ : TimeSet TimeSet -> TimeSet [ctor assoc comm id: emptyTimeSet] .
op possibleMsgDelays : SiteId SiteId -> TimeSet [comm] .
```

A "ripe" message has the form

`msg` *mc* `from` *sender* `to` *receiver*

where *mc* is the *message content*. A message in transit that will be delivered after t time units is modeled by a term `dly(msg` *mc* `from` *sender* `to` *receiver*, t`)`:

```
sort DlyMsg .
subsort Msg < DlyMsg < NEConfiguration .
op dly : Msg Time -> DlyMsg [ctor right id: 0] .
msg msg_from_to_ : MsgContent Oid Oid -> Msg .
```

Nondeterministically selecting *any* possible delay from `possibleMsgDelays(`s_1, s_2`)` can be done using a matching equation in the condition of the rewrite rule. A rule creating a single message with nondeterministic delay should have the form[3]

```
var T : Time .    var TS : TimeSet .

crl [sendMsgAnd...] :
    < SID : Site | ... >   ...
   =>
    < SID : Site | ... >   ...
    dly(msg mc from SID to SID', T)
  if ... /\ T ; TS := possibleMsgDelays(SID,SID') .
```

A site must often *multicast* a message to all other sites replicating an entity group. The delay of each single message must of course be selected nondeterministically. A naïve solution to model such multicast by generating the corresponding single messages in any order would be prohibitively expensive from a model checking perspective: if there are n recipients, there would be $n!$ different orders in which these messages could be *created*. We can therefore use a "partial order reduction" technique, in which the messages are sent in a certain order. In particular, the `replicas` attribute of an `EntityGroup` object contains sets of tuples `egrs(SID,N)`, where the second component is unique in the group. We can therefore order the set of recipients, and generate the single messages in this order, reducing the number of possible orders of sending the messages from $n!$ to 1. The following rewrite rule is used to "dissolve" a "multicast message"

`multiCast` *mc* `from SID to EGRS`

into single messages with nondeterministically selected delays:

```
op multiCast_from_to_ : MsgContent Oid EntityGroupReplicaSet
   -> Configuration [ctor] .
eq multiCast MC from SID to noEGR = none .
crl [multiCastToUnicast] :
```

[3] We do not show most variable declarations, but follow the Maude convention that variables are written with capital letters.

```
   multiCast MC from SID to (egrs(SID', N) ; EGRS)
  =>
   dly(msg MC from SID to SID', T)
   (multiCast MC from SID to EGRS)
  if N == smallest(egrs(SID', N) ; EGRS)
    /\ T ; TS := possibleMsgDelays(SID, SID') .
```

Therefore, to multicast a message with message content *mc* to all other sites replicating the entity group `EG`, a rule of the following form *could* be used:

```
rl [multicastReplicatingSites]
   < SID : Site | entityGroups : < EG : EntityGroup | replicas : EGRS, ... > ...
  =>
   < SID : Site | .... >  ...
   (multiCast mc from SID to EGRS) .
```

However, this would still involve $n + 1$ rewrite steps needed to get to a state where all the single messages have been generated, unnecessarily increasing the state space explored during model checking. By using rewrite conditions, we can replace the above rewrite rule with a rule

```
var SINGLE-MSGS : NEConfiguration .

crl [multicastReplicatingSitesEfficient]
    < SID : Site | entityGroups : < EG : EntityGroup | replicas : EGRS, ... > ...
   =>
    < SID : Site | .... >  ...
    SINGLE-MSGS
   if (multiCast mc from SID to EGRS) => SINGLE-MSGS .
```

where `SINGLE-MSGS` is a variable of some sort containing sets of delayed messages, but no occurrences of the `multiCast` operator. In this rewrite rule, all the single messages are created in *one* rewrite step, drastically reducing the reachable state space. (The local "partial order reduction" is still important, since it significantly reduces the number of behaviors explored by Maude during the evaluation of the rewrite condition; however, it does not reduce the reachable state space.)

Dynamic Behavior. The dynamic behavior of Megastore *without* fault tolerance features is modeled by 16 rewrite rules, 7 of which are given below. A transaction request with operations *ol* and name *t* is sent to a site *s* by a message `newTrans(`*s*`,`*t*`,`*ol*`)`. When a site gets such a transaction request, the site adds a corresponding transaction object to its `localTransactions`.

