Design, Formal Modeling, and Validation of Cloud Storage Systems using Maude

Rakesh Bobba[1], Jon Grov[2], Indranil Gupta[3], Si Liu[3], José Meseguer[3], Peter Csaba Ölveczky[3,4], and Stephen Skeirik[3]

[1] Oregon State University
[2] Gauge AS
[3] University of Illinois at Urbana-Champaign
[4] University of Oslo

Abstract. To deal with large amounts of data while offering high availability, throughput and low latency, cloud computing systems rely on distributed, partitioned, and replicated data stores. Such cloud storage systems are complex software artifacts that are very hard to design and analyze. We argue that formal specification and model checking analysis should significantly improve their design and validation. In particular, we propose rewriting logic and its accompanying Maude tools as a suitable framework for formally specifying and analyzing both the correctness and the performance of cloud storage systems. This chapter largely focuses on how we have used rewriting logic to model and analyze industrial cloud storage systems such as Google's Megastore, Apache Cassandra, Apache ZooKeeper, and RAMP. We also touch on the use of formal methods at Amazon Web Services.

1 Introduction

Cloud computing relies on software systems that store large amounts of data correctly and efficiently. These cloud systems are expected to achieve high performance, defined as high availability and throughput, and low latency. Such performance needs to be *assured* even in the presence of congestion in parts of the network, system or network faults, and scheduled hardware and software upgrades. To achieve this, the data must be *replicated* across both servers within a site, and across geo-distributed sites. To achieve the expected scalability and elasticity of cloud systems, the data may need to be *partitioned*. However, the *CAP theorem* [11] states that it is impossible to have both high availability and strong consistency (correctness) in replicated data stores in today's Internet. Different storage systems therefore offer different tradeoffs between the levels of availability and of consistency that they provide. For example, weak notions of consistency of multiple replicas, such as "eventual consistency," are acceptable for applications, such as social networks and search, where availability and efficiency are key requirements, but where one can tolerate that different replicas store somewhat different versions of the data. Other cloud applications, including online commerce and medical information systems, require stronger consistency guarantees.

The key challenge addressed in this chapter is:

> *How can cloud storage systems be designed with high assurance that they satisfy desired correctness, performance, and quality-of-service requirements?*

1.1 State of the Art

Standard system development and validation techniques are not well suited for addressing the above challenge. Designing cloud storage systems is hard, as the design must take into account wide-area asynchronous communication, concurrency, and fault tolerance. Experimentation with modifications and extensions of an existing system is often impeded by the lack of a precise description at a suitable level of abstraction and by the need to understand and modify large code bases (if available) to test the new design ideas. Furthermore, test-driven system development [32]—where a suite of tests for the planned features are written before development starts, and is used both to give the developer quick feedback during development, and as a set of regression tests when new features are added—has traditionally been considered to be unfeasible for ensuring fault tolerance in complex distributed systems due to the lack of tool support for testing large numbers of different scenarios.

It is also very difficult or impossible to obtain high assurance that the cloud storage system satisfies given correctness and performance requirements using traditional validation methods. Real implementations are costly and error-prone to implement and modify for experimentation purposes. Simulation tool implementations require building an additional artifact that cannot be used for much else. Although system executions and simulations can give an idea of the performance of a design, they cannot give any (quantified) assurance on the performance measures. Furthermore, such implementations cannot verify consistency guarantees: even if we executed the system and analyzed the read/write operations log for consistency violations, that would only cover certain scenarios and cannot guarantee the absence of subtle bugs. In addition, nontrivial fault-tolerant storage systems are too complex for "hand proofs" of key properties based on an informal system description. Even if attempted, such proofs can be error-prone, informal, and usually rely on implicit assumptions.

The inadequacy of current design and verification methods for cloud storage systems in industry has also been pointed out by engineers at Amazon in [34] (see also Section 6). For example, they conclude that "the standard verification techniques in industry are necessary but not sufficient. We routinely use deep design reviews, code reviews, static code analysis, stress testing, and fault-injection testing but still find that subtle bugs can hide in complex concurrent fault-tolerant systems."

1.2 Vision: Formal Methods for Cloud Storage Systems

Our vision is to use *formal methods* to design cloud storage systems and to provide high levels of assurance that their designs satisfy given correctness and performance requirements. In a formally-based system design and analysis methodology, a *mathematical model* S describes the system design at the appropriate level of abstraction. This system specification S should be complemented by a formal *property specification* P that describes mathematically (and therefore precisely) the requirements that the system S should satisfy. Being a mathematical object, the model S can be subjected to mathematical reasoning (preferably fully automated or at least machine-assisted) to guarantee that the design satisfies the properties P. If the mathematical description S is *executable*, then it can be immediately simulated; there is no need to generate an extra artifact for testing and verification. An executable model can also be subjected to various kinds of *model checking* analyses that automatically explore *all possible* system behaviors from a given initial system configuration. From a system developer's perspective, such model checking can be seen as a powerful debugging and testing method that can automatically find subtle "corner case" bugs and that automatically executes a comprehensive "test suite" for complex fault-tolerant systems. We advocate the use of formal methods throughout the design process to quickly and easily explore many design options and to validate designs as early as possible, since errors are increasingly costly the later in the development process they are discovered. Of course, one can also do a *postmortem* formal analysis of an existing system by defining a formal model of it in order to analyze the system formally; we show the usefulness of such *postmortem* analysis in Section 2.

Performance is as important as correctness for storage systems. Some formal frameworks provide probabilistic or statistical model checking that can give performance assurances with a given confidence level.

What properties should a formal framework have in order to be suitable for developing and analyzing cloud storage systems in an industrial setting? In the paper [33], Chris Newcombe of Amazon Web Services, the world's largest cloud computing provider, who has used formal methods during the development of key components of Amazon's cloud computing infrastructure, lists key requirements for formal methods to be used in the development of such cloud computing systems in industry. These requirements can be summarizes as follows:

1. *Expressive languages and powerful tools that can handle very large and complex distributed systems.* Complex distributed systems at different levels of abstraction must be expressible without tedious workarounds of key concepts (such as, e.g., time and different forms of communication). This requirement also includes the ability to express and verify complex liveness properties. In addition to automatic methods that help users diagnose bugs, it is also desirable to be able to machine-check proofs of the most critical parts.
2. *The method must be easy to learn, apply, and remember, and its tools must be easy to use.* The method should have clean simple syntax and semantics, should avoid esoteric concepts, and should use just a few simple language

constructs. The author also recommends against distorting the language to make it more accessible, as the effect would be to obscure what is really going on.

3. *A single method should be effective for a wide range of problems, and should quickly give useful results with minimal training and reasonable effort.* A single method should be useful for many kinds of problems and systems, including data modeling and concurrent algorithms.
4. *Modeling and analyzing performance,* since performance is almost as important as correctness in industry.

1.3 The Rewriting Logic Framework

Satisfying the above requirements is a tall order. We suggest the use of *rewriting logic* [29] and its associated Maude tool [12], and their extensions, as a suitable framework for formally specifying and analyzing cloud storage systems.

