Formalizing and Validating the P-Store Replicated Data Store in Maude

Peter Csaba Ölveczky

▶ **To cite this version:**

Peter Csaba Ölveczky. Formalizing and Validating the P-Store Replicated Data Store in Maude. 23th International Workshop on Algebraic Development Techniques (WADT), Sep 2016, Gregynog, United Kingdom. pp.189-207, 10.1007/978-3-319-72044-9_13 . hal-01767476

HAL Id: hal-01767476

https://inria.hal.science/hal-01767476

Submitted on 16 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Formalizing and Validating the P-Store Replicated Data Store in Maude*

Peter Csaba Ölveczky

[1] University of Oslo
[2] University of Illinois at Urbana-Champaign

Abstract. P-Store is a well-known partially replicated transactional data store that combines wide-area replication, data partition, some fault tolerance, serializability, and limited use of atomic multicast. In addition, a number of recent data store designs can be seen as extensions of P-Store. This paper describes the formalization and formal analysis of P-Store using the rewriting logic framework Maude. As part of this work, this paper specifies group communication commitment and defines an abstract Maude model of atomic multicast, both of which are key building blocks in many data store designs. Maude model checking analysis uncovered a non-trivial error in P-Store; this paper also formalizes a correction of P-Store whose analysis did not uncover any flaw.

1 Introduction

Large cloud applications—such as Google search, Gmail, Facebook, Dropbox, eBay, online banking, and card payment processing—are expected to be *available* continuously, even under peak load, congestion in parts of the network, server failures, and during scheduled hardware or software upgrades. Such applications also typically manage huge amounts of (potentially important user) data. To achieve the desired availability, the data must be *replicated* across geographically distributed sites, and to achieve the desired scalability and elasticity, the data store may have to be *partitioned* across multiple partitions.

Designing and validating cloud storage systems are hard, as the design must take into account wide-area asynchronous communication, concurrency, and fault tolerance. The use of formal methods during the design and validation of cloud storage systems has therefore been advocated recently [9,11]. In [9], engineers at the world's largest cloud computing provider, Amazon Web Services, describe the use of TLA+ during the development of key parts of Amazon's cloud infrastructure, and conclude that the use of formal methods at Amazon has been a success. They report, for example, that: (i) "formal methods find bugs in system designs that cannot be found though any other technique we know of"; (ii) "formal methods [...] give good return on investment"; (iii) "formal methods are routinely applied to the design of complex real-world software, including public

* This work was partially supported by AFOSR/AFRL Grant FA8750-11-2-0084 and NSF Grant CNS 14-09416.

cloud services"; (iv) formal methods can analyze "extremely rare" combination of events, which the engineer cannot do, as "there are too many scenarios to imagine"; and (v) formal methods allowed Amazon to "devise aggressive optimizations to complex algorithms without sacrificing quality."

This paper describes the application of the rewriting-logic-based Maude language and tool [3] to formally specify and analyze the P-Store data store [14]. P-Store is a well-known partially replicated transactional data store that provides both serializability and some fault tolerance (e.g., transactions can be validated even when some nodes participating in the validation are down).

Members of the University of Illinois Center for Assured Cloud Computing have used Maude to formally specify and analyze complex industrial cloud storage systems such as Google's Megastore and Apache Cassandra [4,8]. Why is formalizing and analyzing P-Store interesting? First, P-Store is a well-known data store design in its own right with many good properties that combines wide-area replication, data partition, some fault tolerance, serializability, and limited use of atomic multicast. Second, a number of recent data store designs can be seen as extensions and variations of P-Store [15,1,2]. Third, it uses atomic multicast to order concurrent transactions. Fourth, it uses "group communication" for atomic commit. The point is that both atomic multicast and group communication commit are key building blocks in cloud storage systems (see, e.g., [2]) that have not been formalized in previous work. Indeed, one of the main contributions of this paper is an abstract Maude model of atomic multicast that allows any possible ordering of message reception consistent with atomic multicast.

I have modeled (both versions of) P-Store, and performed model checking analysis on small system configurations. Maude analysis uncovered some significant errors in the supposedly-verified P-Store algorithm, like read-only transactions never getting validated in certain cases. An author of the original P-Store paper [14] confirmed that I had indeed found a nontrivial mistake in their algorithm and suggested a way of correcting the mistake. Maude analysis of the corrected algorithm did not find any error. I also found that a key assumption was missing from the paper, and that an important definition was very easy to misunderstand because of how it was phrased in English. All this emphasizes the need for a formal specification and formal analysis in addition to the standard prose-and-pseudo-code descriptions and informal correctness proofs.

The rest of the paper is organized as follows. Section 2 gives a background on Maude. Section 3 defines an abstract Maude model of the atomic multicast "communication primitive." Section 4 gives an overview of P-Store. Sections 5 and 6 present the Maude model and the Maude analysis, respectively, of P-Store, and Section 7 describes a corrected version of P-Store. Section 8 discusses some related work, and Section 9 gives some concluding remarks.

Due to space limitations, only parts of the specifications and analyses are given. I refer to the longer report [10] for more details. Furthermore, the executable Maude specifications of P-Store, together with analysis commands, are available at `http://folk.uio.no/peterol/WADT16`.

2 Preliminaries: Maude

Maude [3] is a rewriting-logic-based formal language and simulation and model checking tool. A Maude module specifies a *rewrite theory* $(\Sigma, E \cup A, R)$, 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*, with E a set of possibly conditional equations and membership axioms, and A a set of equational axioms such as associativity, commutativity, and identity. The theory $(\Sigma, E \cup A)$ specifies the system's state space as an algebraic data type.
- R is a set of *labeled conditional rewrite rules*[3] $l : t \longrightarrow t'$ **if** $\bigwedge_{j=1}^{m} u_j = v_j$ specifying the system's local transitions. The rules are universally quantified by the variables in the terms, and are applied *modulo* the equations $E \cup A$.[4]

I briefly summarize the syntax of Maude and refer to [3] 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, and equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. 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 having the form $var{:}sort$. An equation $f(t_1, \ldots, t_n) = t$ with the `owise` ("otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied. 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 is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>` 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`.

