View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2019 Si Liu

DESIGN, VERIFICATION AND AUTOMATIC IMPLEMENTATION OF CORRECT-BY-CONSTRUCTION DISTRIBUTED TRANSACTION SYSTEMS IN MAUDE

BY

SI LIU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Professor José Meseguer, Chair
Professor Gul Agha
Professor Indranil Gupta
Professor Peter Csaba Ölveczky

ABSTRACT

Designing, verifying, and implementing highly reliable distributed systems is at present a hard and very labor-intensive task. Cloud-based systems have further increased this complexity due to the desired consistency, availability, scalability, and disaster tolerance. This dissertation addresses this challenge in the context of distributed transaction systems (DTSs) from two complementary perspectives: (i) designing DTSs with high assurance such that they satisfy desired *correctness* and *performance* requirements; and (ii) transforming *verified system designs* into *correct-by-construction distributed implementations.*

Regarding *correctness* requirements, we provide an object-based framework for formally modeling DTSs in Maude, explain how such models can be automatically instrumented to record relevant events during a run, formally define a wide range of consistency properties on such histories of events, and implement a tool which fully automates the entire specification instrumentation and model checking process.

Regarding *performance* requirements, we propose a general, though not yet automated, method that transforms the untimed, non-probabilistic, and nondeterministic formal Maude models of DTSs into probabilistic rewrite theories, explain how we can monitor the system executions of such probabilistic theories, and shows how we can evaluate the performance of the DTS designs based on the recorded log for different performance parameters and workloads by statistical model checking.

To bridge the *formality gap* between verified designs and distributed implementations we present a correct-by-construction automatic transformation mapping a verified formal specification of an actor-based distributed system design in Maude to a distributed implementation enjoying the same safety and liveness properties as the original formal design. Two case studies, applying this automatic transformation to state-of-the-art DTSs analyzed within the same formal framework for *both* logical and performance properties, show that high-quality implementations with acceptable performance and meeting performance predictions can be automatically generated in this way.

To the bad romance

&

To my parents

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor José Meseguer for the continuous support of my Ph.D. study and research, for his immense knowledge, motivation, and patience. Apart from our academic collaboration, I am deeply grateful for our friendship over the years.

I would also like to extend my appreciation to the rest of my dissertation committee members: Gul Agha, Indranil Gupta, and Peter Csaba Ölveczky, for their insightful feedbacks and invaluable advice. Special thanks are due to Indranil Gupta for mentoring me on distributed systems, and to Peter Csaba Ölveczky for being a co-author in many papers.

I am very thankful to my current and past lab mates and fellow students at University of Illinois at Urbana-Champaign. In particular, I thank Stephen Skeirik, Liyi Li, Kyungmin Bae, Musab AlTurki, Camilo Rocha, Qi Wang, and Atul Sandur, for the stimulating discussions and their friendship. I also thank Jatin Ganhotra, Keshav Santhanam, Sihan Li, Wei Yang, Everett Hildenbrandt, Daejun Park, Muntasir Raihan Rahman, Owolabi Legunsen, Rohit Mukerji, Andrei Stefanescu, Andrew Cholewa, and Fan Yang. I apologize for any of the inevitable omissions.

I would like to thank my master advisor Huibiao Zhu at ECNU. Without him I would never have the opportunity to start my Ph.D. study at UIUC.

The work in this dissertation was supported in part by NSF CNS 1409416, NSF CNS 1319527, AFOSR/AFRL FA8750-11-2-0084, and NRL contract N00173-17-1-G002.

Finally, I would like to express my deepest gratitude to my parents. This dissertation would not have been possible without their love, friendship, support, and sacrifices throughout my life. I dedicate this dissertation to the memory of my grandmother, whose role in my life was, and remains, immense. This last word of acknowledgment I have saved for the bad romance in my life: thanks for dancing with me!

TABLE OF CONTENTS

CHAPTER 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. Providing strong consistency guarantees, even in the failure-free case, would require a lot of communication which would seriously impact the system's performance. Different storage systems therefore offer different tradeoffs between the levels of availability and of consistency that they provide. For example, weak notions of consistency, such as read atomicity, are acceptable for applications, such as social networks, where availability and efficiency are key requirements, but where one can tolerate "stale" data. Other cloud applications, including online banking and medical information systems, require stronger consistency guarantees such as serializability.

Designing and implementing highly reliable high-performance distributed systems is at present a hard and very labor-intensive task. Cloud-based systems have further increased this complexity due to the desired consistency, availability, scalability, and disaster tolerance. For example, the communication needed to maintain strong consistency across sites may incur unacceptable latencies, so that designers must balance consistency and performance. Both *performance* and *functional correctness* are therefore critical system requirements.

This dissertation aims at addressing this challenge. Specifically, this dissertation answers the following two major questions:

1. How can cloud storage systems be designed with high assurance that they satisfy desired *correctness* and *performance* requirements?

2. How can a *verified system design* be transformed into a *correct-by-construction distributed implementation*?

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 made very difficult 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 [72] 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 compliance with a design 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 develop and modify for experimentation purposes. Simulation tools 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 [73]. 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 DESIGN AND VERIFICATION OF DISTRIBUTED SYSTEMS IN MAUDE

Formal methods have been advocated to develop and analyze high-level models of distributed system designs. In a formally-based system design and analysis methodology, a mathematical model M describes the system design at the appropriate level of abstraction. This system specification M should be complemented by a formal property specification P that describes mathematically (and therefore precisely) the requirements that the system M should satisfy. Being a mathematical object, the model M can be subjected to mathematical reasoning (preferably fully automated or at least machine-assisted) to guarantee that the design satisfies the properties P.

However, today's distributed systems present a number of challenges to formal methods:

(i) the sheer complexity and heterogeneity of such systems requires a flexible and expressive formal framework, which nevertheless must be simple and intuitive to be usable by system developers [73]; (ii) the correctness properties that these systems must satisfy can be quite complex, and there is a desire in industry for *automatic* verification techniques [73]; and (iii) both *correctness* and *performance* are, as mentioned, crucial requirements; a correct design that performs worse than similar designs is usually worthless.

One formal framework that has shown promise in meeting these challenges is Maude [32], a high-performance language and formal framework for executable specification, verification and programming of concurrent systems based on rewriting logic [67, 24, 69]. Maude meets challenge (i) by being based on a simple and intuitive formalism (algebraic equational specifications define data types and rewrite rules define dynamic behaviors) that is at the same time general and expressive. Maude also provides a natural model of concurrent objects, which is ideal for modeling distributed systems. Regarding challenge (ii), Maude provides a range of *automatic model checking methods*, including reachability analysis and LTL and LTLR temporal logic model checking [32, 13], which allows us to express and analyze complex properties (see, e.g., [62]). The Maude tool environment also provides theorem proving verification of invariants in the InvA tool [77], and of reachability logic properties in Maude's reachability logic prover [84]. For challenge (iii), the Maude tool environment includes the PVeStA [7] *statistical model checker*, which can be used to statistically predict the performance of a design.

These features have made possible the use of Maude to model and analyze both the correctness and performance of high-level designs of a wide range of state-of-the art systems (see the survey [69]). To cite just one example area, Maude has been used to formally model and analyze, often for the first time, state-of-the-art industrial and academic cloud-based transaction systems such as Cassandra [2], ZooKeeper [49], Google's Megastore [16], P-Store [79], RAMP [14], and Walter [85]; and to design the entirely new system ROLA [59] (see the survey [22]). Furthermore, model-based performance predictions using PVeStA have shown good correspondence with experimental evaluation of implementations of systems such as Cassandra, RAMP, and Walter [55, 57, 61].

Despite of the above promising methodology, there is very little work on its automation. There is therefore a need for algorithmic methods for automatically verifying consistency properties, and quantifying performance properties based on executable specifications of both the systems and their properties. Chapter 3 takes the first step towards this goal by presenting a general framework for automatic analysis of consistency properties of DTSs in Maude, followed by three case studies on RAMP, Walter and ROLA in Chapter 4, 5 and 6, respectively.

1.3 FROM DISTRIBUTED SYSTEM DESIGNS TO IMPLEMENTATIONS

We can develop mature designs satisfying given correctness criteria and having good predicted performance in the above way. However, this still leaves open the problem of how to pass from a *verified system design* to a *correct-by-construction distributed implementation.*

Maude provides TCP/IP sockets as Maude external objects so that they can interact with standard Maude objects by message passing [32], and a Maude concurrent object system can be deployed as a *distributed system* across several machines. Message passing within a single machine is executed by rewriting; but message-passing across machines is achieved by Maude TCP/IP socket objects.

Since many different distributed deployments can be chosen for the same concurrent object system *design* expressed as a Maude program M, various distributed implementations can be programmed *within Maude* by manually transforming the design M into a distributed Maude program $D(M)$ by importing the `SOCKET` module [32] and programming the remote message passing communication through such sockets. This, however, leaves open a *formality gap.* Suppose that a given property φ has been verified for the system design M. Does φ still hold true for $D(M)$? Up to now, this formality gap has been filled by developing a formal model $D'(M)$ of $D(M)$ in Maude and verifying that $D'(M)$ verifies φ. For example, the correctness of both the distributed implementations of the Mobile Maude language, and of the Orc orchestration language have been verified this way by model checking in, respectively, [32] and [6].

This situation is unsatisfactory because: (i) one has to *manually hand program* $D(M)$, and has to do so for each particular choice of deployment; and (ii) *checking the preservation of formal properties* when passing from M to $D(M)$ is required *for each* M, which defeats the purpose of carrying out the verification on the simpler model M. One major contribution of this dissertation (Chapter 7) is to *fully automate* the passage from M to $D(M)$ and to *prove* that M and an abstract model $D_0(M)$, which hides the details of $D(M)$'s TCP/IP-based network communication, are *stuttering bisimilar* [71, 66] and therefore satisfy the exact same *CTL** temporal logic properties for any formulas not using the "next" operator $\bigcirc$. Therefore, both *safety* and *liveness* properties are preserved by the bisimulation. What a Maude user provides as *input* to the automatic $M \mapsto D(M)$ transformation is a three-tuple $(M, \mathit{init}, \mathit{di})$, where M is the Maude module specifying the given system's design, *init* is an initial state in such a design, and *di* is a *distribution function*, indicating the specific IP address and Maude *session*[1] where each object in *init* will be located.

[1]Several concurrent Maude sessions can be executed on the same machine.

1.4 SUMMARY OF CHAPTERS AND CONTRIBUTIONS

The work presented in this dissertation can be mainly classified in three categories: (i) a general framework for formally specifying a distributed transaction system (DTS), and for automatically analyzing its consistency properties; (ii) case studies of state-of-the-art DTSs in this framework; and (iii) automatic transformation of a formal Maude design of a DTS into a correct-by-construction distributed implementation.

The rest of this dissertation is organized as follows:

Chapter 2: This chapter gives preliminaries on rewriting logic and Maude, the statistical model checking of probabilistic rewrite theories using PVeStA, stuttering bisimulations, and the consistency properties in DTSs.

Chapter 3: This chapter presents a general framework for formally specifying a DTS in Maude, and formalizes in Maude nine common consistency properties for DTSs so defined. Furthermore, it provides a fully automated method for analyzing whether a DTS design satisfies the desired consistency properties for all initial states up to user-given bounds on system parameters.

This chapter is based on the joint work [62] with José Meseguer, Peter Csaba Ölveczky, Qi Wang, and Min Zhang.

Chapter 4: This chapter shows a case study of using framework on formal modeling of the RAMP transaction system, its variants and new RAMP-like designs of my own, and on model checking eight such RAMP designs against the desired consistency properties. Moreover, statistical model checking is used to explore and extend the design space of RAMP by showing that our results: (i) are consistent with the experimental evaluations of the implemented designs; (ii) are also consistent with conjectures made by the RAMP developers for other unimplemented designs; and (iii) uncover some promising new designs that seem attractive for some applications.

This chapter is based on the joint work [58, 57, 62] with Jatin Ganhotra, Indranil Gupta, José Meseguer, Peter Csaba Ölveczky, Muntasir Raihan Rahman, Qi Wang, and Min Zhang.

Chapter 5: This chapter describes the second case study of using the framework on formal modeling of Walter, and on formal verification of various consistency properties by model checking. To also analyze Walter's performance we extend the Maude specification of Walter to a probabilistic rewrite theory and perform statistical model checking analysis to evaluate

Walter's throughput for a wide range of workloads. Our performance results are consistent with a previous experimental evaluation and throw new light on Walter's performance for different workloads not evaluated before.

This chapter is based on the joint work [61, 62] with José Meseguer, Peter Csaba Ölveczky, Qi Wang, and Min Zhang.

Chapter 6: This chapter presents (i) the design of a new transaction protocol, ROLA, having useful applications and guaranteeing a new consistency model, update atomicity, meeting read atomicity and prevention of lost updates consistency properties with competitive performance; (ii) formal specification and model checking analysis of ROLA; (iii) a detailed performance comparison by statistical model checking between ROLA and the Walter and Jessy protocols showing that ROLA outperforms both Walter and Jessy in all such comparisons, including higher throughput and lower average latency; and (iv) a demonstration that, by a suitable use of formal methods, a completely new distributed transaction protocol can be designed and thoroughly analyzed, as well as be compared with other designs, very early on, *before* its implementation.

This chapter is based on the joint work [59, 60, 62] with Indranil Gupta, José Meseguer, Peter Csaba Ölveczky, Keshav Santhanam, Qi Wang, and Min Zhang.

Chapter 7: This chapter shows (i) the formal definition of the $M \mapsto D(M)$ transformation mapping a Maude concurrent system formal design M to its distributed implementation $D(M)$; (ii) the proof that for any actor-like Maude specification M the system $D_0(M)$ modeling $D(M)$ and the system design M are stuttering bisimilar and satisfy the same safety and liveness properties; (iii) a Maude prototype automation of the $M \mapsto D(M)$ transformation allowing us to generate, deploy and evaluate correct-by-construction implementations of state-of-the art system designs, and allowing interaction of such implementations with workload generators such as YCSB [36]; (iv) two case studies using state-of-the-art DTSs (including ROLA) evaluating the implementations obtained by the $M \mapsto D(M)$ transformation with respect to: (a) the statistical-model-checking-based performance predictions for M; and (b) a conventional implementation. Similar performance trends have been respectively shown for both case studies.

This chapter is based on the joint work with José Meseguer, Peter Csaba Ölveczky, Atul Sandur, and Qi Wang.

Chapter 8: This chapter ends the dissertation with some concluding remarks and future research directions.

Acknowledgments. I thank Andrea Cerone, Alexey Gotsman, Jatin Ganhotra, and Rohit Mukerji for helpful early discussions on the work in Chapter 6, and thank Jatin Ganhotra, Indranil Gupta, José Meseguer, Son Nguyen, Muntasir Raihan Rahman, and Stephen Skeirik for the joint work [63, 56, 55] on modeling and analysis of Cassandra, which has inspired my work in this dissertation.

CHAPTER 2: PRELIMINARIES

2.1 REWRITING LOGIC AND MAUDE

A *membership equational logic* (Mel) [68] *signature* is a triple $\Sigma = (K, \Sigma, S)$ with K a set of *kinds*, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a K-kinded family of disjoint sets of sorts. The kind of a sort s is denoted by $[s]$. A Σ-*algebra* A consists of a set A_k for each kind k, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each operator $f \in \Sigma_{k_1\cdots k_n,k}$, and a subset inclusion $A_s \subseteq A_k$ for each sort $s \in S_k$. The set $T_{\Sigma,k}$ denotes the set of ground Σ-terms with kind k, and $T_\Sigma(X)_k$ denotes the set of Σ-terms with kind k over the set X of kinded variables.

A Mel *theory* is a pair (Σ, E) with Σ a Mel-signature and E a finite set of Mel sentences, which are either conditional equations or conditional memberships of the forms:

$$(\forall X)\ t = t' \ \textbf{if} \ \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j, \qquad (\forall X)\ t : s \ \textbf{if} \ \bigwedge_i p_i = q_i \wedge \bigwedge_j w_j : s_j,$$

where $t, t' \in T_\Sigma(X)_k$ and $s \in S_k$ for some kind $k \in \Sigma$, the latter stating that t is a term of sort s, provided the condition holds. In Maude, an individual equation in the condition may also be a *matching equation* p_l `:=` q_l, which is mathematically interpreted as an ordinary equation. However, operationally, the new variables occurring in the term p_l become instantiated by matching the canonical form of the instance of q_l against the pattern term p_l (see [32] for further explanations). Order-sorted notation $s_1 < s_2$ abbreviates the conditional membership $(\forall x : [s_1])\ x : s_2$ **if** $x : s_1$. Similarly, an operator declaration $f : s_1 \cdots s_n \to s$ corresponds to declaring f at the kind level and giving the membership axiom $(\forall\, x_1 : [s_1], \ldots, x_n : [s_n])\ f(x_1, \ldots, x_n) : s$ **if** $\bigwedge_{1\leq i\leq n} x_i : s_i$.

A Maude module specifies a *rewrite theory* [67] of the form $(\Sigma, E \cup B, R)$, where:

1. $(\Sigma, E \cup B)$ is a membership equational logic theory specifying the system's state space as an algebraic data type with B a set of equational axioms (such as a combination of associativity, commutativity, and identity axioms), to perform equational deduction with the equations and memberships in E (the equations are oriented from left to right for equational simplification) *modulo* the axioms B, and

2. R is a set of *labeled conditional rewrite rules* specifying the system's local transitions,

each of which has the form:

$$l : q \longrightarrow r \textbf{ if } \bigwedge_i p_i = q_i \ \wedge \ \bigwedge_j w_j : s_j \ \wedge \ \bigwedge_m t_m \longrightarrow t'_m,$$

where l is a *label*, and q, r are Σ-terms of the same kind. Intuitively, such a rule specifies a *one-step transition* from a substitution instance of q to the corresponding substitution instance of r, *provided* the condition holds; that is, that the substitution instance of each condition in the rule follows from $\mathcal{R}$ (where condition $t_m \longrightarrow t'_m$ is understood as $t_m \longrightarrow^* t'_m$).

We briefly summarize the syntax of Maude and refer to [32] for more details. Sorts and subsort relations are declared by the keywords `sort` and `subsort`, and operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ \ldots \ s_n$ are the sorts of its arguments, and s is its (value) *sort*. Operators can have user-definable syntax, with underbars '`_`' marking each of the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators can have equational *attributes*, such as `assoc`, `comm`, and `id`, stating, for example, that the operator is associative and commutative and has a certain identity element. Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. An operator can also be declared to be a *constructor* (`ctor`) that builds up the data elements of its sort.

There are three kinds of logical statements in the Maude language, *equations*, *memberships* (declaring that a term has a certain sort), and *rewrite rules*, introduced with the following syntax:

- equations: `eq` $u = v$ or `ceq` $u = v$ `if` *condition*;
- memberships: `mb` $u : s$ or `cmb` $u : s$ `if` *condition*;
- rewrite rules: `rl [`l`]:` u `=>` v or `crl [`l`]:` u `=>` v `if` *condition*.

An equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied. The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they have the form *var*`:`*sort*. Finally, a comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

In object-oriented Maude specifications, a *class* declaration `class` C `|` att_1 `:` s_1`, ...,` att_n `:` s_n declares a class C of objects with attributes att_1 to att_n of sorts s_1 to s_n. An *object* *instance* of class C is represented as a term < $O : C \mid att_1 :\ val_1, \ldots, att_n :\ val_n$ >, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n (of respective sorts $s_1, \ldots, s_n$) are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`. A system's distributed state, called a *configuration*, is modeled as a term of the sort `Config`, and has the structure of a *multiset* made up of objects and messages built up with an empty syntax (juxtaposition) multiset union operator `__`.

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

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

defines the family of concurrent transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`, the attribute `a1` of the object `O` is changed to `x + w`, and a new message `m'(O',x)` is generated. Attributes whose values do not change and do not affect the next state, such as `a3`, need not be mentioned in a rule; all such "superfluous" attributes can be replaced by a variable (such as `AS`) of sort `AttributeSet`, so that the above rewrite rule can also be written

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

Example 2.1. The following example shows parts of a Maude specification of a very simple partitioned database system, where each client performs a sequence of read operations on data items[1] (or keys). A `table` assigns to each data item `K` the `DB` object that stores the data item. When a client wants to read the value of a data item `K`, it sends a `read(K)` message to the `DB` object storing `K` (rule `getValue`). This `DB` object replies with a `value(K,V)` message (rule `replyRead`). When the client reads this response, it stores the pair `(K,V)` in its log (rule `readValue`). We just show snippets of the Maude code, and omit, e.g., the declarations of the data types and messages involved:

[1]By a "data item" we mean (as usual in this area) a key used to store values, and by "data" we mean (as usual in this area) the values stored in such keys.

```
class Client | operations : OperationList, log : Log .
class DB | database : Map{Key,Value} .
op table : -> Map{Key,Oid} .

vars O O' : Oid .  var K : Key .  var OL : OperationList .
var V : Value .    var B : Map{Key,Oid} .  var LOG : Log .

rl [getValue] :
   < O : Client | operations : read(K) :: OL >
  =>  < O : Client | operations : OL >
      (to table[K] from O : read(K)) .

rl [replyRead] :
   (to O from O' : read(K))  < O : DB | database : B >
  =>  < O : DB | >  (to O' from O : value(K,B[K])) .

rl [readValue] :
   (to O from O' : value(K,V))  < O : Client | log : LOG >
  =>  < O : Client | log : LOG ++ (K,V) > .
```

The following shows an initial configuration with two clients and two database partitions, each storing two data items:

```
ops c1 c2 db1 db2 : -> Oid .
ops k1 k2 k3 k4 : -> Key .

op init : -> Configuration .
eq init =
 < c1 : Client | operations : read(k3) :: read(k2), log : empty >
 < c2 : Client | operations : read(k4) :: read(k3), log : empty >
 < db1 : DB | database : k1 |-> 54, k2 |-> 8 >
 < db2 : DB | database : k3 |-> 9, k4 |-> 7 > .

eq table = k1 |-> db1, k2 |-> db1, k3 |-> db2, k4 |-> db2 .
```

Reflection and Metaprogramming in Maude. Rewriting logic is reflective [33, 34], in the sense that important aspects of its metatheory can be represented at the object level in a consistent way. One important application of reflection is to support *metaprogramming* in

Maude in the sense that a term t in a rewrite theory R is meta-represented as a meta-term $\overline{t}$ of sort Term, and a rewrite theory R is meta-represented as a term $\overline{R}$ of sort Module. A Maude program (or specification) can then be transformed by means of a module transformation equationally defined as a Maude function f : Module $\rightarrow$ Module.

