Design and Validation of Cloud Computing Data Stores using Formal Methods*

Peter Csaba Ölveczky

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

Abstract. To deal with large amounts of data and to ensure high availability, cloud computing systems rely on distributed, partitioned, and replicated data stores. However, such data stores are complex artifacts that are very hard to design and analyze. I argue that formal specification and model checking should be beneficial during their design and validation. In particular, I propose rewriting logic and its associated tools as a suitable framework for formally specifying and analyzing both the correctness and the performance of such data stores. I also give an overview of the use of rewriting logic at the University of Illinois' Assured Cloud Computing center on industrial data stores such as Google's Megastore and Apache Cassandra, and summarize the experiences of the use of a different formal method by engineers at Amazon Web Services.

1 Introduction

Cloud computing services and applications—such as Google search, Gmail, Facebook, Dropbox, eBay, online banking and card payment processing—are expected to be *available* continuously, even under peak load, congestion in parts of the network, and during scheduled hardware or software upgrades. Cloud computing services also typically manage large amounts of data: In 2014, Facebook stored more than 300 petabytes of data and users uploaded 350 million photos every day. To achieve the desired availability, the data must be *replicated* across geographically distributed sites, and to achieve the desired scalability and elasticity, the data store may have to be *partitioned* across multiple partitions.

Traditional databases offer strong notions of correctness, such as ACID properties, when clients access data through transactions. However, the *CAP theorem* [6] states that it is impossible to have both high availability and strong consistency (correctness) in a replicated data stores in today's Internet. Weak notions of consistency of multiple replicas, such as "eventual consistency," is perfectly acceptable for applications such as Facebook and Google search, where it is crucially important that the application is always available and deliver results quickly, and where we can easily tolerate that different replicas store

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

somewhat different versions of the appropriate data. However, other cloud computing applications, such as online banking, online commerce (eBay and airplane reservation systems), and medical information systems may need transactions with stronger consistency guarantees. There is therefore now a plethora of replicated/partitioned data stores which support transactions with different consistency guarantees. Examples of transactional data stores are Google's Megastore [9], Facebook/Apache's Cassandra [11], Google's Spanner [8], UC Berkeley's RAMP [4], TAPIR, DrTM, COPS, and so on. Examples of consistency guarantees include snapshot isolation, prefix consistency, causal consistency, update atomic, and read atomic consistency. The paper [27] gives an excellent introduction to some of these consistency guarantees.

Designing and evaluating replicated/partitioned data stores are challenging tasks. Their design must take account wide-area asynchronous communication, concurrency, and fault tolerance. Instead of designing a data store from scratch, one may want to extend or optimize existing data stores. However, this is hindered by the lack of precise descriptions at a suitable level of abstraction; for example, the only way to understand Cassandra is to study its 345,000 lines of code, and there was only a short informal overview paper of Megastore available.

Evaluating the consequence of different design choices is difficult or impossible using traditional system development methods: real implementations or implementations in dedicated simulation tools are typically needed to analyze the performance of the system. Real implementations are obviously 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 such implementations can give an estimate of the performance of a design, they are almost useless to verify consistency guarantees: even if we execute the system and analyze the read/write operations log for consistency violations, this would cover certain scenarios and cannot guarantee the absence of subtle bugs. Likewise, hand-proofs of an informal description is error-prone, informal, and tend to rely on key implicit assumptions.

In this paper, I argue for the use of executable *formal methods* to: (i) quickly develop new data store designs, and (ii) to quickly analyze the effect of different design choices on both correctness and performance. In such a formal system development and analysis methodology, we define a formal executable specification S of the system; this defines the design at the desired level of abstraction as a precise mathematical object that makes assumptions precise and explicit and that, being mathematical, is amenable to mathematical correctness analysis. In particular, such a specification would give us *both* a precise description of the design *and* an executable model that can be simulated and further analyzed *for free*. Such a system description S should be complemented by a formal *property specification* P that describes mathematically (and therefore precisely and unambiguously) the consistency guarantees that the system should satisfy. Using model checking, we can then *automatically* check whether all possible behaviors—from a given initial system configuration—of the system S satisfy the desired consistency guarantee P. This technique can be seen as an

exhaustive analysis of all possible behaviors in order to "systematically" search for "corner-case bugs." From a system developer's perspective, model checking can be considered as a very powerful "testing methodology" that allows us to run more "test scenarios" in one shot than would be possible to define in standard test-based development.