The state is a term of the sort `Configuration`, and is 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.

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) .
```

[3] 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.

[4] Operationally, a term is reduced to its E-normal form modulo A before a rewrite rule is applied.

defines a family of transitions 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. 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. Likewise, attributes that are 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, modulo the structural axioms, and the theory being coherent [3]. Maude provides a range of analysis methods, including simulation for prototyping, search for reachability analysis, and LTL model checking. This paper uses Maude's *search* command

`(search` [`[`n`]`] t_0 `=>*` *pattern* [`such that` *cond*] `.)`

which uses a breadth-first strategy to search for at most n states that are reachable from the initial state t_0, match the pattern *pattern* (a term with variables), and satisfy the (optional) condition *cond*. If '`[`n`]`' is omitted, then Maude searches for all solutions. If the arrow '`=>!`' is used instead of '`=>*`', then Maude searches for *final* states; i.e., states that cannot be further rewritten.

3 Atomic Multicast in Maude

Messages that are *atomically multicast* from (possibly) different nodes in a distributed system must be read in (pairwise) the same order: if nodes n_3 and n_4 both receive the atomically multicast messages m_1 and m_2, they must receive (more precisely: "be served") m_1 and m_2 in the same order. Note that m_2 may be read before m_1 even if m_2 is atomically multicast *after* m_1.

Atomic multicast is typically used to order events in a distributed system. In distributed data stores like P-Store, atomic multicast is used to order (possibly conflicting) concurrent transactions: When a node has finished its local execution of a transaction, it atomically multicasts a validation request to other nodes (to check whether the transaction can commit). The validation requests therefore impose an order on concurrent transactions.

Atomic multicast does not necessarily provide a global order of all events. If each of the messages m_1, m_2, and m_3 is atomically multicast to two of the receivers A, B, and C, then A can read m_1 before m_2, B can read m_2 before m_3, and C can read m_3 before m_1. These reads satisfy the *pairwise total order* requirement of atomic multicast, since there is no conflict between any *pair* of receivers. Nevertheless, atomic multicast has failed to globally order the messages m_1, m_2, and m_3. If atomic multicast is used to impose something resembling a global order (e.g., on transactions), it should also satisfy the following *uniform acyclic order* property: the relation $<$ on (atomic-multicast) messages is acyclic, where $m < m'$ holds if there exists a node that reads m before m'.

Atomic multicast is an important concept in distributed systems, and there are a number of well-known algorithms for achieving atomic multicast [5]. To model P-Store, which uses atomic multicast, I could of course formalize a specific algorithm for atomic multicast and include it in a model of P-Store. Such a solution would, however, have a number of disadvantages, including:

1. *Messy non-modular specifications.* Atomic multicast algorithms involve some complexity, including maintaining Lamport clocks during system execution, keeping buffers of received messages that cannot be served, and so on. This solution could also easily yield a messy non-modular specification that fails to separate the specification of P-Store from that of atomic multicast.
2. *Increased state space.* Running a distributed algorithm concurrently with P-Store would also lead to much larger state spaces during model checking analyses, since also the states generated by the rewrites involving the atomic multicast algorithm would contribute to new states.
3. *Lack of generality.* Implementing a particular atomic multicast algorithm might exclude behaviors possible with other algorithms. That would mean that model checking analysis might not cover all possible behaviors of P-Store, but only those possible with the selected atomic multicast algorithm.

I therefore instead define, for each of the two "versions" of atomic multicast, a general atomic multicast primitive, which allows *all possible* ways of reading messages that are consistent with the selected version of atomic multicast. In particular, such a solution will not add states during model checking analysis.

3.1 Atomic Multicast in Maude: "User Interface"

To define an atomic multicast primitive, the system maintains a "table" of read and sent-but-unread atomic-multicast messages for each node. This table must be consulted before reading an atomic-multicast message, to check whether it can be read/served already, and must be updated when the message is read.

The "user interface" of my atomic multicast "primitive" is as follows:

- *Atomically multicasting a message.* A node n that wants to atomically multicast a message m to a set of nodes $\{n_1, \ldots, n_k\}$ just "sends" the "message"

  `atomic-multicast` m `from` n `to` n_1 `...` n_k

  where the message m should be a term of sort `MsgContent`.
- *Reading an atomically multicast message.* A node must check the multicast table whether a given atomic-multicast message can be read. If so, this table must be updated to reflect that the message has been read. A rewrite rule where an object o_1 reads an atomically multicast message m should therefore have the following form, where `AM-TABLE` is a variable of sort `AM-Table`:

```
crl [read-atomically-multicast-m] :
    (msg m from o1 to o2)
    < o2 : ... |  ... >    AM-TABLE
```

```
=>
  < o2 : ... | ... >
  updateAM(m, o2, AM-TABLE)  if okToRead(m, o2, AM-TABLE) .
```

– The user must add the term `[emptyAME]` (denoting the "empty" atomic multicast table) to the initial state.

3.2 Maude Specification of Atomic Multicast

To keep track of atomic-multicast messages sent and received, the table