Example 2.2. To get a feeling for the similarity between the object-level and meta-level notations, let us consider the metarepresentation of the module on the left as the term (called a *meta-module*) displayed on the right:

```
fmod NAT is                                 fmod 'NAT is
  pr BOOL .                                   protecting 'BOOL .
  sorts Zero Nat .                            sorts 'Zero ; 'Nat .
  subsort Zero < Nat .                        subsort 'Zero < 'Nat .
  op 0 : -> Zero [ctor] .                     op '0 : nil -> Zero [ctor] .
  op s : Nat -> Nat [ctor] .                  op 's : 'Nat -> 'Nat [ctor] .
  op _+_ : Nat Nat -> Nat [ctor] .            op '_+_ : 'Nat 'Nat -> 'Nat [ctor] .
  vars N M : Nat .
  eq 0 + N = N .                              eq '_+_['0.Nat, 'N:Nat] = 'N:Nat .
  eq s(N) + M = s(N + M) .                    eq '_+_['s['N:Nat], 'M:Nat]
                                                = 's['_+_['N:Nat, 'M:Nat]] .
endfm                                       endfm
```

where sorts and kinds are metarepresented as terms in subsorts Sort and Kind of the sort Qid of quoted identifiers. For example, the natural number term s(N) + M shown on the left is meta-represented as the meta-term '_+_['s['N:Nat], 'M:Nat] on the right using the meta-level operator _[_] : Qid Term -> Term, the meta-representation of the NAT operators _+_ and s as quoted identifiers '_+_ and 's, and the meta-representation of variables N and M of sort Nat as 'N:Nat and 'M:Nat.

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

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

where *bound* provides an upper bound on the number of solutions to be found (if omitted, there is no such upper bound). The arrow `=>!` means that Maude only searches for reachable *final* states (i.e., states that cannot be further rewritten) that match *pattern* and satisfies *cond*. If the arrow used is instead `=>*` then Maude searches for all reachable states matching the search pattern and satisfying *cond*.

Sockets in Maude. Maude's `erewrite` command supports rewriting with external objects (that do not reside in the configuration) when the "portal" object `<>` is present in the configuration. Objects in a Maude process, called here a *session*, can communicate with so-called *external objects* in the *same session* by message passing. In particular, they can communicate with Maude's built-in *socket manager* object, with object name `socketManager`, that supports establishing communication and communicating through TCP sockets with other *remote Maude objects* in other Maude sessions, as well as with *remote foreign objects* (see Section 7.1.3) in other processes. Some of the messages defining the interface between a Maude process and Maude's socket manager are the following:

A message `createServerTcpSocket(socketManager,` *myOid*`,` *port*`, ...)` asks Maude's socket manager to create a server socket. If the socket is created successfully, Maude's socket manager sends the message `createdSocket(`*myOid*`,socketManager,`*socketName*`)`, where *socketName* is the name of the created socket. The message `send(`*socketName*`,`*myOid*`,`*string*`)` asks Maude to send *string* through the socket *socketName*. The message `receive(`*socketName*`,` *myOid*`)` solicits data through a socket. When some data (*string*) is received through a socket, the socket manager sends the message `received(`*myOid*`,`*socketName*`,`*string*`)`.

2.2 STATISTICAL MODEL CHECKING AND PVESTA

Distributed systems are often probabilistic in nature, e.g., network latency such as message delay may follow a certain probability distribution, plus some algorithms may be probabilistic. Systems of this kind can be modeled by *probabilistic rewrite theories* [5] with rules of the form:

$$[l] : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \;\; \textbf{if} \;\; cond(\overrightarrow{x}) \;\; with \;\; probability \;\; \overrightarrow{y} := \pi(\overrightarrow{x})$$

where the term t' has additional new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. Since for a given matching instance of the variables $\overrightarrow{x}$ there can be many (often infinite) ways to instantiate the extra variables $\overrightarrow{y}$, such a rule is *nondeterministic*. The probabilistic nature of the rule stems from the probability distribution $\pi(\overrightarrow{x})$, which depends on the matching instance of $\overrightarrow{x}$, and governs the probabilistic choice of the instance of $\overrightarrow{y}$ in the result $t'(\overrightarrow{x}, \overrightarrow{y})$ according to $\pi(\overrightarrow{x})$.

Statistical model checking [80, 94] is an attractive formal approach to analyzing probabilistic systems against quantitative temporal logic properties. Instead of offering a binary yes/no answer, it provides a quantitative real-valued answer and can verify a property up to a user-specified level of confidence by running Monte-Carlo simulations of the system model. The quantitative answer, however, need not be a percentage or a probability: it may instead be a latency estimation, or a quantitative estimation of some other performance property. For example, a statistical model checking result may be "86.87% of ROLA transactions commit successfully with 95% confidence". Existing statistical model checking verification techniques assume that the system model is purely probabilistic. Using the methodology in [5, 41] we can eliminate nondeterminism in the choice of firing rules. We then use PVeStA [7], an extension and parallelization of the tool VeStA [81], to statistically model check purely probabilistic systems against properties expressed by QuaTEx quantitative temporal logic [5]. The expected value of a QuaTEx expression is iteratively evaluated w.r.t. two parameters α and δ provided as input by sampling until the size of `(1-`α`)100%` confidence interval is bounded by δ, where the result of evaluating a formula is not a Boolean value, but a real number.

2.3 STUTTERING BISIMULATIONS

For preservation of temporal logic properties between a Maude design of a concurrent system and its automatic implementation as a distributed system we will use the notion of a *stuttering bisimulation map* between Kripke structures. Recall that a *Kripke structure* $\mathcal{A}$ on a set AP of *atomic propositions* is a 4-tuple $\mathcal{A} = (A, \to_{\mathcal{A}}, a_0, L_{\mathcal{A}})$ where A is a set of *states*, $\to_{\mathcal{A}} \subseteq A \times A$ is the *total transition relation on states* (total means that $\forall a \in A\ \exists a' \in A$ s.t. $a \to_{\mathcal{A}} a'$), $a_0 \in A$ is the *initial state*, and $L_{\mathcal{A}}$, called the *labeling function*, is a function $L_{\mathcal{A}} : A \to \mathcal{P}(AP)$ assigning to each state $a \in A$ the set of atomic state predicates $L_{\mathcal{A}}(a)$ true in state a. A *path* π in $\mathcal{A}$ is function $\pi : \mathbb{N} \to A$ such that $\pi(0) = a_0$ and $\forall n \in \mathbb{N}\ \pi(n) \to_{\mathcal{A}} \pi(n+1)$.

Given Kripke structures $\mathcal{A}$ and $\mathcal{B}$, intuitively, a *bisimulation* is a proposition-preserving correspondence between states of $\mathcal{A}$ and states of $\mathcal{B}$ such that any action of $\mathcal{A}$ can be replicated by an action of $\mathcal{B}$, and vice versa. In this dissertation, $\mathcal{B}$ will be a concurrent system's formal *design* in Maude, and $\mathcal{A}$ will be its distributed Maude implementation. The implementation $\mathcal{A}$ will be *correct by construction* if we can prove that the design $\mathcal{B}$ and its implementation $\mathcal{A}$ are *bisimilar*. Since what is an *atomic transition* in a design may be realized by a *sequence of transitions* in its implementation, our bisimulation needs to be a *stuttering bisimulation map* in the following sense:

Definition 2.1. [71] Given Kripke structures $\mathcal{A} = (A, \to_{\mathcal{A}}, a_0 L_{\mathcal{A}})$ and $\mathcal{B} = (B, \to_{\mathcal{B}}, b_0, L_{\mathcal{B}})$, a *stuttering bisimulation map*, denoted $h : \mathcal{A} \to \mathcal{B}$, is a function $h : A \to B$ such that: (1) given any path π in $\mathcal{A}$ there is a path ρ in $\mathcal{B}$ and a strictly monotonic function $\kappa : \mathbb{N} \to \mathbb{N}$ such that: (i) for each $n \in \mathbb{N}$ and each i, $\kappa(n) \leq i < \kappa(n{+}1)$, (ii) $h(\pi(\kappa(n))) = h(\pi(\kappa(i))) = \rho(n)$, and (iii) $L_{\mathcal{A}}(\pi(\kappa(n))) = L_{\mathcal{A}}(\pi(i)) = L_{\mathcal{B}}(\rho(n))$. And (2) given any path ρ in $\mathcal{B}$ there is a path π in $\mathcal{A}$ and a strictly monotonic function $\kappa : \mathbb{N} \to \mathbb{N}$ satisfying the above conditions (i)–(iii).

The states $\pi(i)$, $\kappa(n) \leq i < \kappa(n{+}1)$, can be called the "stuttering states" of $\mathcal{A}$ bisimulated by $\rho(n)$ in $\mathcal{B}$. The key property of a stuttering bisimulation map $h : \mathcal{A} \to \mathcal{B}$ is that all formulas $\varphi \in CTL^* \setminus \bigcirc$ satisfied by $\mathcal{B}$ are also satisfied by $\mathcal{A}$, and vice versa, where $CTL^* \setminus \bigcirc$ denotes the subset of the CTL^* temporal logic not involving the "next" operator $\bigcirc$ (for more on CTL^*, its LTL sublogic, and the satisfaction relation $\mathcal{A} \models \varphi$ between a Kripke structure and a CTL^* formula φ see [31]). That is, we have:

Theorem 2.1. [71], Thm. 3 (Implementation Correctness). If $h : \mathcal{A} \to \mathcal{B}$ is a stuttering bisimulation map, for each $\varphi \in CTL^* \setminus \bigcirc$ we have: $\mathcal{B} \models \varphi \Leftrightarrow \mathcal{A} \models \varphi$.

Definition 2.1 is conceptually appealing but hard to check directly. As explained in [71], a more easily checkable characterization by Manolios [66] can be adapted to our setting as the following theorem:

Theorem 2.2. (Adapted from [66, 71]). If all states in Kripke structures $\mathcal{A}$ and $\mathcal{B}$ are reachable from their corresponding initial states a_0 and b_0, then a function $h : A \to B$ such that $h(a_0) = b_0$ and $L_{\mathcal{A}} = L_{\mathcal{B}} \circ h$ is a stuttering bisimulation map $h : \mathcal{A} \to \mathcal{B}$ iff there is a well-founded order $(W, >)$ and a function $\mu : A \times B \to W$ such that whenever $h(a) = b$ and $a \to_{\mathcal{A}} a'$, then either (i) there is a $b' \in B$ s.t. $b \to_{\mathcal{A}} b'$ and $h(a') = b'$, or (ii) $h(a') = b$ and $\mu(a, b) > \mu(a', b)$, and, in addition, whenever $h(a) = b$ and $b \to_{\mathcal{B}} b'$, there is a finite sequence of transitions $a \to_{\mathcal{A}} a_1 \ldots a_n \to_{\mathcal{A}} a_{n+1}$, with $n \geq 0$, such that for $1 \leq i < n + 1$ $h(a_i) = b$, and $h(a_{n+1}) = b'$.

A concurrent system design is formally specified in Maude as a rewrite theory. For temporal logic reasoning we can associate to a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and an initial state $init \in T_{\Sigma/E}$ a corresponding *Kripke structure* $\mathcal{K}(\mathcal{R}, init) = (Reach(init), \longrightarrow^{\bullet}_{R/E}, init, L)$ as follows [32]: (i) $Reach(init)$ is the set of all states $[u] \in T_{\Sigma/E}$ *reachable* from $init$, i.e., such that $[t] \longrightarrow^{*}_{R/E} [u]$, where $\longrightarrow_{R/E}$ denotes the relation of rewriting E-equivalence classes with the rules R modulo E; (ii) $\longrightarrow^{\bullet}_{R/E}$ is the (totalization of) the one-step rewrite relation

$\longrightarrow_{R/E}$; and (iii) L maps each reachable state $[u]$ to the set $L([u]) = \{p \in AP \mid u \models p =_E true\}$, where $=_E$ denotes equality modulo E and we assume that Σ contains a sort whose ground terms are the atomic propositions AP, and E contains equations defining the satisfaction relation $u \models p$ between states and atomic propositions as a Boolean-valued function $\models$.

2.4 TRANSACTION CONSISTENCY

Different applications require different consistency guarantees. There are therefore many consistency properties for DTSs on partially replicated distributed data stores. This dissertation focuses on the following nine, which span a spectrum from weak consistency such as read committed to strong consistency like serializability:

- *Read committed* (*RC*) [18] disallows a transaction[2] from seeing any uncommitted or aborted data.
- *Cursor stability* (*CS*) [38], widely implemented by commercial SQL systems (e.g., IBM DB2 [3]) and academic prototypes (e.g., MDCC [50]), guarantees *RC* and in addition prevents the *lost update* anomaly.
- *Read atomicity* (*RA*) [14] guarantees that either *all* or *none* of a (distributed) transaction's updates are visible to other transactions. For example, if Alice and Bob become friends on social media, then Charlie should not see that Alice is a friend of Bob's, and that Bob is not a friend of Alice's.
- *Update atomicity* (*UA*) [29, 59] guarantees read atomicity and prevents lost updates.
- *Snapshot isolation* (*SI*) [18] requires a multi-partition transaction to read from a snapshot of a distributed data store that reflects a single commit order of transactions across sites, even if they are independent of each other. Alice sees Charlie's post before seeing David's post if and only if Bob sees the two posts in the same order. Charlie and David must therefore coordinate the order of committing their posts even if they do not know each other.
- *Parallel snapshot isolation* (*PSI*) [85] weakens *SI* by allowing different commit orders at different sites, while guaranteeing that a transaction reads the most recent version

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

committed at the transaction execution site, as of the time when the transaction begins. For example, Alice may see Charlie's post before seeing David's post, even though Bob sees David's post before Charlie's post, as long as the two posts are independent of each other. Charlie and David can therefore commit their posts without waiting for each other.

- *Non-monotonic snapshot isolation* (*NMSI*) [9] weakens *PSI* by allowing a transaction to read a version committed after the transaction begins. Alice may see Bob's post that committed after her transaction started executing.

- *Serializability* (*SER*) [75] ensures that the execution of concurrent transactions is equivalent to one where the transactions are run one at a time.

- *Strict Serializability* (*SSER*) strengthens *SER* by enforcing the serial order to follow real time.

CHAPTER 3: AUTOMATIC ANALYSIS OF CONSISTENCY PROPERTIES OF DISTRIBUTED TRANSACTION SYSTEMS

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

In particular, we show how one can automatically add a monitoring mechanism which records relevant history during a run of a DTS specified in our framework, and we define the consistency properties on such histories so that the DTS can be directly model checked in Maude. We have implemented a tool that uses Maude's meta-programming features to automatically add the monitoring mechanism, that automatically generates all the desired initial states, and that performs the Maude model checking of the desired consistency properties. We have applied our tool to model check the consistency properties of state-of-the-art DTSs such as variants of RAMP [14], P-Store [79], ROLA [59], Walter [85], and Jessy [9]. To the best of our knowledge, this is the first time that model checking of all these properties in a unified, systematic manner is investigated and supported.

This chapter is organized as follows. Section 3.1 presents our framework for modeling DTSs in Maude, and Section 3.2 explains how to record the history in such models. Section 3.3 formally defines consistency models as Maude functions on such recorded histories. Section 3.4 introduces our tool which automates the entire process. Finally, Section 3.5 discusses related work and Section 3.6 gives some concluding remarks.

Acknowledgements. This chapter is based on the work [62] partially supported by NRL contract N00173-17-1-G002, and NSFC Project No. 61872146.

3.1 MODELING DISTRIBUTED TRANSACTION SYSTEMS IN MAUDE

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

- We can identify and record "when"[1] a transaction starts executing at its server/proxy

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

and "when" the transaction is committed and aborted at the different sites involved in its validation.

- The transactions record their read and write sets.

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

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

- The database is distributed across of a number of *sites*, or *servers* or *replicas*, that communicate by asynchronous *message passing*. Data are *partially replicated* across these sites: a data item may be replicated/stored at more than one site. The sites replicating a data item are called that item's *replicas*.

- Systems evolve by message passing or by local computations. Servers communicate by asynchronous message passing with arbitrary but finite delays.

- A client forwards a transaction to be executed to some server (called the transaction's *executing server* or *proxy*), which executes the transaction.

- Transaction execution should terminate in either a *commit* or an *abort*.

3.1.1 Modeling DTSs in Maude

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

Basic Data Types. There are user-defined sorts `Key` for data items (or keys) and `Version` for versions of data items, with a partial order `<` on versions, with $v < v'$ denoting that v' is a later version of v in `<`. We then define key-version pairs `<`*key*`,`*version*`>` and sets of such pairs, that model a transaction's read and write sets, as follows:

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

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

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

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

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

The attributes `executing`, `committed`, and `aborted` contain, respectively, transactions that are being executed, and have been committed or aborted on the executing server; `decided` is the status of transactions executed on other servers. Recall that transactions themselves are modeled as objects *internal* to the replica storing them. That is why the `executing`, `committed`, and `aborted` attributes have sort `Configuration`.

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

Example 3.1. A replica in our Maude model of Walter [61] is specified as an object instance of the following subclass `Walter-Replica` of class `Replica` that adds 14 new attributes (only 4 shown below):

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

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

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

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

Example 3.2. Walter transactions can be modeled as object instances of the subclass `Walter-Txn` with four new attributes:

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

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

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

  Example 3.3. A Walter replica starts executing a transaction by moving the transaction `TID` in `gotTxns` (buffering transactions from clients) to `executing`:[2]

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

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

  Example 3.4. In Walter, when all operations of an executing read-only transaction have been performed, the proxy commits the transaction directly:

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

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

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

  Example 3.5. If either of the two conflict checks for fast commit fails, the executing Walter replica aborts the transaction:

```
crl [fast-commit-failed] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica |
        executing : < TID : Walter-Txn | operations : nil, writeSet : WS,
                                  readSet : RS, startVTS : VTS > TRANSES,
        aborted : TRANSES',  history : DS,  locked : LOCKS >
 =>
    < TABLE : Table | >
    < RID : Walter-Replica | executing : TRANSES,
                            aborted : TRANSES' < TID : Walter-Txn | > >
    if WS =/= empty /\ allLocalPreferred(WS,RID,REPLICA-TABLE)
       /\ (modified(WS,VTS,DS) or locked(WS,LOCKS)) .
```

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

  Example 3.6. Upon receiving the "disaster-safe durable" message, the remote Walter replica "commits" the transaction `TID` by setting its status to `true`:

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

These requirements can be easily met by many systems. The Maude models of the DTSs RAMP [58], Faster [57], Walter [61], ROLA [59], Jessy [60], and P-Store [74] can all be seen as instantiations of our modeling framework, with very small syntactic changes, such as defining transaction and replica objects as subclasses of `Txn` and `Replica`, changing the names of the attributes and sorts, etc. Google's Megastore [16] is a cloud storage system with transaction support; its Real-Time Maude model [45] can be adapted into our framework by removing the nondeterministic communication delays for messages. The Apache Cassandra [2] NoSQL key-value store can be seen as a transaction system where each transaction is a single operation; the Maude model of Cassandra in [63] can also be easily modified to fit within our modeling framework.

3.2 ADDING EXECUTION LOGS AND MONITORS

To formalize and analyze consistency properties of distributed transaction systems we add an "execution log" that is stored in a monitor and records the *history* of relevant events during a system execution. This section explains how this history recording can be added *automatically* to a model of a DTS that has been specified according to the requirements explained in Section 3.1.

3.2.1 Execution Log

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

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

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

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

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

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

3.2.2 Logging Execution History

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

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

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

The log is updated each time an interesting event (see Section 3.1.1) happens. Our tool identifies those events and *automatically* transforms their corresponding rewrite rules by adding and updating the monitor object.

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

Example 3.7. The rewrite rule in Example 3.3 where a Walter replica is served a transaction is modified by adding and updating the monitor object (in blue):

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

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

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

Example 3.8. The monitor object is added to the rewrite rule in Example 3.4 for committing a read-only transaction:

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

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

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

Example 3.9. The monitor object is added to the rewrite rule in Example 3.5 for aborting a transaction:

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

```
    < O@M : Monitor | clock : GT@M + 1 , log : LOG@M ,
               (TID |-> < RID, T@M, insert(RID,GT@M,VTS@M), false, RS, WS >)
    < TABLE : Table | >
    < RID : Walter-Replica | executing : TRANSES,
                             aborted : TRANSES' < TID : Walter-Txn | > >
  if WS =/= empty /\ allLocalPreferred(WS,RID,REPLICA-TABLE)
     /\ (modified(WS,VTS,DS) or locked(WS,LOCKS)) .
```

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

Example 3.10. The rewrite rule from Example 3.6 for committing a transaction remotely is transformed into the following rewrite rule:

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

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

3.2.3 Implementing the Monitoring Mechanism

We have formalized and implemented the transformation from a Maude specification of a DTS into one having a monitor as a meta-level function monitorRules : Module -> Module in Maude. Specifically, the transformation takes as input Maude object-oriented modules satisfying the requirements in Section 3.1 and including the rewrite rules specifying the system dynamics, instruments the "interesting" rewrites rules at the meta-level according to the monitoring mechanism, and outputs a new *flattened* model of the system and the monitor.

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

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


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

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

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

Example 3.11. The definition of `monitorTerms` in the case of COMMIT is defined by the following meta-level equation:

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

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

3.3 FORMALIZING CONSISTENCY MODELS IN MAUDE

This section formalizes the consistency properties informally described in Section 2.4 as functions on the "history log" of a *completed* run.

Read Committed (RC). (A transaction cannot read any writes by uncommitted transactions.) Note that standard definitions for single-version databases disallow reading versions that are not committed at the time of the read. We follow the definition for multi-versioned systems by Adya, summarized by Bailis et al. [14], that defines the *RC* property as follows: (i) a committed transaction cannot read a version that was written by an aborted transaction; and (ii) a transaction cannot read *intermediate values*: that is, if T writes two versions `< X,V >` and `< X,V' >` with `V < V'`, then no $T' \neq T$ can read `< X,V >`.

The first equation defining the function `rc`, specifying when *RC* holds, checks whether some (committed) transaction `TID1` reads version `V` of key `X` (i.e., `< X,V >` is in `TID`'s read set `< X,V > , RS`, where `RS` matches the rest of `TID`'s read set), and this version `V` was written by some transaction `TID2` that was never committed (i.e., `TID2`'s commit flag is `false`, and its write set is `< X,V > , WS'`). The second equation checks whether there was an *intermediate* read of a version `< X,V >` that was overwritten by the same transaction `TID2` that wrote the version:[3]

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

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

Read Atomicity (RA). A system guarantees *RA* if it prevents fractured reads and prevents transactions from reading uncommitted or aborted data. A transaction T_j exhibits *fractured reads* if transaction T_i writes versions x_m and y_n, T_j reads version x_m and version y_k, and $k < n$ [14]. The function `fracRead` checks whether there are fractured reads in the log. There is a fractured read if a transaction `TID2` reads `X` and `Y`, transaction `TID1` writes `X` and `Y`, `TID2` reads the version `VX` of `X` written by `TID1`, and reads a version `VY'` of `Y` written *before* `VY` (`VY' < VY`):

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

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

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

Cursor Stability (CS) [38] strengthens *RC* by also preventing lost updates (LU). LU can only happen with multiple *conditional writes* (i.e., a transaction first fetches some value of an data item, and then updates it with a new value) fetching the same data. Once one of those transactions commits its writes, the others must be aborted. The function `lu` captures the case when there are two committed transactions `TID1` and `TID2`, both of which read the same version `V` of data item `X` (i.e., `< X,V >` is in the read sets of both `TID1` and `TID2`), and commit on the same key (i.e., the two transactions wrote `< X,VX >` and `< X,VX' >`, respectively):

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

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

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

Update Atomicity (UA) [29, 59] provides read atomicity and prevents lost updates:

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

Snapshot Isolation (SI) is defined by two properties in [85]:

- SI-1 (snapshot read): All operations in a transaction read the most recent committed version as of time when the transaction began.
- SI-2 (no write-write conflicts): The write sets of each pair of committed concurrent[4] transactions must be disjoint.