In rewriting logic, data types are defined by *algebraic equational specifications.* That is, we declare sorts and function symbols; some function symbols are *constructors* used to define the *values* of the data type; the others denote *defined functions* functions that are defined in a functional programming style using equations. Transitions are defined by *rewrite rules* of the form $t \longrightarrow t'$ if *cond*, where t and t' are terms (possibly containing variables) representing *local state patterns*, and *cond* is a condition. Rewriting logic is particularly suitable for specifying distributed systems in an object-oriented way, in which case the states are multisets of objects and messages (traveling between the objects), and where an object o of class C with attributes att_1 to att_n, having values val_1 to val_n, is represented by a term `<` o `:` C `|` att_1 `:` val_1`, ... ,` att_n `:` val_n `>`. A rewrite 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) .
```

then 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.

Maude [12] is a specification language and high-performance simulation and model checking tool for rewriting logic. *Simulations*—which simulate *single* runs of the system—provide a first quick initial feedback of the design. Maude *reachability analysis*—which checks whether a certain (un)desired state pattern can be reached from the initial state—and *linear temporal logic (LTL) model checking*—which checks whether all possible behaviors from the initial state satisfy a given LTL formula—can be used to analyze all possible behaviors from a given initial configuration.

The Maude tool ecosystem also includes Real-Time Maude [35], which extends Maude to *real-time systems*, and *probabilistic rewrite theories* [3], a specification formalism for specifying distributed systems with probabilistic features. A fully probabilistic subset of such theories can be subjected to statistical model checking analysis using the PVeStA tool [4]. Statistical model checking [42] performs randomized simulations until a probabilistic query can be answered (or the value of an expression be estimated) with the desired statistical confidence.

Rewriting logic and Maude address the above requirements as follows:

1. Rewriting logic is an expressive logic in which a wide range of complex concurrent systems, with different forms of communication and at various levels of abstractions, can be modeled in a natural way. In addition, its real-time extension supports the modeling of real-time systems. The Maude tools have been applied to a range of industrial and state-of-the-art academic systems (see, e.g., [30,38]). Complex system requirements, including safety and liveness properties, can be specified in Maude using *linear temporal logic*, which seems to be the most intuitive and easy-to-understand advanced property specification language for system designers [47]. We can also define functions on states to express nontrivial reachability properties.
2. Equations and rewrite rules: these intuitive notions are all that have to be learned. In addition, object-oriented programming is a well-known programming paradigm, which means that Maude's simple model of concurrent objects should be attractive to designers. We have experienced in other projects that system developers find object-oriented Maude specifications easier to read and understand than their own use case descriptions [36], and that students with no previous formal methods background can easily model and analyze complex distributed systems in Maude (e.g., [24]). The Maude tools provide automatic ("push-button") reachability and temporal logic model checking analysis, and simulation for rapid prototyping.
3. As mentioned, this simple and intuitive formalism has been applied to a wide range of systems, and to all aspects of those systems. For example, data types are modeled as equational specification and dynamic behavior is modeled by rewrite rules. Maude simulations and model checking are easy to use and provide useful feedback automatically: Maude's search and LTL model checking provides a counterexample trace if the desired property does not hold.
4. We show in [37] that randomized Real-Time Maude simulations (of wireless sensor networks) can give performance estimates as good as those of domain-specific simulation tools. More importantly, we can analyze performance measures and provide performance estimations with given confidence levels using probabilistic rewrite theories and statistical model checking; e.g.: "I can claim with 90% confidence that at least 75% of the transactions satisfy the property P." For performance estimation for cloud storage systems see Sections 2–3 and 5.

To summarize, a formal executable specification in Maude or one of its extensions allows us to define *a single artifact* that is, simultaneously, a mathematically precise high-level description of the system design and an executable system model that can be used for rapid prototyping, extensive testing, correctness analysis, and performance estimation.

1.4 Summary: Using Formal Methods on Cloud Storage Systems

In this chapter, we summarize some of the work performed at the Assured Cloud Computing Center at the University of Illinois at Urbana-Champaign using Maude and its extensions to formally specify and analyze the correctness and performance of several important industrial cloud storage systems and a state-of-the-art academic one. In particular, we describe the following contributions:

(i) **Apache Cassandra** [19] is a popular open-source industrial key-value data store that only guarantees *eventual consistency*. We were interested in: (i) evaluating a proposed variation of Cassandra, and (ii) analyzing under what circumstances—and how often in practice—Cassandra also provides stronger consistency guarantees, such as read-your-writes or strong consistency. After studying Cassandra's 345,000 lines of code, we first developed a 1,000-line Maude specification, that captured the main design choices. Standard model checking allowed us to analyze under what conditions Cassandra guarantees strong consistency. By modifying a single function in our Maude model we obtained a model of our proposed optimization. We subjected both of our models to statistical model checking using PVeStA; this analysis indicated that the proposed optimization did *not* improve Cassandra's performance. But how reliable are such formal performance estimates? To investigate this question, we modified the Cassandra code to obtain an implementation of the alternative design, and executed both the original Cassandra code and the new system on representative workloads. These experiments showed that PVeStA statistical model checking provides reliable performance estimates. To the best of our knowledge this was the first time that, for key-value stores, model checking results were checked against a real system deployment, especially on performance-related metrics.

(ii) **Megastore** [8] is a key part of Google's celebrated cloud infrastructure. Megastore's trade-off between consistency and efficiency is to guarantee consistency only for transactions that access a single *entity group*. It is obviously interesting to study such a successful cloud storage system. Furthermore, one of us had an idea on how to extend Megastore so that it would also guarantee strong consistency for certain transactions accessing *multiple* entity groups *without* sacrificing performance. The first challenge was to develop a detailed formal model of Megastore from the short high-level description in [8]. We used Maude simulation and model checking throughout the formalization of this complex system until we obtained a model that satisfied all desired properties. This model also provided the first reasonable detailed public description of Megastore. We then developed a formal model of our extension, and estimated the performance of both systems using randomized simulations in Real-Time Maude; these simulations indicated that Megastore and our extension had about the same performance. (Note that such ad hoc randomized simulations do not give a precise level of confidence in the performance estimates.)

(iii) **RAMP** [7] is a state-of-the-art academic partitioned data store that provides efficient lightweight transactions that guarantee the simple "read atomicity" consistency property. The paper [7] gives hand proofs of correctness properties and proposes a number of variations of RAMP without giving details. We used Maude to: (i) check whether RAMP indeed satisfies the guaranteed properties, and (ii) develop detailed specifications of the different variations of RAMP and check which properties they satisfy.

(iv) **ZooKeeper** [20] is a fault-tolerant distributed key/value data store that provides reliable distributed coordination. In [43] we investigate whether a useful group key management service can be built using ZooKeeper. PVeStA statistical model checking showed that such a ZooKeeper-based service handles faults better than a traditional centralized group key management service, and that it scales to a large number of clients while maintaining low latencies.

To the best of our knowledge, the above-mentioned work at the Assured Cloud Computing Center represents the first published papers on the use of formal methods to model and analyze such a wide swathe of industrial cloud storage systems. Our results are encouraging, but is the use of formal methods feasible in an industrial setting? The recent paper [34] from Amazon tells a story very similar to ours, and formal methods are now a key ingredient in the system development process at Amazon. The Amazon experience is summarized in Section 6, which also discusses the formal framework used at Amazon.

The rest of this chapter is organized as follows: Sections 2 to 5 summarize our work on Cassandra, Megastore, RAMP, and ZooKeeper, respectively, while Section 6 gives an overview of the use of formal methods at Amazon. Section 7 discusses related work, and Section 8 gives some concluding remarks.

2 Apache Cassandra

Apache Cassandra [19] is a popular open-source key-value data store originally developed at Facebook.[5] According to the DB-Engines Ranking [1], Cassandra has advanced into the top 10 most popular database engines among 315 systems, and is currently used by, e.g., Amadeus, Apple, IBM, Netflix, Facebook/Instagram, GitHub, and Twitter.

Cassandra only guarantees *eventual consistency* (if no more writes happen, then eventually all reads will see the *last* value written). However, it might be possible that Cassandra offers stronger consistency guarantees in certain cases.

[5] A key-value store can be seen as a transactional data store where transactions are single read or write operations.

It is therefore interesting to analyze both the circumstances under which Cassandra offers stronger consistency guarantees, and how often stronger consistency properties hold in practice.

The task of accurately predicting when consistency properties hold is nontrivial. To begin with, building a large-scale distributed key-value store is a challenging task. A key-value store usually consists of a large number of components (e.g., membership management, consistent hashing, and so on), and each component is given by source code that embodies many complex design decisions. If a developer wishes to improve the performance of a system (e.g., to improve consistency guarantees, or reduce operation latency) by implementing an alternative design choice for a component, then the only option available was to make changes to huge source code bases (Apache Cassandra has about 345,000 lines of code). Not only does this require many man-months of effort; it also comes with a high risk of introducing new bugs, requires understanding a huge code base before making changes, and is not repeatable. Developers can only afford to explore very few design alternatives, which may in the end fail to lead to a better design.

To be able to reason about Cassandra experiment with alternative design choices and understand their effects on the consistency guarantees and the performance of the system, we have developed in Maude both a formal nondeterministic model [28] and a formal probabilistic model [27] of Cassandra, as well as a model of an alternative Cassandra-like design [27]. To the best of our knowledge, these were the first formal models of Cassandra ever created. Our Maude models include main components of Cassandra such as data partitioning strategies, consistency levels, and timestamp policies for ordering multiple versions of data. Each Maude model consists of about 1000 lines of Maude code with 20 rewrite rules. We use the nondeterministic model to answer *qualitative* consistency queries about Cassandra (e.g., whether a key-value store read operation is strongly (resp. weakly) consistent); and we use the probabilistic model to answer *quantitative* questions like: how often are these stronger consistency properties satisfied in practice?

Apache Cassandra is a distributed, scalable, and highly available NoSQL database design. It is distributed over collaborative servers that appear as a single instance to the end client. Data items are dynamically assigned to several servers in the cluster (called the *ring*), and each server (called a *replica*) is responsible for different ranges of the data stored as key-value pairs. Each key-value pair is stored at multiple replicas to support fault-tolerance. In Cassandra a client can perform *read* or *write* operations to query or update data. When a client requests a read/write operation to a cluster, the server connected to the client acts as a *coordinator* and forwards the request to all replicas that hold copies of the requested key. According to the specified *consistency level* in the operation, after collecting sufficient responses from replicas, the coordinator will reply to the client with a value. Cassandra supports tunable consistency levels, with `ONE`, `QUORUM` and `ALL` being the three major ones, meaning that the coordinator will reply with the most recent value (namely, the value with the highest timestamp) to the client after hearing from *one* replica, a *majority* of the replicas, or *all* replicas, respectively.

We show below one rewrite rule to illustrate our specification style. This rewrite rule describes how the coordinator `S` reacts upon receiving a read reply message `{T, S <- ReadReplySS(ID,KV,CL,A)}` from a replica at global time `T`, with `KV` the returned key-value pair of the form `(key,value,timestamp)`, `ID` and `A` the read operation's and the client's identifiers, respectively; and `CL` the read's consistency level. The coordinator `S` adds `KV` to its local buffer (which stores the replies from the replicas) by `add(ID,KV,BF)`. If the coordinator `S` now has collected the required number of responses (according to the desired consistency level `CL` for the operation), which is determined by the function `cl?`, then the coordinator returns to `A` the highest timestamped value, determined by the function `tb`, by sending the message `[D, A <- ReadReplyCS(ID,tb(BF'))]` to `A`. This outgoing message is equipped with a message delay `D` nondeterministically selected from the delay set `delays`, where `DS` describes the other delays in the set. If the coordinator has not yet received the required number of responses, then no message is sent. (Below, `none` denotes the empty multiset of objects and messages).