```
[am-entry(o1, read1, unread1)   ...   am-entry(on, readn, unreadn)]
```

is added to the state. This table contains a record `am-entry(`o_k`,` $read_k$`,` $unread_k$`)` for each object o_k that has been sent an atomically multicast message. $read_k$ is a *list* of atomic-multicast messages read by o_k, in the order in which the messages were read, and $unread_k$ is a *set* of atomic-multicast messages not yet read by o_k.

The "wrapper" used for atomic multicast takes as arguments the message (content), the sender's identifier, and the (identifiers of the) set of receivers:

```
op atomic-multicast_from_to_ : MsgCont Oid OidSet -> Configuration .
```

The equation

```
eq (atomic-multicast MC from O to OS)  [AM-ENTRIES]
 = (distribute MC from O to OS)        [insert(MC, OS, AM-ENTRIES)] .
```

"distributes" such an `atomic-multicast` *msg* `from` o `to` $o_1 \ldots o_n$ message by: (1) "dissolving" the above multicast message into a set of messages

`(msg` *msg* `from` o `to` o_1`)  ...  (msg` *msg* `from` o `to` o_n`)`,

one for each receiver o_k in the set $\{o_1, \ldots, o_n\}$; and (2) by adding, for each receiver o_k, the message (content) *msg* to the set $unread_k$ of unread atomic-multicast messages in the atomic-multicast table.

The `update` function, which updates the atomic-multicast table when `O` reads a message `MC`, just moves `MC` from the set of unread messages to the end of the list of read messages in `O`'s record in the table.

The expression `okToRead(`*mc*`,` *o*`,` *amTable*`)` is used to check whether the object *o* can read the atomic-multicast message *mc* with the given global atomic-multicast table *amTable*. The function `okToRead` is defined differently depending on whether atomic multicast must satisfy the *uniform acyclic order* requirement.

okToRead for Pairwise Total Order Atomic Multicast. The following equations define `okToRead` by first characterizing the cases when the message *cannot* be read; the last equation uses Maude's `owise` construct to specify that the message can be read in all other cases:

```
vars MC MC2 : MsgContent .   vars MCS MCS2 : MsgContSet .
vars MCL MCL2 MCL3 MCL4 : MsgContList .

eq okToRead(MC, O, [am-entry(O, MCL, MCS MC MC2)
                    am-entry(O2, MCL2 :: MC2 :: MCL3 :: MC :: MCL4, MCS2)
                    AM-ENTRIES]) = false .

eq okToRead(MC, O, [am-entry(O, MCL, MCS MC MC2)
                    am-entry(O2, MCL2 :: MC2 :: MCL4, MCS2 MC)
                    AM-ENTRIES]) = false .

eq okToRead(MC, O, [AM-ENTRIES]) = true [owise] .
```

In the first equation, `O` wants to read `MC`, and its AM-entry shows that `O` has not read message `MC2`. However, another object `O2` has already read `MC2` *before* `MC`, which implies that `O` cannot read `MC`. In the second equation some object `O2` has read `MC2` and has `MC` in its sets of unread atomic-multicast messages, which implies that `O` cannot read `MC` yet (it must read `MC2` first).

okToRead for Uniform Acyclic Order Atomic Multicast. To define atomic multicast which satisfies the uniform acyclic order requirement, the above definition must be generalized to consider the induced relation $<$ instead of pairwise reads.

The above definition checks whether a node o can read a message m_1 by checking whether it has some other *unread* message m_2 pending such reading m_1 before m_2 would conflict with the m_1/m_2-reading order of another node. This happens if another node has read m_2 before reading m_1, or if it has read m_2 and has m_1 pending (which implies that eventually, that object would read m_2 before m_1). In the more complex uniform acyclic order setting, that solution must be generalized to check whether reading m_1 before any other pending message m_2 would violate the *current* or the (necessary) *future* "global order." That is, is there some m_1 elsewhere that has been read or must eventually be read *after* m_2 somewhere? If so, node o obviously cannot read m_1 at the moment.

The function `receivedAfter` takes a set of messages and the global AM-table as arguments, and computes the $<^*$-closure of the original set of messages; i.e., the messages that cannot be read before the original set of messages:

```
op receivedAfter : MsgContSet AM-Table -> MsgContSet .

ceq receivedAfter(MC MCS, [am-entry(O2, MCL :: MC :: MC2 :: MCL2, MCS2)
                           AM-ENTRIES])
  = receivedAfter(MC MCS MC2, [am-entry(O2, MCL :: MC :: MC2 :: MCL2, MCS2)
                               AM-ENTRIES])
  if not (MC2 in MCS) .
```

In the above equation, there is a message MC in the current set of messages in the closure. Furthermore, the global atomic-multicast table shows that some node O2 has read MC2 right after reading MC, and MC2 is not yet in the closure. Therefore, MC2 is added to the closure.

In the following equation, there is a message MC in the closure; furthermore, some object O2 has already read MC. This implies that all *unread* messages MCS2 of O2 must eventually be read after MC, and hence they are added to the closure:

```
ceq receivedAfter(MC MCS, [am-entry(O2, MCL2 :: MC :: MCL4, MCS2)
                           AM-ENTRIES])
  = receivedAfter(MC MCS MCS2,
                  [am-entry(O2, MCL2 :: MC :: MCL4, emptyMsgContSet)
                   AM-ENTRIES])     if MCS2 =/= emptyMsgContSet .
```

Finally, the current set is returned when it cannot be extended:

```
eq receivedAfter(MCS, AM-TABLE) = MCS [owise] .
```

The function okToRead can then be defined as expected: O can read the pending message MC if MC is not (forced to be) read after any other pending message (in the set MCS):