Performance is as important as correctness for data stores. Can formal methods also be used for performance estimation? Indeed, some formal tools for real-time and probabilistic systems can be used to simulate and statistically evaluate also the performance of our designs.

Cloud computing data stores are a challenging domain for formal methods. Section 3 mentions some desired properties of a formal framework for specifying and analyzing such systems, and argues that rewriting logic and its associated Maude tools are a suitable formal specification and analysis framework for specifying and analyzing such complex system designs.

To the best of my knowledge, the first published papers on formally specifying and analyzing state-of-the-art industrial cloud computing data stores were written by me and my colleagues at the Assured Cloud Computing University Center of Excellence (ACC) at the University of Illinois at Urbana-Champaign (see the related work sections in [9,10,17,16]). In this paper, I motivate the use of formal methods in general, and rewriting logic and Maude in particular, during the design and validation of cloud computing data stores, from a software engineering perspective (Sections 2 and 3). I then give an overview of some work done at the ACC, including formalizing, extending, and analyzing Google's Megastore and Facebook/Apache's Cassandra data stores (Sections 4-6), and mention on-going work and future challenges in this domain (Sections 6 and 8).

Our results are encouraging, but are formal methods feasible also in an industrial setting? A recent paper from the world's largest cloud computing provider, 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 7.

2 Suitable Formal Frameworks for Modeling and Analyzing Cloud Computing Data Stores

What is a suitable formal framework for modeling and analyzing data store designs? This section lists some requirements for such a framework.

Expressive, simple, and intuitive modeling formalism. The modeling formalism should meet the seemingly contradictory requirements of being both *expressive* and very *simple* and *intuitive*. Transactional data stores are large and complex systems, incorporating different data types, communication forms (unicast, multicast, "atomic multicast," with or without using time), and so on. To make the specification of such systems in feasible, the specification formalism must be "powerful" and expressive. For example, even if, say, automata somehow theoretically *could* be used for such systems, which is highly dubious, specifying complex

data stores using such low-level formalisms would be a futile exercise. On the flip side, the language must be easy learn and use for designers of such complex systems, who are unlikely to have much, or any, training in formal methods.

Expressive and intuitive property specification language. There are a wide range of consistency properties that need to be formalized and analyzed, which means that a suitable *intuitive* property specification language is needed.

Automatically check whether a design satisfies a property. Automatic *model checking* analysis—as opposed to, say, theorem proving that involves highly nontrivial user interaction during the verification of desired properties—gives a quick feedback "at the push of a button." Such model checking, which checks whether *all possible* behaviors from a given initial system configuration satisfy the desired property, can be seen as a large "test suite" with more coverage than test-based development methods used in industry.

Support analyzing both functional correctness and performance. Performance measures, such as degree of availability, average latency per transaction, percentage of successfully committed transactions, percentage of reads that read the latest written values, etc., are as important for widely distributed data stores as functional correctness. We should be able to use the formal model of the system also for performance estimation, preferably *without* creating additional artifacts.

Support for real-time systems specification. The functionality of many advanced cloud computing systems depends crucially on real-time features, such as timeouts and retransmission of certain messages in case a response has not been received in a certain amount. The formal framework should support the specification of real-time systems and timed properties.

3 Our Framework: Rewriting Logic and Maude

Satisfying the desired properties above is a tall order. We advocate in the Assured Cloud Computing center the use of the *rewriting logic* formalism [18] and its associated Maude tool [7], and their extensions, as a suitable framework for formally specifying and analyzing transactional data store designs.

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. For example, the following Maude module defines a data type for natural numbers, where the constructors `0` and `s` define the values (`0`, `s(0)` (representing the number 1), `s(s(0))` (representing the number 2, etc.) of the sort `Nat`, and with defined functions `+` (for addition) and `double` (which doubles its argument) defined by equations:

```
fmod NAT is
  sort Nat .
  op 0 : -> Nat [ctor] .              op _+_ : Nat Nat -> Nat .
  op s : Nat -> Nat [ctor] .          op double : Nat -> Nat .
  vars M N : Nat .
  eq 0 + N = N .                      eq double(0) = 0 .
  eq s(M) + N = s(M + N) .            eq double(s(M)) = s(s(double(M))) .
endm
```

(The function `double` could also be defined by `eq double(M) = M + M .`)