```
crl [on-rec-rrep-coord-nondet] :
    {T, S <- ReadReplySS(ID,KV,CL,A)}
    < S : Server | buffer: BF, delays: (D,DS), AS >
 =>
    < S : Server | buffer: BF', delays: (D,DS), AS >
    (if cl?(CL,BF') then
       [D, A <- ReadReplyCS(ID,tb(BF'))]
     else none fi)
 if BF' := add(ID,KV,BF) .
```

We analyze *strong consistency* (where each read returns the value of the last write that occurred before that read), and *eventual consistency* using experimental scenarios with *one* or *multiple* clients. The purpose of our experiments is to answer the following question: does strong/eventual consistency depend on:

– with one client, the combination of consistency levels of consecutive requests?
– with multiple clients, also on the latency between consecutive requests?

Our model checking results show that strong consistency holds in some scenarios, and that eventual consistency holds in all scenarios regardless of the consistency level combinations. Although Cassandra is expected to violate strong consistency under certain conditions, previously there was no formal way of discovering under which conditions such violations could occur.

Table 1 shows the results of model checking strong/eventual consistency with one client, with '×' denoting a violation. Three out of nine combinations of consistency levels violate strong consistency, where at least one of the read and the write operations has a consistency level of `ONE`.

The results of model checking strong consistency with *two* clients is shown in Table 2. We experiment with different relations between the set of possible

Table 1: Results of checking strong consistency (left) and eventual consistency (right) with one client.

Write$_1$ \ Read$_2$	`ONE`	`QUORUM`	`ALL`
`ONE`	×	×	✓
`QUORUM`	×	✓	✓
`ALL`	✓	✓	✓

Write$_1$ \ Write$_2$	`ONE`	`QUORUM`	`ALL`
`ONE`	✓	✓	✓
`QUORUM`	✓	✓	✓
`ALL`	✓	✓	✓

message delays for the coordinator (`D1` and `D2`, with `D1` < `D2`) and the latencies (`L1`, `L2`, and `L3`, with `L1` < `L2` < `L3`) between consecutive requests. We observe that satisfaction of strong consistency depends on the time between requests. For eventual consistency, no violation occurs, regardless of the consistency level and time between requests.

Table 2: Results of checking strong consistency (top) and eventual consistency (bottom) with two clients; the delay set for the coordinator to nondeterministically select a message delay from is {`D1,D2`} with `D1` < `D2`.

Strong

Latency \ Consistency Lv.	`ONE`	`QUORUM`	`ALL`
`L1 (L1<D1)`	×	×	×
`L2 (D1<L2<D2)`	×	×	×
`L3 (D2<L3)`	✓	✓	✓

Eventual

Latency \ Consistency Lv.	`ONE`	`QUORUM`	`ALL`
`L1 (L1<D1)`	✓	✓	✓
`L2 (D1<L2<D2)`	✓	✓	✓
`L3 (D2<L3)`	✓	✓	✓

We then wanted to experiment with a possible optimization of the Cassandra design in which the *values* of the keys are considered instead of their *timestamps* when deciding which value should be returned upon a read operation. Our goal was to see whether that would provide better consistency or at least lower operation latency. Having a formal specification of Cassandra, we were able to easily specify this possible optimization by just modifying the function `tb`.

To estimate how often Cassandra satisfies stronger consistency *in practice*, and to compare the original Cassandra design with our proposed optimization, we transformed our nondeterministic specification into a *fully probabilistic* rewrite theory that can be subjected to statistical model checking using the PVeStA tool [4]. The main idea behind turning a nondeterministic model into a fully probabilistic model is to let the message delays be sampled from a dense probabilistic distribution. Density implies that the probability that two messages arrive at the same time, and hence that two events happen at the same time, is zero.

To illustrate this transformation from a nondeterministic model to a probabilistic one, we show below the probabilistic rewrite rule obtained by transforming the above rewrite rule. In the transformed rule, the message delay `D` of the generated `ReadReply` message is now probabilistically chosen according to the parameterized probability distribution function `distr(...)`, which in our model was selected to be a lognormal distribution:

```
crl [on-rec-rrep-coord-prob] :
    {T, S <- ReadReplySS(ID,KV,CL,A)}
    < S : Server | buffer: BF, AS >
 =>
    < S : Server | buffer: BF', AS >
    (if cl?(CL,BF') then
     [D , A <- ReadReplyCS(ID,tb(BF'))]
     else none fi)
 if BF' := add(ID,KV,BF)
 with probability D := distr(...)  .
```

We quantitatively analyzed the following five consistency guarantees in Cassandra and in our alternative design:

– *Strong consistency* (SC) ensures that each read returns the value of the last write that occurred before that read.
– *Read your writes* (RYW) guarantees that the effects of all writes performed by a client are visible to that client's subsequent reads.
– *Monotonic reads* (MR) ensure that a client observes a key-value store increasingly up to date over time.
– *Consistent prefix* (CP) guarantees that a client will observe an ordered sequence of writes starting with the first write to the system.
– *Causal consistency* (CC) guarantees that effects are observed only after their causes: reads will not see a write unless its dependencies are also seen.

Our PVeStA analysis indicated that Cassandra frequently achieves much higher consistency (up to strong consistency) than the promised eventual consistency, especially with `QUORUM` and `ALL` reads, in which cases the probability of achieving strong consistency starts to approach 1 even with fairly short latencies (see the left plot in Fig. 1).

Another interesting observation comes from the PVeStA analysis of consistent prefix. By fixing the consistency level of writes (see the three left-hand side plots in Fig. 2 for `ONE`/`QUORUM`/ALL write), we can see how the consistency level of reads affect consistent prefix over issuing latency. Surprisingly, it appears that lower reads can achieve higher consistent prefix consistency.

Our analysis shows that our alternative design does not outperform the original Cassandra design in terms of the consistency models we considered, except for *consistent prefix*, even though the alternative design's behavior is quite close to that of the original one in most cases.

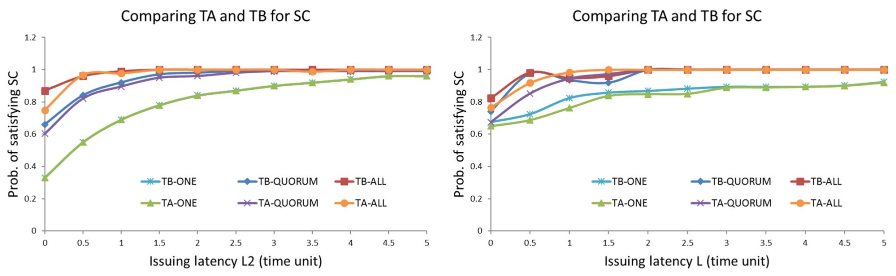

Fig. 1: PVeStA-based estimation (left) and actual measures running Cassandra (right) of the probability of satisfying strong consistency (SC) as a function of the time between a read request and the latest previous write request. TA stands for *time-agnostic strategy*, our proposed alternative design, while TB stands for *timestamp-based strategy*, the original Cassandra design.

To investigate whether such PVeStA-based analysis really provides realistic performance estimates, we also executed the real Cassandra system on representative workloads (see the right hand side plots in Figs. 1 and 2). Both the model predictions and the implementation-based evaluations reached the same conclusion. We could show that results derived from model checking agree reasonably well with experimental reality. We showed that this agreement holds true for various consistency models, even if changes are introduced on how timestamps are used in responding to queries. For more details on our quantitative analysis of Cassandra and of the alternative design we considered, we refer the reader to the journal paper [26], which substantially extends our conference paper [27].

3 Formalizing, Analyzing, and Extending Google's Megastore

Megastore [8] is a distributed data store developed and widely applied at Google. It is a key component of Google's celebrated cloud computing infrastructure, and is used internally at Google for Gmail, Google+, Android Market, and Google AppEngine. It is one of a few industrial replicated data stores that provide both data replication, fault tolerance, and support for transactions with some data consistency guarantees. By 2011, Megastore handled 3 billion write and 20 billion read transactions per day [8].

In Megastore, data are divided into different *entity groups* (such as, e.g., "Peter's email" or "books on rewriting logic"). Google Megastore's tradeoff among high availability, concurrency, and consistency is that data consistency is only guaranteed for transactions that access a *single* entity group. There are no guarantees if a transaction reads *multiple* entity groups.

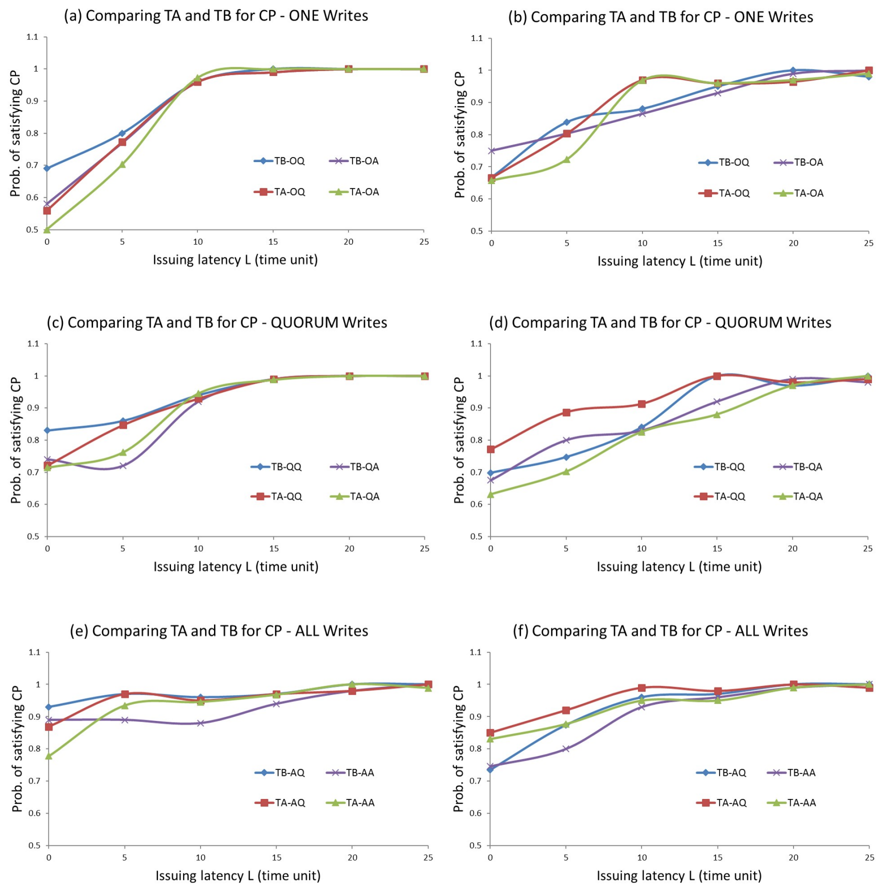

Fig. 2: Probabilities of satisfying consistent prefix (CP) by statistical model checking (left) and by real deployment run (right). TA stands for Time-agnostic Strategy, our proposed alternative design, while TB for Timestamp-based Strategy, the original Cassandra design.

One of us, Jon Grov, a researcher on transactional systems with no background in formal methods, had some ideas about how to add consistency also for certain transactions that access multiple entity groups, *without* significant performance penalty. Grov was therefore interested in experimenting with his ideas to extend Megastore, to analyze the correctness of his extension, and to compare its performance with that of Megastore.

There was, however, a problem: there was no detailed description of Megastore, or publicly available code for this industrial system that could be used for experimentation. The only publicly available description of Megastore was a brief overview paper [8]. To fully understand Megastore (more precisely, the Megastore algorithm/approach), two of us, Grov and Ölveczky, first had to develop our own sufficiently detailed executable specification of Megastore from the description in [8]. This specification could then be used to estimate the performance of Megastore, which could then be compared with the estimated performance of our extension. We employed Real-Time Maude simulations and, in particular, LTL model checking throughout our development effort.

Specifying Megastore. Our specification of Megastore [15] is the first publicly available formalization and reasonably detailed description of Megastore. It contains 56 rewrite rules, of which 37 deal with fault tolerance features. An important point is that even *if* we had had access to Megastore's code base, understanding and extending it would likely have been much more time-consuming than developing our own 56-rule description and simulation model in Real-Time Maude.

To show an example of the specification style, the following shows one (of the smallest) of the 56 rewrite rules:

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

In this rule, the `Site` object `SID` has a number of transaction objects to execute in its `localTransactions` attribute; one of them is the transaction `TID` (the variable `LOCALTRANS` ranges over multisets of objects, and therefore captures all the other objects in the site's transaction set). The next operation that the transaction `TID` should execute is the write operation `write(KEY, VAL)`, which represents writing the value `VAL` to the key `KEY`. The effect of applying this rewrite rule is that the write operation is removed from the transaction's list of operations to perform, and is added to its *write set* `writes`.

The second rewrite rule we show concern the validation phase. In Megastore, a replicated transaction log is maintained for each entity group. When a transaction t is ready to commit, its coordinating site s prepares a new log entry for each entity group written by t, does some leader election part of Paxos, and (if the leader election phase is successful) multicasts the new proposed log entry to the other replicating sites. Each recipient of this message must then verify that it has not already granted an accept for the new log position. If so, the recipient replies with an accept message to the originating site.

The following rule shows the part when the replicating site `THIS` receives the multicasted message `acceptAllReq` with the proposed new log entry `(TID' LP SID OL)` for entity group `EG` from the coordinator `THIS`. The site `THIS` verifies that it has not already granted an accept for that 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 `acceptAllRsp(...)`, and stores its accept in its `proposals` attribute. The record `(TID' LP SID OL)` represents the candidate log entry, which contains 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(msg acceptAllRsp(TID, EG, LP, PROPNUM) from THIS to SENDER, T)
   if not (containsLPos(LP, LEL) or hasAcceptedForPosition(LP, PROPSET))
      /\ T ; TS := possibleMessageDelays(THIS, SENDER) .
```