```
eq okToRead(MC, O, [am-entry(O, MCL, MCS MC)  AM-ENTRIES])
 = not (MC in receivedAfter(MCS, [am-entry(O, MCL, MCS) AM-ENTRIES])) .
```

I have model-checked both specifications of atomic multicast on a number of scenarios and found no deadlocks or inconsistent multicast read orders.

4 P-Store

P-Store [14] is a partially replicated data store for wide-area networks developed by Schiper, Sutra, and Pedone that provides transactions with serializability. P-Store executes transactions *optimistically*: the execution of a transaction T at site s (which may involve remote reads of data items not replicated at s) proceeds without worrying about conflicting concurrent transactions at other sites. After the transaction T has finished executing, a *certification process* is executed to check whether or not the transaction T was in conflict with a concurrent transaction elsewhere, in which case T might have to be aborted. More precisely, in the certification phase the site s atomically multicasts a request to certify T to all sites storing data accessed by T. These sites then perform a voting procedure to decide whether T can commit or has to be aborted.

P-Store has a number of attractive features: (i) it is a *genuine* protocol: only the sites replicating data items accessed by a transaction T are involved in the certification of T; and (ii) P-Store uses atomic multicast at most once per transaction. Another issue in the certification phase: in principle, the sites certify the transactions in the order in which the certification requests are read. However, if for some reason the certification of the first transaction in a site's certification

queue takes a long time (maybe because other sites involved in the voting are still certifying other transactions), then the certification of the next transaction in line will be delayed accordingly, leading to the dreaded *convoy effect*. P-Store has an "advanced" version that tries to mitigate this problem by allowing a site to start the certification also of other transactions in its certification queue, as long as they are not in a possible conflict with "older" transactions in that queue.

The authors of [14] claim that they have proved the P-Store algorithm correct.

4.1 P-Store in Detail

This section summarizes the description of P-Store in [14].

System Model and Assumptions. A database is a set of triples (k, v, ts), where k is a key, v its value, and ts its time stamp. Each site holds a partial copy of the database, with $Items(s)$ denoting the keys replicated at site s. I do not consider failures in this paper (as failure treatment is not described in the algorithms in [14]). A transaction T is a sequence of read and write operations, and is executed locally at site $proxy(T)$. $Items(T)$ is the set of keys read or written by T; $WReplicas(T)$ and $Replicas(T)$ denote the sites replicating a key written, respectively read or written, by T. A transaction T "is *local* iff for any site s in $Replicas(T)$, $Items(T) \subseteq Items(s)$; otherwise, T is *global*."

Each site ensures *order-preserving serializability* of its local executions of transactions. As already mentioned, P-Store assumes access to an atomic multicast service that guarantees uniform acyclic order.

Executing a Transaction. While a transaction T is executing (at site $proxy(T)$), a read on key k is executed at some site that stores k; k and the item time stamp ts read are stored as a pair (k, ts) in T's *read set* $T.rs$. Every write of value v to key k is stored as a pair (k, v) in T's set of updates $T.up$. If T reads a key that was previously updated by T, the corresponding value in $T.up$ is returned.

When T has finished executing, it can be committed immediately if T is read-only and local. Otherwise, we need to run the certification protocol, which also propagates T's updates to the other (write-) replicating sites.

If the certification process, described next, decides that T can commit, all sites in $WReplicas(T)$ apply T's updates. In any case, $proxy(T)$ is notified about the outcome (commit or abort) of the certification.

Certification Phase. When T is submitted for certification, T is atomically multicast to all sites storing keys read (to check for stale reads) or written (to propagate the updates) by T. When a site s reads such a request, it checks whether the values read by T are up-to-date by comparing their versions against those currently stored in the database. If they are the same, T passes the certification test; otherwise T fails at s.

Algorithm $\mathcal{A}_{ge}$
A Genuine Certification Protocol - Code of site s

```
1: Initialization
2:    Votes ← ∅

3: function ApplyUpdates(T)
4:    foreach ∀(k, v) ∈ T.up : k ∈ Items(s) do
5:       let ts be Version(k, s)
6:       w_T[k, v, ts + 1]                          {write to the database}

7: function Certify(T)
8:    return ∀(k, ts) ∈ T.rs s.t. k ∈ Items(s) : ts = Version(k, s)

9: To submit transaction T                                        {Task 1}
10:   A-MCast(T) to Replicas(T)                  {Executing → Submitted}

11: When receive(VOTE, T.id, vote) from s'                        {Task 2}
12:   Votes ← Votes ∪ (T.id, s', vote)

13: When A-Deliver(T)                                             {Task 3}
14:   if T is local then
15:      if Certify(T) then
16:         ApplyUpdates(T)
17:         commit T                              {Submitted → Committed}
18:      else abort T                               {Submitted → Aborted}
19:   else
20:      if ∃(k, -) ∈ T.rs : k ∈ Items(s) then
21:         Votes ← Votes ∪ (T.id, s, Certify(T))
22:         send(VOTE, T.id, Certify(T)) to all s' in WReplicas(T) s.t.
                                                    s' ∉ group(s)
23:      if s ∈ WReplicas(T) then
24:         wait until ∃VQ ∈ VQ(T) :
                        ∀s' ∈ VQ : (T.id, s', -) ∈ Votes
25:         if ∀s' ∈ VQ : (T.id, s', yes) ∈ Votes then
26:            ApplyUpdates(T)
27:            commit T                           {Submitted → Committed}
28:         else abort T                            {Submitted → Aborted}
29:   if s ∈ WReplicas(T) then send T's outcome to Proxy(T)
```

Fig. 1. The P-Store certification algorithm in [14].

The site s may not replicate all keys read by T and therefore may not be able to certify T. In this case there is a voting phase where each site s replicating keys read by T sends the result of its local certification test to all sites s_w replicating a key written by T. A site s_w can decide on T's outcome when it has received (positive) votes from a *voting quorum* for T, i.e., a set of sites that together replicate all keys read by T. If some site votes "no," the transaction must be aborted. The pseudo-code description of this certification algorithm in [14] is shown in Fig. 1.

As already mentioned, a site does not start the certification of another transaction until it is done certifying the first transaction in its certification queue. To avoid the convoy effect that this can lead to, the paper [14] also describes a version of P-Store where different transactions in a site's certification queue can be certified concurrently as long as they do not read-write conflict.

5 Formalizing P-Store in Maude

I have formalized both versions of P-Store (i.e., with and without sites initiating multiple concurrent certifications) in Maude, and present parts of the formalization of the simpler version. The executable specifications of both versions, with analysis commands, are available at `http://folk.uio.no/peterol/WADT16`, and the longer report [10] provides more detail.

5.1 Class Declarations

Transactions. Although the actual values of keys in the databases are sometimes ignored during analysis of distributed data stores, I choose for purposes of illus-

tration to represent the concrete values of *keys* (or *data items*). This should not add new states that would slow down the model checking analysis.

A transaction (sometimes also called a transaction request) is modeled as an object of the following class `Transaction`:

```
class Transaction | operations : OperationList,   destination : Oid,
                    readSet : ReadSet,            writeSet : WriteSet,
                    status : TransStatus,         localVars : LocalVars .
```

The `operations` attribute denotes the list of read and write operations that remain to be executed. Such an operation is either a *read* operation x `:= read` k, where x is a "local variable" that stores the value of the (data item with) key k read by the operation, or a *write* operation `write(`k`,` *expr*`)`, where *expr* in our case is a simple arithmetic expression involving the transaction's local variables. `waitRemote(`k`,` x`)` is an "internal operation" denoting that the transaction execution is awaiting the value of a key k (to be assigned to the local variable x) which is not replicated by the transaction's proxy. An operation list is a list of such operations, with list concatenation denoted by juxtaposition. `destination` denotes the (identity of the) proxy of the transaction; that is, the site that should execute the transaction. The `readSet` attribute denotes the ','-separated set of pairs `versionRead(`k`,` *version*`)`, each such pair denoting that the transaction has read version *version* of the key k. The `writeSet` attribute denotes the write set of the transaction as a map `(`k_1 `|->` val_1`), ..., (`k_n `|->` val_n`)`. The `status` attribute denotes the commit state of the transaction, which is either `commit`, `abort`, or `undecided`. Finally, `localVars` is a map from the transaction's local variables to their current values.

Replicas. A *replicating site* (or *site* or *replica*) stores parts of the database, executes the transactions for which it is the proxy, and takes part in the certification of other transactions. A replica is formalized as an object instance of the following subclass `Replica`:

```
class Replica | datastore : DataStore,        executing : Configuration,
                submitted : Configuration,    committed : Configuration,
                aborted   : Configuration,        queue : ObjectList .
                transToCertify : CertificationData,
                decidedTranses : TransStatusSet .
```

The `datastore` attribute represents the replica's local database as a set `<` key_1`,` val_1`,` ver_1 `> , ... , <` key_l`,` val_l`,` ver_l `>` of triples `<` key_i`,` val_i`,` ver_i `>` denoting a *version* of the data item with key key_i, value val_i, and version number ver_i.[5] The attributes `executing`, `submitted`, `committed`, and `aborted` denote the transactions executed by the replica and which are/have been, respectively, currently executing, submitted for certification, committed, and aborted. The `queue` holds the certification queue of transactions to be certified by the replica (in collaboration with other replicas). `transToCertify` contains data used for

[5] The paper [14] does not specify whether a replica stores multiple versions of a key.

the certification of the first element in the certification queue (in the simpler algorithm), and `decidedTranses` show the status (aborted/committed) of the transactions that have previously been (partly) certified by the replica.

Clients. Finally, I add an "interface/application layer" to the P-Store specification in the form of *clients* that send transactions to be executed by P-Store:

```
class Client | txns : ObjectList,   pendingTrans : TransIdSet .
```

`txns` denotes the list of transaction (objects) that the client wants P-Store to execute, and `pendingTrans` is either the empty set or (the identity of) the transaction the client has submitted to P-Store but whose execution is not yet finished.

Initial State. The following shows an initial state `init4` (with some parts replaced by '`...`') used in the analysis of P-Store. This system has: two clients, `c1` and `c2`, that want P-Store to execute the two transactions `t1` and `t2`; three replicating sites, `r1`, `r2`, and `r3`; and three data items/keys `x`, `y`, and `z`. Transaction `t1` wants to execute the operations `(xl :=read x) (yl :=read y)` at replica `r1`, while transaction `t2` wants to execute `write(y,5) write(x,8)` at replica `r2`. The initial state also contains the empty atomic multicast table and the table which assigns to each key the sites replicating this key. Initially, the value of each key is `[2]` and its version is `1`. Site `r2` replicates both `x` and `y`.