```
rl [newTrans] :
   newTrans(SID, TID, OL)
   < SID : Site | localTransactions : LOCALTRANS >
  =>
   < SID : Site | localTransactions : LOCALTRANS
       < TID : Transaction | operations : OL, readState : emptyReadState,
                             paxosState : emptyPaxosState, reads : emptyEntitySet,
                             writes : emptyOpList, status : idle > .
```

If the next operation in an `idle` transaction `TID` is a current read (`cr`) of an entity `entity(EG,N)` in entity group `EG`, the transaction goes to the local state `executing(LP,readDelay)`, where `LP` is the local coordinator's current log position for `EG`, and `readDelay` is the time it takes to perform a read operation:

```
crl [startCurrentLocalRead] :
    < SID : Site | coordinator : (EG upToDateAt LP ; CES),
                   entityGroups : EGROUPS
                      < EG : EntityGroup | pendingWrites : emptyPWList >
                   localTransactions : LOCALTRANS
                      < TID : Transaction | operations : cr(entity(EG,N)) :: OL,
                                            status : idle > >
   =>
    < SID : Site | localTransactions : LOCALTRANS
                      < TID : Transaction | operations : cr(entity(EG,N)) :: OL,
                                            status : executing(LP, readDelay) > >
   if not (containsUpdate(entity(EG,N), OL) and
           inConflictWithRunning(EG, LOCALTRANS)) .
```

To avoid locals conflicts, a site only allows one active update transaction for each entity group. The condition of the rewrite rule blocks the read request if the transaction `TID` contains an update operation on `entity(EG,N)` until there are no other active conflicting transactions.

When the `executing` timer expires (i.e., becomes zero), the read operation completes and adds the version read at the given log position to `reads`. The transaction status is then set to `idle`, allowing execution to proceed:

```
rl [endCurrentLocalRead] :
   < SID : Site |
      entityGroups : EGROUPS
        < EG : EntityGroup | entitiesState : (entity(EG,N) |-> EVERSIONS) ; BSTATE >,
      localTransactions : LOCALTRANS
        < TID : Transaction | operations : cr(entity(EG,N)) :: OL, readState : RSTATE,
                              status : executing(LP, 0), reads : READS > >
   =>
    < SID : Site |  localTransactions : LOCALTRANS
        < TID : Transaction |
             operations : OL, readState : readpos(EG, LP) ; RSTATE, status : idle,
             reads : READS ; (entity(EG,N) |-> getVersion(LP, EVERSIONS)) > > .
```

A write operation is moved to the buffer `writes`, and will be executed once the transaction is committed:

```
rl [bufferWriteOperation] :
   < SID : Site | localTransactions : LOCALTRANS
       < TID : Transaction | operations : w(EID, VAL) :: OL, writes : WRITEOPS,
                             status : idle > >
  =>
   < SID : Site | localTransactions : LOCALTRANS
       < TID : Transaction | operations : OL, writes : WRITEOPS :: w(EID, VAL) > > .
```

When all operations in the `operations` list are completed (reads) or buffered (writes), the transaction is ready to commit. All buffered updates are merged

into a candidate log entry. If the transaction updates entities from several entity groups, one log entry is created for each group.

For each such entity group, the first step is to send the candidate log entry to the leader for the *next* log position, which was selected during the previous coordination round. The rule for initiating Paxos is modeled as follows:

```
crl [initiateCommit] :
    < SID : Site |
        entityGroups : EGROUPS,
        localTransactions : LOCALTRANS
           < TID : Transaction | operations : emptyOpList,
                                 writes : WRITEOPS, status : idle
                                 readState : RSTATE, paxosState : PSTATE > >
  =>
    < SID : Site |
        localTransactions : LOCALTRANS
           < TID : Transaction | paxosState : NEW-PAXOS-STATE,
                                 status : in-paxos > >
    ACC-LEADER-REQ-MSGS
  if EIDSET := getEntityGroupIds(WRITEOPS) /\
     NEW-PAXOS-STATE := initiatePaxosState(EIDSET, TID, WRITEOPS,
                                           SID, RSTATE, EGROUPS)
  /\ (createAcceptLeaderMessages(SID, NEW-PAXOS-STATE)) => ACC-LEADER-REQ-MSGS .
```

`getEntityGroupIds(WRITEOPS)` contains entity groups accessed by operations in `WRITEOPS`, and `NEW-PAXOS-STATE` contains one record for each entity group. These records contain the log position that `TID` requests to update and the candidate log entry le. The operator `createAcceptLeaderMessages` generates an `acceptLeaderReq` message to the leader of each entity group containing the transaction id `TID` and candidate log entry le.

The execution then proceeds as follows for each entity group:

1. When the leader s_l receives an `acceptLeaderReq` message from the originating site s_o for the transaction `TID`, the leader site inspects the `proposals` set for the given entity group, to check whether it has previously accepted some value for this log position and entity group. If so, there is a conflict, and s_l signals this with a message to the originating site of `TID`, which aborts the transaction. Otherwise, s_l sends an `acceptLeaderRsp` message to s_o.
2. When it receives an `acceptLeaderRsp` message, the originating site proceeds by multicasting the log entry to the other replicating sites. Each recipient of this message must verify that it has not already granted an accept for this log position. If so, the recipient replies with an accept message to the originating site. We show this rule below.
3. After receiving an `acceptAllRsp` message from all replicating sites, the originating site confirms the commit by multicasting an `applyReq` message. When receiving this message, a recipient appends the proposed log entry to the `transactionLog` of the entity group, and the update operations are added to the `pendingWrites` list. With this, the transaction is committed.

The following rule shows the rule from step 2 where a replicating site receives an `acceptAllReq` message. The site verifies that it has not already granted an

accept for this log position (since messages could be delayed for a long time, it checks both the transaction log and received proposals). If there are no such conflicts, the site responds with an accept message, and stores its accept in `proposals` for this entity group. The record `(TID' LP SID OL)` represents the candidate log entry, containing the transaction identifier `TID'`, the log position `LP`, the proposed leader site `SID`, and the list of update operations `OL`.