Our model assumes that `possibleMessageDelays(`s_1`,`s_2`)` gives the set d_1 `;` d_2 `;` $\cdots$ `;` d_n of possible messaging delays between sites s_1 and s_2, where the union operator `_;_` is associative and commutative. The matching equation

```
  T ; TS := possibleMessageDelays(THIS, SENDER)
```

then nondeterministically assigns to the variable `T` (of sort `Time`) *any* delay d_i from the set `possibleMessageDelays(THIS, SENDER)` of possible delays, and uses this value as the messaging delay when sending the message

```
  dly(msg acceptAllRsp(TID, EG, LP, PROPNUM) from THIS to SENDER, T).
```

Analyzing Megastore. We wanted to analyze both the correctness and the performance of our model of Megastore throughout its development, to catch functional errors and performance bottlenecks quickly. For our analysis, we generated two additional models from the main model described above, as follows:

1. Our model of Megastore is a real-time model. However, this means that any exhaustive model checking analysis of our model only analyzes those behaviors that are possible *within the given timing parameters* (for messaging

delays, etc.). To exhaustively analyze all possible system behaviors *irrespective* of particular timing parameters, we generated an *untimed* model from our real-time model by just ignoring messaging delays in our specification. For example, in this specification, the above rewrite rule becomes (where the parts represented by '...' are as before):

```
crl [rcvAcceptAllReq] :
    (msg acceptAllReq(...) from SENDER to THIS)
    < THIS : Site | entityGroups : EGROUPS  < EG : EntityGroup | ... > >
   =>
    < THIS : Site | entityGroups :  EGROUPS  < EG : EntityGroup | ... > >
    (msg acceptAllRsp(TID, EG, LP, PROPNUM) from THIS to SENDER)
   if not (containsLPos(LP, LEL) or hasAcceptedForPosition(LP, PROPSET)) .
```

2. For *performance estimation* purposes we also defined a real-time model in which certain parameters, such as the messaging delays between two nodes, are selected probabilistically according to a given probability distribution. For example, we used the following probability distribution for the network delays (in milliseconds):

	30%	30%	30%	10%
London $\leftrightarrow$ Paris	10	15	20	50
London $\leftrightarrow$ New York	30	35	40	100
Paris $\leftrightarrow$ New York	30	35	40	100

An important difference between our Megastore work and the Cassandra effort described in Section 2 is that in the Cassandra work we used (fully) probabilistic rewrite theories and the dedicated statistical model checker PVeStA for the probabilistic analysis. In contrast, our Megastore work stayed within Real-Time Maude and simulated the selection of a value from a distribution by using an ever-changing seed and Maude's built-in function `random`. In this "lightweight" probabilistic real-time model, we maintain the value n of the seed in an object `< seed : Seed | value :` n `>` in the state, and the above rewrite rule is transformed to

```
crl [rcvAcceptAllReq] :
    (msg acceptAllReq(...) from SENDER to THIS)
    < THIS : Site | entityGroups : EGROUPS  < EG : EntityGroup | ... > >
    < seed : Seed | value : N >
   =>
    < THIS : Site | entityGroups :  EGROUPS  < EG : EntityGroup | ... > >
    < seed : Seed | value : N + 1 >
    dly(msg acceptAllRsp(...) from THIS to SENDER,
          selectFromDistribution(delayDistribution(THIS, SENDER), random(N)))
   if not (containsLPos(LP, LEL) or hasAcceptedForPosition(LP, PROPSET)) .
```

where `selectFromDistribution` "picks" the appropriate delay value from the distribution `delayDistribution(THIS, SENDER)` using the random number `random(N)`. While such lightweight probabilistic performance analysis using Real-Time Maude has previously been shown to give performance estimates on par with those of dedicated domain-specific simulation tools, e.g., for wireless sensor networks [37], such analysis cannot provide a (quantified) level of confidence in the estimates; for that we need statistical model checking.

Performance Estimation. For performance estimation, we also added a transaction generator that generates transaction requests at random times, with an adjustable average rate measured in *transactions per second*, and used the above probability distribution for the network delays.

The key performance metrics to analyze are the average transaction latency, and the number of committed/aborted transactions. We also added a *fault injector* that randomly injects short outages in the sites, with mean time to failure 10 seconds, and mean time to repair 2 seconds for each site. The results from the randomized simulations of our probabilistic real-time model for 200 seconds, with an average of 2.5 transactions generated per second, in this fairly challenging setting, are given in the following table:[6]

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

These latency figures are consistent with Megastore itself [8]: "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."

Model Checking Analysis. For model checking analysis—which exhaustively analyzes all possible system behaviors from a given initial system configuration—we added to the state a *serialization graph*, which is updated whenever a transaction commits. The combination of properties we analyzed was that: (i) the serialization graph never contains cycles; and that eventually (ii) all transactions have finished executing; and (iii) all entity groups and all transaction logs are equal (or invalid). Model checking these properties in combination can be done by giving the following Real-Time Maude command:

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

This commands returns `true` if the temporal logic formula (the last three lines in the command) holds from the initial state `initMegastore`, and a counterexample showing a behavior that does satisfy the formula in case the formula does not hold for all behaviors.

[6] Because of site failures, not all the generated 500 transactions were committed or aborted; however, our analysis shows that all 500 transactions are validated when there are no site failures. See [15] for details.

We used model checking throughout the development of our model, and discovered many unexpected corner cases. To give an idea of the size of the configurations that can be model checked, we summarize below the execution time of the above model checking command for different system parameters, where $\{n_1, \ldots, n_k\}$ means that the corresponding value was selected nondeterministically from the set. All the model checking commands that finished executing returned `true`. *DNF* means that the execution was aborted after 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

As mentioned, we also model checked an untimed model (of the non-fault-tolerant part of Megastore) that covers all possible behaviors from an initial system configuration irrespective of timing parameters. The disadvantage of such untimed model checking is that we can only analyze smaller systems: model checking the untimed system proved unfeasible even for four transactions. Furthermore, failure detection and fault tolerance features rely heavily on timing, which means that the untimed model is not suitable for modeling and analyzing the fault-tolerant version of Megastore.