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

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

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

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

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

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

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

Parallel snapshot isolation (PSI) is given by three properties [85]:

- PSI-1 (site snapshot read): All operations read the most recent committed version at the transaction's site as of the time when the transaction began.
- PSI-2 (no write-write conflicts): The write sets of each pair of committed *somewhere-concurrent*[5] transactions must be disjoint.
- PSI-3 (commit causality across sites): If a transaction T_1 commits at a site S before a transaction T_2 starts at site S, then T_1 cannot commit after T_2 at any site.

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

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

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

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

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

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

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

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

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

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

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

Non-monotonic snapshot isolation (NMSI) is the same as *PSI* except that a transaction may read a version committed even after the transaction begins [8]. *NMSI* can therefore be defined as the conjunction of PSI-2 and PSI-3:

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

Serializability (SER) means that the concurrent execution of transactions is equivalent to executing them in some (non-overlapping in time) sequence [75].

A formal definition of *SER* is based on *direct serialization graphs* (DSGs): an execution is serializable if and only if the corresponding DSG is acyclic. Each node in a DSG corresponds to a committed transaction, and directed edges in a DSG correspond to the following types of direct dependencies [4]:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where the function `committedBetween` returns true iff (for if and only if) there is a version committed between the two versions in the version order. It is defined as:

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

If there are no more edges to be added for the current version read, we move to the next version read:

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

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

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

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

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

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

If there are no more edges to be added for the current version written, we move to the next version written:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4 FORMAL ANALYSIS OF CONSISTENCY PROPERTIES

3.4.1 Parametric Generation of Initial States

Explicit-state model checkers like Maude are typically quite expressive but only analyze the system from a *single* initial configuration. To increase coverage, we would like to model check system models for *all possible* configurations satisfying certain bounds, for example having j transactions and k clients. Despite the wealth of Maude applications, we are not aware of any work doing such comprehensive model checking in Maude. We therefore present a general technique in Maude for model checking a system from a range of different initial configurations.

Specifically, we introduce a new operator `init` so there is a one-step rewrite `init(`*params*`)` $\longrightarrow c_0$ for each possible initial configuration c_0 satisfying the given parameter requirements, and declare a sort for *sets* of configurations:

```
sort ConfigSet .  subsort Configuration < ConfigSet .
op empty : -> ConfigSet .
op _;_ : ConfigSet ConfigSet -> ConfigSet [assoc comm id: empty] .
```

We define a function

```
op initAux : s1 ... sn sn+1 ... sn+m  -> ConfigSet .
```

such that s_1 ... s_n are the sorts for the user-specified parameters *params*, and s_{n+1} ... s_{n+m} those for the auxiliary parameters *params*′, and then `initAux(`*params*`,`*params*′`)` generates all possible initial states for such parameters. We also add the following rewrite rule to our model:

```
var C : Configuration .  var CS : ConfigSet .
crl [init] : init(params) => C if C ; CS := initAux(params,params') .
```

`init`'s parameters specify the number of each of the following parameters: read-only, write-only, and read-write transactions; operations for each type of transaction; clients; servers; keys; and replicas per key.

Here we only illustrate how to generate replicas, and how to replicate keys. The entire specification is given at `https://github.com/siliunobi/cat`.

We start by generating the replica table and key-variable pairs. `keyVars` consists of "`;`"-separated key-variable pairs $< k_1, var_1 >$; ... ; $< k_n, var_n >$, each of which has a key k and the associated local variable var. The function `kvars` extracts `KEYS` key-variable pairs from `keyVars`:

```
--- generate table and key-var pairs:
crl initAux(RTX,WTX,RWTX,ROP,WOP,RWOP,CLS,SVS,KEYS,RF,none)
 => $initAux(RTX,WTX,RWTX,ROP,WOP,RWOP,CLS,SVS,KVARS,genKeyVarSet(KVARS),RF,
    < 0 : Table | table : initTable(KVARS) >)
 if KVARS := kvars(KEYS,keyVars) .
```

The function `initTable` initializes the replica table for each key with its replicas of `nil`:

```
--- initialize table with generated keys:
op initTable : KeyVars -> ReplicaTable .
op $initTable : KeyVars ReplicaTable -> ReplicaTable .
eq initTable(KVARS) = $initTable(KVARS,[emptyTable]) .
eq $initTable((< K,VAR > ; KVARS),[KEYREPLICAS]) =
     $initTable(KVARS,[sites(K,nil) ;; KEYREPLICAS]) .
eq $initTable(noKeyVar,[KEYREPLICAS]) = [KEYREPLICAS] .
```

We now generate replicas, assign the keys to them, and update the replica table accordingly.

Example 3.12. Walter replicas are generated with the attributes in Example 3.1:

```
--- generate Walter replicas:
rl $initAux(RTX,WTX,RWTX,ROP,WOP,RWOP,CLS,s PARS,KVARS,KS,RF,C)
=> $initAux(RTX,WTX,RWTX,ROP,WOP,RWOP,CLS,PARS,KVARS,KS,RF,C
     < s PARS : Walter-Replica | executing : none, committed : none,
            aborted : none, decided : empty, store : empty, sqn : 0,
            locked : empty, votes : empty, ... >) .

--- assign keys to Walter replicas and update table accordingly:
```

```
crl $initAux(RTX,WTX,RWTX,ROP,WOP,RWOP,CLS,0,(< K,VAR > ; KVARS),KS,s RF,
    < RID : Walter-Replica | store : VS >
    < 0 : Table | table : [sites(K,RIDS) ;; KEYREPLICAS] > C)
 => $initAux(RTX,WTX,RWTX,ROP,WOP,RWOP,CLS,0,(< K,VAR > ; KVARS),KS,RF,
    < RID : Walter-Replica | store : (VS,K |-> (< [0],version(0,0) >)) >
    < 0 : Table | table : [sites(K,RIDS RID) ;; KEYREPLICAS] > C)
    if not $hasMapping(VS,K) .
```

Note that, to assign a key to a replica, we *nondeterministically* add a replica `RID` to key `K`'s replicating sites. Once the key has been assigned to *replication factor* replicas, we continue to the next key by resetting the replication factor to `rf`:

```
--- next key
rl $initAux(RTX,WTX,RWTX,ROP,WOP,RWOP,CLS,0,(< K,VAR > ; KVARS),KS,0,C)
=> $initAux(RTX,WTX,RWTX,ROP,WOP,RWOP,CLS,0,KVARS,KS,rf,C) .
```

3.4.2 The CAT Tool

We have implemented the *Consistency Analysis Tool* (CAT) that automates the new method for model checking consistency properties explained in this chapter. CAT takes as input:

- A Maude model of the DTS specified as explained in Section 3.1.
- The desired *number* of each of the following initial state parameters: read-only, write-only, and read-write transactions; operations for each type of transaction; clients; servers; keys; and replicas per key. The tool analyzes the desired property for *all* initial states having the desired number of each of these parameters.
- The consistency property to be analyzed.

Given these inputs, CAT performs the following steps:

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

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

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

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

Table 3.1: Model Checking Results w.r.t. Consistency Properties. "✓", "×", and "-" refer to satisfying or violating the property, and "not applicable", respectively.

Maude Model	LOC	Consistency Property								
		RC	RA	CS	UA	NMSI	PSI	SI	SER	SSER
RAMP-F [58]	330	✓	✓	×	×	-	-	×	×	×
RAMP-F+1PW [57]	302	✓	✓	×	×	-	-	×	×	×
RAMP-F+FC [57]	305	✓	✓	×	×	-	-	×	×	×
RAMP-F¬2PC [57]	320	✓	×	×	×	-	-	×	×	×
RAMP-S [58]	255	✓	✓	×	×	-	-	×	×	×
RAMP-S+1PW [57]	237	✓	✓	×	×	-	-	×	×	×
RAMP-S¬2PC [57]	248	✓	×	×	×	-	-	×	×	×
Faster [57]	300	✓	×	×	×	-	-	×	×	×
ROLA [59]	411	✓	✓	✓	✓	-	-	×	×	×
Jessy [60]	413	✓	✓	✓	✓	✓	×	×	×	×
Walter [61]	830	✓	✓	✓	✓	✓	✓	×	×	×
P-Store [74]	438	✓	✓	✓	✓	✓	✓	✓	✓	×
NO_WAIT (Chapter 7)	600	✓	✓	✓	✓	-	-	✓	✓	×

We have applied our tool to 13 Maude models of state-of-the-art DTSs (different variants of RAMP and Walter, ROLA, Jessy, P-Store, and NO_WAIT) against all nine properties (three case studies, namely RAMP, Walter, and ROLA, will be presented in detail in Chapter 4, 5, and 6, respectively). Table 3.1 summarizes our experience with CAT: all model checking results are as expected. It is worth remarking that our automatic analysis found all the violations of properties that the respective systems should violate. There are also some cases where model checking is not applicable ("-" in Table 3.1): some system models do not include a mechanism for committing a transaction on remote servers (i.e., no commit time on any remote server is recorded by the monitor). Thus, model checking *NMSI* or *PSI* is not applicable.

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

Data Availability. The system models, properties specifications, and tool implementation are available at `https://github.com/siliunobi/cat`.

3.5 RELATED WORK

Formalizing Consistency Properties in a Single Framework. Adya [4] uses dependencies between reads and writes to define different isolation models in database systems. Bailis et al. [14] adopt this model to define read atomicity. Burckhardt et al. [26] and Cerone et al. [29] propose axiomatic specifications of consistency models for transaction systems using visibility and arbitration relationships. Shapiro et al. [83] propose a classification along three dimensions (total order, visibility, and transaction composition) for transactional consistency models. Crooks et al. [37] formalize transactional consistency properties in terms of observable states from a client's perspective. On the non-transactional side, Burckhardt [25] focuses on session and eventual consistency models. Viotti *et al.* [89] expand his work by covering more than 50 non-transactional consistency properties. Szekeres *et al.* [86] propose a unified model based on result visibility to formalize both transactional and non-transactional consistency properties.

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

Model Checking Distributed Transaction Systems. There is very little work on model checking state-of-the-art DTSs, maybe because the complexity of these systems requires expressive formalisms. Engineers at Amazon Web Services successfully used TLA+ to model check key algorithms in Amazon's Simple Storage Systems and DynamoDB database [73]; however, they do not state which consistency properties, if any, were model checked. The designers of the TAPIR transaction protocol have specified and model checked correctness properties of their design using TLA+ [95]. The IronFleet framework [48] combines TLA+

analysis and Floyd-Hoare-style imperative verification to reason about protocol-level concurrency and implementation complexities, respectively. Their methodology requires "considerable assistance from the developer" to perform the proofs. Cai [27] proposes some basic patterns for modeling real-time transactions, and uses Timed Computation Tree Logic (TCTL) to specify the timeliness and ACID properties. Li [54] models three multi-version concurrency control mechanisms using the patterns in [27], and verifies transaction timeliness and *SER* in UPPAAL. These case studies are based on timed automata, and none of them checks state-of-the-art DTSs.

Distributed model checkers [51, 93] are used to model check *implementations* of distributed systems such as Cassandra, ZooKeeper, the BerkeleyDB database and a replication protocol implementation.

Our previous work [45, 46, 63, 55, 74, 57, 59, 61, 60, 22] specifies and model checks *single* DTSs and consistency properties in different ways, as opposed to in a single framework that, furthermore, automates the "monitoring" and analysis process.

Other Formal Reasoning about Distributed Database Systems. Cerone et al. [30] develop a new characterization of *SI* and apply it to the static analysis of DTSs. Bernardi et al. [19] propose criteria for checking the robustness of transactional programs against consistency models. Bouajjani et al. [23] propose a formal definition of eventual consistency, and reduce the problem of checking eventual consistency to reachability and model checking problems. Gotsman *et al.* [44] propose a proof rule for reasoning about non-transactional consistency choices.

There is also work [91, 53, 82] that focuses on specifying, implementing and verifying distributed systems using the Coq proof assistant. Their executable Coq "implementations" can be seen as executable high-level formal specifications, but the theorem proving requires nontrivial user interaction.

Finally, the authors in [40] apply both model checking and meta-programming techniques in Maude to a distributed snapshot algorithm and its reachability property, but they do not consider either transaction systems or consistency properties.

3.6 CONCLUDING REMARKS

In this chapter we have provided an object-based framework for formally modeling distributed transaction systems (DTSs) in Maude, have explained how such models can be automatically instrumented to record relevant events during a run, and have formally defined a wide range of consistency properties on such histories of events. We have implemented a tool which automates the entire instrumentation and model checking process. Our framework is very general: we could easily adapt previous Maude models of state-of-the-art DTSs such as P-Store, RAMP, Walter, Jessy, ROLA, and NO_WAIT to our framework. We then model checked the DTSs w.r.t. all the consistency properties for all initial states with 4 transactions, 2 sites, and so on. This analysis was sufficient to differentiate all these DTSs according to which consistency properties they satisfy.

CHAPTER 4: CASE STUDY: RAMP TRANSACTION SYSTEMS

Database systems can provide scalability by partitioning data across several database partitions. Multipartition transactions are often expensive due to coordination-intensive mechanisms, which, however, provide useful semantics for correct behaviors of such transactions. Several efforts have recently emerged to reach an acceptable trade-off between consistency and performance. One of the most promising is the RAMP transaction system proposed by Bailis *et al.* [15, 14]. RAMP allows clients to execute transactions (like in relational databases) in NoSQL-like storage systems. It offers a correctness property called "Read Atomicity" (RA) which ensures that a given transaction's updates are either all visible or not visible at all, to other transactions. For example, if A and B become "friends" in one transaction, then other transactions should *not* see that A is a friend of B but that B is not a friend of A: either both relationships are visible or neither of them is.

The main contributions of this chapter are fourfold: (i) we instantiate the ideas of Chapter 3 by showing how we can model RAMP and its extensions in the CAT framework; (ii) we verify the consistency properties of RAMP and its extensions by model checking in CAT; (iii) we transform the RAMP designs specified in the CAT framework into probabilistic Maude models; and (iv) we evaluate eight RAMP designs w.r.t. the performance and consistency measures for different experimental parameters and workloads. Regarding (iv), our results: (a) are consistent with the experimental results obtained by the RAMP developers for their implemented designs; (b) confirm the conjectures made by the RAMP developers for their other three unimplemented designs; and (c) uncover some promising new RAMP-like designs that seem attractive for some applications.

This chapter is structured as follows. Sections 4.1 and 4.2 give some background on RAMP, its variants, and our new RAMP design alternative. Sections 4.3 and 4.4 present the Maude models of RAMP and its extensions in the CAT framework, and show how we can use them to model check their consistency properties. Section 4.5 shows how we can specify our RAMP designs as probabilistic rewrite theories. Sections 4.6 and 4.7 explain how we can monitor the system executions, and how we can evaluate the performance of the designs based on the recorded log for different performance parameters and workloads. Section 4.8 gives some concluding remarks.

Acknowledgements. This chapter is based on the work [58, 57, 62] supported in part by NSF CNS 1409416, NSF CNS 1319527, NSF CCF 0964471, AFOSR/AFRL FA8750-11-2-0084, NRL contract N00173-17-1-G002, and NSFC Project No. 61872146.

4.1 RAMP TRANSACTIONS

To deal with large amounts of data, distributed databases *partition* their data across multiple servers. However, many systems do not provide useful transactional semantics for operations accessing multiple partitions, since the latency needed to ensure correct multi-partition transactional access is often high. Therefore, trade-offs that combine efficiency with weaker transactional guarantees are needed.

In [14], Bailis *et al.* propose a new isolation model, called *read atomic* (RA) isolation, and *Read Atomic Multi-Partition* (RAMP) transactions, that together combine efficient multi-partition operations and partial fault tolerance with some transactional guarantee: either all or none of a transaction's updates are visible to other transactions.

RAMP writers attach metadata to each write and the reads use this metadata to get the correct version. There are three versions of RAMP: RAMP-Fast, RAMP-Small, and RAMP-Hybrid. The write protocols in these algorithms only differ in the amount of attached metadata. To guarantee that all partitions perform a transaction successfully or that none do, RAMP performs two-phase writes by using the two-phase commit protocol (2PC) [20]:[1] 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 which contains the highest-timestamped committed version of each item. The RAMP algorithms described in [14] only deal with read-only and write-only transactions. This dissertation focuses on RAMP-Fast and RAMP-Small, which lie at the end points. Below we only detail the RAMP-Fast algorithm, and refer to [14] for the details of the other RAMP algorithms.

RAMP-Fast (abbreviated RAMP-F). In RAMP-Fast, read transactions first fetch the highest-timestamped committed *version* of each requested data item from the corresponding partition, and then decide if they have missed any version that has been prepared but not yet committed. The timestamp and metadata from each version read produces a mapping from items to timestamps that represent the highest-timestamped write for each transaction, appearing in the first-round read set. If the reader has a lower timestamp version than indicated in the mapping for that item, a second-round read will be issued to fetch the missing version. Once all the missing versions have been fetched, the client can return the resulting set of versions, which include both the first-round reads as well as any missing versions fetched in the second round of reads. The pseudo-code of RAMP-Fast in [14] is shown in Appendix A.

[1]2PC is an atomic commitment protocol in database systems, which coordinates all the processes that participate in a distributed atomic transaction on whether to commit or abort the transaction.

RAMP-Small (abbreviated RAMP-S). RAMP-Small read transactions proceed by first fetching the highest committed *timestamp* of each requested data item; the readers then send the entire set of those timestamps in a second message. The highest-timestamped version that also exists in the received set will be returned to the reader by the corresponding partition. RAMP-Small transactions require two round-trip times (RTTs) for reads and writes. RAMP-Small writes only store the transaction timestamp, instead of attaching the entire write set to each write.

Extensions of RAMP. The paper [14] briefly discusses the following extensions and optimizations of the basic RAMP algorithms, but without giving any details:

- *RAMP with one-phase writes* (RAMP-F+1PW and RAMP-S+1PW), where writes only require one *prepare* phase, as the client can execute the *commit* phase asynchronously.
- *RAMP with faster commit detection* (RAMP-F+FC). If a server returns a version with the timestamp fresher than the highest committed version of the item, then the server can mark the version as committed. This allows faster updates to correct versioning and thus fewer round trip time delays.

4.2 NEW RAMP-LIKE DESIGNS

In this section we propose two new RAMP-like designs, both of which trade consistency off for higher performance.

RAMP without two-phase commit (RAMP¬2PC). RAMP uses 2PC to ensure that all partitions successfully execute a transaction or that none do. Specifically, writes start to commit only after all of them are prepared on the partitions. This results in high latency, even "resource leak" on partitions during failures [14], since one blocked write will cause the transaction to stall. RAMP¬2PC decouples 2PC from RAMP by committing a prepared version directly without waiting for all of the writes to be prepared.

RAMP-Faster (abbreviated Faster). Both RAMP and RAMP¬2PC require two RTTs to commit a write transaction. To further optimize the performance we propose the RAMP-Faster design (based on RAMP-Fast) that also decouples two-phase commit, but commits a write transaction in only one RTT.

In RAMP-Fast, upon receiving a *prepare* message, the partition adds the timestamped write to its local database, and upon receiving the *commit* message, updates an index containing the highest-timestamped committed version of each item. Instead, in Faster, a partition performs both operations upon receiving the *prepare* message, and hence requires only one RTT. Note that all information required to complete the two operations is provided by the *prepare* message: Faster does not need to store more data than RAMP-Fast.

Since each write in Faster needs only one RTT, it should incur lower latency per transaction and provide higher throughput. Since writes are faster, it also seems reasonable to conjecture that there is a higher chance that reads fetch the latest write; this means that Faster should provide better strong consistency (i.e., reads reading the “latest writes) than other RAMP designs. Even though Faster does not guarantee read atomicity, as the client does not ensure that each partition has received the *prepare* message before issuing the *commit* message, it would be interesting to check whether Faster indeed provides better performance, and a high degree of read atomicity, for classes of transactions encountered in practice. If so, Faster would be an attractive option for multi-partition transactions where a high degree of read atomicity, good consistency properties, and low latency are desired.

4.3 MODELING RAMP AND ITS VARIANTS IN CAT

This section presents formal models of RAMP and its variants in the CAT framework described in Chapter 3. We show the specification of RAMP-Fast in detail, and only show the main differences for the other RAMP algorithms.

4.3.1 Data Types, Objects, and Messages

Data Types. A *version* is a timestamped version of a data item (or key) and is modeled as a four-tuple `version(`*key*`,`*value*`,` *timestamp*`,`*metadata*`)` consisting of the key, its value, and the version's timestamp and metadata. A timestamp is modeled as a pair `ts(`*Oid*`,`*sqn*`)` consisting of a Replica's identifier *Oid* and a local sequence number *sqn* that together uniquely identify a write transaction. Metadata are modeled as a set of keys, denoting, for each key, the other keys that are written in the same transaction. For example, if a transaction writes keys x, y, and z, then versions of x have as metadata the set $\{y, z\}$.

```
sorts Key Value Timestamp Version KeySet Versions
      KeyTimestampEntry KeyTimestamps .
subsort Key < KeySet .
```

```
  subsort KeyTimeEntry < KeyTimestamps .
  subsort Version < Versions .

  op ts : Oid Nat -> Timestamp .
  op version : Key Value Timestamp KeySet -> Version [ctor] .

  op empty : -> KeySet [ctor] .
  op _,_ : KeySet KeySet -> KeySet [ctor assoc comm id: empty] .

  op empty : -> KeyTimestamps [ctor] .
  op _|->_ : Key Timestamp -> KeyTimestampEntry [ctor] .
  op _,_ : KeyTimestamps KeyTimestamps ->
               KeyTimestamps [ctor assoc comm id: empty] .

  op empty : -> Versions [ctor] .
  op _,_ : Versions Versions -> Versions [ctor assoc id: empty] .
```

A set of keys of sort `KeySet` is built from singleton sets (identified with keys of sort `Key` by means of a `subsort` declaration) with an associative, commutative, and idempotent union operator _,_ having `empty` as its identity element. A set of entries, or mappings from keys to timestamps, of sort `KeyTimestamps` is built from singleton sets (identified with entries of sort `KeyTimestampEntry` by means of a `subsort` declaration) with an associative, commutative, and idempotent union operator _,_ having `empty` as identity element. A set of versions of sort `Versions` is built from singleton lists (identified with versions of sort `Version` by means of a `subsort` declaration) with an associative concatenation operator _,_ with identity `empty`.