```
crl [rcvAcceptAllReq] :
    (msg acceptAllReq(TID, EG, (TID' LP SID OL), PROPNUM) from SENDER to THIS)
    < THIS : Site |
      entityGroups : EGROUPS
        < EG : EntityGroup | proposals : PROPSET, transactionLog : LEL > >
  =>
    < THIS : Site |
      entityGroups :  EGROUPS
        < EG : EntityGroup |
          proposals : accepted(SENDER, (TID' LP SID OL), PROPNUM) ;
                      removeProposal(LP, PROPSET) > >
    dly(acceptAllRsp(TID, EG, LP, PROPNUM) from THIS to SENDER), T)
  if not (containsLPos(LP, LEL) or hasAcceptedForPosition(LP, PROPSET))
     /\ T ; TS := possibleMessageDelay(THIS, SENDER) .
```

Modeling Time and Time Elapse. We follow the guidelines in [17] for modeling time in object-oriented specifications. Since an action can only be triggered by the arrival of a message, the expiration of a timer, or by another event, we use the following tick rule to advance time until the next event will take place:

```
crl [tick] : {SYSTEM}  =>  {delta(SYSTEM, mte(SYSTEM))
             if mte(SYSTEM) > 0 /\ mte(SYSTEM) =/= INF .
```

The function `mte` denotes the minimum time that can elapse until the next event will take place, and `delta` defines the effect of time elapse on the state. For example, `mte(dly(M,T) REST) = min(T, mte(REST))`, which means that `mte`(m) is zero for a ripe message m (since m is identical to `dly`(m,`0`)). Therefore, time cannot advance when there are ripe messages in the configuration.

We import the built-in module `NAT-TIME-DOMAIN-WITH-INF`, which defines the time domain `Time` to be the natural numbers, with an additional constant `INF` (for ∞) of a supersort `TimeInf`.

4.3 Modeling Megastore's Fault Tolerance Mechanisms

Megastore is supposed to tolerate: (i) site failures (except for the coordinators); (ii) message loss; and (iii) arbitrarily long message delays. We have formalized these fault tolerance features using 37 rewrites rules, out of which we show only 1 rule in this paper. Our model provides fault tolerance and consistency through the following mechanisms:

– A Paxos-based commit protocol to ensure that even in the presence of multiple failure and recovery events, all available replicas agree on the value for the

next log position. If the originating site s_o, after sending an `acceptLeaderReq` message for log position lp, does not receive a response from the leader of lp within a certain amount of time, it attempts to become the leader itself by sending a `prepareAllReq` message to all replicating sites. When receiving a positive response from a majority of sites, s_o proceeds with the accept phase by multicasting an `acceptAllReq` message to all replicating sites. If at this point s_o fails to receive an `acceptAllRsp` message from a majority of sites, it re-initiates the prepare step after a nondeterministic backoff.
- If a replicating site s_r is unable to apply an update, the coordinator at s_r must ensure that the site avoids serving invalid data. After obtaining a `acceptAllRsp` message from a majority of the replicating sites, the originating site sends an `invalidateCoordinator` message to each site which did not respond in time to the `acceptAllReq` message.
- A majority read and catchup procedure is used to bring a replica up-to-date in case of failures. When executing a current read operation requesting an entity from an invalid entity group eg (according to the coordinator), the originating site s_o broadcasts a `majorityRead` request to all sites replicating eg. Each available recipient responds with the highest log position seen so far. When a majority of replicating sites have responded, s_o sends a `catchupRequest` containing the highest received log position to *one* of the responding sites. If this site does not have a complete log, s_o sends several catchup requests. Once s_o's log is complete, the entity group is marked as valid in the coordinator.

The following rule belongs to the first mechanism above, and shows how we meet a requirement of Paxos: after a site has accepted a log entry, it can never accept another log entry for this log position. Therefore, if a replicating site receives a `prepareAllReq` message for a log position where it has already accepted a log entry, the entry is sent to the originating site in a `prepareAllRsp` message. At the originating site, the log entry for the highest proposal number seen so far is stored within the `prepare` record of `paxosState`. If the originating site has received `prepareAllRsp` from a majority of the participating sites (`hasQuorum(size(SIS ; SENDER), REPLICAS)`), it initiates the `acceptAll` step by multicasting an `acceptAllReq` to all sites replicating the entity group `EG`:

```
crl [rcvPrepareAllRspWithValue] :
    (msg prepareAllRsp(TID,EG, (TID2 LP MSID1 OL1), PROPNUM, PN)
         from SENDER to THIS)
    < THIS : Site |
       entityGroups : < EG : EntityGroup | replicas : EGRS > EGROUPS,
       localTransactions : LOCALTRANS
          < TID : Transaction | status : in-paxos,
              paxosState : prepare(EG, (TID3 LP MSID2 OL2),
                                   PROPNUM, SEEN-PROPNUM, SIS, EXP) ; PSTATE > >
   =>
    < THIS : Site |
       localTransactions : LOCALTRANS
          (if hasQuorum(size(SIS ; SENDER), REPLICAS) then
             < TID : Transaction | status : in-paxos, paxosState :
```