Dynamic behaviors are defined by *rewrite rules* of the form $t \longrightarrow t'$, where t and t' are terms (possibly containing variables) representing *local state patterns*; the above rule then defines a family of local transitions (one for each substitution of the variables in t and t'). Rewriting logic is particularly suitable for specifying distributed systems in an object-oriented way. The state is then a multiset of objects and messages (traveling between the objects), 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 ` >`. Then, 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) .
```

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 [7] 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 feedback of the design (also called rapid prototyping). Maude also provides high-performance *reachability analysis*—which checks whether a certain (un)desired state pattern can be reached from the initial state—and *linear temporal logic model checking*—which checks whether all possible behaviors from the initial state satisfies a given *linear temporal logic* formula—to analyze all possible behaviors from a given initial configuration.

The Maude ecosystem also includes Real-Time Maude [20], which extends Maude to model and analyze *real-time systems*, and *probabilistic rewrite theories* [2], 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 [3]. Statistical model checking [25] consists of performing 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 is a promising formal framework for cloud computing data stores, as they address the desired properties in Section 2 as follows:

Expressive, simple, and intuitive specification formalism. Equations and rewrite rules. These intuitive notions are all that have to be learned and mastered. In

addition, object-oriented programming is a well-known programming paradigm, which means that the simple model of concurrent objects should be attractive to system designers. I have experienced in other projects that system developers find it easier to read and understand object-oriented Maude specifications than their own 50-page use case description of a state-of-the-art multicast protocol [21] and that a student with no previous formal methods background could easily model and analyze a complex IETF multicast in Maude [14].

Simple and intuitive property specification language. Complex system requirements, such as desired consistency levels, 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 [28]. In addition, we can define functions on states to express nontrivial reachability properties.

Automatic analysis. Maude provides automatic ("push-button") reachability and temporal logic model checking analysis, and simulation for rapid prototyping.

Performance estimation. I show in [22] that randomized Real-Time Maude simulations (of wireless sensor networks) can give as good performance estimates as domain-specific simulation tools. We can further analyze performance measures using probabilistic rewrite theories and statistical model checking. For example, "I can claim with 90% confidence that at least 75% of the transactions will satisfy the property P."

Support for real-time systems. Real-Time Maude supports the object-oriented specification of real-time systems in rewriting logic and the model checking of *timed* temporal logic properties [13], and has successfully been applied to a large number of advanced applications [23].

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

4 Case Study I: Formalizing, Analyzing, and Extending Google's Megastore

Megastore [5] 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. Megastore handled 3 billion write and 20 billion read transactions per day already in 2011 [5].

In Megastore, data are divided into different *entity groups* (such as, e.g., "Peter's email" or "books on rewriting logic"). Google's tradeoff between high

availability, concurrency, and consistency is that data consistency is only guaranteed for transactions accessing a *single* entity group. There are no guarantees if a transaction reads *multiple* entity groups.

Jon Grov, a researcher on transactional systems with no background in formal methods, became intrigued by the Megastore approach. He had some ideas about how to add consistency also for certain transactions accessing multiple entity groups, *without* significant performance penalty. He 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 of this industrial system that could be used for experimentation. The only publicly available description of Megastore was the brief overview paper [5]. To fully understand Megastore (more precisely, the Megastore algorithm/approach), Jon Grov and I first had to develop our own sufficiently detailed executable specification of Megastore from the description in [5]. 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.

Our specification of Megastore [9] is the first publicly available formalization and reasonably detailed description of Megastore. It contains 56 rewrite rules, 37 of which concern fault tolerance features. The point is that even *if* we would have had access to Megastore's code base, understanding it and extending it would likely be much more time-consuming than to develop 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`.

For performance estimation, we added a transaction generator that generates transaction requests at random times, with an adjustable average rate measured in *transactions per second*, and used the following probability distribution for the network delays (in milliseconds):