The sort `OperationList` represents lists of read and write operations as terms such as (x `:= read` $k1$) (y `:= read` $k2$) `write(`$k1, x+y$`)`, where `LocalVar` denotes the "local variable" that stores the value of the key read by the operation, and `Expression` is an expression involving the transaction's local variables:

```
  sorts Expression LocalVar LocalVarEntry LocalVars Operation .
  subsort Operation < OperationList .
  subsort LocalVarEntry < LocalVars .

  op write : Key Expression -> Operation [ctor] .
  op _:=read_ : LocalVar Key -> Operation [ctor] .
```

```
op nil : -> OperationList [ctor] .
op __ : OperationList OperationList -> OperationList [ctor assoc id: nil] .

op empty : -> LocalVars [ctor] .
op _|->_ : LocalVal Value -> LocalVarEntry [ctor] .
op _,_ : LocalVars LocalVars -> LocalVars [ctor assoc comm id: empty] .
```

A list of operations of sort `OperationList` is built from singleton lists (identified with operations of sort `Operation` by means of a `subsort` declaration) with an associative concatenation operator `__` with identity `nil`. A set of entries, or mappings from local variables to their values, of sort `LocalVars` is built from singleton sets (identified with entries of sort `LocalVarEntry` by means of a `subsort` declaration) with an associative, commutative, and idempotent union operator `_,_` with identity `empty`.

We define a collection of votes of sort `Vote` as a multiset (built with the associative and commutative operator `_;_` with identity `noVote`) of votes, where each vote, as a triple `vote(`*`txn`*`,`*`part`*`,`*`result`*`)`, indicates the voting result by some replica for a certain transaction:

```
sort Vote .
op noVote : -> Vote [ctor] .
op vote : Tid Oid Bool -> Vote [ctor] .
op _;_ : Vote Vote -> Vote [ctor assoc comm id: noVote] .
```

The data type `TxnOidSet` is defined for the situation when a replica is waiting for messages such as votes from other replicas w.r.t. some transaction:

```
sorts OidSet TxnOidSet .
subsort Oid < OidSet .

op empty : -> OidSet [ctor] .
op _,_ : OidSet OidSet -> OidSet [ctor assoc comm id: empty] .

op empty : -> TxnOidSet [ctor] .
op Oids : Tid OidSet -> TxnOidSet [ctor] .
op _;_ : TxnOidSet TxnOidSet -> TxnOidSet [ctor assoc comm id: empty] .
```

where `OidSet` is defined as a set of replicas (more precisely, each replica is represented by its identifier of sort `Oid`), while `TxnOidSet` is a mapping (set of pairs) from transaction identifiers (of sort `Oid`) to sets of replicas.

Transactions. RAMP transactions can be modeled as object instances of the subclass `RAMP-Txn` with four new attributes:

```
class RAMP-Txn | operations : OperationList,  localVars : LocalVars,
                 latest : KeyTimestamps,  txnSqn : Nat .
subclass RAMP-Txn < Txn .
```

The `operations` attribute denotes the transaction's operations. `localVars` maps the transaction's local variables to their current values. `latest` stores the local view as a mapping from keys to their respective latest committed timestamps. `txnSqn` stores the transaction's sequence number.

Replicas. A replica in our Maude model of RAMP is modeled as an object instance of the following subclass `RAMP-Replica` of class `Replica` that adds nine new attributes:

```
class RAMP-Replica | datastore : Versions,       sqn : Nat,
                     gotTxns : ObjectList,       latestCommit : KeyTimestamps,
                     votes : Vote,               voteSites : TxnOidSet,
                     1stGetSites : TxnOidSet,    2ndGetSites : TxnOidSet,
                     commitSites : TxnOidSet .
subclass RAMP-Replica < Replica .
```

The `datastore` attribute represents the replica's local database as a set of versions for each key stored at the replica. The attribute `latestCommit` maps each key to the timestamp of its last committed version. `sqn` refers to the local sequence number. The attributes `gotTxns` stores the transaction object(s) waiting to be executed. The attribute `votes` stores the votes from the replicas which participate in the two-phase commit. The remaining attributes `voteSites`, `commitSites`, `1stGetSites`, and `2ndGetSites` store, respectively, the replicas from which the executing replica is awaiting votes, committed acks, first-round get replies, and second-round get replies.

The state also contains a "table" object of class `Table` mapping each data item to the replica storing the item:

```
class Table | table : ReplicaTable .
```

where the table (of sort `ReplicaTable`) stored inside a `Table` object is built with the associative and commutative operator `_;;_` as a set of mappings, each of which is a pair `sites(`*`key`*`,`*`part`*`)`.

Messages travel between clients and replicas, and have the form:

`msg` *`msgContent`* `from` *`sender`* `to` *`receiver`*

where the message content ***`msgContent`*** is defined in our RAMP-Fast model as follows:

- `prepare(`*txn*`,`*version*`)` sends a version from a write-only transaction to its replica;
- `prepare-reply(`*txn*`,`*vote*`)` is the reply to the corresponding prepare message, where *vote* tells whether this replica can commit the transaction;
- `commit(`*txn*`,`*ts*`)` marks the versions with timestamp *ts* as committed;
- `committed(`*txn*`)` is the reply to `commit`;
- `get(`*txn*`,`*key*`,`*ts*`)` asks for the highest-timestamped committed version or a missing version for *key* by timestamp *ts*;
- `response1(`*txn*`,`*version*`)` responds to first-round `get` request;
- `response2(`*txn*`,`*version*`)` responds to second-round `get` requests.

Initial State. The following shows an automatically generated initial state (with some parts replaced by '...') with two replicas, `r1` and `r2`, that are coordinators for, respectively, transactions `t1` and `t2`. `r1` stores the data items `x` and `z`, and `r2` stores `y`. Transaction `t1` is the read-only transaction `(xl :=read x) (yl :=read y)`, while transaction `t2` is a write-only transaction `write(y,3) write(z,8)`. The states also include a table which assigns to each data item the replica storing it. Initially, the value of each item is `[0]`; the version's timestamp is empty (`eptTS`), and metadata is an empty set:

```
eq init =
< tb : Table | table : [sites(x,r1) ;; sites(y,r2) ;; sites(z,r1)] >
< r1 : RAMP-Replica |
    gotTxns : < t1 : RAMP-Txn |
    operations : ((xl :=read x) (yl :=read y)), readSet : empty, writeSet : empty,
    latest : empty, localVars : (xl |-> [0], yl |-> [0]), txnSqn : 0 >,
    datastore : (version(x,[0],eptTS,empty) version(z,[0],eptTS,empty)),
    sqn : 1, executing : none, committed : none, latestCommit : empty,
    votes : noVote, voteSites : empty, 1stGetSites : empty,
    2ndGetSites : empty, commitSites : empty >
< r2 : RAMP-Replica |
    gotTxns : < t2 : RAMP-Txn | operations : (write(y,3) write(z,8)), ... >,
    datastore : version(y,[0],eptTS,empty), ... >
```

4.3.2 Formalizing RAMP-Fast

This section formalizes the dynamic behaviors of RAMP-Fast using rewrite rules.[2] We also refer to the corresponding lines of code in the description in Fig. A.1. The entire specification is given at `https://github.com/siliunobi/cat`.

Starting a New Transaction (Lines 14–19 for writes and lines 22–26 for reads). A replica starts executing a transaction by moving the first transaction (`TID`) in `gotTxns` to `executing`. If the new transaction is a write-only transaction (`write-only(OPS)`), the replica: (i) uses the function `genPuts` to generate all `prepare` messages; (ii) uses a function `prepareSites` to remember the sites `RIDS` from which it awaits votes for transaction `TID` in the `voteSites` attribute; and (iii) increments its local sequence number by one:

```
crl [start-wo-txn] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica |
              gotTxns : (< TID : RAMP-Txn | operations : OPS, localVars : VARS,
                                                    txnSqn : N > ;; TXNS),
              executing : TRANSES,   sqn : SQN,   voteSites : VSTS >
   =>
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica |
              gotTxns : TXNS,
              executing : < TID : RAMP-Txn | operations : OPS, localVars : VARS,
                                                     txnSqn : SQN' > TRANSES,
              sqn : SQN',   voteSites : (VSTS ; addrs(TID,RIDS)) >
    genPuts(OPS,RID,TID,SQN',VARS,REPLICA-TABLE)
    if SQN' := SQN + 1 /\  write-only(OPS) /\
       RIDS := prepareSites(OPS,RID,REPLICA-TABLE) .
```

The above function `genPuts` is defined as follows:

```
op genPuts : OperationList Oid Oid Nat LocalVars ReplicaTable -> Config .
op $genPuts : OperationList Oid Oid Nat LocalVars ReplicaTable
                                                 OperationList -> Config .
eq genPuts(OPS,RID,TID,SQN,VARS,REPLICA-TABLE)
 = $genPuts(OPS,RID,TID,SQN,VARS,REPLICA-TABLE,OPS) .
eq $genPuts((write(K,EXPR) OPS),RID,TID,SQN,VARS,REPLICA-TABLE,
```

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

```
    (OPS' write(K,EXPR) OPS''))
 = $genPuts(OPS,RID,TID,SQN,VARS,REPLICA-TABLE,(OPS' write(K,EXPR) OPS''))
    msg prepare(TID,version(K,eval(EXPR,VARS),ts(RID,SQN),md(OPS' OPS'')))
    from RID to preferredSite(K,REPLICA-TABLE) .
eq $genPuts(((X :=read K) OPS),RID,TID,SQN,VARS,REPLICA-TABLE,OPS')
 = $genPuts(OPS,RID,TID,SQN,VARS,REPLICA-TABLE,OPS') .
eq $genPuts(nil,RID,TID,SQN,VARS,REPLICA-TABLE,OPS') = null .
```

Otherwise, if the first transaction in gotTxns is a read-only transaction, the replica updates 1stGetSites instead to keep track of the replicas from which it receives the versions from the first-round gets. Similar to genPuts, the function genGets generates all get messages for the keys concerned by TID. The expression 1stSites gives the corresponding replicas for those keys:

```
crl [start-ro-txn] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica | gotTxns :
                (< TID : RAMP-Txn | operations : OPS, latest: empty > ;; TXNS),
            executing : TRANSES,
            1stGetSites : 1STGETS >
  =>
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica | gotTxns : TXNS,
            executing : < TID : RAMP-Txn | operations: OPS,
                                           latest : vl(OPS) > TRANSES,
            1stGetSites : (1STGETS ; addrs(TID,RIDS)) >
    genGets(OPS,RID,TID,REPLICA-TABLE)
    if (not write-only(OPS)) /\
       RIDS := 1stSites(OPS,RID,REPLICA-TABLE) .
```

Receiving Prepare Messages (Lines 3–5). When a Replica receives a prepare message for a write-only transaction, the replica simply adds the received version to its local datastore. The out-going messages always consider successful preparations:

```
rl [receive-prepare-wo] :
   msg prepare(TID,VER) from RID' to RID
   < RID : RAMP-Replica | datastore: VS >
 =>
   < RID : RAMP-Replica | datastore: (VS, VER) >
   msg prepare-reply(TID,true) from RID to RID' .
```

Receiving Prepared Messages (Lines 20–21). Upon receiving a "true" vote, the replica first checks whether all votes have now been collected. The expression `VSTS'[TID]` extracts for `TID` the remaining replicas from which it is awaiting votes. If all received votes are "yes," the replica starts to commit `TID` at the associated replicas by invoking `genCommits` to generate all commit messages with the commit timestamp including the current sequence number `SQN`. The replica also adds to `commitSites` the replicas from which it is awaiting `committed` messages to commit the transaction:

```
crl [receive-prepare-reply-true-executing] :
    msg prepare-reply(TID,true) from RID' to RID
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica | executing : < TID : RAMP-Txn | operations : OPS,
                                  txnSqn : SQN > TRANSES,
                      voteSites : VSTS, commitSites : CMTS >
  =>
    < TABLE : Table | table : REPLICA-TABLE >
    if VSTS'[TID] == empty  --- all votes received and all yes!
    then < RID : RAMP-Replica | executing : < TID : RAMP-Txn | operations : OPS,
                                  txnSqn : SQN > TRANSES,
                           voteSites : VSTS',
                           commitSites : (CMTS ; addrs(TID,RIDS)) >
         genCommits(TID,SQN,RIDS,RID)
    else < RID : RAMP-Replica | executing : < TID : RAMP-Txn | operations: OPS,
                                  txnSqn : SQN > TRANSES,
                           voteSites : VSTS', commitSites : CMTS >
    fi
    if VSTS' := remove(TID,RID',VSTS) /\
       RIDS := commitSites(OPS,RID,REPLICA-TABLE) .
```

Receiving Commit Messages (Lines 6–8). Upon receiving a `commit` message, the replica invokes the function `cmt` to commit the transaction. `cmt` looks up `LC` for the latest committed version's timestamp, and updates the latest committed version if `TS` is higher. A `committed` message is then sent back to confirm the commit:

```
rl [receive-commit] :
   msg commit(TID, TS) from RID' to RID
   < RID : RAMP-Replica | datastore : VS, latestCommit : LC >
  =>
   < RID : RAMP-Replica | datastore : VS, latestCommit : cmt(LC,VS,TS) >
   msg committed(TID) from RID to RID' .
```

Receiving Committed Message. Upon receiving a `committed` message, the replica first checks if all committed messages have now been collected. The expression `CMTS'[TID]` extracts for `TID` the remaining replicas from which it is awaiting `committed` messages. If the projection is `empty`, the replica commits the transaction:

```
crl [receive-committed] :
    msg committed(TID) from RID' to RID
    < RID : RAMP-Replica | executing : < TID : RAMP-Txn | > TRANSES,
                      committed : TRANSES', commitSites : CMTS >
  =>
    if CMTS'[TID] == empty  --- all "committed" received
    then < RID : RAMP-Replica | executing : TRANSES,
                           committed : < TID : RAMP-Txn | > TRANSES',
                           commitSites : CMTS' >
    else < RID : RAMP-Replica | executing : < TID : RAMP-Txn | > TRANSES,
                           committed : TRANSES', commitSites : CMTS' >
    fi
    if CMTS' := remove(TID,RID',CMTS) .
```

Receiving Get Messages (Lines 9–13). Upon receiving a `get` message, depending on the associated timestamp `TS` (if `TS` is an empty timestamp `eptTS`, the incoming message is the first-round `get`; otherwise, it is the second-round `get`), the replica replies with the corresponding version determined by the function `vmatch`. For a first-round `get`, `vmatch` looks up `LC` for the latest committed version; for the second-round `get`, `vmatch` returns the matched timestamped version of `TS`:

```
rl [receive-get] :
   msg get(TID,K,TS) from RID' to RID
   < RID : RAMP-Replica | datastore : VS, latestCommit : LC >
 =>
   < RID : RAMP-Replica | datastore : VS, latestCommit : LC >
   if TS == eptTS
     then msg response1(TID,vmatch(K,VS,LC)) from RID to RID'
     else msg response2(TID,vmatch(K,VS,TS)) from RID to RID'
   fi .
```

Receiving Response to First-round Get Messages (Lines 25, 27–33). Upon receiving a returned version for the first-round get, the replica adds it to the read set, and updates `localVars` accordingly. When the replica has collected all replies to the first-round gets, it determines whether a second-round `get` is needed. The expression `gen2ndGets(TID,VL',RS',RID,REPLICA-TABLE)` generates possible second-round `get` messages based on the updated `latest`, `VL'`, and `readSet`, `RS'`:

```
crl [receive-response1] :
    msg response1(TID,version(K,V,TS,MD)) from RID' to RID
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica | executing : < TID : RAMP-Txn |
                      operations : (OPS (X :=read K) OPS'),
                      readSet : RS, localVars : VARS, latest : VL > TRANSES,
                      1stGetSites : 1STGETS, 2ndGetSites : 2NDGETS >
  =>
    < TABLE : Table | table : REPLICA-TABLE >
    if 1STGETS'[TID] == empty
    then < RID : RAMP-Replica | executing : < TID : RAMP-Txn |
                             operations : (OPS (X :=read K) OPS'),
                             readSet : RS', localVars : insert(X,V,VARS),
                             latest : VL' > TRANSES,
                           1stGetSites : 1STGETS',
                           2ndGetSites : (2NDGETS ; addrs(TID,RIDS)) >
         gen2ndGets(TID,VL',RS',RID,REPLICA-TABLE)
    else < RID : RAMP-Replica | executing : < TID : RAMP-Txn |
                             operations : (OPS (X :=read K) OPS'),
                             readSet : RS', localVars : insert(X,V,VARS),
                             latest : VL' > TRANSES,
                           1stGetSites : 1STGETS', 2ndGetSites : 2NDGETS >
    fi
    if RS' := RS, version(K,V,TS,MD) /\
       VL' := lat(VL,MD,TS) /\
       1STGETS' := remove(TID,RID',1STGETS) /\
       RIDS := 2ndSites(VL',RS',RID,REPLICA-TABLE) .
```

Receiving Response to Second-round Get Messages (Lines 32–33). Upon receiving a returned version for the second-round get, the Replica simply overwrites the version fetched by the first-round get (the `readSet` is updated). It then updates the local variables `localVars` and the remaining replicas from which it is awaiting second-round `get`s:

```
rl [receive-response2] :
  msg response2(TID,version(K,V,TS,MD)) from RID' to RID
  < RID : RAMP-Replica | executing:
                     < TID : RAMP-Txn | operations : (OPS (X :=read K) OPS'),
                                        readSet : (RS, version(K,V',TS',MD')),
                                        localVars : VARS >,
                         2ndGetSites : 2NDGETS >
 =>
  < RID : RAMP-Replica | executing:
                     < TID : RAMP-Txn | operations : (OPS (X :=read K) OPS'),
                                        readSet : (RS, version(K,V,TS,MD)),
                                        localVars : insert(X,V,VARS) >,
                         2ndGetSites : remove(TID,RID',2NDGETS) > .
```

Committing Reads (Lines 18–22). If the replica has no remaining replicas from which it is awaiting replies to either first-round gets or second-round gets (`1STGETS[TID] == empty` and `2NDGETS[TID] == empty`), it commits the reads by storing the `TID` object in `committed`:

```
crl [commit-reads] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica | executing : < TID : RAMP-Txn | > TRANSES,
                     committed : TRANSES', 1stGetSites: 1STGETS,
                     2ndGetSites : 2NDGETS >
 =>
   < TABLE : Table | table : REPLICA-TABLE >
   < RID : RAMP-Replica | executing : TRANSES,
                    committed : (TRANSES' < TID : RAMP-Txn | >),
                    1stGetSites : 1STGETS, 2ndGetSites : 2NDGETS >
   if 1STGETS[TID] == empty /\ 2NDGETS[TID] == empty .
```

4.3.3 Formalizing RAMP-Small

Instead of attaching the entire write set to each write, RAMP-Small only stores the transaction timestamp. This correspond to changing the rule `start-wo-txn` by letting `genPuts` not instantiate metadata for each outgoing write, but instead leave it as an `empty` set. Apart from that, only the following two rules in RAMP-Fast need to be modified to define RAMP-Small.

Receiving Response to First-round Get Messages. When a replica has fetched the (highest-timestamped) committed timestamp for the requested item in the received version, it proceeds in a similar way as in RAMP-Fast, except that it will not update `readSet` or `localVars`, since RAMP-Small always requires two RTTs for reads. Each outgoing `get` message generated by `gen2ndGets` includes the entire set of timestamps:

```
crl [receive-response1-small] :
    msg response1(TID,version(K,V,TS,MD)) from RID' to RID
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica | executing :
                < TID : RAMP-Txn | latest : VL > TRANSES,
                      1stGetSites : 1STGETS, 2ndGetSites : 2NDGETS >
  =>
    < TABLE : Table | table : REPLICA-TABLE >
    if 1STGETS'[TID] == empty
    then < RID : RAMP-Replica | executing :
                < TID : RAMP-Txn | latest : VL' > TRANSES, 1stGetSites : 1STGETS',
                           2ndGetSites : (2NDGETS ; addrs(TID,RIDS)) >
         gen2ndGets(TID,VL',RID,REPLICA-TABLE)
    else < RID : RAMP-Replica | executing :
                < TID : RAMP-Txn | latest : VL' > TRANSES,
                           1stGetSites : 1STGETS', 2ndGetSites : 2NDGETS >
    fi
    if VL' := lat(VL,MD,TS) /\
       1STGETS' := remove(TID,RID',1STGETS) /\
       RIDS := 2ndSites(VL',RID,REPLICA-TABLE) .
```

Receiving Get Messages. When a Replica receives a `get` message for the first time, it proceeds in the same way as in RAMP-Fast; however, when the second `get` message arrives that contains *the entire set of timestamps* for the requested item, it returns the highest-timestamped version (determined by `maxts`) of that key that also exists in the received set of timestamps (determined by `tsmatch`). Note that the incoming `get` message includes a set of timestamps `TSS`:

```
rl [receive-get-small] :
   msg get(TID,K,TSS) from RID' to RID
   < RID : RAMP-Replica | datastore : VS, latestCommit : LC >
 =>
```

```
   < RID : RAMP-Replica | datastore : VS, latestCommit : LC >
   if TSS == empty
   then msg response1(TID,vmatch(K,VS,LC)) from RID to RID'
   else msg response2(TID,vmatch(K,VS,maxts(tsmatch(X,VS,TSS)))) from RID to RID'
   fi .
```

4.3.4 Formalizing RAMP Extensions

RAMP Without 2PC. We decouple 2PC from RAMP by changing the rule `receive-prepare-reply-true-executing` to the following rule, in which a replica simply removes the write from the pending set and commits it on the RAMP replica, instead of waiting for all `prepare-reply` messages to arrive:

```
rl [receive-prepare-reply-true-executing-decouple-2pc] :
   msg prepare-reply(TID,true) from RID' to RID
   < RID : RAMP-Replica | executing : < TID : RAMP-Txn | txnSqn : SQN > TRANSES,
                      voteSites : VSTS >
  =>
    < RID : RAMP-Replica | executing : < TID : RAMP-Txn | txnSqn : SQN > TRANSES,
                       voteSites : remove(TID,RID',VSTS) >
    msg commit(TID,ts(RID,SQN)) from RID to RID' .
```

RAMP with Faster Commit. A RAMP replica can mark as committed the version (by sending a `committed` message to the replica `RID'`) that has a fresher timestamp than the highest committed version of the requested item (indicated by `LC[K] < TS`). We model this optimization of RAMP-Fast by replacing the rule `receive-get` with the following rule. The other rules are unchanged:

```
rl [receive-get-faster-commit] :
   msg get(TID,K,TS) from RID' to RID
   < RID : RAMP-Replica | datastore : VS, latestCommit : LC >
 =>
   if TS == eptTS
     then < RID : RAMP-Replica | datastore : VS, latestCommit : LC >
          msg response1(TID,vmatch(K,VS,LC)) from RID to RID'
     else < RID : RAMP-Replica | datastore : VS, latestCommit :
            (if LC[K] < TS then insert(K,TS,LC) else LC fi) >
          msg response2(TID,vmatch(K,VS,TS)) from RID to RID'
          (if LC[K] < TS
            then msg committed(TID) from RID to RID'
            else none
           fi)
   fi .
```

RAMP with One-Phase Writes. After collecting all `prepare-reply` messages, a replica commits the transaction besides invoking `genCommits` to generate `commit` messages. The other rules are unchanged:

```
crl [receive-prepare-reply-true-executing-1pw] :
    msg prepare-reply(TID,true) from RID' to RID
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica | executing : < TID : RAMP-Txn | operations : OPS,
                             txnSqn : SQN > TRANSES,
                      committed : TRANSES',
                      voteSites : VSTS, commitSites : CMTS >
  =>
    < TABLE : Table | table : REPLICA-TABLE >
    if VSTS'[TID] == empty  --- all votes received and all yes!
    then < RID : RAMP-Replica | executing : TRANSES,
                           committed : TRANSES'
                             < TID : RAMP-Txn | operations : OPS, txnSqn : SQN >,
                           voteSites : VSTS',
                           commitSites : (CMTS ; addrs(TID,RIDS)) >
         genCommits(TID,SQN,RIDS,RID)
    else < RID : RAMP-Replica | executing : < TID : RAMP-Txn | operations: OPS,
                             txnSqn : SQN > TRANSES,
                           committed : TRANSES',
                           voteSites : VSTS', commitSites : CMTS >
    fi
    if VSTS' := remove(TID,RID',VSTS) /\
       RIDS := commitSites(OPS,RID,REPLICA-TABLE) .
```

RAMP-Faster. This alternative design integrates the two phases in writes: upon receiving a `prepare` message, the RAMP replica adds the incoming version to its local database `VS`, and also updates the index containing the highest-timestamped committed version of the item by invoking the function `cmt`. The following rule replaces the rules `receive-prepare-wo`, `receive-prepare-reply-true-executing`, and `receive-commit` in our RAMP-Fast model:

```
crl [receive-prepare-faster] :
    msg prepare(TID,version(K,V,TS,MD)) from RID' to RID
    < RID : RAMP-Replica | datastore : VS, latestCommit : LC >
 =>
    < RID : RAMP-Replica | datastore : VS', latestCommit : cmt(LC,VS',TS) >
    msg committed(TID) from RID to RID'
 if VS' := (VS, version(K,V,TS,MD)) .
```

4.4 MODEL CHECKING CONSISTENCY PROPERTIES

As shown in Table 3.1, we have applied our tool CAT to model check eight RAMP designs against nine properties. All model checking results are as expected. In particular, original RAMP designs (i.e., RAMP-F and RAMP-S) and the proposed RAMP variants (i.e., RAMP-F+1PW, RAMP-F+FC, and RAMP-S+1PW) by the developers satisfy RA, as well as RC that is weaker than RA, and violate all consistency properties stronger than RA; our proposed new RAMP-like designs (i.e., RAMP-F¬2PC, RAMP-S¬2PC, and Faster) satisfy only RC, and violate RA and any stronger consistency properties.