Megastore-CGC. As mentioned, Jon Grov had some ideas on how to extend Megastore to also provide consistency for transactions that access *multiple* entity groups, while maintaining Megastore's performance and strong fault tolerance features. He observed that, in Megastore, a site replicating a set of entity groups participates in all updates of these entity groups, and should therefore be able to maintain an ordering on those updates. The idea behind our extension, called Megastore-CGC, is that by making this ordering explicit, such an "ordering site" can validate transactions [16].

Since Megastore-CGC exploits the implicit ordering of updates during Megastore commits, it *piggybacks* ordering and validation onto Megastore's commit protocol. Megastore-CGC therefore does not require additional messages for validation and commit, and should maintain Megastore's performance and strong fault tolerance. A *failover* protocol deals with failures of the ordering sites.

We again used both simulations (to discover performance bottlenecks) and Maude model checking extensively during the development of Megastore-CGC, whose formalization contains 72 rewrite rules. The following table compares the performance of Megastore and Megastore-CGC in a setting without failures, where sites London and New York also handle transactions accessing multiple entity groups. Notice that Megastore-CGC will abort some transactions that access multiple entity groups ("validation aborts") that Megastore does not care about:

	Megastore			**Megastore-CGC**			
	Commits	Aborts	Avg.latency	Commits	Aborts	Validation aborts	Avg.latency
Paris	652	152	126	660	144	0	123
London	704	100	118	674	115	15	118
New York	640	172	151	631	171	10	150

Designing and validating a sophisticated protocol like Megastore-CGC is very challenging. Maude's intuitive and expressive formalism allowed a domain expert to define both a precise, formal description and an executable prototype in a single artifact. We found that anticipating all possible behaviors of Megastore-CGC is impossible. A similar observation was made by Google's Megastore team, which implemented a pseudo-random test framework, and state that *"the tests have found many surprising problems"* [8]. Compared to such a testing framework, Real-Time Maude model checking analyzes not only a set of pseudo-random behaviors, but all possible behaviors from an initial system configuration. Furthermore, we believe that Maude provides a more effective and low-overhead approach to testing than that of a real testing environment.

In a test-driven development method, a suite of tests for the planned features are written before development starts. This set of tests is then used both to give the developer quick feedback during development, and as a set of regression tests when new features are added. However, test-driven development has traditionally been considered to be unfeasible when targeting fault tolerance in complex concurrent systems, due to the lack of tool support for testing a large number of different scenarios. Our experience with Megastore-CGC showed that with Maude a test-driven approach is possible also in such systems, since many complex scenarios can be quickly tested by model checking.

Simulating and model checking this prototype automatically provided quick feedback about both the performance and the correctness of different design choices, even for very complex scenarios. Model checking was especially helpful, both to verify properties and to find subtle "corner case" design errors that were not found during extensive simulations.

4 RAMP Transaction Systems

Read-Atomic Multi-Partition (RAMP) transactions were proposed by Peter Bailis *et al.* [7] to offer light-weight multi-partition transactions that guarantee one of the fundamental consistency levels, namely, *read atomicity*: either all updates or no updates of a transaction are visible to other transactions. To guarantee that either all partitions perform a transaction successfully or none do, RAMP performs two-phase writes by using the two-phase commit protocol (2PC): In the *prepare* phase, each timestamped write is sent to its partition, which adds the write to its local database. In the *commit* phase, each partition updates an index that contains the highest-timestamped committed version of each item.

The paper [7] presents a pseudo-code description of the RAMP algorithms and "hand proofs" of key properties. It also mentions a number of optimizations and variations of RAMP, but without providing any details or correctness arguments. There was therefore a clear need for formally specifying and analyzing RAMP and its proposed extensions. Providing correctness in the protocol and its extensions is a critical step towards making RAMP a production-capable system.

We have therefore formalized RAMP and its variations in Maude. We have modeled and analyzed the effect of the following RAMP building blocks: fast commit, one-phase write, and 2PC.

To show an example of the specification, the following rewrite rule illustrates what happens when a client `O` receives a `prepared` message for the current write `ID` from a partition `O'`: it deletes `ID` from the set `NS` of pending `prepared` messages for writes; and if the resulting set `NS'` is empty, meaning that all `prepared` messages have been received, the client starts committing the transaction using the function `startCommit`, which generates a `commit` message for each write:

```
crl [receive-prepared] :
    msg prepared(ID) from O' to O
    < O : Client | pendingPrep : NS, pendingOps : OI, sqn : SQN >
=>
    < O : Client | pendingPrep : NS' >
    (if NS' == empty then startCommit(OI,SQN,O) else none fi)
if NS' := delete(ID,NS) .
```

We used reachability analysis to analyze whether RAMP and its variants satisfy all the following properties (from [7]) that RAMP transactions should satisfy:

– *Read atomic isolation*: either all updates or no updates of a transaction are visible to other transactions.
– *Companions present*: if a version is committed, then each of the version's sibling versions are present on their respective partitions.
– *Synchronization independence*: one client's transactions cannot cause another client's one to block, and if a client can contact the partition responsible for each item in its transaction, then the transaction will eventually commit (or abort of its own volition).
– *Read your writes*: a client's writes are visible to her subsequent reads.

We analyzed those properties for our seven versions of RAMP, for all initial configurations with four operations and two clients, as well as for a number of configurations with six operations. Our model checking results agree with the proved and conjectured results in [7], i.e., all versions satisfy the above properties, except that:

– RAMP without 2PC does not satisfy read atomicity, "companions present" and read-your-writes; and
– RAMP with one-phase writes does not satisfy read-your-writes.

5 Group Key Management via ZooKeeper

Group key management is the management of cryptographic keys so that multiple authorized entities can securely communicate with each other. A central group key controller can fulfill this need by: (a) authenticating/admitting authorized users into the group, and (b) generating a *group key* and distributing it to authorized group members [49]. The group key needs to be updated periodically to ensure the secrecy of the group key, and whenever a new member joins or leaves the group to preserve *backward secrecy* and *forward secrecy* respectively. In particular, the group has some secure mechanism to designate a *key encrypting key (KEK)* which the group controller then uses to securely distribute group key updates. In settings with a centralized group controller, its failure can impact both group dynamics and periodic key updates, leaving the group key vulnerable. If the failure occurs during a key update, then the group might be left in an inconsistent state where both the updated and old keys are still in use. This is especially significant when designing a cloud-based group key management service, since such a service will likely manage many groups. [7]

In [43] we investigated whether a fault-tolerant cloud-based group key management service could be built by leveraging existing coordination services commonly available in cloud infrastructures and if so, how to design such a system. In particular, we: (a) designed a group key management service built using Zookeeper [20], a reliable distributed coordination service supporting Internet-scale distributed applications, (b) developed a rewriting logic model of our design in Maude [12], based on [17], where key generation is handled by a centralized key management server and key distribution is offloaded to a ZooKeeper cluster and where the group controller stores its state in ZooKeeper to enable quick recovery from failure, and (c) analyzed our model using the PVeStA [4] statistical model checking tool. The analysis centered on two key questions: (1) can a ZooKeeper-based group key management service handle faults more reliably than a traditional centralized group key manager, and (2) can it scale to a large number of concurrent clients with a low enough latency to be useful?

Zookeeper Background. ZooKeeper is used by many distributed applications such as Apache Hadoop MapReduce, and Apache HBase, and by many companies including Yahoo and Zynga in their infrastructures. From a bird's eye view, the ZooKeeper system consists of two kinds of entities: servers and clients. All of the servers together form a distributed and fault-tolerant key/value store which the clients may read data from or write data to. In ZooKeeper terminology, each key/value pair is called a *znode*, and these znodes are then organized in a tree structure similar to that of a standard filesystem. In order to achieve fault-tolerance, ZooKeeper requires that a majority of servers acknowledge and log each write to disk before it is committed. Since in order to operate, ZooKeeper requires a majority of servers to be alive and aware of each other, updates will

[7] Please refer to [40] for a survey of group key management schemes.

not be lost. A detailed description of ZooKeeper design and features can be found in [20] and in ZooKeeper documentation.[8]

ZooKeeper provides a set of guarantees that are useful for key management applications. Updates to ZooKeeper state are atomic, and once committed they will persist until they are overwritten. Thus, a client will have a consistent view of the system regardless of which server it connects to and is guaranteed to be up-to-date within a certain time-bound. For our purposes, this means updates, once committed, will not be lost. Furthermore, ZooKeeper provides an event-driven notification mechanism through *watches*. A client that sets a *watch* on a znode which will be notified whenever the znode is changed or deleted. Watches can enable us, for example, to easily propagate key change events to interested clients.

System Design. In our design, if a user wishes to join/leave a group, she will contact the group controller who will: (a) perform the necessary authentication and authorization checks, (b) if the user is authorized, add/remove the user to/from the group, (c) update the group membership state in the ZooKeeper service, (d) generate and update the group key in the ZooKeeper service, and (e) provide any new users with the updated group key and necessary information to connect to ZooKeeper and obtain future key updates.

The system state is stored in ZooKeeper as follows: each secure group and authorized user is assigned a unique znode storing an encrypted key. Whenever a user joins a group, the znode corresponding to that user is added as a child of the znode representing the group. The znode corresponding to the group also stores the current group key encrypted by the KEK. During periodic group key updates, the old group key is overwritten by the new group key (encrypted by the same KEK). However, when an authorized user joins or leaves a group, the group controller generates a *new* KEK and distributes it using the client's pairwise keys. Specifically, the group controller updates the value stored at each user's assigned znode with the new KEK encrypted by the client's the pairwise key. Since each group member sets a watch on the znode corresponding to itself and its group, it will be notified whenever the group key or the KEK changes.