	30%	30%	30%	10%
HCMC ↔ Hanoi	10	15	20	50
HCMC ↔ Oslo	30	35	40	100
Hanoi ↔ Oslo	30	35	40	100

The key performance metrics to analyze are 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 *simulations* of this fairly challenging setting are given in the following table:

	Avg. latency (ms)	Commits	Aborts
HCMC	218	109	38
Oslo	336	129	16
Hanoi	331	116	21

These latency figures are consistent with Megastore itself [5]: "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."

For *model checking* analysis, we added a *serialization graph*, which is updated whenever a transaction commits, to the state. The property to analyze is that eventually: (i) all transactions have finished executing, (ii) all entity groups and all transaction logs are equal (or invalid), and (iii) the serialization graph does not contain cycles. 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 the execution time of the above model checking command for different system parameters, where $\{n_1, \ldots, n_k\}$ means that the corresponding value is selected nondeterministically from the set. All the model checking commands that finished executing returned `true`. *DNF* means that the execution was aborted after 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

Megastore-CGC. As mentioned, Jon Grov had some ideas to extend Megastore to provide consistency also for transactions accessing *multiple* entity groups, while maintaining Megastore's performance and strong fault tolerance features. Jon Grov observed that, in Megastore, a site replicating a set of entity groups participates in all updates on 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 [10].

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 HCMC and Oslo also handle transactions accessing multiple entity groups. Notice that Megastore-CGC will abort some transactions accessing multiple entity groups ("validation aborts") that Megastore does not care about:

	Megastore			**Megastore-CGC**			
	Commits	Aborts	Avg.latency	Commits	Aborts	Validation aborts	Avg.latency
Hanoi	652	152	126	660	144	0	123
HCMC	704	100	118	674	115	15	118
Oslo	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 (Jon Grov) to define both a precise, formal description and an executable prototype in a single artifact. We experienced 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"* [5]. 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 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 large number of different scenarios. Our experience from Megastore-CGC is 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.

5 Apache Cassandra

Apache Cassandra [11] is a popular open-source key-value data store originally developed at Facebook. (A key-value store can be seen as a transactional data store where transactions are single read or write operations.) Cassandra is used by, e.g., Amadeus, Apple, IBM, Netflix, Facebook/Instagram, and Twitter.

Cassandra only guarantees *eventual consistency* (if no more writes happen, then eventually all reads will read the *last* value written). However, it would be interesting to know: (i) under what conditions does Cassandra *guarantee* stronger properties, such as *strong consistency* (a read will always return the last value written before the read) or *read-your-writes* (the effect of a client's writes are visible to its subsequent reads)?; and (ii) how often are these stronger consistency properties satisfied in practice?

To understand how Cassandra works, and/or to experiment with alternative design choices, one has to study and modify Cassandra's 345,000 lines of code. Colleagues at the UIUC Assured Cloud Computing center therefore developed a detailed Maude specification of Cassandra, which captures all the main design decisions of Cassandra, and which consists of about 1000 lines of Maude code [17].

Standard model checking of the Maude specification showed that Cassandra indeed guarantees eventual consistency. Furthermore, they show that Cassandra guarantees strong consistency under certain conditions.

They then wanted to experiment with a possible optimization of Cassandra (where the *values* of the keys are considered instead of their *timestamps* when deciding which value should be returned upon a read operation) and see whether this provided better consistency or at least lower estimated operation latency. Having a formal specification of Cassandra, they could easily specify the proposed optimization by just modifying one function. To estimate how often Cassandra satisfies strong consistency or read-your-writes in practice, and to compare Cassandra with their proposed optimization, the authors in [16] transform their nondeterministic specification into a *fully probabilistic* probabilistic rewrite theory that is subjected to statistical model checking using PVeStA. The PVeStA analysis indicated that the proposed optimization of Cassandra did not improve the performance of Cassandra; their analysis also showed that, in practice, using Cassandra with consistency levels QUORUM or ALL provides strong consistency for almost all read operations as long as the time between a read and the latest write is above a certain threshold (see Fig. 1).

To investigate whether such PVeStA-based analysis really provides realistic performance estimates, the authors also executed the (real) Cassandra system on representative workloads. They could conclude that the statistical-model-checking-based performance estimation only differs from running the actual Cassandra system by 10-15%.