Here we focus on RA and use the same example to illustrate the impact of decoupling two-phase commit protocol (the major difference between the developers' RAMP designs and our new proposals).

In our model checking experiment we provide the CAT tool with one read-only transaction, one write-only transaction, two operations for each type of transaction, two replicas, and two keys. Under the hood, CAT executes the following command to search, from all generated initial states, for one reachable final state where the RA consistency property is violated:

```
search [1] init(1,1,0,2,2,0,0,2,2,1) =>! C:Configuration
    < M:Oid : Monitor | log: LOG:Log  clock: N:Nat > such that not ra(LOG:Log) .
```

CAT outputs "`No solution`," meaning that all runs from all the given initial states satisfy RA, for the developers' RAMP designs, while providing a counterexample showing a behavior that violates RA for RAMP-F¬2PC, RAMP-S¬2PC, and Faster, respectively.

Specifically, the counterexample obtained by analyzing the initial state with the two transactions $[\mathit{write}(k1, v1) \quad \mathit{write}(k2, v2)] \quad [\mathit{read}(k1) \quad \mathit{read}(k2)]$ shows that the read operations return $v1$, and $v0$ (the default version, older than $v2$), respectively. The reason is that our new proposals, despite of different mechanisms, do not wait for all prepared messages to arrive before committing a prepared version.

4.5 PROBABILISTIC MODELING OF RAMP DESIGNS

The RAMP models specified in the CAT framework are untimed, non-probabilistic, and nondeterministic for model checking purpose. In this section we are interested in estimating the *performance* (expected latency, percentage of transactions satisfying certain properties, etc.) of our designs. We therefore need to: (i) include time and probabilities in our models, and (ii) eliminate any nondeterminism, so that our models become purely probabilistic and can be subjected to statistical model checking.

The key idea to address both of these issues is to *probabilistically* assign to each message a *delay*. The point regarding issue (ii) is that if: (a) each rewrite rule is triggered by the arrival of a message, and (b) the delay is sampled probabilistically from a dense/continuous time interval, then the probability that two messages have the same delay is 0, and hence no two actions could happen at the same time, eliminating nondeterminism.

All our models are available at `https://sites.google.com/site/siliunobi/ramp-smc`.

4.5.1 Scheduler

To obtain a deterministic model from the corresponding nondeterministic one we implement in Maude the scheduling algorithm in the actor PMaude framework [5] for totally ordering message-triggered rules.

The *scheduler* object has the form {*time* | *msgs*}, where `msgs` buffers a list of (unripe) messages ordered by their delays, which need to be scheduled; `time` of sort `Float` represents the global time in real numbers.

In *transformed* models there are two types of messages, *unripe* and *ripe* messages, both of which wrap messages of sort `Msg` in original models with timing information (of sort `Float`):

```
sorts UnripeMsg RipeMsg .
subsorts UnripeMsg RipeMsg < Config .

op [_,_] : Float Msg -> UnripeMsg .
op {_,_} : Float Msg -> RipeMsg .
```

An unripe message `[d, msg]` contains the message delay `d` for the message `msg` to become ripe, while a ripe message `{t, msg}` includes the global time when `msg` became ripe. The scheduler only enqueues unripe messages, and dequeues them once they are ripe. Whenever the scheduler dequeues a message, it also advances the global time by `d` time units.

Example 4.1. Without loss of generality, let us assume that there are only two unripe messages in the configuration, `[d1, mc1 from o1 to o]` and `[d2, mc2 from o2 to o]`, generated at the global time `gt` with the respective message delays `d1` and `d2` with `d1` $<$ `d2`. The scheduler enqueues both messages for scheduling, and becomes:

```
{ gt | [d1, mc1 from o1 to o] [d2, mc2 from o2 to o] }
```

At the global time `gt + d1`, the scheduler dequeues the first message as a ripe one {`gt + d1, mc1 from o1 to o`}, and the configuration becomes (for simplicity we do not show the objects):

```
{ gt + d1 | [d2, mc2 from o2 to o] }  { gt + d1, mc1 from o1 to o }
```

Similarly, the message `mc2` is ripe and dequeued at the global time `gt + d2`:

```
{ gt + d2 | nil }  { gt + d2, mc2 from o2 to o }
```

Note that the message `mc1` has been consumed by the object `o`, and thus disappears from the configuration. Eventually, the message `mc2` will also be consumed. The global time has been advanced twice by the message delays `d1` and `d2`, respectively.

4.5.2 Wrapping Messages

The transformation decorates the messages in the original model in the following two manners:

- An incoming messages is wrapped with the *current global time*.
- An outgoing message is wrapped with a *delay* sampled probabilistically from the user-chosen continuous distribution of network latency.

Example 4.2. In the previous Maude model of RAMP-F, when receiving the `prepare` message, the replica adds the version to its local data store and replies with a `prepared` message (shown in black):

```
rl [receive-prepare-wo-prob] :
   { GT, prepare(TID,VER) from RID' to RID }
   < RID : RAMP-Replica | datastore : VS >
 =>
   < RID : RAMP-Replica | datastore : (VS, VER) >
   [ delay, prepare-reply(TID,true) from RID to RID' ] .
```

The transformed rule contains the message wrappers (in blue), indicating that the `prepare` message is consumed by `RID` at the global time `GT`, and the `prepare-reply` message will be consumed by `RID'` after `delay` time units. `delay` of sort `Float` is a parameter instantiated with a certain probability distribution, e.g., the lognormal distribution with $\mu = 3.0$ and $\sigma = 2.0$:

```
op delay : -> Float .
--- e.g., 'delay' is instantiated as:
eq delay = sampleLogNormal(3.0,2.0) .
```

4.6 MONITORING EXECUTIONS

Inspired by the monitoring mechanism of the CAT framework (Section 3.2) we also equip the transformed model with an execution log recording the history of relevant events during a system execution.

Specifically, we also define a "time vector" using Maude's map data type that maps replica identifiers (of sort `Oid`) to global times. The main difference is that global times here are of sort `Float`:

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

where each entry in the mapping is of the form `Oid |-> Float`.

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

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

We add to the configuration a `Monitor` object storing the current log in the `log` attribute:

```
< M : Monitor | log : Log >
```

Note that, thanks to the scheduler, the `Monitor` object here does not need to store the current logical global time (by the attribute `clock`) as in the CAT framework.

The log is updated each time an interesting event happens (i.e., the start and commit of a transaction).[3] We (manually) identify those events in a Maude model, and transform the corresponding rules by adding and updating the monitor object.

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

Example 4.3. The rewrite rule (in black) where a RAMP replica starts executing a write-only transaction is modified by adding and updating the monitor object (in blue):

```
crl [start-wo-txn-monitor] :
    < O@M : Monitor | log : LOG@M > { GT, start RID }
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica |
              gotTxns : (< TID : RAMP-Txn | operations : OPS, localVars : VARS,
                                                     txnSqn : N > ;; TXNS),
              executing : TRANSES,   sqn : SQN,   voteSites : VSTS >
   =>
    < O@M : Monitor | log : LOG@M, (TID |-> < RID, GT, empty, false, empty, empty >) >
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : RAMP-Replica |
              gotTxns : TXNS,
              executing : < TID : RAMP-Txn | operations : OPS, localVars : VARS,
                                                     txnSqn : SQN' > TRANSES,
              sqn : SQN',   voteSites : (VSTS ; addrs(TID,RIDS)) >
    genPuts(OPS,RID,TID,SQN',VARS,REPLICA-TABLE)
    if SQN' := SQN + 1 /\  write-only(OPS) /\
       RIDS := prepareSites(OPS,RID,REPLICA-TABLE) .
```

[3]We do not consider replication or transaction failure in our RAMP models, and therefore interesting events exclude the abort of a transaction, or transaction commits on remote sites.

where the monitor `O@M` initializes a new record for the transaction `TID` in the log, with starting time (the current global time `GT`) at its executing server `RID`, finish time (`empty`), flag (`false`), read set (`empty`), and write set (`empty`).

Commit. A transaction commits at its executing server when the transaction object appears in the attribute `committed` in the right-hand side, but not in the left-hand side, of a rewrite rule. The corresponding record is updated with commit status, versions read and written, and commit time.

Example 4.4. The monitor object is added to the rule `receive-committed` for committing a write-only transaction. The monitor updates the log entry for the transaction `TID` by setting its finish time at the proxy `RID` to `GT` (`insert(RID,GT,VTS@M)`) provided by the incoming message, the committed flag to `true`, the read set to `RS`, and write set to `WS`:

```
crl [receive-committed-monitor] :
    < O@M : Monitor | log : LOG@M, (TID |->
        < RID, T@M, VTS@M, FLAG@M, READS@M, WRITES@M)) >
    { GT, msg committed(TID) from RID' to RID }
    < RID : RAMP-Replica | executing : < TID : RAMP-Txn |
                               writeSet : WS, readSet : RS > TRANSES,
                          committed : TRANSES', commitSites : CMTS >
  =>
    if CMTS'[TID] == empty  --- all "committed" received
    then < O@M : Monitor | log : LOG@M, (TID |->
         < RID, T@M, insert(RID,GT,VTS@M), true, RS, WS >)
         < RID : RAMP-Replica | executing : TRANSES,
                               committed : < TID : RAMP-Txn |
                                    writeSet : WS, readSet : RS > TRANSES',
                               commitSites : CMTS' >
    else < O@M : Monitor | log : LOG@M, (TID |->
         < RID, T@M, VTS@M, FLAG@M, READS@M, WRITES@M)) >
         < RID : RAMP-Replica | executing : < TID : RAMP-Txn |
                                    writeSet : WS, readSet : RS > TRANSES,
                               committed : TRANSES', commitSites : CMTS' >
    fi
    if CMTS' := remove(TID,RID',CMTS) .
```

4.7 QUANTITATIVE ANALYSIS OF RAMP DESIGNS

The main difference between the RAMP designs in [14] and the new designs we have proposed is that those in [14] guarantee read atomicity whereas ours do not. On the other hand, as mentioned in Section 4.2, we conjecture that our designs—in particular, RAMP-Faster—provide not only better performance (throughput, average latency, etc.) but also in some ways better "consistency" in the sense of reads more often reading the latest value written. If this is indeed the case, and, furthermore, a large fraction of transactions in representative workloads satisfy read atomicity, then our designs should be interesting for applications where read atomicity is highly desirable but not an absolute requirement. For example, in a social network, read atomicity is desirable (if A befriends B in a transaction, then another transaction should not observe a "fractured read" where A is a friend of B but where B is not a friend of A), but a small percentage of fractured reads might be acceptable if the performance becomes significantly better.

In this section we compare the performance—along a number of performance parameters, including throughput, average latency, percentage of strongly consistent reads—of our own RAMP-like designs with the original RAMP designs using statistical model checking. Besides, we also intend to answer the question "Does statistical model checking of probabilistic Maude models provide realistic performance estimates for RAMP designs?" by comparing the performance estimates obtained by our method with the implementation-based evaluations in [14].[4]

4.7.1 Performance Measures

We start by formalizing the performance metrics as functions of the "history log" of a *completed* run. The common performance measures for DTSs are *throughput* and *average latency.*

Throughput. The function `throughput` computes the number of committed transactions per time unit. `committedNumber` computes the number of committed transactions in `LOG`, and `totalRunTime` returns the time when all transactions are finished (i.e., the largest *finishTime* in `LOG`):

```
op throughput : Log -> Float .
eq throughput(LOG) = committedNumber(LOG) / totalRunTime(LOG) .
```

[4]Strong consistency is not considered in [14].

```
op committedNumber : Log -> Float .
op $committedNumber : Log Float -> Float .
eq committedNumber(LOG) = $committedNumber(LOG,0.0) .
eq $committedNumber((TID |-> < O,T,VT,true,RS,WS >,LOG),N)
   = $committedNumber(LOG,N + 1.0) .
eq $committedNumber((TID |-> < O,T,VT,false,RS,WS >,LOG),N)
   = $committedNumber(LOG,N) .
eq $committedNumber(noRecord,N) = N .
```

Average Latency. The function `avgLatency` computes the average transaction latency by dividing the sum of the latencies of all committed transactions by the number of such transactions:

```
op avgLatency : Log -> Float .
eq avgLatency(LOG) = totalLatency(LOG) / committedNumber(LOG) .
```

where `totalLatency` computes the sum of all transaction latencies (time between the start time and the finish time of a committed transaction).

```
op totalLatency : Log -> Float .
op $totalLatency : Log Float -> Float .
eq totalLatency(LOG) = $totalLatency(LOG,0.0) .
eq $totalLatency((TID |-> < O,T1,(O |-> T2,VT),true,RS,WS >,LOG),T)
   = $totalLatency(LOG,T + T2 - T1) .
eq $totalLatency((TID |-> < O,T1,VT,false,RS,WS >,LOG),T)
   = $totalLatency(LOG,T) .
eq $totalLatency(noRecord,T) = T .
```

4.7.2 Consistency Measures

There is considerable interest in how well data consistency properties of interest are met by different cloud storage systems [88, 43, 10, 64, 76]. Consistency properties include read consistency guarantees (e.g., *strong consistency*, and *read my writes* advocated by Doug Terry [87]) in general, and transaction isolation models (e.g., the classic *read committed* isolation [18], and state-of-the-art *read atomicity* [14]) in particular for DTSs.

We focus on the following two in this chapter:

- *Strong Consistency* (SC) guarantees that each read returns the value of the last write that occurred before that read.

- *Read Atomicity* (RA) ensures that either *all* or *none* of a (distributed) transaction's updates are visible to other transactions.

Strong Consistency. As all transactions can be totally ordered by their issue times, we define the function `sc` that computes the fraction of read transactions satisfying SC:

```
op sc : Log -> Float .
eq sc(LOG) = scTxns(LOG) / totalReadTxns(LOG) .
```

where `totalReadTxns` returns the total number of (committed) read transactions in `LOG`. `scTxns` checks, for each (committed) read transaction in `LOG`, whether the versions read match those of the last (committed) write transaction (indicated by `WS'`):

```
 op scTxns : Log -> Float .
 op $scTxns : Log Float -> Float .
 eq scTxns(LOG) = $scTxns(rtx(LOG),wtx(LOG),0.0) .
ceq $scTxns((TID1 |-> < O, T, VT, true, WS',WS >, LOG),
            (TID2 |-> < O', T', VT', true, RS',WS') >, LOG'), N)
    = $scTxns(LOG, (TID2 |-> < O', T', VT', true, RS', WS' >, LOG'), N + 1.0)
    if T' < T /\ noWtx(T',T,LOG') .
 eq $scTxns(noRecord, LOG', N) = N .
```

where the functions `rtx` and `wtx` return all committed read and write transactions in `LOG`, indicated by the non-empty read and write set, respectively. `noWtx` ensures that there is no (committed) write transaction issued between the last write transaction's issue time `T'` and the read transaction's issue time `T`.

Read Atomicity. We define the function `ra` computing the fraction of read transactions that satisfy RA:

```
op ra : Log -> Float .
eq ra(LOG) = raTxns(LOG) / totalReadTxns(LOG) .
```

where `raTxns` checks, for each (committed) read transaction in `LOG`, whether it reads RA-consistent versions.

Specifically, if there is a fractured read (indicated by the matching version `V`, and `V' < V''`), then that transaction does not count; otherwise, the total number of RA-consistent transactions (`N`) increases by one:

```
 op raTxns : Log -> Float .
 op $raTxns : Log Float -> Float .
 eq raTxns(LOG) = $raTxns(rtx(LOG),wtx(LOG),0.0) .
ceq $raTxns((TID1 |-> < O, T, VT, true, (< X,V >, < Y,V'>,RS), WS >, LOG),
            (TID2 |-> < O', T', VT', true, RS', (< X,V >, < Y,V''>,WS') >, LOG'), N)
    = $raTxns(LOG, (TID2 |-> < O', T', VT', true, RS',(< X,V >, < Y,V'' >,WS') >, LOG'),
      N) if V' < V'' .
 eq $raTxns((TID1 |-> < O, T, VT, true, RS, WS >, LOG),
            (TID2 |-> < O', T', VT', true, RS', WS' >, LOG'), N)
    = $raTxns(LOG, (TID2 |-> < O', T', VT', true, RS',WS' >, LOG'),N + 1.0) [owise] .
 eq $raTxns(noRecord, LOG', N) = N .
```

4.7.3 Generating Initial States

We use an operator `init` to *probabilistically* generate initial states:

`init(`*rtx* , *wtx* , *rwtx* , *repl* , *keys* , *rops* , *wops* , *rwops* , *distr*`)`

generates an initial state with *rtx* read-only transactions, *wtx* write-only transactions, *rwtx* read-write transactions, *repl* replicas, *keys* data items, *rops* operations per read-only transaction, *wops* operations per write-only transaction, *rwops* operations per read-write transactions, and *distr* the key access distribution (the probability that an operation accesses a certain data item). To capture the fact that some data items may be accessed more frequently than others, we also use Zipfian distributions in our experiments.

Each PVeStA simulation starts from `init(`*parameters*`)`, which rewrites to a *different* initial state in each simulation. The reason is that this expression involves generating certain values—such as the transactions—probabilistically. The entire specification is given at `https://sites.google.com/site/siliunobi/ramp-smc`.

4.7.4 Statistical Model Checking Results

This section shows the result of using statistical model checking from many initial states to compare all eight RAMP versions w.r.t. the performance (i.e., throughput and average latency) and consistency measures (i.e., strong consistency and read atomicity).

In our experiments we use lognormal distribution for message delay with the mean μ `= 0.0` and standard deviation σ `= 1.0` [17]. All properties are computed with a 99% confidence level of size at most 0.01. Our analyses consider 2 data items, 2 operations per transaction, up to 50 clients (or proxies), and up to 400 transactions. We consider not only the 95% read transaction and 5% write transaction proportion workloads in [14], but also explore how the RAMP designs behave for different read/write rates.

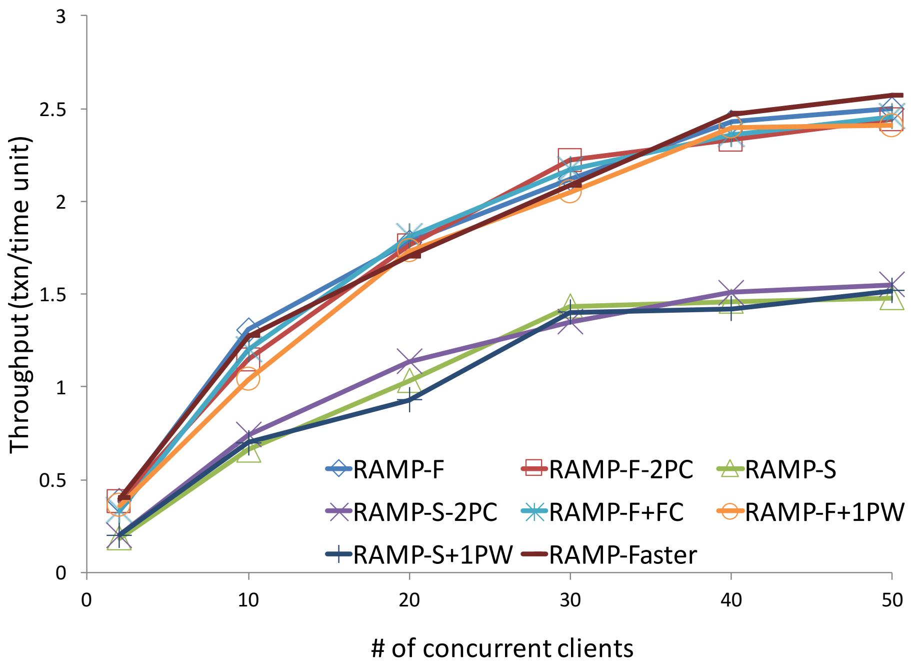
Throughput. Figure 4.1 shows the resulting of analyzing throughput against the number of concurrent clients (top) and percentage of read transactions (bottom).

For the original RAMP designs, under a 95% read proportion, as the number of clients increases, both RAMP-F and RAMP-S's throughput increases, and RAMP-F provides higher throughput than RAMP-S. As the read proportion increases, RAMP-F's throughput increases, while RAMP-S's throughput keeps nearly constant; and RAMP-F also outperforms RAMP-S in throughput. These observations are consistent with the experimental results in [14].