```
eq init4
 = [emptyAME]
   [replicatingSites(x,r2) ;; replicatingSites(y, (r2,r3))
    ;; replicatingSites(z,r1)]
   < c1 : Client |
         txns : < t1 : Transaction | operations : ((xl :=read x) (yl :=read y)),
                                     destination : r1,    readSet : emptyReadSet,
                                     status : undecided,  writeSet : emptyWriteSet,
                                     localVars : (xl |-> [0] , yl |-> [0]) >,
         pendingTrans : empty >
   < c2 : Client |
         txns : < t2 : Transaction | operations : (write(y, 5) write(x, 8)),
                                     destination : r2,  ... >
         pendingTrans : empty >
   < r1 : Replica | datastore : (< z, [2], 1 >),
                    committed : none,   aborted : none,   executing : none,
                    submitted : none,   queue : emptyTransList,
                    transToCertify : noTrans,   decidedTranses : noTS >
   < r2 : Replica | datastore : ((< x, [2], 1 >) , (< y, [2], 1 >)),  ... >
   < r3 : Replica | datastore : (< y, [2], 1 >),  ... > .
```

5.2 Local Execution of a Transaction

The execution of a transaction has two phases. In the first phase, the transaction is executed locally by its proxy: the transaction performs its reads and writes, but the database is not updated; instead, the reads are recorded in the transaction's read set, and its updates are stored in the `writeSet` attribute.

The second phase is the certification (or validation) phase, when all appropriate nodes together decide whether or not the transaction can be committed or must be aborted. If it can be committed, the replicas update their databases.

This section specifies the first phase, which starts when a client without pending transactions sends its next transaction to its proxy. I do not show the variable declarations (see [10]), but follow the convention that variables are written with (all) capital letters.

```
rl [sendTxn] :
   < C : Client | pendingTrans : empty,
                  txns : < TID : Transaction | destination : RID > ; TXNS >
 =>
   < C : Client | pendingTrans : TID, txns : TXNS >
   (msg executeTrans(< TID : Transaction | >) from C to RID) .
```

P-Store assumes that the local executions of multiple transactions on a site are equivalent to some serialized executions. I model this assumption by executing the transactions one-by-one. Therefore, a replica can only receive a transaction request if its set of currently `executing` transactions is empty (`none`):

```
rl [receiveTxn] :
   (msg executeTrans(< TID : Transaction | >) from C to RID)
   < RID : Replica | executing : none >
 =>
   < RID : Replica | executing :  < TID : Transaction | > > .
```

There are three cases to consider when executing a read operation `X :=read K`: (i) the transaction has already written to key `K`; (ii) the transaction has not written `K` and the proxy replicates `K`; or (iii) the key `K` has not been read and the proxy does not replicate `K`. I only show the specification for case (i). I do not know what version number should be associated to the read, and I choose not to add the item to the read set. (The paper [14] does not describe what to do in this case; the problem disappears if we make the common assumption that a transaction always reads a key before updating it.) As an effect, the local variable `X` gets the value `V`:

```
rl [executeRead1] :
   < RID : Replica | executing :
           < TID : Transaction | operations : (X :=read K) OPLIST,
                                 writeSet : (K |-> V), WS,   localVars : VARS > >
=>
   < RID : Replica | executing :
           < TID : Transaction | operations : OPLIST,
                                 localVars : insert(X, V, VARS) > > .
```

Write operations are easy: evaluate the expression `EXPR` to write and add the update to the transaction's `writeSet`:

```
rl [executeWrite] :
   < RID : Replica | executing :
           < TID : Transaction | operations : write(K, EXPR) OPLIST,
```

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

5.3 Certification Phase

When all the transaction's operations have been executed by the proxy, the proxy's next step is to try to commit the transaction. If the transaction is read-only and *local*, it can be committed directly; otherwise, it must be submitted to the certification protocol.

Some colleagues and I all found the definition of *local* in [14] (and quoted in Section 4) to be quite ambiguous. We thought that "for any site s in $Replicas(T)$, $Items(T) \subseteq Items(s)$" means either "for each site s ..." or that $proxy(T)$ replicates all items in T. The first author of [14], Nicolas Schiper, told me that it actually means "for *some* s ...." In hindsight, we see that this is also a valid interpretation of the definition of *local*. To avoid misunderstanding, it is probably good to avoid the phrase "for any" and use either "for each" or "for some."

If the transaction T cannot be committed immediately, it is submitted for certification by atomically multicasting a certification request—with the transaction's identity `TID`, read set `RS`, and write set `WS`—to all replicas storing keys read or updated by T (lines 9-10 in Fig. 1):

```
crl [commit/submit2] :
    < RID : Replica | executing :
            < TID : Transaction | operations : nil, readSet : RS, writeSet : WS >,
                      submitted : TRANSES >
    REPLICA-TABLE
  =>
    < RID : Replica | executing : none,  submitted : TRANSES  < TID : Transaction | > >
    REPLICA-TABLE
    (atomic-multicast certify(TID, RS, WS) from RID
                  to replicas((keys(RS) , keys(WS)), REPLICA-TABLE))
  if WS =/= emptyWriteSet or not localTrans(keys(RS), REPLICA-TABLE) .
```

According to lines 7–8 in Fig. 1, a replica's local certification succeeds if, for each key in the transaction's read set that is replicated by the replica in question, the transaction read the same version stored by the replica:

```
op certificationOk : ReadSet DataStore -> Bool .
eq certificationOk((versionRead(K, VERSION) , READSET), (< K, V, VERSION2 > , DS))
 =  (VERSION == VERSION2) and certificationOk(READSET, (< K, V, VERSION2 > , DS)) .
eq certificationOk(RS, DS) = true [owise] .
```

If the transaction to certify is not local, the certifying sites must together decide whether or not the transaction can be committed. Each certifying site therefore checks whether the transaction passes the local certification test, and sends the outcome of this test to the other certifying sites (lines 13 and 19–22):