6 RAMP and P-Store Transaction Systems

Read-Atomic Multi-Partition (RAMP) transactions were proposed by Peter Bailis *et al.* [4] to offer light-weight multi-partition transactions that guarantee one of

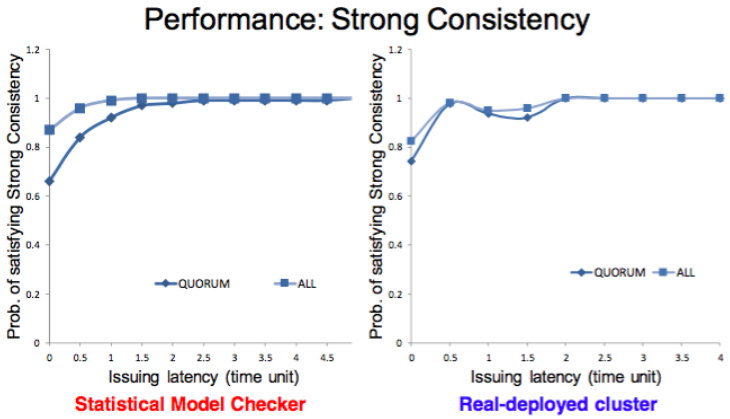

Fig. 1. PVeStA-based estimation (left) and actual measures running Cassandra (right) of the probability of reading the latest write as a function of the time between a read request and the latest previous write request.

the fundamental consistency levels, *read atomicity*: either all updates or no updates of a transaction are visible to other transactions. The paper [4] presents a pseudo-code description of the RAMP algorithm and "hand proofs" of key properties. The paper also mentions a number of optimizations and variations of RAMP, but without providing any details or correctness arguments. We have therefore formalized RAMP and its variations in Maude [15]. Maude model checking demonstrated that certain properties do not hold for certain variations of RAMP; for example, RAMP with one-phase commits does not satisfy the read-your-writes property.

I have recently formalized the P-Store transaction system [24] for partially replicated data stores. In contrast to our analysis of RAMP, which did not uncover any unexpected errors in the main algorithms, the Maude analysis of P-Store uncovered a number of errors in the P-Store algorithms in [24]. The authors of [24] confirmed the errors and also suggested some corrections. I could incorporate the proposed corrections into my Maude specification in less than an hour. Maude model checking of the corrected version did not encounter any problem.

7 How Amazon Web Services Uses Formal Methods

The previous sections have tried to make the case for the use of formal methods, and rewriting logic and Maude in particular, during the design and development of cloud computing storage systems. Nevertheless, the reported work took place in academic settings. How about industry?

About a year ago, engineers at Amazon Web Services (AWS) published a paper "How Amazon Web Services Uses Formal Methods" [19]. 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 [26]. Key components of its cloud computing infrastructure are the DynamoDB highly available replicated database and the Simple Storage System (S3), which stores more than two trillion objects and is handling more than 1.1 million requests per second [19].

The developers at AWS have used formal specification and model checking extensively since 2011. In this section I summarize their use of formal methods and their reported experiences.

Use of Formal Methods. The AWS developers used the approach that I advocate in this paper: 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+ [12], 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 $T == x' = x + 1 \wedge sum' = sum + x$. 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 experiences made by the AWS engineers was remarkably similar to the ones generally advocated by the formal methods community, and the experiences by Jon Grov. The quotations below are all taken from the paper [19].

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 psuedo-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. Whereas TLA+ was effective to find bugs, the main limitation was that TLA+ was not, or could not be, used to analyze performance degradation.

Why TLA+? In this paper, 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 specification of any computable data type as an equational specification. These features do not seem to be supported by TLA+, which of course might make TLA+ specification even more intuitive and understandable to a novice user. 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 $S \rightarrow P$.

Maybe 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 supports for the specification and analysis of real-time and probabilistic systems. (Leslie Lamport argues that special treatment of real-time systems is not needed: just add a system variable *clock* that denotes the current time [1].) 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 I have showed 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 maybe 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."

8 Concluding Remarks

I 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 computing data stores. 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 search for "corner case" bugs, or as defining a more comprehensive "test suite" than is possible in standard test-driven system development.

The intuitive Maude framework allowed a designer without formal methods background, Jon Grov, to develop a formal model of Google's complex fault-tolerant replicated data store Megastore, to design a significant extension (Megastore-CGC) of Megastore, and to check the correctness of the designs and compare their performance using randomized simulations. The key in the design process was to run simulations and model checking (for free) *throughout* the design process for quick and extensive feedback that helped making Grov aware of possible scenarios that he had not considered.