Thus, the group controller uses the ZooKeeper service to maintain group information and to distribute and update cryptographic keys. Since the group controller's operational state is already saved within the ZooKeeper service (e.g. group member znodes it has updated, last key update time, etc.) if the controller were to fail a back-up controller could take over with minimal downtime.

Maude Model. The distributed system state in our model is a multiset structure, populated by objects, messages, and the scheduler. Messages are state fragments passed from one object to another marked with a timestamp, an address, and a payload. The scheduler's purpose is to deliver messages to an object at the time indicated by the message's timestamp and to provide a total ordering of messages in case some have identical timestamps. ZooKeeper system state is modeled

[8] `http://zookeeper.apache.org/doc/trunk/`

by three classes of objects: the ZooKeeper *service*, *servers*, and *clients*. The group key management system also consists of three different classes of objects: the group key *service*, *managers*, and *clients*. All these objects are designed to operate according to the system design discussed above, but we must be careful to clarify a few details.

To simplify the model, we abstracted away many complex—but externally *invisible*—details of ZooKeeper. We also say a few words about failure modes. We assume that both ZooKeeper servers and group key managers may fail. When a ZooKeeper server fails, it will no longer respond to any messages. After a variable repair timeout, the server will come back online. When any connected clients discover the failure, they will attempt to migrate to another randomly chosen server. Similarly, when a key manager fails, any client waiting on that key manager will time out and try again. A key manager buffers clients' requests and answers them in order. Before answering each request, it saves information about the requested operation to ZooKeeper. If a manager dies, the succeeding manager loads the stored state from ZooKeeper and completes any pending operations before responding to new requests. So that our model would accurately reflect the performance of a real ZooKeeper cluster, we chose parameters that agree with the data gathered in [20]. In particular, we set the total latency of a read or write request to ZooKeeper to under $2ms$ on average (which corresponds to a ZooKeeper system under *full* load as in [20]). Note that this average time only occurs in practice if there is no server failure; our model permits servers to fail. We also assume that all of the ZooKeeper servers are located in the same data center, which is the recommended way to deploy a ZooKeeper cluster.

Here we show an example Maude rewrite rule from our specification.

```
crl [ZKSERVER-PROCESS] :
    < A : ZKServer | leader: L, live: true, txid: Z, store: S,
                     clients: C, requests: R, updates: U, AS >
    {T, A <- process}
  =>
    < A : ZKServer | leader: L, live: true, txid: Z', store: S',
                     clients: C, requests: null, updates: null, AS >
      M M'
if (Z',S',M) := commit(C,U,S) /\ M' := process(L,R,S') .
```

This rule, specified in Maude, illustrates how a live ZooKeeper server object processes a batch of updates (service internal messages) and requests from clients during a processing cycle. The condition of this rule invokes two auxiliary functions, `commit` and `process`, which repeatedly commit key updates and process requests from connected clients. When this rule completes, the server will have a new transaction id and key/value store, an empty update and request list, and a set of messages to be sent `M M'` where `M` are notifications to interested clients that keys were updated and `M'` are client requests forwarded onto to the ZooKeeper leader.

Analysis and Discussion. Our analysis consisted of two experiments. Both were run hundreds of times via PVeStA and average results were collected. The first experiment was designed to test whether saving snapshots of the group key manager's state in the ZooKeeper store could increase the overall reliability of the system. In the experiment, we set the failure rate for the key manager to 50% per experiment, the time to failover from one server to another to 5 seconds, and the experiment duration to 50 seconds. We then compared the average key manager availability (i.e., the time it is available to distribute keys to clients) between a single key manager and two key managers where they share a common state saved in the ZooKeeper store. In former case case, the average availability was 32.3 seconds whereas in the latter case it was 42.31 seconds. This represents an availability improvement from 65% to 85%. Of course, we expect a system with replicated state to be more reliable as there is no longer a single point of failure.

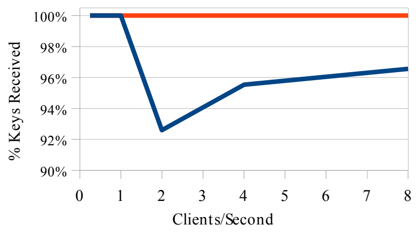

Fig. 3: % Keys vs. join rate

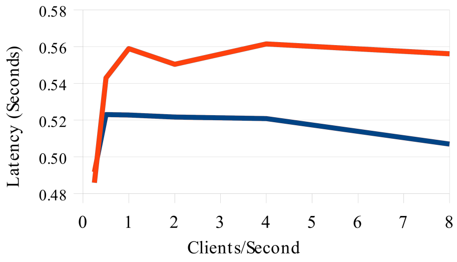

Fig. 4: Latency vs. join rate

Our second experiment was designed to examine whether using ZooKeeper to distribute shared keys is efficient and scalable enough for real-world use. The experiment measured the variations in: (a) the percentage of keys successfully received by group members, and (b) the key distribution latency, as increasing numbers of clients joined a group per second. The percentage of keys successfully received is defined as the total number of keys expected to be received by any client over its lifetime versus the amount actually received; and distribution latency is the average time measured from when a key is generated by the key manager until that key is received by a client. We sampled these parameters at client join rates of once every 4, 2, 1, 0.5, 0.25, and 0.125 second(s). We kept the same duration as in the first experiment, but picked failure probabilities such that system will have 99.99% availability and specified link latency between the ZooKeeper service and ZooKeeper clients to vary according to a uniform distribution on the interval from .05 to .25 seconds.

We present our results in Figure 3 and Figure 4, where the blue line corresponds to our initial experiments while the red line corresponds to a slightly modified model where we added a 2 second wait time between key updates from the key manager. While our initial experiments show that naively using ZooKeeper as a key distribution agent works well, at high client join rates, the key reception rate seems to level out around 96%. This occurs because ZooKeeper can apply key updates internally more quickly then clients can download them; after all, the ZooKeeper servers enjoy a high-speed intra-cloud connection. By adding extra latency between key updates, the ZooKeeper servers are forced to wait enough time for the correct keys to propagate to clients. As shown in Figure 2, this change achieves a 99% key reception in all cases. On the other hand, key distribution latency remained relatively constant, at around half a second, regardless of the join rate because ZooKeeper can distribute keys at a much higher rate than a key manager can update them [20]. Of course, the artificial latency added in the second round of experiments has a cost; it increases the time required for a client to join or leave the group by the additional wait time.

In essence, our analysis confirmed that a scalable and fault-tolerant key-management service can indeed be built using ZooKeeper, settling various doubts raised about the effectiveness of ZooKeeper for key management by an earlier, but considerably less-detailed, model and analysis [13]. This result is not particularly surprising, especially considering that many man-hours would be needed to optimize an actual system. More interestingly, the analysis also showed that system designs may suffer from performance bottlenecks not readily apparent in the original description—highlighting the power of formal modeling and analysis as a method to explore the design space.

6 How Amazon Web Services Uses Formal Methods

The previous sections have made the case for the use of formal methods in general, and rewriting logic and Maude in particular, during the design and development of cloud computing storage systems. The reported work was conducted in academic settings. How about industry?

In 2015, engineers at Amazon Web Services (AWS) published a paper entitled "How Amazon Web Services Uses Formal Methods" [34]. AWS is probably the world's largest provider of cloud computing services, with more than a million customers and almost $10 billion in revenue in 2015, and is now more profitable than Amazon's North American retail business [45]. Key components of its cloud computing infrastructure include the DynamoDB highly available replicated database and the Simple Storage System (S3), which stores more than two trillion objects and handles more than 1.1 million requests per second [34].

The developers at AWS have used formal specification and model checking extensively since 2011. This section summarizes their use of formal methods and their reported experiences.

Use of Formal Methods. The AWS developers used the approach that we advocate in this chapter: the use of an intuitive, expressive, and executable (sort of) specification language together with model checking for automatic push-button exploration of all possible behaviors

More precisely, they used Leslie Lamport's specification formalism TLA+ [21], and its associated model checker TLC. The language is based on set theory, and has the usual logic and temporal logic operators. In TLA+ a transition T is defined as a logical axiom relating the "current value" of a variable x with the *next* value x' of the variable. For example, a transition T that increases the value of x by one and the value of *sum* with x can be defined as $T == x' = x + 1 \wedge \mathit{sum}' = \mathit{sum} + x$. The model checker TLC can analyze invariants and generate random behaviors.

Formal methods were applied on different components of S3, DynamoDB, and other components, and a number of subtle bugs were found.

Outcomes and Experiences. The experience of the AWS engineers was remarkably similar to that generally advocated by the formal methods community. The quotations below are all from [34].

Model checking finds "corner case" bugs that would be hard to find with standard industrial methods:

- "We have found that standard verification techniques in industry are necessary but not sufficient. We routinely use deep design reviews, static code analysis, stress testing, and fault-injection testing but still find that subtle bugs can hide in complex fault-tolerant systems."
- "T.R. learned TLA+ and wrote a detailed specification of [components of DynamoDB] in a couple of weeks. [...] the model checker found a bug that could lead to losing data if a particular sequence of failures and recovery steps would be interleaved with other processing. This was a very subtle bug; the shortest error trace exhibiting the bug included 35 high-level steps. [...] The bug had passed unnoticed through extensive design reviews, code reviews, and testing."