```
                    acceptAll(EG, NEW-LE, PROPNUM, THIS, defTimeout) ; PSTATE >
          else
            < TID : Transaction | paxosState :
                    prepare(EG, NEW-LE, PROPNUM, maxPn(PN, SEEN-PROPNUM),
                            (SIS ; SENDER), EXP) ; PSTATE >
          fi) >
    MSGS
   if REPLICAS := getSites(EGRS) /\
      NEW-LE := chooseValue(PN, SEEN-PROPNUM,
                            (TID2 LP MSID1 OL1),(TID3 LP MSID2 OL2))
      /\ (if hasQuorum(size(SIS ; SENDER), REPLICAS) then
            multiCast acceptAllReq(TID, EG, NEW-LE, PROPNUM) from THIS to EGRS
          else none fi) => MSGS .
```

Site Failures. All processing is blocked and incoming messages are dropped when a site has failed. The exception is that the (co-located) coordinator of the site is supposed to be available, and be able to receive and respond to `invalidateCoordinator` messages even when the site is otherwise failed.

We model site failures in a modular way by enclosing the failed site object by a "wrapper": a failed site is modeled as a term `failed(< s : Site | ... >)`. This wrapper is declared to be a *frozen* operator (see [5])

```
op failed : Object -> Object [ctor frozen (1)] .
```

which ensures that no activity takes place inside the failed object.

A message arriving at a failed site is dropped, unless it is a message to the coordinator:

```
crl [msgWhenSiteFailure] :
    (msg MC from SENDER to SID)  failed(< SID : Site | >)
   =>
    failed(< SID : Site | >)
   if not isInvalidateCoordinator(MC) .

crl [invalidateCoordinator] :
    (msg invalidateCoordinator(EG, LP) from SENDER to THIS)
    failed(< THIS : Site | coordinator : CES >)
   =>
    failed(< THIS : Site | coordinator : applyInvalidate(EG, LP, CES) >)
    (dly invalidateConfirmed(EG, LP) from THIS to SENDER, T)
   if T ; TS := possibleMsgDelays(THIS,SENDER) .
```

In our analysis, we use "messages" `siteFailure(`s`)` and `siteRepair(`s`)` to inject failures and repairs as follows:

```
msgs siteFailure siteRepair : SiteId -> Msg .

crl [siteDown] :
    siteFailure(SID) < SID : Site | > => failed(< SID | >) dly(siteRepair(SID), T)
  if T ; TS := possibleSiteRepairTimes .

rl [siteUp] :
   siteRepair(SID)  failed(< SID : Site | >)  =>  < SID : Site | > .
```

5 Formally Analyzing our Model of Megastore

We used both simulation and temporal logic model checking throughout the development of our formal model from the description in [2]. Simulation provided quick feedback; allowed us to analyze large systems with many sites, transactions, and failures; and "probabilistic" simulation was used for quality of service (QoS) estimation of the model. Model checking, which explores all possible system behaviors, turned out to be very useful to find a number of subtle design flaws that were not uncovered during extensive simulations.

This section shows how our model of Megastore can be formally analyzed in (Maude and) Real-Time Maude. In particular, Section 5.1 lists the main properties to analyze; Section 5.2 gives some parameters of our model; Section 5.3 shows how we can simulate our model for QoS estimation; Section 5.4 explains our model checking of the model *without* fault-tolerance features; and Section 5.5 describes the model checking of the entire model. Finally, Section 5.6 presents a general technique for formally analyzing the *serializability* property of transactional systems: each execution is equivalent to one in which all operations of a transaction are completed before the next transaction begins.

5.1 Properties to Analyze

We use Real-Time Maude to analyze both *quality of service* and *correctness* properties of our model. The important quality of service parameters are:

1. Transaction latency: the delay between the reception of a transaction request and the response to the caller.
2. The fraction of received transactions that are committed and aborted, respectively.

If there are a *finite* number of transactions to be executed, then the main correctness properties that the system should satisfy are:

3. All transactions will eventually finish their execution.
4. All replicas of an entity must eventually have the same value.
5. All logs for the same entity group must eventually contain the same entries.
6. The execution is serializable; i.e., it gives the same result as some execution where the transactions are executed one after the other.
7. Furthermore, from some point on, the properties 3-6 above must hold for all future states.

5.2 System Parameters

There are a number of system parameters in our model, including:

- the number of sites;
- the set of possible message delays between each pair of sites;
- the number of transactions and their arrival times;

- the set of operations in each transaction;
- the number of entities and their organization into entity groups;
- the degree of replication of the different entity groups;
- the number and time distribution of site failures, and the set of possible durations of a site failure;
- the amount of message losses; and
- the duration of the timeouts before initiating fault handling procedures.

Changing these parameters allows us to analyze the model under different scenarios. For example, to define the set of possible message delays, we need to define the function `possibleMsgDelays`. In some of the model checking commands, we use three sites and the following message delays:

```
eq possibleMsgDelays(PARIS, LONDON) = (10 ; 30 ; 80) .
eq possibleMsgDelays(PARIS, NEW-YORK) = (30 ; 60 ; 120) .
eq possibleMsgDelays(LONDON, NEW-YORK) = (30 ; 60 ; 120) .
```

Transactions and failures are injected into the system by (delayed) messages `dly(newTrans(`s`,`t`,`ol`),`$startTime$`)` and `dly(siteFailure(`s`),`$failureTime$`)`. For example, some of our analyses use `initTransactions` and `initFailures`, where the start time of each transaction is nondeterministically selected from the set of possible start times `transStartTime`, and the time of each failure is nondeterministically selected from the set `ttf`:

```
vars T1 T2 T3 : Time .    vars TS1 TS2 TS3 : TimeSet .

crl [delayTransactions] :
    initTransactions
   =>
    dly(newTrans(PARIS, T-K, cr(entity(EG1,0)) :: w(entity(EG1,0),value(2))), T1)
    dly(newTrans(LONDON, T-L, cr(entity(EG1,0)) :: w(entity(EG1,0),value(5))), T2)
    dly(newTrans(NEW-YORK, T-M, cr(entity(EG2,0)) :: w(entity(EG2,0),value(4))), T3)
   if  T1 ; TS1 := transStartTime /\ T2 ; TS2 := transStartTime
    /\ T3 ; TS3 := transStartTime .

eq transStartTime = (10 ; 50 ; 200) .

crl [delayFailures] :
    initFailures => dly(siteFailure(LONDON), T1) dly(siteFailure(NEW-YORK), T2)
  if T1 ; TS1 := ttf /\ (T2 ; TS2) := ttf .

eq ttf = (40 ; 100) .
```

The initial state `initMegastore` can then be defined as follows:

```
op initMegastore : -> GlobalSystem .
eq initMegastore = {initSites initTransactions initFailures} .

eq initSites =
  < PARIS : Site | coordinator : EG1 upToDateAt lpos(0) ; EG2 upToDateAt lpos(0),
                   entityGroups : entityGroupsParis, localTransactions : none >
  < LONDON : Site | coordinator : EG1 upToDateAt lpos(0) ; EG2 upToDateAt lpos(0),
                    entityGroups : entityGroupsLondon, localTransactions : none >
  < NEW-YORK : Site | coordinator : EG1 upToDateAt lpos(0) ; EG2 upToDateAt lpos(0),
                      entityGroups : entityGroupsNY, localTransactions : none > .
```

5.3 Simulation

We can use Real-Time Maude's timed rewrite command to simulate the system for a certain duration:

```
Maude> (tfrew initMegastore in time <= 850 .)

{< LONDON : Site | coordinator : EG1 upToDateAt lpos(0) ; EG2 upToDateAt lpos(1),
  entityGroups :(
   < EG1 : EntityGroup |
     entitiesState : entity(EG1,0) |-> lpos(0)value(0) ; entity(EG1,1) |-> lpos(0)value(0),
     pendingWrites : emptyPWList,
     proposals : accepted(LONDON,T-L lpos(1) LONDON w(entity(EG1,0),value(5)),2),
     replicas : egr(LONDON,0,lpos(0)) ; egr(NEW-YORK,2,lpos(0)) ; egr(PARIS,1,lpos(0)),
     transactionLog : initTrans lpos(0) PARIS emptyOpList >
  < EG2 : EntityGroup | ... >),
  localTransactions :
     < T-L : Transaction | operations : emptyOpList,
         paxosState : acceptAll(EG1,T-L lpos(1) LONDON w(entity(EG1,0),value(5)),
                                1,LONDON ; NEW-YORK, 240),
         reads : entity(EG1,0)|-> lpos(0)value(0), writes : w(entity(EG1,0),value(5)),
         readState : readpos(EG1,lpos(0)), status : in-paxos > >
< NEW-YORK : Site | ... >
< PARIS : Site | ... >} in time 850
```

Although this gives very quick and useful feedback, each application of a rule which selects a value nondeterministically will select the same value. To simulate more random behaviors, and to obtain more realistic QoS estimates, we have also defined a "probalistic" version of our model where the different delays are given by discrete probability distributions. We then add an object containing the seed to Maude's built-in `random` function to the configuration, and use this random value to sample a message delay from the probability distribution. Our probability distribution for the network delays is as follows:[4]

	30%	30%	30%	10%
London ↔ Paris	10	15	20	50
London ↔ New York	30	35	40	100
Paris ↔ New York	30	35	40	100

We generate transactions with a *transaction generator* for each site, which generates transaction requests at random times, with an adjustable average rate measured in *transactions per second (TPS)*. We simulated two fully replicated entity groups. We assume a delay of 10 ms for a local read operation in accordance with the real-world measurements reported in [2].

Simulation without Fault Injection. With an average of 2.5 TPS and no failures, we observe the following results in a run of 200 seconds:

	Avg. latency (ms)	Commits	Aborts
London	122	149	15
New York	155	132	33
Paris	119	148	18

[4] The delays New York–Paris and New York–London are the same, assuming transatlantic backbone links from each of these cities. The delay between Paris and London reflect that network equipment and local lines increase delivery times.

The relatively high abort rate is expected, since we have only two entity groups. While our calibration data are estimates based on a typical setup for this type of cloud service combined with information given in [2], our measured latency appears fairly consistent with Megastore itself [2]: "Most users see average write latencies of 100–400 milliseconds, depending on the distance between datacenters, the size of the data being written, and the number of full replicas."

Simulation with Fault Injection. We have modified the above experiment by adding a fault injector that randomly injects short outages in the sites. The mean time to failure and the mean time to repair for each site was set to 10 and 2 seconds, respectively. This is a challenging scenario where a large fraction of the transactions will experience failure on one or multiple sites. The results from our simulation are given in the following table.

	Avg. latency (ms)	Commits	Aborts
London	218	109	38
New York	336	129	16
Paris	331	116	21

Although both the average latency and the abort rate increase significantly, these results indicate that Megastore is able to maintain an acceptable quality of service under this challenging failure scenario.

5.4 Model Checking the Model without Fault Tolerance

We use linear temporal logic model checking to verify that all possible executions from a given initial state satisfy the correctness properties 3-5 and 7 in Section 5.1 (the serializability analysis is explained in Section 5.6).

The state proposition `allTransFinished` is `true` in all states where all transactions have finished executing. That is, there are no `Transaction` objects remaining in a site's `localTransactions` and there are no messages in the system:

```
vars SYSTEM REST LOCALTRANS EGS1 EGS2 : Configuration .
var M : Msg .     vars ES1 ES2 : EntitySet .     vars TL1 TL2 : LogEntryList .

op allTransFinished : -> Prop [ctor] .
eq {initTransactions REST}  |= allTransFinished = false .
eq {< S1 : Site | localTransactions : < TID : Transaction | > LOCALTRANS > REST}
      |= allTransFinished = false .
eq {M REST}  |= allTransFinished = false .
eq {SYSTEM}  |= allTransFinished = true [owise] .
```

This definition first characterizes the states where `allTransFinished` does *not* hold; the last equation, with the `owise` attribute, then defines `allTransFinished` to be `true` for all other states.

The following proposition `entityGroupsEqual` is `true` for all states where all replicas of each entity have the same value:

```
op entityGroupsEqual : -> Prop [ctor] .
ceq {< S1 : Site | entityGroups : < EG1 : EntityGroup | entitiesState : ES1 > EGS1 >
     < S2 : Site | entityGroups : < EG1 : EntityGroup | entitiesState : ES2 > EGS2 >
     REST}  |= entityGroupsEqual = false if ES1 =/= ES2 .
eq  {SYSTEM}  |= entityGroupsEqual = true [owise] .
```

In the same way, we can define when all transitions logs for each entity group are equal:

```
op transLogsEqual : -> Prop [ctor] .
ceq {< S1 : Site | entityGroups : < EG1 : EntityGroup | transactionLog : TL1 > EGS1 >
     < S2 : Site | entityGroups : < EG1 : EntityGroup | transactionLog : TL2 > EGS2 >
     REST}  |= transLogsEqual = false if TL1 =/= TL2 .
eq  {SYSTEM}  |= transLogsEqual = true [owise] .
```

The temporal logic formula

```
<> [] (allTransFinished /\ entityGroupsEqual /\ transLogsEqual)
```

says that in *all possible executions* from the initial state, a state satisfying Properties 1–3 and where all subsequent states also satisfy those properties, will eventually be reached.

In the absence of the sophisticated failure handling, this formula should hold for all possible message delays and transaction (start and execution) times. We have therefore abstracted from the real-time features of our model, such as message delays, execution times, and timers, and have transformed our model into an *untimed model* that will exhibit *all possible* behaviors of the system. Model checking this property for the initial state `initMegastore` (without delays) with the three sites and three transactions can be done in Maude as follows:

```
Maude> (red modelCheck(initMegastore,
               <> [] (allTransFinished /\ entityGroupsEqual /\ transLogsEqual)) .)

result Bool :  true
```

That is, the desired property holds. The model checking took 950 seconds on an Intel Xeon 1.87Ghz CPU with 128 GB RAM. Reachability analysis showed that this untimed model has 992,992 states reachable from `initMegastore`. Both model checking and reachability analysis from `initMegastore` extended with a fourth transaction were aborted due to lack of memory after 11 hours.

5.5 Model Checking the Model with Failure Handling

The analysis in Section 5.4 shows that model checking the *untimed* model is unfeasible for four transactions even *without* the large fault-tolerance part. Furthermore, the fault-tolerance features of Megastore require an extensive use of timers. Therefore, we model check only the real-time version described in Section 4 when including the fault-tolerance part.

Since we consider a finite number of transactions, the desired property must now also take into account the following possibility: if a failure causes one or

more of the sites to miss the *last* update, leaving its coordinator invalidated, then no further transactions will arrive to initiate a majority read. Therefore, we use modified versions of the propositions in Section 5.4, that make sure that we only require equal `entitiesState` and `transactionLog` among sites where the coordinator indicates that the given entity group is up-to-date:

```
op entityGroupsEqualOrInvalid : -> Prop [ctor] .
ceq  {< S1 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                    entityGroups : < EG1 : EntityGroup | entitiesState : ES1 > EGS1 >
      < S2 : Site | coordinator : eglp(EG1, LP) ; EGLP,
                    entityGroups : < EG1 : EntityGroup | entitiesState : ES2 > EGS2 >
      REST} |= entityGroupsEqual = false if ES1 =/= ES2 .
eq {SYSTEM} |= entityGroupsEqualOrInvalid = true [owise] .
```

We have model checked a number of scenarios, all with three sites, two entity groups, three transactions (each accessing one item in each entity group). The parameters we modify are: the number of possible message delays, the possible start times of a transaction, and the number of failures and their start times. In the case with possible message delays $\{20, 100\}$, possible transaction start times $\{10, 50, 200\}$, and one failure at time 60, the following (unbounded) Real-Time Maude model checking command verifies the desired property in 1164 seconds:

```
Maude: (mc initMegastore |=u <> [] (allTransFinished /\ entityGroupsEqualOrInvalid
                                    /\ transLogsEqualOrInvalid) .)

result Bool :  true
```

We summarize the execution time of the above model checking command for different system parameters, where $\{n_1, \ldots, n_k\}$ means that the corresponding value is selected nondeterministically from the set. All the model checking commands that finished executing returned `true`. *DNF* means that the execution was aborted after more than 4 hours.

Msg. delay	#Trans	Trans. start time	#Fail.	Fail. time	Run (sec)
$\{20, 100\}$	4	$\{19, 80\}$ and $\{50, 200\}$	0	-	1367
$\{20, 100\}$	3	$\{10, 50, 200\}$	1	60	1164
$\{20, 40\}$	3	20, 30, and $\{10, 50\}$	2	$\{40, 80\}$	872
$\{20, 40\}$	4	20, 20, 60, and 110	2	70 and $\{10, 130\}$	241
$\{20, 40\}$	4	20, 20, 60, and 110	2	$\{30, 80\}$	DNF
$\{10, 30, 80\}$,and $\{30, 60, 120\}$	3	20, 30, 40	1	$\{30, 80\}$	DNF
$\{10, 30, 80\}$,and $\{30, 60, 120\}$	3	20, 30, 40	1	60	DNF

5.6 Model Checking Serializability

The *serialization graph* for a given execution of a set of committed transactions is a directed graph where each transaction is represented by a node, and where there is an edge from a node t_1 to another node t_2 iff the transaction t_1 has executed an operation on entity e before transaction t_2 executed an operation

on the same entity, and at least one of the operations was a write operation. It is well known that an execution of multiple transactions is serializable if and only if its *serialization graph* is acyclic [21].

If there is only one version of each entity, and every update therefore overwrites the previous version, the *before* relation follows real time. In a multi-versioned replicated data store like Megastore, we require a defined *version order* $<<$ on the written entity values to decide the *before* relation when constructing the serialization graph. For example: a write operation `w(`e`,`v`)` which creates a version k of entity e occurs *before* a current read `cr(`e`)` iff `cr(`e`)` reads a version l where $k << l$ according to the selected version order.

Since we require serializability within each entity group only, and every committed transaction is assigned a unique log position for each entity group it updates, we use log positions for the version order. This means that if, for example, t_i reads from log position lp and t_k commits an update at log position lp', then $t_i \rightarrow t_k$ in the serialization graph iff $lp < lp'$.

When an update transaction t_i commits, it produces a message containing:

- the log position and value of each entity it has read; and
- the set of entities written, all of them have the log position assigned to t_i.

We therefore add to the state an object of class `TransactionHistory` containing the current serialization graph. Each time a transaction commits, this object reads the above message and updates its serialization graph.

The sort `SerGraph` defines a set of edges:

```
var E : Edge .
sort SerGraph .      sort Edge .      subsort Edge < SerGraph .
op _<->_ : TransId TransId -> Edge [ctor] .
op emptyGraph : -> SerGraph [ctor] .
op _;_ : SerGraph SerGraph -> SerGraph [ctor assoc comm id: emptyGraph] .
eq E ; E = E .

class TransactionHistory | graph : SerGraph .
```

The proposition `isSerializable` can then be defined as expected:

```
op isSerializable : -> Prop [ctor] .
eq {< th : TransactionHistory | graph : GRAPH > REST}
          |= isSerializable = not hasCycle(GRAPH) .
```

We can therefore verify that for each state, the execution up to the current state is serializable:

```
Maude: (mc initMegastore |=u [] isSerializable .)