There are no conjectures in [14] about the throughput of the designs that were only sketched in [14]. We observe that unlike other RAMP-F-like algorithms, whose throughput increases as read activities increase, RAMP-F+1PW's throughput keeps high with all reads/writes. As the right plot shows, at the beginning, when there are more writes than reads, RAMP-F+1PW and RAMP-Faster perform better than other RAMP-F-like designs. This happens because RAMP-F requires two RTTs for a write, RAMP-F+1PW needs only one RTT and RAMP-Faster, our proposed design, performs commit when the PREPARE message is received. Hence, with all write transactions, RAMP-F+1PW and RAMP-Faster will always provide higher throughput. However, as read activities increase, other RAMP-F-like designs increase their throughput, as they require one RTT for all reads. Even though as the percentage of reads increases, RAMP-F+1PW and RAMP-Faster compensate the extra RTT incurred due to the races, with the RTT saved during the write operations.

The RAMP-S-like designs provide lower throughput than the RAMP-F-like designs, which is consistent with the observations in [14]. As expected, as the read percentage increases, RAMP-S+1PW's throughput converges with those of other RAMP-S-like designs, because all RAMP-S-like designs require more RTTs for reads compared to RAMP-F even when there is no race between reads and writes. In the worst case, when there is a race between read and write operations, all designs require two RTTs for reads.

Regarding our own designs, RAMP-Faster provides the highest throughput with varying read load and with larger number of concurrent clients among all RAMP versions. One reason is that RAMP-Faster's writes need only one RTT. RAMP-F-2PC (or RAMP-S-2PC) is not competitive with RAMP-F (or RAMP-S) regarding throughput. The reason is that, although they sacrifice 2PC, they still need to commit each write operation before committing the write transaction, which brings no apparent difference in throughput.



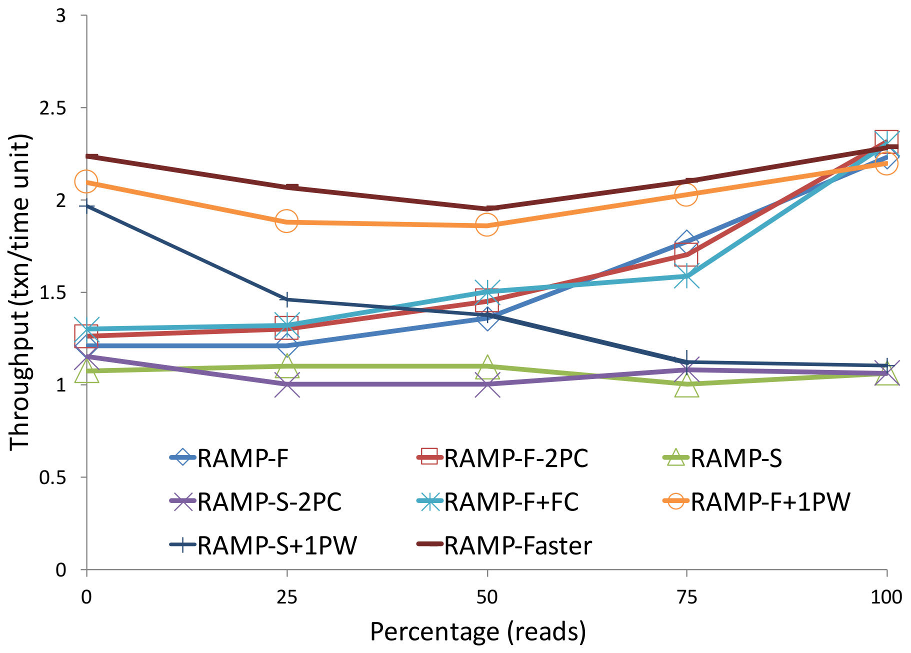


Figure 4.1: Throughput under varying client and read load.

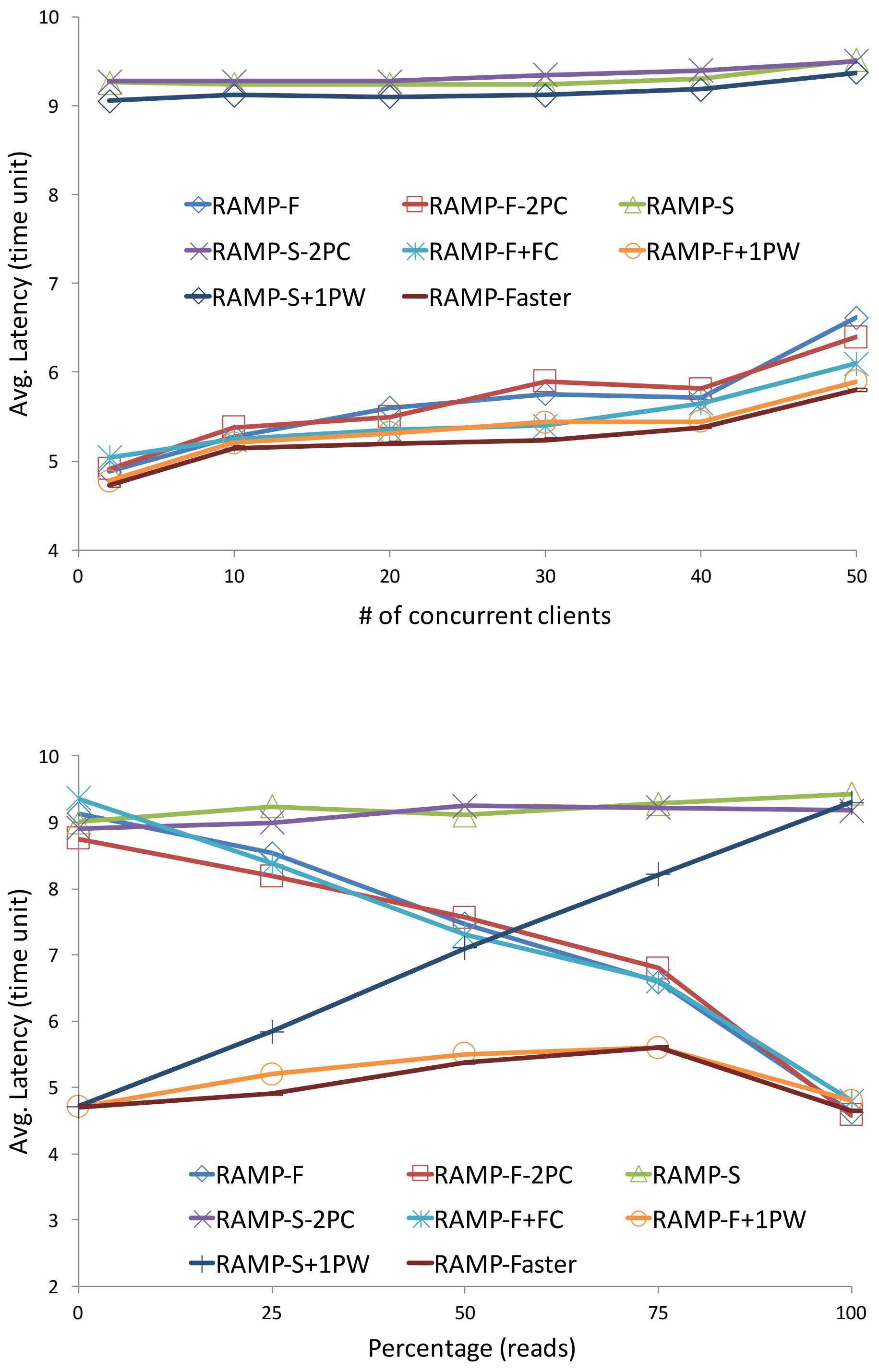


Figure 4.2: Average transaction latency under varying client and read load.

Average Latency. Figure 4.2 shows the average transaction latency as the number of concurrent clients (top) and the proportion of read transactions (down) increases.

Under a 95% read proportion, as the number of concurrent clients increases, the RAMP-F versions' average latency increases slightly, and the RAMP-S versions are almost twice as slow as the RAMP-F variations. And although RAMP-F+1PW and RAMP-S+1PW as expected have lower latencies than RAMP-F and RAMP-S, respectively, the differences are surprisingly small. In the same way, removing 2PC does not seem to help much. Although the differences are small, RAMP-Faster is the fastest, followed by RAMP-F with one-phase writes.

In Fig. 4.2 (bottom) we see that RAMP-F+1PW and RAMP-Faster significantly outperform all the other algorithms when the proportion of write transactions is between 25% and 75-80%.

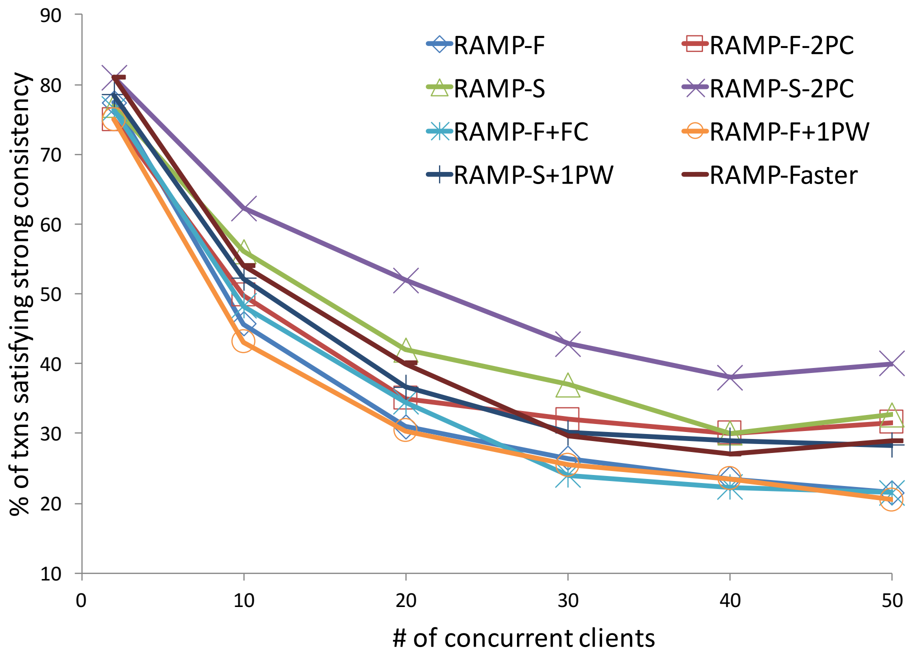
Regarding our own designs, it seems that RAMP-F-2PC (resp. RAMP-S-2PC) is not competitive with RAMP-F (resp. RAMP-S) regarding average latency. The reason is that, although RAMP-F-2PC and RAMP-S-2PC sacrifice 2PC, they still need to commit each write operation before committing the write transaction, which brings no noticeable difference in latency. RAMP-Faster incurs the lowest average latency among all RAMP versions with varying client and read loads.

Strong Consistency. Figure 4.3 shows the percentage of transactions satisfying strong consistency under varying number of clients and read/write proportions by using statistical model checking.

In all RAMP designs, the probability of satisfying strong consistency decreases as the number of clients increases, since there are more races between reads and writes, which decreases the probability of reading the preceding write.

It is natural that the percentage of transactions satisfying strong consistency increases as the reads increase: the chance of reading the latest preceding write should increase when writes are few and far between.

We also observe that RAMP-S-like designs (i.e., RAMP-S/+1PW/-2PC) provide greater degrees of strong consistency than their RAMP-F counterparts. The reason is that RAMP-S-like designs always use second-round reads, which might increase the chance of reading the latest write. The only exception seems to be that RAMP-Faster outperforms all other RAMP designs for 25-75% read workloads. The reason is that RAMP-Faster only requires one RTT for a write to commit, which increases a read transaction's chance to fetch the latest write.



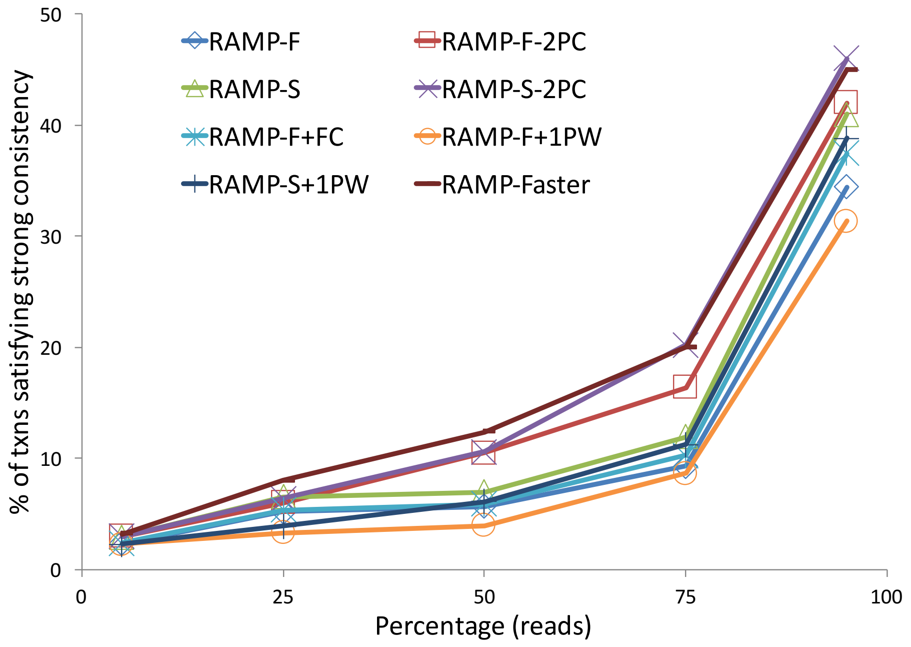
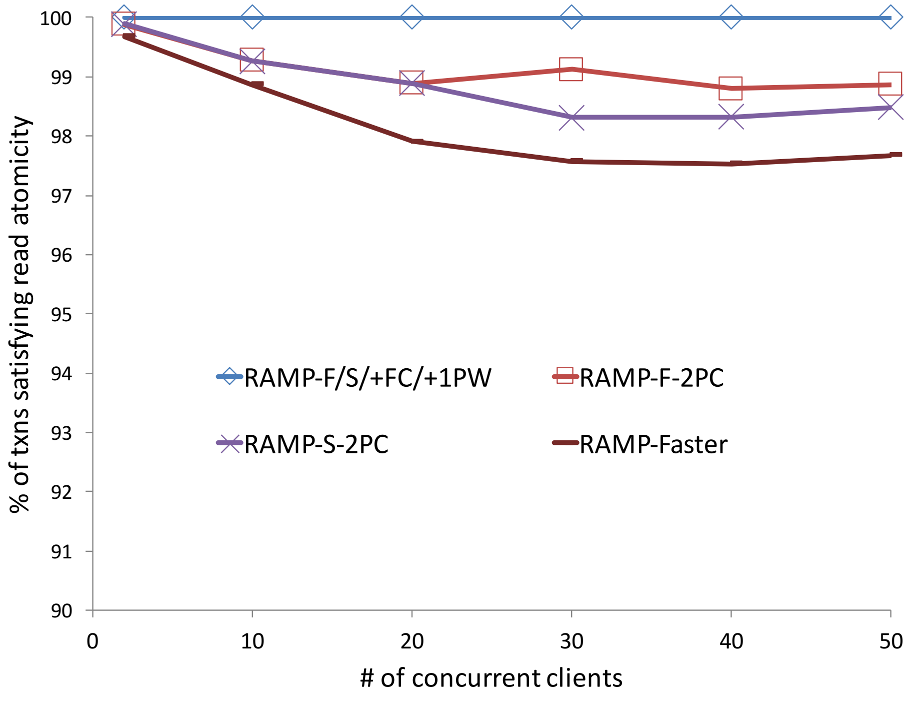


Figure 4.3: Probability of satisfying strong consistency under varying client and read load.



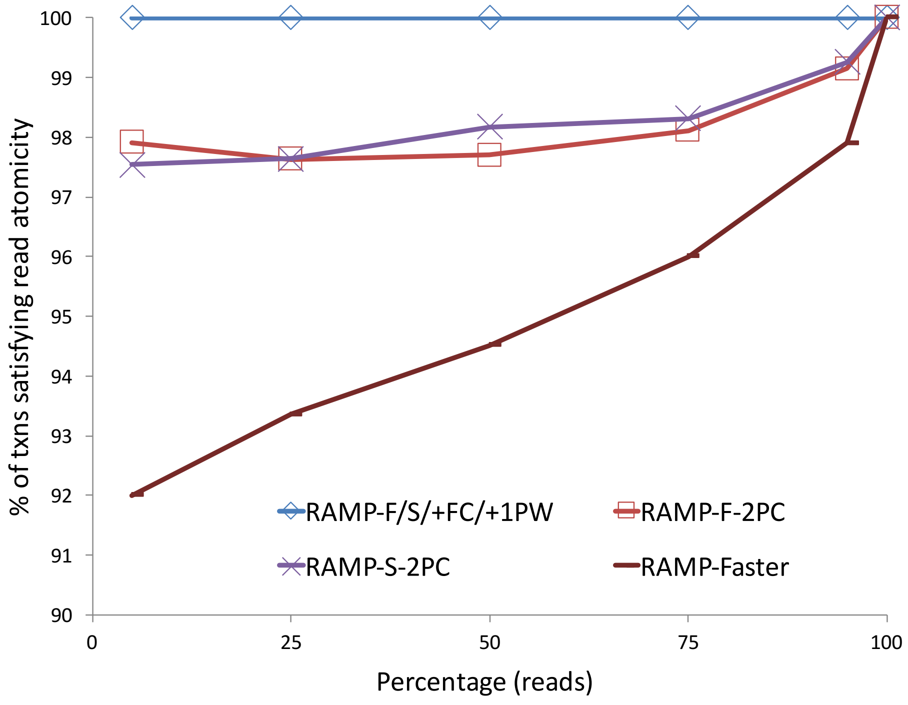


Figure 4.4: Probability of satisfying read atomicity under varying client and read load.

Read Atomicity. Figure 4.4 shows the percentage of transactions satisfying read atomicity by using statistical model checking. As it should be, all designs in [14] satisfy read atomic isolation. Our own design alternatives provide 92-100% read atomicity under all scenarios considered. With 95% read transactions, they all offer 97-100% read atomicity.

Summary. Our formal model-based methodology has allowed us to quickly and easily analyze the expected performance of a large number of RAMP designs along a number of performance parameters and with varying number of concurrent clients and read/write transaction proportions. This allows to predict which design is the best fit for a particular application.

Our results are consistent with the experimental results in [14]. For example: the throughput of both RAMP-F and RAMP-S increases with the number of concurrent clients, and RAMP-F provides higher throughput than RAMP-S; the latency also increases with the increase of concurrent clients (very minimally for RAMP-S, however).

Our results also confirm the conjectures about the sketched designs in [14], which were never experimentally validated by the RAMP developers. For example: RAMP-F+FC and RAMP-F+1PW have lower latency than RAMP-F (and similarly for RAMP-S). We can also compare RAMP-F-FC with RAMP-F+1PW, and see that RAMP-F+1PW typically provides better performance among these two optimizations.

We can also evaluate our own designs. It turns out that RAMP without 2PC does not improve the performance of RAMP. On the other hand, RAMP-Faster is an interesting design, as it generally provides the smallest average latency and highest throughput among all RAMP designs, in particular when there are a fair amount of write transactions, while providing more than 92% read atomicity even for very write-heavy workloads. Maybe slightly surprisingly, RAMP-Faster does not provide the highest percentage of strongly consistent reads for read-heavy workloads, but does so for workloads with 25-75% read transactions.

Note that the actual values might differ between the experiments in [14] and our statistical analysis, due to factors like hard-to-match experimental configurations, the inherent difference between statistical model checking and implementation-based evaluation[5], processing delay at client/replica side, and different distributions of item accesses. The important observation is that the relative performance in both sides are similar (see Appendix B for the RAMP performance in [14]).

It is also worth remarking we only use two data items, while the experiments in [14] use

[5]In general, implementation-based evaluation is based on a single trace of hundreds of thousands of transactions, while statistical model checking is based on sampling hundreds of thousands of Monte-Carlo simulations of hundreds of transactions up to a certain statistical confidence.

up to thousands. This implies that we "stress" the algorithms much more, since there are much fewer potential clashes between small transactions (typically with four operations) in a 1000-data-object setting that between our two-operation transactions on two data objects.

The time to compute the probabilities for strong consistency is around 15 hours (in the worst-case), and for other metrics is around 8 hours (in the worst-case) with a workload of 400 transactions on a 2.7 GHz Intel Core i5 CPU with 8 GB of memory. Each point in the plots represents the average of three statistical model checking results.

4.8 CONCLUDING REMARKS

In this chapter, we have investigated the RAMP transaction systems from both qualitative and quantitative perspectives.

Regarding correctness analysis, while the original RAMP paper included hand proofs only for the basic RAMP algorithms, we have adopted a model checking approach where we have: (i) first developed fully-executable formal models of many RAMP variants and extensions in the CAT framework; (ii) formally analyzed these models to confirm the original correctness properties; and (iii) used our models to analyze RAMP's extensions and optimizations (which the original RAMP paper did not do).

Regarding performance estimation, we have explored eight design alternatives for RAMP transactions by using statistical model checking. Substantial knowledge about both implemented and unimplemented RAMP designs has thus been gained. This knowledge can help find the best match between a given RAMP version and a class of applications. For example, we now know how the different designs behave not just for read-intensive workloads, but understand their behavior across the entire spectrum from read-intensive to write-intensive tasks. We have also identified promising new design alternatives for given classes of applications relatively easily *before* they are implemented. This of course does not replace the need for implementation and experimental validation, but it allows us to focus implementation and validation efforts where they are most likely to pay off.

CHAPTER 5: CASE STUDY: THE WALTER TRANSACTIONAL DATA STORE

Walter [85] is a distributed partially replicated data store providing Parallel Snapshot Isolation (PSI), an important consistency property that offers attractive performance while ensuring adequate guarantees for certain kinds of applications. In this chapter we formally model Walter's design in the CAT framework described in Chapter 3, and formally verify the desired consistency properties by model checking. To the best of our knowledge, this is the first formal specification, as well as the first model checking analysis, of Walter. To also analyze Walter's performance we extend the Maude specification of Walter to a probabilistic rewrite theory and perform statistical model checking analysis to evaluate Walter's throughput for a wide range of workloads. Our performance results are consistent with a previous experimental evaluation and throw new light on Walter's performance for different workloads not evaluated before.

This chapter is structured as follows. Section 5.1 gives an overview of Walter. Section 5.2 provides a detailed formal executable specification of Walter. Section 5.3 formally analyzes whether the Walter model satisfies PSI or SI (short for Snapshot Isolation). In Section 5.4 we transform the Maude model of Walter from the CAT framework into a probabilistic rewrite theory, and carry out a systematic statistical model checking analysis of the key performance metric, transaction throughput, under a wide range of workloads. Finally, concluding remarks are given in Section 5.5.

Acknowledgements. This chapter is based on the work [61, 62] supported in part by NSF Grant CNS 1409416, NRL contract N00173-17-1-G002, and NSFC Project No. 61872146.