A formal specification is a valuable precise description of an algorithm:

- "There are at least two major benefits to writing precise design: the author is forced to think more clearly, helping eliminating "hand waving," and tools can be applied to check for errors in the design, even while it is being written. In contrast, conventional design documents consist of prose, static diagrams, and perhaps pseudo-code in an ad hoc untestable language."
- "Talk and design documents can be ambiguous or incomplete, and the executable code is much too large to absorb quickly and might not precisely reflect the intended design. In contrast, a formal specification is precise, short, and can be explored and experimented on with tools."
- "We had been able to capture the essence of a design in a few hundred lines of precise description."

Formal methods are surprisingly feasible for mainstream software development and give good return on investment:

- "In industry, formal methods have a reputation for requiring a huge amount of training and effort to verify a tiny piece of relatively straightforward code, so the return on investment is justified only in safety-critical domains (such as medical systems and avionics). Our experience with TLA+ shows this perception to be wrong. [...] Amazon engineers have used TLA+ on 10 large complex real-world systems. In each, TLA+ has added significant value. [...] Amazon now has seven teams using TLA+, with encouragement from senior management and technical leadership. Engineers from entry level to principal have been able to learn TLA+ from scratch and get useful results in two to three weeks."

– “Using TLA+ in place of traditional proof writing would thus likely have improved time to market, in addition to achieving greater confidence in the system’s correctness.”

Quick and easy to experiment with different design choices:

– “We have been able to make innovative performance optimizations [...] we would not have dared to do without having model-checked those changes. A precise, testable description of a system becomes a what-if tool for designs.”

The paper’s conclusions include the following:

> “Formal methods are a big success at AWS, helping us to prevent subtle but serious bugs from reaching production, bugs we would not have found using other techniques. They have helped us devise aggressive optimizations to complex algorithms without sacrificing quality. [...] seven Amazon teams have used TLA+, all finding value in doing so [...] Using TLA+ will improve both time-to-market and quality of our systems. Executive management actively encourages teams to write TLA+ specs for new features and other significant design changes. In annual planning, managers now allocate engineering time to TLA+.”

Limitations. The authors point out that there are two main classes of problems with large distributed systems: bugs and performance degradation when some components slow down, leading to unacceptable response times from a user’s perspective. While TLA+ was effective to find bugs, it had one significant limitation in that it was not, or could not be, used to analyze performance degradation.

Why TLA+? We have advocated using the Maude framework for the specification and analysis of cloud computing storage systems. What are the differences between Maude and its toolset and TLA+ and its model checker? And why did the Amazon engineers use TLA+ instead of, e.g., Maude?

On the specification side, Maude supports hierarchical system states, object-oriented specifications, with dynamic object creation deletion, subclasses, and so on, as well as the ability to specify any computable data type as an equational specification. These features do not seem to be supported by TLA+. Maude also has a clear separation between the system specification and the property specification, whereas TLA+ uses the same language for both parts. Hence, the fact that a system S satisfies its property P can be written in TLA+ as the logical implication $S \to P$.

Perhaps the most important difference is that real-time systems can be modeled and analyzed in Real-Time Maude, and that probabilistic rewrite theories can be statistically model checked using PVeStA, whereas TLA+ seems to lack support for the specification and analysis of real-time and probabilistic systems. (Lamport argues that special treatment of real-time systems is not needed: just add a system variable *clock* that denotes the current time [2].) The lack of support for real-time and probabilistic analysis probably explains why the TLA+

engineers could only use TLA+ and TLC for correctness analysis but *not* for performance analysis, whereas we have shown that the Maude framework can be used for both aspects.

So, why did the Amazon engineers choose TLA+? The main reason seems to be that TLA+ was developed by one of the most prominent researcher in distributed systems, Leslie Lamport, whose algorithms (like the many versions of Paxos) are key components in today's cloud computing systems:

> "C.N. eventually stumbled on a language [...] when he found a TLA+ specification in the appendix of a paper on a canonical algorithm in our problem domain—the Paxos consensus algorithm. The fact that TLA+ was created by the designer of such a widely used algorithm gave us confidence that TLA+ would work for real-world systems."

Indeed, it seems that they did not explore too many formal frameworks:

> "When we found [that] TLA+ met those requirements, we stopped evaluating methods."

7 Related Work

Regarding related work on Cassandra, on the model-based performance estimation side, Osman and Piazzola use queueing Petri nets (an extension of colored stochastic Petri nets) to study the performance of a single Cassandra node on realistic workloads. They also compare their model-based predictions with actual running times of Cassandra. The main difference between our work and this and other work on model-based performance estimation [14,9] is that we do both functional correctness analysis and model-based performance estimation.

We discuss the use of formal methods at Amazon [34,33] in Section 6, and note that their approach is very similar to ours, except that they do not use their models also for performance estimation. In the same vein, the designers of the TAPIR transaction protocol targeting large-scale distributed storage systems have specified and model checked correctness properties of their system design using TLA+ [52,51].

Instead of developing and analyzing high-level formal models to quickly analyze different design choices and finding bugs early, other approaches [22,50] use distributed model checkers to model check the *implementation* of cloud systems such as Cassandra and ZooKeeper, as well as the BerkeleyDB database and a replication protocol implementation. This method can discover implementation bugs as well as protocol-level bugs.

Verifying both protocols and code is the goal of the IronFleet framework at Microsoft Research [18]. Their methodology combines TLA+ analysis to reason about protocol-level concurrency (while ignoring implementation complexities) and Floyd-Hoare-style imperative verification to reason about implementation complexities (while ignoring concurrency). Their verification methodology includes a wide range of methods and tools, including SMT solving, and requires "considerable assistance from the developer" to perform the proofs, as well as writing the code in a new verification-friendly language instead of a standard implementation language. To illustrate their method's applicability, they built and proved the correctness of a Paxos-based replicated-state-machine library and a sharded key-value store.

Whereas most of the work discussed in this chapter employs model checking, the Verdi framework [48] focuses on specifying, implementing and verifying distributed systems using the higher-order theorem prover Coq. The Verdi framework provides libraries and a toolchain for writing distributed systems (in Coq) and verifying them. The Verdi methodology has been used to mechanically check correctness proofs for the Raft consensus algorithm [39], a back-up replication system, and a key-value data store. Much like our approach, their executable Coq "implementations" can be seen as executable high-level formal models/specifications, an approach that eliminates the "formality gap between the model and the implementation." The key difference is that Verdi relies on theorem proving, which requires nontrivial user interaction, whereas model checking is automatic. Model checking and theorem proving are somewhat orthogonal analysis methods. On the one hand, model checking only verifies the system for single initial configurations, whereas theorem proving can prove the model correct for all initial configurations. On the other hand, model checking can find subtle bugs automatically, whereas theorem proving is not a good method for finding bugs. (Failure of a proof attempt does not imply that there is a bug in the system, only that the particular proof attempt did not work.)

Coq is also used in the Chapar framework which aims to verify the causal consistency property for key-value store implementations [23].

Finally, in [10], Bouajjani et al. study the problem of verifying the *eventual consistency* property (guaranteed by, e.g., Cassandra) for optimistic replication systems (ORSs) in general. They show that verifying eventual consistency for such systems can be reduced to an LTL model checking problem. For this purpose, they formalize both the eventual consistency property and, in particular, ORSs, and characterize the classes of ORSs for which the problem of verifying eventual consistency is decidable.

8 Concluding Remarks

We have proposed rewriting logic, with its extensions and tools, as a suitable framework for formally specifying and analyzing both the correctness and the performance of cloud storage systems. Rewriting logic is a simple and intuitive yet expressive formalism for specifying distributed systems in an object-oriented way. The Maude tool supports both simulation for rapid prototyping and automatic "push-button" model checking exploration of all possible behaviors from a given initial system configuration. Such model checking can be seen as an exhaustive search for "corner case" bugs, or as a way to automatically execute a more comprehensive "test suite" than is possible in standard test-driven system development. Furthermore, PVeStA-based statistical model checking can provide assurance about quantitative properties measuring various performance and quality of service behavior of a design with a given confidence level, and Real-Time Maude supports model checking analysis of real-time distributed systems.

We have used Maude and Real-Time Maude to develop quite detailed formal models of a range of industrial cloud storage systems (Apache Cassandra, Megastore, and Zookeeper) and an academic one (RAMP), and have also designed and formalized significant extensions of these systems (a variant of Cassandra, Megastore-CGC, a key management system on top of ZooKeeper, and variations of RAMP) and have provided assurance that they satisfy desired correctness properties; we have also analyzed their performance. Furthermore, in the case of Cassandra, we compared the performance estimates provided by PVeStA analysis with the performance actually observed when running the real Cassandra code on representative workloads; they differed only by 10-15%.

We have also summarized the experience of developers at Amazon Web Services, who have successfully used formal specification and model checking with TLA+ and TLC during the design of critical cloud computing services such as the DynamoDB database and the Simple Storage System. Their experience showed that formal methods: (i) can feasibly be mastered quickly by engineers, (ii) reduce time-to-market and improves product quality; (ii) are cost-effective; and (iv) discover bugs not discovered during traditional system development. Their main complaint was that they could not use formal methods analyze the performance of a design (since TLA+ lacks the support for analyzing real-time and probabilistic systems).

We believe that the Maude formalism should be at least as easy to master as TLA+ (as our experience with Megastore indicates), and might be even more convenient; furthermore, as shown in this chapter, the Maude tools can also be used to analyze the performance of a system.

8.1 The Future

Much work remains ahead. To begin with, one current limitation is that explicit-state model checking explores the behaviors starting from a *single* initial system configuration, and hence—since this can only be done for a finite number of states—it cannot be used to verify that an algorithm is correct for all possible initial system configurations, which can be infinite.