```
crl [certify-nonLocal] :
    (msg certify(TID, RS, WS) from RID2 to RID)
    < RID : Replica | datastore : DS, transToCertify : noTrans, decidedTranses : TSS >
    AM-TABLE      REPLICA-TABLE
  =>
    < RID : Replica | transToCertify : (if LOCAL-CERTIFICATION-OK
                                        then certify(TID, RID2, RS, WS, RID)
                                        else noTrans fi),
                      decidedTranses : (if LOCAL-CERTIFICATION-OK then TSS
                                        else (transStatus(TID, abort) ; TSS) fi) >
    (if intersection(keys(DS), keys(RS)) =/= noKey
     then (distribute
              vote(RID, TID, if LOCAL-CERTIFICATION-OK then commit else abort fi)
           from RID to  (replicas(keys(WS), REPLICA-TABLE)  RID))
     else none fi)
    (if (not LOCAL-CERTIFICATION-OK) and          --- if certification fails ...
         intersection(keys(DS), keys(WS)) =/= noKey        --- write replica ...
     then (msg abort(TID) from RID to RID2) else none fi)  --- notifies proxy
    REPLICA-TABLE
    updateAM(certify(TID, RS, WS), RID, AM-TABLE)
  if okToRead(certify(TID, RS, WS), RID, AM-TABLE)
     /\ not localTrans((keys(RS) , keys(WS)), REPLICA-TABLE)
     /\ LOCAL-CERTIFICATION-OK := certificationOk(RS, DS) .
```

If the local certification fails, the site sends an `abort` vote to the other *write* replicas and also notifies the proxy of the outcome. Otherwise, the site sends a `commit` vote to all other site replicating an item written by the transaction.

The voting phase ends when there is a voting quorum; that is, when the voting sites together replicate all keys read by the transaction. This means that a certifying site must keep track of the votes received during the certification of a transaction. The set of sites from which the site has received a (positive) vote is the fourth parameter of the `certify` record it maintains for each transaction. If a site receives a positive vote, it stores the sender of the vote (lines 11-12). If a site receives a negative vote, it decides the fate of the transaction and notifies the proxy if it replicates an item written by the transaction (lines 28–29).

If a write replica has received positive votes from a voting quorum (lines 23–27 and 29), the transaction can be committed, and the write replica applies the updates and notifies the proxy. The following rule models the behavior when a site has received votes from a voting quorum `RIDS` for transaction `TID`:

```
crl [quorum] :
    < RID : Replica | transToCertify : certify(TID, RID3, RS, WS, RIDS),
                      decidedTranses : TSS,   datastore : DS >
    REPLICA-TABLE
  =>
    < RID : Replica | transToCertify : noTrans,   datastore : applyUpdates(DS, WS),
                      decidedTranses : TSS ; transStatus(TID, commit) >
    REPLICA-TABLE
    (if intersection(keys(DS), keys(WS)) =/= noKey       --- if write replica ...
     then (msg commit(TID) from RID to RID3) else none fi)    --- notify proxy
  if (keys(RS) subset replicatedKeys(RIDS, REPLICA-TABLE)) .  --- voting quorum!
```

Finally, the proxy of transaction `TID` receives the outcome from one or more sites in `TID`'s certification set (the `abort` case is similar):

```
rl [readCommit] :
   (msg commit(TID) from RID2 to RID)
   < RID : Replica | submitted : < TID : Transaction | >,  committed : TRANSES >
 =>
   < RID : Replica | submitted : none,
                     committed : (TRANSES < TID : Transaction | >) >
   done(TID) .      --- notify client
```

6 Formal Analysis of P-Store

In the absence of failures, P-Store is supposed to guarantee *serializability* of the committed transactions, and that a decision (commit/abort) is made on all transactions.

To analyze P-Store, I search for all *final* states—i.e., states that cannot be further rewritten—reachable from a given initial state, and inspect the result. This analysis therefore also discovers undesired deadlocks. In the future, I should instead automatically check serializability, possibly using the techniques in [4], which adds to the state a "serialization graph" that is updated whenever a transaction commits, and then checks whether the graph has cycles.

The search for final states reachable from state `init4` in Section 5.1 yields a state which shows that `t1`'s proxy is not notified about the outcome of the certification (see [10] for details). The problem seems to be line 29 in the algorithm in Fig. 1: only sites replicating items *written* by transaction T ($WReplicas(T)$) send the outcome of the certification to T's proxy. It is therefore not surprising that the outcome of the *read-only* transaction `t1` does not reach `t1`'s proxy.

The transactions in `init4` are local. What about non-local transactions? The initial state `init5` is the same as `init4` in Section 5.1, except that item `y` is only replicated at site `r3`, which means that `t1` and `t2` become non-local transactions.

Searching for final states reachable from `init5` shows a result where the certification process cannot reach a decision on the outcome of transaction `t1`:

```
Maude> (search init5 =>! C:Configuration .)
...
Solution 4
...
< r1 : Replica | submitted :
        < t1 : Transaction | localVars :(xl |->[8], yl |->[5]), operations : nil,
                             readSet : versionRead(x,2), versionRead(y,2), ... >,
                 transToCertify : noTrans >
< r2 : Replica | committed : <t2 : Transaction | writeSet : (x |-> [8], y |-> [5]), ...>,
                 datastore : <x,[8],2>, decidedTranses : transStatus(t2,commit),
                 transToCertify : certify(t1,r1,(versionRead(x,2),versionRead(y,2)),
                                          emptyWriteSet,r2) ,  ... >
< r3 : Replica | aborted : none, committed : none, datastore : < y,[5],2 >,
                 decidedTranses : transStatus(t2,commit),
                 transToCertify : certify(t1, r1, ..., emptyWriteSet, r3) ,  ... >
```

The fate of `t1` is not decided: both `r2` and `r3` are stuck in their certification process. The problem seems to be lines 22 and 23 in the P-Store certification algorithm: why are only *write* replicas involved in sending and receiving votes during the certification? Shouldn't both read and write replicas vote? Otherwise, it is impossible to certify non-local read-only transactions, such as `t1` in `init5`.

7 Fixing P-Store

Nicolas Schiper confirmed that the errors pointed out in Section 6 are indeed errors in P-Store. He also suggested the fix alluded to in Section 6: replace $WReplicas(T)$ with $Replicas(T)$ in lines 22, 23, and 29. The Maude specification of the proposed correction is given in `http://folk.uio.no/peterol/WADT16/`.

Missing Assumptions. One issue seems to remain: why can read-only local transactions be committed without certification? Couldn't such transactions have read stale values? Nicolas Schiper kindly explained that local read-only transactions are handled in a special way (all values are read from the same site and some additional concurrency control is used to ensure serializability), but admitted that this is indeed not mentioned anywhere in their paper. My specifications consider the algorithm as given in [14], without taking the unstated assumptions into account, and also subjects the local read-only transactions to certification.

Analysis of the Updated Specification. I have analyzed the corrected specification on five small initial configurations (3 sites, 3 data items, 2 transactions, 4 operations). All the final states were correct: the committed transactions were indeed serializable.

The Advanced Algorithm. I have also specified and successfully analyzed the (corrected) version of P-Store where multiple transactions can be certified concurrently. It is beyond the scope of this paper to describe that specification.

8 Related Work

Different communication forms/primitives have been defined in Maude, including wireless broadcast that takes into account the geographic location of nodes and the transmission strength/radius [12], as well as wireless broadcast in mobile systems [6]. However, I am not aware of any model of atomic multicast in Maude.

Maude has been applied to a number of industrial and academic cloud storage systems, including Google's Megastore [4], Apache Cassandra [8], and UC Berkeley's RAMP [7]. However, that work did not address issues like atomic multicast and group communication commit.

Lamport's TLA+ has also been used to specify and model check large industrial cloud storage systems like S3 at Amazon [9] and the academic TAPIR transaction protocol targeting large-scale distributed storage systems.

On the validation of P-Store and similar designs, P-Store itself has been proved to be correct using informal "hand proofs" [13]. However, such hand proofs do not generate precise specifications of the systems and tend to be error-prone and rely on missing assumptions, as I show in this paper. I have not found any model checking validation of related designs, such as Jessy [1] and Walter [15].

9 Concluding Remarks

Cloud computing relies on partially replicated wide-area data stores to provide the availability and elasticity required by cloud systems. P-Store is a well-known such data store that uses atomic multicast, group communication commitment, concurrent certification of independent transactions, etc. Furthermore, many other partially replicated data stores are extensions and variations of P-Store.

I have formally specified and analyzed P-Store in Maude. Maude reachability analysis uncovered a numbers of errors in P-Store that were confirmed by one of the P-Store developers: both read and write replicas need to participate in the certification of transactions; write replicas are not enough. I have specified the proposed fix of P-Store, whose Maude analysis did not uncover any error.

Another main contribution of this paper is a general and abstract Maude "primitive" for both variations of atomic multicast.

One important advantage claimed by proponents of formal methods is that even precise-looking informal descriptions tend to be ambiguous and contain missing assumptions. In this paper I have pointed at a concrete case of ambiguity in a precise-looking definition, and at a crucial missing assumption in P-Store.

This work took place in the context of the University of Illinois Center for Assured Cloud Computing, within which we want to identify key building blocks of cloud storage systems, so that they can be built and verified in a modular way by combining such building blocks in different ways. Some of those building blocks are group communication commitment certification and atomic multicast. In more near term, this work should simplify the analysis of other state-of-the-art data stores, such as Walter and Jessy, that can be seen as extensions of P-Store.

The analysis performed was performed using reachability analysis; in the future one should also be able to specify the desired consistency property "directly."

Acknowledgments. I would like to thank Nicolas Schiper for quick and friendly replies to my questions about P-Store, the anonymous reviewers for helpful comments, and Si Liu and José Meseguer for valuable discussions about P-Store and atomic multicast.

References

1. Ardekani, M.S., Sutra, P., Shapiro, M.: Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In: Proc. SRDS'13. IEEE Computer Society (2013)
2. Ardekani, M.S., Sutra, P., Shapiro, M.: G-DUR: a middleware for assembling, analyzing, and improving transactional protocols. In: Middleware'14. ACM (2014)

3. Clavel, M., et al.: All About Maude, LNCS, vol. 4350. Springer (2007)
4. 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)
5. Guerraoui, R., Schiper, A.: Genuine atomic multicast in asynchronous distributed systems. Theoretical Computer Science 254(1-2), 297–316 (2001)
6. Liu, S., Ölveczky, P.C., Meseguer, J.: Modeling and analyzing mobile ad hoc networks in Real-Time Maude. Journal of Logical and Algebraic Methods in Programming 85(1), 34–66 (2016)
7. Liu, S., Ölveczky, P.C., Rahman, M.R., Ganhotra, J., Gupta, I., Meseguer, J.: Formal modeling and analysis of RAMP transaction systems. In: Proc. SAC'16. ACM (2016)
8. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: ICFEM'14. LNCS, vol. 8829. Springer (2014)
9. 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)
10. Ölveczky, P.C.: Formalizing and validating the P-Store replicated data store in Maude, report available at `http://folk.uio.no/peterol/WADT16/`
11. Ölveczky, P.C.: Design and validation of cloud computing data stores using formal methods. In: Proc. International Symposium on Intelligent Systems and Applications (ISA'16) (2016), available at `http://assured-cloud-computing.illinois.edu/publications/`
12. 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)
13. Schiper, N., Sutra, P., Pedone, F.: P-Store: Genuine partial replication in wide area networks. Tech. rep., University of Lugano (2010)
14. Schiper, N., Sutra, P., Pedone, F.: P-Store: Genuine partial replication in wide area networks. In: Proc. SRDS'10. IEEE Computer Society (2010)
15. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proc. SOSP'11. ACM (2011)