5.1 WALTER DATA STORE

Walter [85] is a partially replicated geo-distributed data store that supports multi-partition transactions and guarantees PSI. The key idea to ensure that all operations in a transaction read a consistent "snapshot" of the distributed data store is that *each site* s maintains a (local) *vector timestamp* $\{site_1 \mapsto k_1, \ldots, site_n \mapsto k_n\}$ representing a current snapshot of the state, as seen by site s, where $site_j \mapsto k_j$ means that the snapshot includes the first k transactions executed at site $site_i$. Each time a transactions starts executing at s, the transaction is assigned the current local snapshot/vector timestamp of site s. Remote reads can then be performed consistently according to this snapshot. Another key Walter feature is that each data item has a preferred site, so that writes at preferred sites can be committed fast (e.g., the sites that you usually use could be the preferred site for "your" data).

A transaction is executed as follows. When the "host" site s starts executing transaction t, t is assigned the current snapshot of s. The site s then executes the read and write operations in t. For writes, Walter buffers the versions written in the transaction's write set. For reads, Walter fetches the latest appropriate version according to t's start snapshot, by checking any updates in the write set and its history of previous updates. If the associated key is not replicated locally, Walter retrieves the right version remotely from the data item's preferred site.

When the host site has finished executing the operations in the transaction, it starts committing the transaction. Read-only transactions and transactions that only write data items whose preferred site is the host site s can commit locally (*fast commit*). Walter then checks whether all versions of each data item in the history of the local site are *unmodified* since the start vector timestamp, and whether all data items are *unlocked* (i.e., not being committed by another transaction). If either check fails, Walter aborts the transaction; otherwise, Walter can commit the transaction. If a transaction cannot commit locally (*slow commit*), the executing site s uses the two-phase commit (2PC) protocol to check whether the transaction can be committed, by asking all the preferred sites of data items written by t whether t can be committed. If the data items written by t are unmodified and unlocked at such a site, the site replies with a "yes" vote and locks the corresponding data items. Otherwise, the site votes "no." If the executing site receives a "no" vote, the transaction is aborted and the other preferred sites are notified and release the appropriate locks. If all votes are "yes" votes, the transaction can be committed.

If the transaction t can be (fast or slow) committed, the site s marks t as committed, assigns it a *version* $(s, seqNo)$ (where *seqNo* is a local sequence number), updates the local history with the updates, and propagates t to other sites, which update their histories and their vector timestamps. To allow f site failures, a transaction is marked *disaster-safe durable* if its writes have been logged at $f+1$ sites. The propagation protocol first checks whether the transaction can be marked as disaster-safe durable by collecting acknowledgments from $f+1$ sites for each data item. Upon receiving the propagation of a transaction, a site acknowledges it only after it receives all transactions that causally precede the propagated transaction (by using the transaction's start vector timestamp), and all transactions at the same executing site with a smaller sequence number. The protocol then checks whether the transaction can be marked as globally visible. This is done by committing the transaction at all sites. A transaction can be committed at a remote site when it learns that the transaction is disaster-safe durable, all transactions causally preceding the transaction have been committed locally, and all transactions at the same executing site with a smaller sequence number have been committed locally.

The paper [85] briefly discusses failure handling, but does not give much detail. The authors have implemented Walter in about 30K lines of code, and have implemented Facebook- and Twitter-like applications on top of Walter using the Amazon EC2 cloud platform to experiment with and evaluate Walter's performance in isolation, and as a backend for social networking, in a distributed setting (with nodes in US, Ireland, and Singapore). They use their distributed deployment to estimate the transaction latency and throughput (committed transactions per second) for read-only, write-only, and 90% read workloads.

The authors do not prove or justify that Walter actually guarantees PSI.

5.2 FORMAL MODELING OF WALTER

This section defines a formal executable model of Walter in the CAT framework.

5.2.1 Data Types, Classes, and Messages

We formalize Walter in an object-oriented style, where the state consists of a number of *replica* (or *site*) objects, each modeling a local database, and a number of messages traveling between the objects. A *transaction* is formalized as an object which resides inside the replica object that executes the transaction.

Some Data Types. A *version* is a pair `version(oid,sqn)` consisting of a site `oid` where the transaction is executed, and a sequence number `sqn` local to that site. A vector timestamp is a map from site identifiers to sequence numbers:

```
pr MAP{Oid,Nat} * (sort Map{Oid,Nat} to VectorTimestamp) .
```

The sort `OperationList` represents lists of read and write operations as terms such as (x `:= read` $k1$) (y `:= read` $k2$) `write(`$k1, x+y$`)`, where `LocalVar` denotes the "local variable" that stores the value of the key read by the operation, and `Expression` is an expression involving the transaction's local variables:

```
op write : Key Expression -> Operation [ctor] .
op _:=read_ : LocalVar Key -> Operation [ctor] .
op waitRemote : Key LocalVar -> Operation [ctor] .
pr LIST{Operation} * (sort List{Operation} to OperationList) .
```

`waitRemote(`k, x`)` means that the transaction execution is awaiting the value of the key (or data item) k from a remote site to be assigned to the local variable x.

Classes. A Walter transaction is modeled as an object instance of the subclass `Walter-Txn` of the class `Txn` defined in Chapter 3.1:

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

The `operations` attribute denotes the transaction's remaining operations. `localVars` maps the transaction's local variables to their current values. `startVTS` refers to the vector timestamp assigned to the transaction when it starts to execute, and `txnSQN` is the transaction's sequence number given upon commit.

A *replica*, or *site*, stores parts of the database, and executes the transactions for which it is the host/server. A Walter replica is specified as an object instance of the following subclass `Walter-Replica` of the class `Replica` defined in Chapter 3.1:

```
class Walter-Replica | history : Datastore,  sqn : Nat,  gotTxns : ObjectList,
                       gotVTS : VectorTimestamp,  locked : Locks,
                       votes : Vote,  voteSites : TxnSites,  abortSites : TxnSites,
                       dsSites : PropagateSites,  vsbSites : TxnSites,
                       dsTxns : OidSet,  gvTxns : OidSet,
                       recPropTxns : PropagatedTxns,  recDurableTxns : DurableTxns .
subclass Walter-Replica < Replica .
```

The `history` attribute represents the site's local database, as well as propagated updates also on data items not stored at the replica, as a map from keys to lists of updates `<` *value* `,` *version* `>`:

```
op <_,_> : Value Version -> ValueVersion [ctor] .

pr LIST{ValueVersion} * (sort List{ValueVersion} to ValueVersionList) .
pr MAP{Key,ValueVersionList} * (sort Map{Key,ValueVersionList} to Datastore) .
```

The `sqn` attribute denotes the replica's current local sequence number. The attribute `gotTxns` denotes the transaction (objects) which are waiting to be executed. `gotVTS` indicates for each site how many transactions of that site have been received by this site. The `locked` attribute denotes the locked keys and their associated transactions at this site:

```
op lock : Oid Key -> Lock .  --- Txn Oid locks Key
pr SET{Lock} * (sort Set{Lock} to Locks) .
```

The `votes` attribute denotes a collection of votes in the two-phase commit:

```
sort Vote .
op noVote : -> Vote [ctor] .
op vote : Oid Oid Bool -> Vote [ctor] .  --- Txn, Participant, vote
op _;_ : Vote Vote -> Vote [ctor assoc comm id: noVote] .
```

The `voteSites` attribute refers to, for each transaction, the remaining replicas from which the coordinator is awaiting votes:

```
sort TxnSites .
op noTS : -> TxnSites [ctor] .
op txnSites : Oid OidSet -> TxnSites [ctor] .
op _;_ : TxnSites TxnSites -> TxnSites [ctor assoc comm id: noTS] .
```

Similarly for each transaction, the attribute `abortSites` denotes the remaining sites from which the coordinator is awaiting the acknowledgments to abort the transaction. (The coordinator first notifies the corresponding sites to abort a transaction, and it will abort it locally after it gets the replies from those sites.)

The remaining attributes refer to the transaction replication. The attribute `dsSites` (resp. `vsbSites`) denotes the remaining sites from which each transaction is awaiting acknowledgments to mark itself as disaster-safe durable (resp. as globally visible). The sort `PropagateSites` contains the keys in each transaction's write set, because for a transaction to be disaster-safe durable each key must be replicated:

```
sort PropagateSites .
op noPS : -> PropagateSites [ctor] .
op propagateSites : Oid Key OidSet -> PropagateSites [ctor] .
op _;_ : PropagateSites PropagateSites ->
           PropagateSites [ctor assoc comm id: noPS] .
```

The attributes `dsTxns` and `gvTxns` denote the set (of sort `OidSet`) of disaster-safe durable and globally visible transactions, respectively. The last two attributes `recPropTxns` and `recDurableTxns` buffer the received propagation and disaster-safe durable messages from the coordinator.

The state also contains an object mapping each key to the sites storing the key (these sites are also called the *replicas* of the key):

```
class Table | table : ReplicaTable .
```

Elements of sort `ReplicaTable` are '`;;`'-separated sets of terms `sites(`k_i`,`*$replicas_i$*`)`, where the list *$replicas_i$* denotes the sites replicating the key k_i. The first element in such a list is the *preferred site* of the corresponding key:

```
sort KeyReplicas .
op [_] : KeyReplicas -> ReplicaTable [ctor] .
op eptTable : -> KeyReplicas [ctor] .
op sites : Key OidList -> KeyReplicas [ctor] .
op _;;_ : KeyReplicas KeyReplicas -> KeyReplicas [ctor assoc comm id: eptTable] .
```

Messages between sites have the form `msg` *content* `from` *sender* `to` *receiver*. The message content (or simply message) `request(`*key*, *txn*, *vts*`)` sends a read request for transaction `txn` to *key*'s preferred site to retrieve its state from the snapshot determined by vector timestamp *vts*. The preferred site replies with a message `reply(`*txn*, *key*, *value_version*`)`, where *value_version* is chosen based on the incoming vector timestamp. The message `prepare(`*txn*`,`*keys*`,` *vts*`)` sends the key(s) *keys* in transaction *txn* to their preferred sites with the transaction's start vector timestamp *vts*. Those preferred sites reply with a message `prepare-reply(`*txn*, *vote*`)`. The messages `abort(`*txn*`)` and `aborted(`*txn*`)` are sent out when the coordinator distributes the "abort" decision to the participants, and when the participants acknowledge the decision. The message `propagate(`*txn*, *sqn*, *vts*, *ws*`)` sends a transaction *txn*'s sequence number *sqn*, vector timestamp *vts*, and write set *ws* to all sites. The sites reply with a message `propagate-ack(txn)` to acknowledge that the transaction *txn* has been propagated successfully. The message `ds-durable(`*txn*`)` is sent to all sites once the transaction *txn* has been marked as disaster-safe durable. The sites then reply with a message `visible(`*txn*`)` to acknowledge the notification.

Initial State. The following shows an automatically generated initial state (with some parts replaced by '...') with three replicas, `r1`, `r2`, and `r3`, where `r1` and `r2` are the coordinators for, respectively, transactions `t1`, and `t2` and `t3`. Key `x` is replicated at `r1` and `r2`, key `y` at `r2` and `r3`, and key `z` at `r3` and `r1`, with `r1`, `r2` and `r3` the respective preferred sites. Transaction `t1` is the read-only transaction `(xl :=read x) (yl :=read y)`, transaction `t2` is a write-only transaction `write(y, 3) write(z, 8)`, while transaction `t3` is a read-write transaction on key `x`. Initially, the value of each key is `[0]`, and its version is `version(0,0)`:

```
eq init =
< tb : Table | table : [sites(x,r1 r2) ;; sites(y,r2 r3) ;; sites(z,r3 r1)] >
< r1 : Walter-Replica |
       gotTxns : < t1 : Walter-Txn | operations : ((xl :=read x) (yl :=read y)),
                    readSet : empty, writeSet : empty, startVTS : empty,
                    localVars : (xl |-> [0], yl |-> [0]), txnSqn : 0 >,
       history : (x |-> (< [0],version(0,0) >),
                   z |-> (< [0],version(0,0) >)), sqn : 0,  ... >
< r2 : Walter-Replica |
       gotTxns : < t2 : Walter-Txn | operations : (write(y,3) write(z,8)), ... >
                 < t3 : Walter-Txn | operations : ((xl := read x)
                                      write(x, xl plus 1)),  ... > ... >
< r3 : Walter-Replica | history : (y |-> (< [0],version(0,0) >),
                                   z |-> (< [0],version(0,0) >)), ... > .
```

5.2.2 Formalizing Walter's Behavior

This section formalizes the dynamic behavior of Walter using rewrite rules.[1]

Starting a transaction. A replica starts executing a transaction by moving the first transaction `TID` in `gotTxns` to `executing`, and assigns its committed vector timestamp `VTS` to the transaction's start vector timestamp:

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

Executing a transaction. We can now execute the operations of the transaction. Assume we start with a read operation `X :=read K`. There are three cases to consider: (i) the transaction has already written to key `K` (buffered in the write set); (ii) there is no preceding write in the transaction but the executing site replicates `K`; or (iii) neither (i) nor (ii) hold.

In case (i), the local variable `X` is given the value `V` buffered in the write set:

```
rl [execute-read-own-write] :
   < RID : Walter-Replica | executing : TRANSES
```

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

```
        < TID : Walter-Txn | operations : ((X :=read K) OPS),
                      writeSet : (K |-> V, WS), localVars : VARS > >
 =>
   < RID : Walter-Replica | executing : TRANSES
        < TID : Walter-Txn | operations : OPS,
                      writeSet : (K |-> V, WS),
                      localVars : insert(X,V,VARS) > > .
```

In case (ii) (the site `RID` replicates `K`: `localReplica(K,RID,REPLICA-TABLE)`), the replica chooses the last update `< V,VERSION >` in its local history `DS` that is visible to the transaction's start snapshot `VTS`:

```
crl [execute-read-local] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica | executing : TRANSES
          < TID : Walter-Txn | operations : ((X :=read K) OPS), writeSet : WS,
                      readSet : RS, localVars : VARS, startVTS : VTS >,
                    history : DS >
  =>
    < TABLE : Table | >
    < RID : Walter-Replica | executing : TRANSES
          < TID : Walter-Txn | operations : OPS,  localVars : insert(X,V,VARS),
                      readSet : (< K,VERSION >,RS) > >
  if (not $hasMapping(WS,K)) /\ localReplica(K,RID,REPLICA-TABLE) /\
     < V,VERSION > := choose(VTS,DS[K]) .
```

In case (iii), the site sends a `request` message (with the transaction's start vector timestamp `VTS`, since the remote site must choose the version consistent with the snapshot) to `K`'s preferred site (`preferredSite(...)`) to fetch the version. The "next operation" of the transaction changes to `waitRemote(K,X)`:

```
crl [execute-read-remote] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica | executing : TRANSES
          < TID : Walter-Txn | operations : ((X :=read K) OPS), writeSet : WS,
                      startVTS : VTS > >
  =>
    < TABLE : Table | >
    < RID : Walter-Replica | executing : TRANSES
```

```
          < TID : Walter-Txn | operations : (waitRemote(K,X) OPS) > >
    (msg request(K,TID,VTS) from RID to preferredSite(K,REPLICA-TABLE))
  if (not $hasMapping(WS,K)) /\ (not localReplica(K,RID,REPLICA-TABLE)) .
```

The remote (preferred) site responds to such a request by sending the snapshot-consistent value and version (`choose(VTS, DS[K])`) of the requested key:

```
rl [receive-remote-request] :
   (msg request(K, TID, VTS) from RID' to RID)
   < RID : Walter-Replica | history : DS >
 =>
   < RID : Walter-Replica | >
   (msg reply(TID, K, choose(VTS, DS[K])) from RID to RID') .
```

The executing site then merges the fetched value and version in the local history, and updates the read set and local variables:

```
rl [receive-remote-reply] :
   (msg reply(TID, K, < V,VERSION >) from RID' to RID)
   < RID : Walter-Replica | history : DS, executing : TRANSES
         < TID : Walter-Txn | operations : (waitRemote(K, X) OPS), readSet : RS,
                              localVars : VARS > >
 =>
   < RID : Walter-Replica | executing : TRANSES
         < TID : Walter-Txn | operations : OPS,
                              readSet : (< K, VERSION >, RS),
                              localVars : insert(X, V, VARS) >,
                           history : merge(K, < V,VERSION >, DS) > .
```

When the next transaction operation is a write operation `write(K, EXPR)`, the expression `EXPR` to be written is evaluated w.r.t. the current values of the local variables, and the resulting value is added to the write set:

```
rl [execute-write] :
   < RID : Walter-Replica | executing : TRANSES
         < TID : Walter-Txn | operations : (write(K, EXPR) OPS),
                              localVars : VARS, writeSet : WS > >
 =>
   < RID : Walter-Replica | executing : TRANSES
         < TID : Walter-Txn | operations : OPS,
                              writeSet : insert(K, eval(EXPR, VARS), WS) > > .
```

Commit a Transaction. When all the currently executing transaction's operations have been performed, the site starts to commit the transaction. A read-only transaction (`writeSet` is `empty`) is committed locally:

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

There are two cases for committing a write transaction: *fast commit* if the executing site is the preferred site of all keys written by the transaction; and *slow commit* if the transaction's write sets contains keys with non-local preferred sites.

Fast Commit. To fast commit a transaction, two checks for conflicts are performed at the site: one check for any modified key, and another check for any locked key, i.e., a key being committed concurrently by another transaction. `modified(WS,VTS,DS)` checks whether there is a key in the write set `WS` and a version of that key in the history `DS` that is not visible to the snapshot `VTS`, and `locked(WS,LOCKS)` checks whether there is a key in `WS` that also appears in `LOCKS`. The following rule shows the case when both checks succeed:

```
crl [fast-commit-success] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica | executing : TRANSES
          < TID : Walter-Txn | operations : nil, writeSet : WS,
                        startVTS : VTS, txnSQN : TXNSQN >,
                      committed : TRANSES', history : DS, locked : LOCKS,
                      sqn : SQN, committedVTS : VTS', dsSites : PSTS >
  =>
    < TABLE : Table | >
    < RID : Walter-Replica | executing : TRANSES,
                      committed : TRANSES' < TID : Walter-Txn | txnSQN : SQN' >,
                      history : update(WS,version(RID,SQN'),DS),
                      sqn : SQN',  committedVTS : insert(RID,SQN',VTS'),
                      dsSites : PSTS ; txnPropagateSites(TID,WS) >
    propagateTxn(TID,SQN',VTS,WS,allSites(REPLICA-TABLE),RID)
    if WS =/= empty /\ allLocalPreferred(WS,RID,REPLICA-TABLE) /\
       (not modified(WS,VTS,DS)) /\ (not locked(WS,LOCKS)) /\
       SQN' := SQN + 1 .
```

The site commits the transaction by assigning a new local sequence number `SQN'`, and updating the local history (`update(...)`). The site then propagates the transaction to remote sites. This is done by generating propagation messages using `propagateTxn`, which produces one propagation message for each site. The site then keeps track of the sites that have acknowledged the propagation (`txnPropagateSites(...)`).

If either check fails, the transaction is aborted:

```
crl [fast-commit-failed] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica | executing : TRANSES
          < TID : Walter-Txn | operations : nil, writeSet : WS, startVTS : VTS >,
                        aborted : TRANSES', history : DS, locked : LOCKS >
 =>
    < TABLE : Table | >
    < RID : Walter-Replica | executing : TRANSES,
                     aborted : TRANSES' < TID : Walter-Txn | > >
    if WS =/= empty /\ allLocalPreferred(WS,RID,REPLICA-TABLE) /\
       (modified(WS,VTS,DS) or locked(WS,LOCKS)) .
```

Slow Commit. Slow commit uses two-phase commit among the preferred sites of the keys in the transaction's write set. The executing site distributes the `prepare` messages to those preferred sites (`allPreferredSites(...)`), asking the participants to vote based on whether the corresponding keys are unmodified and unlocked. The `prepare` messages are produced by the function `prepareTxn`:

```
crl [slow-commit-prepare] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica | voteSites : VSTS, executing : TRANSES
        < TID : Walter-Txn | operations : nil, writeSet : WS, startVTS : VTS > >
  =>
    < TABLE : Table | >
    < RID : Walter-Replica | voteSites : (VSTS ; voteSites(TID,RIDS)),
                     executing : TRANSES < TID : Walter-Txn | > >
    prepareTxn(TID,keys(WS),VTS,RIDS,REPLICA-TABLE,RID)
    if WS =/= empty /\ (not allLocalPreferred(WS,RID,REPLICA-TABLE)) /\
       RIDS := allPreferredSites(WS,REPLICA-TABLE) /\ (not (TID in VSTS)) .
```

The receiver of a `prepare` message performs the two checks as in fast commit: if either check fails, a `false` vote is sent back; otherwise, the participant locks the key(s) and sends back a `true` vote:

```
rl [slow-commit-receive-prepare] :
   (msg prepare(TID,KS,VTS) from RID' to RID)
   < RID : Walter-Replica | locked : LOCKS, history : DS >
 =>
   if (not locked(KS,LOCKS)) and (not modified(KS,VTS,DS))
   then < RID : Walter-Replica | locked : (addLock(KS,TID),LOCKS) >
        (msg prepare-reply(TID,true) from RID to RID')
   else < RID : Walter-Replica | >
        (msg prepare-reply(TID,false) from RID to RID') fi .
```

When the executing replica receives a vote, it first checks whether all votes have been collected (`VSTS'[TID] == empty`), and then checks whether all votes associated to the transaction are `true` votes (`allYes(TID, VOTES')`). If so, the coordinator decides to propagate the transaction as in the fast commit; otherwise, the coordinator aborts the transaction, and notifies the participants that voted `true` to release the locks. This is done by producing "abort" messages for the corresponding participants `RIDS` (`propagateAbort(TID,RIDS,RID)`). The following rule shows the "aborted" branch:

```
crl [slow-commit-receive-vote-abort] :
    (msg prepare-reply(TID, FLAG) from RID' to RID)
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica | votes : VOTES, voteSites : VSTS,
                    abortSites : ABORTS >
  =>
    < TABLE : Table | >
    < RID : Walter-Replica | votes : VOTES', voteSites : VSTS',
                    abortSites : ABORTS ; voteSites(TID,RIDS) >
    propagateAbort(TID, RIDS, RID)
    if VSTS' := remove(TID, RID', VSTS) /\
       VOTES' := VOTES ; vote(TID, RID', FLAG) /\
       VSTS'[TID] == empty /\ (not allYes(TID, VOTES')) /\
       RIDS := yesSites(TID, VOTES') .
```

The abort procedure is straightforward: the participant releases the lock(s) held by the transaction `TID`, and the executing site aborts the transaction:

```
rl [slow-commit-receive-abort] :
   (msg abort(TID) from RID' to RID)
   < RID : Walter-Replica | locked : LOCKS >
 =>
   < RID : Walter-Replica | locked : release(TID,LOCKS) >
   (msg aborted(TID) from RID to RID') .

crl [slow-commit-receive-aborted] :
    (msg aborted(TID) from RID' to RID)
    < RID : Walter-Replica | executing : TRANSES < TID : Walter-Txn | >,
                     aborted : TRANSES', abortSites : ABORTS >
  =>
    (if ABORTS'[TID] == empty  --- all acks received; abort the txn locally
     then < RID : Walter-Replica | executing : TRANSES,
                             aborted : TRANSES' < TID : Walter-Txn | >,
                             abortSites : ABORTS' >
     else < RID : Replica | abortSites : ABORTS' > fi)
   if ABORTS' := remove(TID,RID',ABORTS) .
```