A *first* approach to obtain a more complete coverage is illustrated by [15,25], where we have extended coverage by model checking all initial system configurations up to n operations, m replicas, and so on. However, naïvely generating initial states can yield a large number of symmetric initial states; greater coverage and symmetry reduction methods should therefore be explored. A *second*, more powerful approach is to use *symbolic model checking* techniques, where a possibly infinite number of initial states is described symbolically by formulas in a theory whose satisfiability is decidable by an SMT solver. In the Maude context, this form of symbolic model checking is supported by *rewriting modulo*

SMT [41], which has already been applied to various distributed real-time systems, and by *narrowing-based symbolic model checking* [5,6]. An obvious next step is to verify properties of cloud storage systems for possibly infinite sets of initial states using this kind of symbolic model checking. A *third* approach is to take to heart the complementary nature of model checking and theorem proving, so that these methods do help each other when used *in combination*, as explained below.

In the approach we have presented, *design exploration* based on formal executable specification comes first. Once a promising design has been identified, fairly exhaustive *model checking* debugging and verification of such a design can be carried out using: (i) LTL explicit-state model checking; (ii) statistical model checking of quantitative properties; and (iii) symbolic model checking from possibly infinite sets of initial configurations for greater assurance. Since all these methods are *automatic*, this is relatively easy to do based on the system's formal executable specification with rewrite rules. Only after we are fairly sure that we have obtained a good design and have eliminated many subtle errors by these automatic methods, does it become cost-effective to attempt a more labor-intensive *deductive verification* through theorem proving of properties not yet fully verified. Indeed, some properties may already have been *fully verified* in an automatic way by symbolic model checking, so that only some additional properties not expressible within the symbolic model checking framework need to be verified deductively. Two compelling reasons for such deductive verification are: (i) the *safety-critical* nature of a system or a system component; and (iii) the *high reusability* of a component or algorithm, such as, for example, Paxos or Raft, so that the deductive verification effort becomes amortized over many uses.

In the context of rewriting logic verification, a simple logic to deductively verify reachability properties of a distributed system specified as a rewrite theory has been recently developed, namely, *constructor-based reachability logic* [44], which itself extends and applies to rewrite theories the original reachability logic in [46]. In fact, a number of distributed algorithms have already been proved correct in [44]; and we plan to soon extend and apply the Maude reachability logic tool to verify specific properties of cloud storage systems.

Of course, high reusability of designs is not just a good thing for amortizing verification efforts: it is a good thing in all respects. By breaking distributed system designs into modular, well-understood, and highly reusable components that come with strong correctness guarantees, as advocated by the notion of *formal pattern* in [31], unprecedented levels of assurance and of software quality could be achieved for cloud storage systems in the near future. It is for this reason that, from the beginning of our research on the application of formal methods to gain high assurance for cloud-based systems, we have tried to understand cloud systems designs as suitable compositions of more basic components providing key functionality in the form of generic distributed algorithms. This is still work in progress. Our medium-term goal is to develop a library of generic components formally specified and verified in Maude as formal patterns in the above sense.

Out of these generic components, different system designs for key-value stores, and for cloud storage systems supporting transactions could then be naturally and easily obtained as suitable compositions.

Acknowledgments. We thank Jatin Ganhotra, Muntasir Raihan Rahman, and Son Nguyen for their contributions to the work on Cassandra reported in this chapter. This work is based on research sponsored by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This work is also based on research supported by the National Science Foundation under Grant Nos. NSF CNS 1409416 and NSF CNS 1319527.

References

1. DB-Engines (2016), `http://db-engines.com/en/ranking`
2. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Transactions on Programming Languages and Systems 16(5), 1543–1571 (1994)
3. Agha, G.A., Meseguer, J., Sen, K.: PMaude: Rewrite-based specification language for probabilistic object systems. Electr. Notes Theor. Comput. Sci. 153(2), 213–239 (2006)
4. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: Proc. CALCO'11, LNCS, vol. 6859. Springer (2011)
5. Bae, K., Meseguer, J.: Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In: Rewriting Techniques and Applications (RTA'13). LIPIcs, vol. 21, pp. 81–96. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2013)
6. Bae, K., Meseguer, J.: Infinite-state model checking of LTLR formulas using narrowing. In: WRLA'14. LNCS, vol. 8663. Springer (2014)
7. Bailis, P., Fekete, A., Hellerstein, J.M., Ghodsi, A., Stoica, I.: Scalable atomic visibility with RAMP transactions. In: Proc. SIGMOD'14. ACM (2014)
8. Baker, J., et al.: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR'11. www.cidrdb.org (2011)
9. Barbierato, E., Gribaudo, M., Iacono, M.: Performance evaluation of NoSQL big-data applications using multi-formalism models. Future Generation Comp. Syst. 37, 345–353 (2014)
10. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: Proc. 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14). ACM (2014)
11. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proc. PODC'00. ACM (2000)
12. Clavel, M., et al.: All About Maude, LNCS, vol. 4350. Springer (2007)
13. Eckhart, J.: Security Analysis in Cloud Computing using Rewriting Logic. Master's thesis, Ludwig-Maximilans-Universität München (2012)
14. Gandini, A., Gribaudo, M., Knottenbelt, W.J., Osman, R., Piazzolla, P.: Performance evaluation of NoSQL databases. In: EPEW (2014)
15. 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)

16. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: Proc. SEFM'14. LNCS, vol. 8702. Springer (2014)
17. Gupta, J.: Available group key management for NASPInet. Master's thesis, Univeristy of Illinois at Champaign-Urbana (2011)
18. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: IronFleet: proving practical distributed systems correct. In: Proc. 25th Symposium on Operating Systems Principles (SOSP'15). ACM (2015)
19. Hewitt, E.: Cassandra: The Definitive Guide. O'Reilly Media (2010)
20. Hunt, P., Konar, M., Junqueira, F., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX ATC. vol. 10 (2010)
21. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
22. Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J.F., Gunawi, H.S.: SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14). USENIX Association (2014)
23. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: certified causally consistent distributed key-value stores. In: Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16). ACM (2016)
24. Lien, E., Ölveczky, P.C.: Formal modeling and analysis of an IETF multicast protocol. In: Proc. SEFM'09. IEEE Computer Society (2009)
25. 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)
26. Liu, S., Ganhotra, J., Rahman, M.R., Nguyen, S., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. LITES 3(2), 02:1–02:26 (2016)
27. Liu, S., Nguyen, S., Ganhotra, J., Rahman, M.R., Gupta, I., Meseguer, J.: Quantitative analysis of consistency in NoSQL key-value stores. In: Proc. QEST'15. LNCS, vol. 9259. Springer (2015)
28. Liu, S., Rahman, M.R., Skeirik, S., Gupta, I., Meseguer, J.: Formal modeling and analysis of Cassandra in Maude. In: Proc. ICFEM'14. LNCS, vol. 8829. Springer (2014)
29. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
30. Meseguer, J.: Twenty years of rewriting logic. Journal of Logic and Algebraic Programming 81(7-8), 721–781 (2012)
31. Meseguer, J.: Taming distributed system complexity through formal patterns. Science of Computer Programming 83, 3–34 (2014)
32. Munir, H., Moayyed, M., Petersen, K.: Considering rigor and relevance when evaluating test driven development: A systematic review. Inform. Softw. Techn. (2014)
33. Newcombe, C.: Why Amazon chose TLA+. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ'14). Lecture Notes in Computer Science, vol. 8477. Springer (2014)
34. 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)
35. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)

36. 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)
37. 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)
38. Ölveczky, P.C.: Real-Time Maude and its applications. In: Proc. WRLA'14. LNCS, vol. 8663. Springer (2014)
39. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014. pp. 305–319. USENIX Association (2014)
40. Rafaeli, S., Hutchison, D.: A survey of key management for secure group communication. ACM Comput. Surv. 35(3), 309–329 (2003)
41. Rocha, C., Meseguer, J., Muñoz, C.A.: Rewriting modulo SMT and open system analysis. J. Log. Algebr. Meth. Program. 86(1), 269–297 (2017)
42. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Proc. CAV'05. LNCS, vol. 3576. Springer (2005)
43. Skeirik, S., Bobba, R.B., Meseguer, J.: Formal analysis of fault-tolerant group key management using ZooKeeper. In: 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid'13). IEEE Computer Society (2013)
44. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. Tech. rep., University of Illinois Computer Science Department (March 2017), available at : `http://hdl.handle.net/2142/95770`
45. Statt, N.: Amazon's earnings soar as its hardware takes the spotlight. In The Verge, April 28, 2016, `http://www.theverge.com/2016/4/28/11530336/amazon-q1-first-quarter-2016-earnings`, accessed May 29, 2016
46. Stefanescu, A., Ciobâcă, S., Mereuta, R., Moore, B.M., Serbanuta, T., Rosu, G.: All-path reachability logic. In: Proc. RTA-TLCA 2014. LNCS, vol. 8560. Springer (2014)
47. Vardi, M.Y.: Branching vs. linear time: Final showdown. In: Proc. TACAS'01. LNCS, vol. 2031. Springer (2001)
48. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.E.: Verdi: a framework for implementing and formally verifying distributed systems. In: Proc. 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). ACM (2015)
49. Wong, C.K., Gouda, M.G., Lam, S.S.: Secure group communications using key graphs. IEEE/ACM Trans. Netw. 8(1), 16–30 (2000)
50. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: transparent model checking of unmodified distributed systems. In: Proc. 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09). pp. 213–228. USENIX Association (2009)
51. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. Tech. Rep. UW-CSE-2014-12-01 v2, University of Washington (2014), `https://syslab.cs.washington.edu/papers/tapir-tr14.pdf`
52. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.K.: Building consistent transactions with inconsistent replication. In: Proc. Symposium on Operating Systems Principles, (SOSP'15). ACM (2015)