Likewise, colleagues at the UIUC Assured Cloud Computing center developed a formal specification of another popular industrial data store, Apache Cassandra, from its 345,000 lines of code. Using the 1,000-line formal specification, they could verify the desired *eventual consistency* guarantee of Cassandra, but could in addition also analyze under what condition stronger guarantees can be given. Having a concise specification of Cassandra made it possible to analyze a proposed optimization of Cassandra; this task would have been a nightmare if they would have had to modify the 345,000 lines of Cassandra code instead. Statistical model checking with PVeStA was used to compare the performance of Cassandra with the proposed optimization. Furthermore, they compared the performance predicted by the PVeStA analysis of their probabilistic formal model with the performance actually observed when running the real Cassandra code on representative workloads; these only differed by 10-15%.

We have also specified and analyzed two state-of-the-art academic transactional systems, including variations of RAMP not described in [4]. My analysis uncovered a number of bugs in P-Store.

I have also summarized the experiences of developers at Amazon Web Services, who have very 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) are feasible to master quickly for 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—which is as important as the functionality in a cloud computing system—since TLA+ lacks the support for analyzing (real-time and) probabilistic systems.

Although I am biased, I believe that the Maude formalism should be equally simple to master as TLA+ (as my experience with Jon Grov indicates), and might be even more suitable. Furthermore, as demonstrated by the Megastore and Cassandra work, the Maude tools are also suitable to analyze the performance of a system.

Lot of work remains. First of all, model checking only explores the behaviors from a *single* initial system configuration, and hence cannot be used to verify that an algorithm is correct for all possible initial system configurations. In [9,15] 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 yields a large number of symmetric initial states; greater coverage and symmetry reduction must therefore be explored. We should also develop techniques for *distributing* the model checking of such specifications.

To make Maude a more appealing framework we should develop a library of (verified?) formal cloud computing "building blocks," so that new systems can be quickly developed and tested by splicing in and out such components. It would of course be very desirable to obtain *modular* verification techniques for such a formal plug-and-play design framework.

Finally, we should design even more intuitive domain-specific system and property specification languages for cloud computing systems that can be automatically directly into Maude.

Acknowledgments. I thank Jon Grov for our joint work on Megastore, and Roy Campbell, Indranil Gupta, Si Liu, José Meseguer, and other colleagues at UIUC's Assured Cloud Computing University Center of Excellence for useful discussions on cloud computing and joint work on RAMP. I also thank the organizers of ISA 2016 for inviting me to give a presentation at their event.

References

1. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Transactions on Programming Languages and Systems 16(5), 1543–1571 (1994)
2. 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)
3. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: Proc. CALCO'11, LNCS, vol. 6859. Springer (2011)
4. Bailis, P., Fekete, A., Hellerstein, J.M., Ghodsi, A., Stoica, I.: Scalable atomic visibility with RAMP transactions. In: Proc. SIGMOD'14. ACM (2014)
5. Baker, J., et al.: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR'11. www.cidrdb.org (2011)
6. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proc. PODC'00. ACM (2000)
7. Clavel, M., et al.: All About Maude, LNCS, vol. 4350. Springer (2007)
8. Corbett, J.C., et al.: Spanner: Google's globally-distributed database. In: OSDI'12. USENIX (2012)
9. Grov, J., Ölveczky, P.C.: Formal modeling and analysis of Google's Megastore in Real-Time Maude. In: Specification, Algebra, and Software. LNCS, vol. 8373. Springer (2014)

10. Grov, J., Ölveczky, P.C.: Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In: Proc. SEFM'14. LNCS, vol. 8702. Springer (2014)
11. Hewitt, E.: Cassandra: The Definitive Guide. O'Reilly Media (2010)
12. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
13. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. Science of Computer Programming 99, 128–192 (2015)
14. Lien, E., Ölveczky, P.C.: Formal modeling and analysis of an IETF multicast protocol. In: Proc. SEFM'09. IEEE Computer Society (2009)
15. 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)
16. 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)
17. 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)
18. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
19. 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)
20. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
21. 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)
22. 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)
23. Ölveczky, P.C.: Real-Time Maude and its applications. In: Proc. WRLA'14. LNCS, vol. 8663. Springer (2014)
24. Schiper, N., Sutra, P., Pedone, F.: P-Store: Genuine partial replication in wide area networks. In: Proc. SRDS'10. IEEE Computer Society (2010)
25. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Proc. CAV'05. LNCS, vol. 3576. Springer (2005)
26. 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
27. Terry, D.: Replicated data consistency explained through baseball. Communications of the ACM 56(12), 82–89 (2013)
28. Vardi, M.Y.: Branching vs. linear time: Final showdown. In: Proc. TACAS'01. LNCS, vol. 2031. Springer (2001)