result Bool :  true
```

6 Related Work and Concluding Remarks

Despite the importance of transactional data stores, we are not aware of any work on formalizing and verifying such systems. We are also not aware of any detailed description of Megastore itself beyond [2].

The paper [18] addresses the need for formal analysis of replication and concurrency control in transactional cloud data stores. Using Megastore as a motivating example, the authors propose a generic framework for concurrency control based on Paxos, and include a pseudo-code description of Paxos and a proof of how it can be used to ensure serializability. In contrast, we provide a much more detailed and formal model not only of Paxos, but of Megastore itself.

The value of Maude for formally analyzing other cloud mechanisms is demonstrated in [19], where the authors point out possible bottlenecks in a naïve implementation of ZooKeeper for key distribution, and in [7], where the authors analyze denial-of-service prevention mechanisms using Maude and PVeStA.

Real-Time Maude has been used to model and analyze a wide range of advanced state-of-the-art systems, including multicast protocols [14], wireless sensor network algorithms [15], and scheduling protocols [12]. In all these applications, Real-Time Maude analysis uncovered significant design errors that could be traced back to flaws in the original system. The work presented in this paper differs fundamentally from those applications of Real-Time Maude: in this case, our starting point was a fairly brief and informal overview paper on Megastore – in addition to a number of papers describing the underlying Paxos protocol. We therefore had to "fill in" a lot of details, in essence developing and formalizing our own version of the Megastore approach. The available source on Megastore was not detailed enough to allow us to map flaws found during Real-Time Maude model checking to flaws in the original description of Megastore. Instead, we used Real-Time Maude simulation and model checking extensively throughout our development of this very complex system to improve our model to the point where we cannot find any flaws during our model checking analyses.

Our main contribution is therefore this fairly detailed executable formal model of (our version of) Megastore. Minor contributions include general techniques for: (i) efficiently modeling multicast with nondeterministic message delays in Real-Time Maude; and (ii) model checking the serializability property of distributed transactions on replication data in (Real-Time) Maude.

We hope that our formalization contributes to further research on the Megastore approach to transactional data stores. In particular, we are planning on combining Megastore with the FLACS approach [8] to provide serializability also for transactions accessing multiple entity groups. Other future work includes defining a probabilistic version of our model in a probabilistic extension of Maude, and use the PVeStA tool [1] for statistical model checking and more advanced QoS estimation.

Acknowledgments. We are grateful to the Festschrift editors for giving us the opportunity to honor Kokichi Futatsugi, an excellent researcher and a true gentleman. The second author and his family still have many fond memories from Kokichi and Junko's visit to Oslo a few years ago. We would also like to thank the anonymous reviewers for very insightful comments which have helped us improve the paper.

References

1. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011)
2. Baker, J., et al.: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR (2011), `http://www.cidrdb.org`
3. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360(1-3), 386–414 (2006)
4. Campbell, D.G., Kakivaya, G., Ellis, N.: Extreme scale with full SQL language support in Microsoft SQL Azure. In: SIGMOD 2010, pp. 1021–1024. ACM (2010)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Corbett, J.C., et al.: Spanner: Google's globally-distributed database. In: OSDI 2012, pp. 251–264. USENIX Association, Berkeley (2012)
7. Eckhardt, J., Mühlbauer, T., AlTurki, M., Meseguer, J., Wirsing, M.: Stable availability under denial of service attacks through formal patterns. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 78–93. Springer, Heidelberg (2012)
8. Grov, J., Ölveczky, P.C.: Scalable and fully consistent transactions in the cloud through hierarchical validation. In: Hameurlain, A., Rahayu, W., Taniar, D. (eds.) Globe 2013. LNCS, vol. 8059, pp. 26–38. Springer, Heidelberg (2013)
9. Lamport, L.: Paxos made simple. ACM Sigact News 32(4), 18–25 (2001)
10. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
11. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
12. Ölveczky, P.C., Caccamo, M.: Formal simulation and analysis of the CASH scheduling algorithm in Real-Time Maude. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 357–372. Springer, Heidelberg (2006)
13. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
14. Ölveczky, P.C., Meseguer, J., Talcott, C.L.: Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude. Formal Methods in System Design 29(3), 253–293 (2006)
15. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. Theoretical Computer Science 410(2-3), 254–280 (2009)
16. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theor. Comput. Sci. 285(2), 359–405 (2002)
17. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
18. Patterson, S., Elmore, A.J., Nawab, F., Agrawal, D., El Abbadi, A.: Serializability, not serial: concurrency control and availability in multi-datacenter datastores. Proc. VLDB Endow. 5(11), 1459–1470 (2012)
19. Skeirik, S., Bobba, R.B., Meseguer, J.: Formal analysis of fault-tolerant group key management using ZooKeeper. In: IEEE International Symposium on Cluster Computing and the Grid, pp. 636–641 (2013)
20. Stonebraker, M., Cattell, R.: 10 rules for scalable performance in 'simple operation' datastores. Commun. ACM 54(6), 72–80 (2011)
21. Weikum, G., Vossen, G.: Concurrency Control and Recovery in Database Systems. Morgan Kaufman Publishers (2001)