Transaction Propagation. After a transaction commits, the executing site propagates it to other sites by invoking the propagation protocol. Upon receiving a propagation message for transaction `TID`, the receiving site performs two checks: (i) whether it has gotten all transactions that causally precede transaction `TID`, and (ii) whether all transactions from `TID`'s executing site have a smaller sequence number. (i) is indicated by `VTS' gt VTS`, meaning that the latest snapshot the site got is greater than the incoming snapshot `VTS`, and (ii) by `s(VTS'[RID']) == SQN`, meaning that the corresponding latest sequence number the site got is exactly one smaller than the incoming sequence number `SQN`. If either check fails, the site buffers the propagated information regarding the transaction (`nonPropagatedTxns`), and waits until the "missing" transactions are propagated to it; otherwise, the transaction is considered to be propagated successfully (`propagatedTxns`), and the site updates its local history (if the site is not the coordinator itself), and then sends back the acknowledgment:

```
crl [receive-propagate] :
    (msg propagate(TID,SQN,VTS,WS) from RID' to RID)
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica | gotVTS : VTS', history : DS, recPropTxns : PTXNS >
  =>
    < TABLE : Table | >
    (if s(VTS'[RID']) == SQN and (VTS' gt VTS)
     then if RID =/= RID'
          then < RID : Walter-Replica | gotVTS : VTS'', history : DS',
                                  recPropTxns : PTXNS' >
               (msg propagate-ack(TID) from RID to RID')
          else < RID : Walter-Replica | gotVTS : VTS'', recPropTxns : PTXNS' >
               (msg propagate-ack(TID) from RID to RID')
          fi
     else < RID : Walter-Replica | recPropTxns : PTXNS'' >
     fi)
     if PTXNS' := propagatedTxns(TID,SQN,VTS) ; PTXNS /\
        PTXNS'' := nonPropagatedTxns(TID,SQN,VTS,WS,RID') ; PTXNS /\
        VTS'' := insert(RID',SQN,VTS') /\
        DS' := update(locRepWS(WS,RID,REPLICA-TABLE),version(RID',SQN),DS) .
```

A failed propagated transaction (nonPropagatedTxns) is acknowledged whenever those two checks pass. The site transforms nonPropagatedTxns to propagatedTxns, and sends back the acknowledgment:

```
crl [later-propagate-ack] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica | gotVTS : VTS', history : DS, recPropTxns :
                      (nonPropagatedTxns(TID, SQN, VTS, WS, RID') ; PTXNS) >
  =>
    < TABLE : Table | >
    (if RID =/= RID'
     then < RID : Walter-Replica | gotVTS : VTS'', history : DS', recPropTxns :
                            (propagatedTxns(TID, SQN, VTS) ; PTXNS) >
          (msg propagate-ack(TID) from RID to RID')
     else < RID : Walter-Replica | gotVTS : VTS'', history : DS, recPropTxns :
                            (propagatedTxns(TID, SQN, VTS) ; PTXNS) >
          (msg propagate-ack(TID) from RID to RID')
     fi)
    if s(VTS'[RID']) == SQN /\ VTS' gt VTS /\
       VTS'' := insert(RID', SQN, VTS') /\
       DS' := update(locRepWS(WS, RID, REPLICA-TABLE),version(RID', SQN), DS) .
```

When the executing site has collected propagation acknowledgments from $f + 1$ sites, it marks the transaction as disaster-safe durable. This is done by the function dsDurable, which counts the number of received acks in `dsSites`. The site also distributes the decision to all sites by using the function `dsDurableTxn` to produce a `ds-durable` message to each site, and records that information in `vsbSites`. If there is no need to distribute the decision, the transaction is marked as globally visible directly (by adding it to `gvTxns`):

```
crl [receive-propagate-ack] :
    (msg propagate-ack(TID) from RID' to RID)
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : Walter-Replica | dsSites : PSTS, vsbSites : VSBS,
                      committed : TRANSES < TID : Walter-Txn | writeSet : WS,
                          startVTS : VTS, txnSQN : SQN > ;; TXNS',
                      dsTxns : DSTXNS, gvTxns : GVTXNS >
  =>
    < TABLE : Table | >
    (if dsDurable(TID,PSTS')
     then if RIDS =/= empty
          then < RID : Walter-Replica | dsSites : PSTS', vsbSites : VSBS',
                                   dsTxns : (TID, DSTXNS) >
               dsDurableTxn(TID,RIDS,RID)
          else < RID : Walter-Replica | dsSites : PSTS', vsbSites : VSBS',
                                   dsTxns : (TID, DSTXNS),
                                   gvTxns : (TID, GVTXNS) >
          fi
     else < RID : Walter-Replica | dsSites : PSTS' >
     fi)
    if PSTS' := add(TID,keys(WS),RID',REPLICA-TABLE,PSTS) /\
       (not TID in DSTXNS) /\ RIDS := allServers(REPLICA-TABLE) \ RID /\
       VSBS' := VSBS ; voteSites(TID,RIDS) .
```

A propagation acknowledgment that arrives after the transaction has been marked as disaster-safe durable is ignored:

```
rl [receive-propagate-ack-after-ds-durable-mark] :
   (msg propagate-ack(TID) from RID' to RID)
   < RID : Walter-Replica | dsTxns : TID , DSTXNS >
 =>
   < RID : Walter-Replica | > .
```

Upon receiving the "disaster-safe durable" decision, the site tries to commit the transaction locally:

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

To commit transaction TID, the site must pass three checks: (i) the propagation message has been received and acknowledged (propagatedTxns(TID,SQN,VTS) shown in recPropTxns), (ii) VTS' is greater than VTS, and (iii) all transactions from TID's executing site with a smaller sequence number have been received (s(VTS'[RID']) == SQN). A visible message is then sent back, and all corresponding locks are released.

The site fails to commit the transaction immediately after receiving the decision if any check fails. The following rule shows the case when the site has not yet acknowledged the propagation:

```
rl [receive-ds-durable-not-visible-not-ack-propagated] :
   (msg ds-durable(TID) from RID' to RID)
   < RID : Walter-Replica | recPropTxns : (nonPropagatedTxns(TID,SQN,VTS,WS,RID')
                     ; PTXNS), recDurableTxns : DTXNS >
 =>
   < RID : Walter-Replica | recPropTxns : (nonPropagatedTxns(TID,SQN,VTS,WS,RID')
                     ; PTXNS), recDurableTxns : (nonDurableTxns(TID,RID')
                     ; DTXNS) > .
```

The site commits any failed committed transaction (nonDurableTxns) whenever those checks succeed, by changing nonDurableTxns to durableTxns. It also sends back a visible message, updates the committed vector timestamp, and releases all corresponding locks:

```
crl [later-visible] :
    < RID : Walter-Replica | recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
                      recDurableTxns : (nonDurableTxns(TID,RID') ; DTXNS),
                      committedVTS : VTS', locked : LOCKS >
```

```
  =>
    < RID : Walter-Replica | recPropTxns : (propagatedTxns(TID,SQN,VTS) ; PTXNS),
                             recDurableTxns : (durableTxns(TID) ; DTXNS),
                             committedVTS : insert(RID',SQN,VTS'),
                             locked : release(TID,LOCKS) >
    (msg visible(TID) from RID to RID')
    if VTS' gt VTS /\ s(VTS'[RID']) == SQN .
```

Finally, after receiving `visible` messages from all sites, the executing site marks the transaction as globally visible:

```
crl [receive-visible] :
    (msg visible(TID) from RID' to RID)
    < RID : Walter-Replica | vsbSites : VSBS, gvTxns : GVTXNS >
  =>
    (if VSBS'[TID] == empty
     then < RID : Walter-Replica | vsbSites : VSBS', gvTxns : (TID , GVTXNS) >
     else < RID : Walter-Replica | vsbSites : VSBS' >
     fi)
    if VSBS' := remove(TID,RID',VSBS) .
```

5.3 MODEL CHECKING SI AND PSI

As shown in Table 3.1, we have applied our tool CAT to modeling check the Maude model of Walter against nine properties. All model checking results are as expected. In particular, the Walter model satisfies consistency properties weaker than or equal to PSI, and violates SI and any stronger consistency property.

Here we focus on PSI and SI, and investigate in detail whether or not each sub-property (defined in Section 3.3) of them is satisfied by our Walter model. We have extended the tool CAT with the associated five sub-properties: *snapshot read* (SI-1), *no write-write conflicts* (SI-2), *site snapshot read* (PSI-1), *no write-write conflicts* (PSI-2), and *commit causality across sites* (PSI-3). For example, CAT executes the following command to search, from all generated initial states (with one read-only transaction, two read-write transaction, two operations per each type of transaction, two sites, two keys, and two replicas per key), for one reachable final state where the consistency property PSI-1 does not hold:

```
search [1] init(1,0,2,2,0,2,0,2,2,2) =>! C:Configuration
    < M:Oid : Monitor | log: LOG:Log  clock: N:Nat >
        such that notSiteSnapshotRead(LOG:Log) .
```

Similarly, either of the two results is output: "`No solution`," meaning that all runs from all the given initial states satisfy the desired consistency property, or a counterexample (in Maude at the moment) showing a behavior that violates the property.

Table 5.1: Model Checking Results w.r.t. SI and PSI. "✓" and "×" refer to satisfying or violating the property, respectively.

	SI-1	SI-2	PSI-1	PSI-2	PSI-3
1 read-only, 2 read-write	×	×	✓	✓	✓
1 read-only, 1 write-only, 1 read-write	×	×	✓	✓	✓
3 read-write	×	×	✓	✓	✓
2 read-only, 1 read-write	×	✓	✓	✓	✓
2 read-only, 1 write-only	×	✓	✓	✓	✓

Table 5.1 summarizes our analysis results with the initial states generated with different combinations of three transactions (all cases share two operations per each type of transaction, two sites, two keys, and two replicas per key). In particular, SI-1 is violated in all the cases, despite of SI-2 being satisfied in some cases.

5.4 STATISTICAL MODEL CHECKING OF WALTER

To estimate the performance of Walter we adapt its model specified in the CAT framework into the actor-based framework [5] for statistical model checking using PVeStA.[2]

5.4.1 A Probabilistic Model of Walter

Following the approach in Section 4.5 we eliminate nondeterminism in our Walter model by probabilistically assigning to each message a delay. Specifically, the (manual) transformation wraps the messages in the original Walter model in two ways: (i) an incoming messages is wrapped with the current global time, and (ii) an outgoing message is wrapped with a delay sampled probabilistically from a certain distribution of network latency (e.g., the lognormal distribution [17]).

Example 5.1. In the transformed rule below (the original rule is in black), the incoming message `request` is equipped with the current global time `GT`, and the outgoing message `reply` is equipped with a `delay`:

```
rl [receive-remote-request-prob] :
```

[2]The entire probabilistic model is available at `https://sites.google.com/site/siliunobi/walter`.

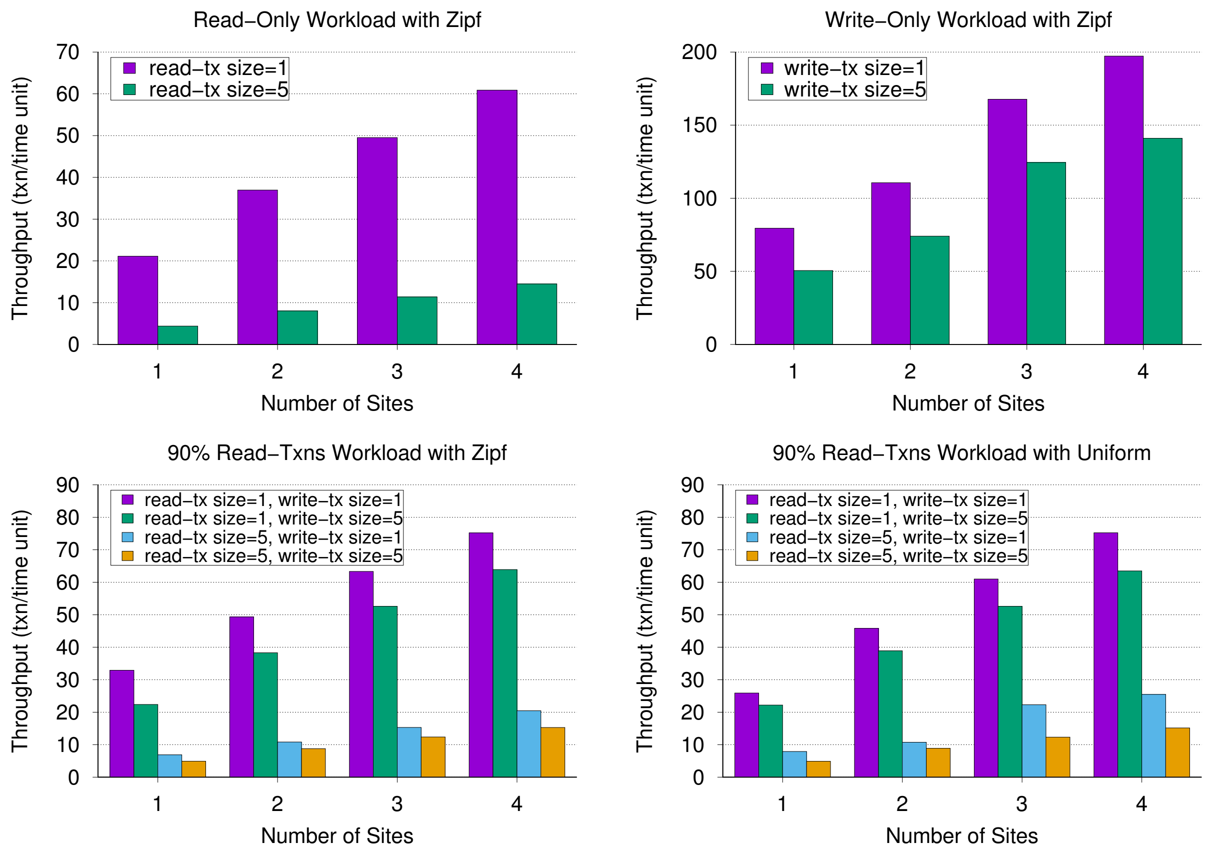


Figure 5.1: Throughput with fast commit under different workloads.

```
   {GT, msg request(K,TID,VTS) from RID' to RID}
   < RID : Walter-Replica | history : DS >
=>
   < RID : Walter-Replica | >
   [delay, msg reply(TID,K,choose(VTS,DS[K])) from RID to RID'] .
```

where the `request` message is consumed by `RID` at the global time `GT`, and the `reply` message will be consumed by `RID'` after `delay` time units. `delay` of sort `Float` is a parameter instantiated with a certain probability distribution (see Section 4.5).

5.4.2 Statistical Model Checking Results

Experimental Setup. We performed our experiments with 100 (read-only and/or write-only) transactions, 1 or 5 operations per transaction, 100 keys, and up to 4 sites (the number of sites and the transaction size are the same as in the experiments in [85]). All initial states

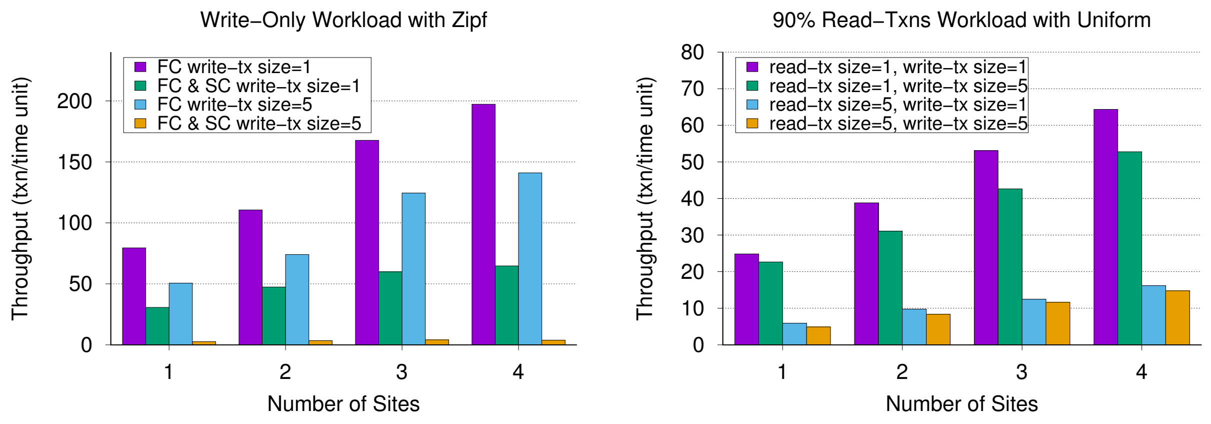


Figure 5.2: Throughput with fast commit (FC) and slow commit (SC).

are generated in the same manner as in Section 4.7. To capture the fact that some keys may be accessed more frequently than others, we use Zipfian distributions in our experiments. We also use lognormal message delay distributions with parameters $\mu = 3$ and $\sigma = 1$ for local delays, and $\mu = 1$ and $\sigma = 2$ for remote delays.

The plots in Fig. 5.1 show the *throughput* with only fast commit as a function of the number of sites, with read-only, write-only or 90% reads workload, and with uniform and Zipfian distributions. The plots show that read throughput scales nearly linearly with the number of sites; write throughput also grows with the number of sites, but not linearly. With a mixed workload, throughput is mostly determined by the transaction size. Our statistical model checking results are consistent with the system evaluation results in [85] (see Appendix C). For uniform distribution we only plot the results with a mixed workload.

The plots in Fig. 5.2 show the throughput with mixed (both fast and slow) commit protocols under the same experimental settings as in Fig. 5.1. As shown in the left plot, throughput is mostly determined by the transaction size in the mixed workload; the trends of, and the differences among, various transaction sizes are consistent with those in Fig. 5.1. We only plot the results with Zipfian distribution, which are consistent with those with uniform distribution.

Our probabilistic model of Walter, including the infrastructure for statistical model checking, is around 1.8K LOC. Computing the probabilities took a couple of minutes on 30 servers, each with a 64-bit Intel Quad Core Xeon E5530 CPU with 12 GB memory. Each point in the plots represents the average of 3 statistical model checking results. The confidence level for all our statistical experiments is 95%.

5.5 CONCLUDING REMARKS

We have formally analyzed and verified in Maude the design of Walter [85], a partially replicated distributed data store providing multi-partition transactions and guaranteeing parallel snapshot isolation (PSI), an important consistency property that offers attractive performance while providing adequate guarantees for certain kinds of applications. No formal specification of Walter existed before this work. Furthermore, PSI was only informally described by pseudo-code in [85] and no formal verification existed. This work has used model checking and systematic generation of initial states to verify that Walter satisfies PSI for all such states. We have also extended the Maude specification of Walter to model time and probabilistic communication delays as a probabilistic rewrite theory, and have then used statistical model checking analysis to study Walter's latency and throughput performance for a wide range of workloads. The results of the statistical model checking analysis are consistent with the experimental results in [85] but offer also new insights about Walter's performance for a wider range of workloads than those evaluated experimentally in [85].

CHAPTER 6: CASE STUDY: THE ROLA TRANSACTIONAL PROTOCOL

Distributed transaction protocols are complex distributed systems whose design is quite challenging because: (i) as for other distributed systems, validating correctness is very hard to achieve by testing alone; (ii) the high performance requirements needed in many applications are hard to measure before implementation and expensive to compare across different implementations; and (iii) there is an unavoidable tension between the *degree of consistency* needed for the intended applications and the *high performance* required of the transaction protocol for such applications: balancing well these two requirements is essential.

In this chapter,[1] we present our results on how to use formal modeling and analysis as early as possible in the design process to arrive at a mature design of a *new* distributed transaction protocol, called ROLA ("*R*ead at*O*micity and prevention of *L*ost upd*A*tes"), meeting specific correctness and performance requirements *before* such a protocol is implemented. In this way, the above-mentioned design challenges (i)–(iii) can be adequately met. We also show how using this formal design approach it is relatively easy to *compare* ROLA with other existing transaction protocols. This is also part of meeting design challenge (iii), since the key comparisons focus on how well each protocol balances the consistency vs. performance trade-offs for the intended applications.

ROLA in a Nutshell. Different applications require negotiating the consistency vs. performance trade-offs in different ways. The key issue is the application's required *degree of consistency*, and how to meet such requirements with *high performance*. Cerone et al. [29] survey a *hierarchy of consistency models* for distributed transaction protocols. Three of the weakest consistency models in [29] are: *read atomicity* (RA), *causal consistency* (CC),[2] and *parallel snapshot isolation* (PSI).

A key property of transaction protocols is the *prevention of lost updates* (PLU). The weakest consistency model in [29] satisfying both RA and PLU is PSI. However, PSI, and the well-known protocol Walter [85] implementing PSI, also guarantee CC. Furthermore, in [9], Ardekani et al. propose a consistency model called *non-monotonic snapshot isolation* (NMSI)—and a distributed transaction protocol called Jessy that implements NMSI—that is weaker than PSI, but still satisfies RA, CC, and PLU. To the best of our knowledge, up to now NMSI has in fact been the weakest consistency model satisfying both RA and PLU,

[1]This chapter is based on the work [59, 60, 62] supported in part by NSF CNS 1409416, NSF CNS 1319527, AFOSR/AFRL FA8750-11-2-0084, NRL contract N00173-17-1-G002, and NSFC Project No. 61872146.

[2]CC strengthens RA as follows: If transaction T_2 is *causally dependent* on transaction T_1, then if another transaction sees the updates by T_2, it must also see the updates of T_1 (e.g., if A posts something on a social media, and C sees B's comment on A's post, then C must also see A's original post).

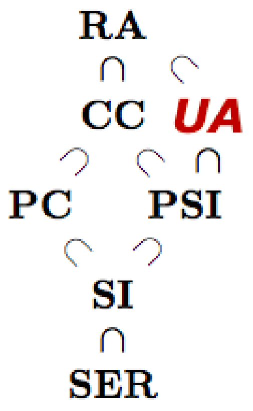


Figure 6.1: ROLA's *update atomic* (UA) consistency model added to the hierarchy of consistency models of Cerone et al. [29]

which means that all current such models also satisfy CC. However, Cerone et al. conjecture in [29] that a system guaranteeing RA and PLU *without* guaranteeing CC should be useful:

> "*existing consistency models do not include a counterpart of Read Atomic obtained by adding the* NoConflict *axiom [preventing lost updates]. Such an 'Update Atomic' consistency model would prevent lost update anomalies without having to enforce causal consistency [...]. Update Atomic could be particularly useful [...].*"

There was until now no distributed database design supporting such "update atomicity" without also providing CC. Filling this gap, that is, presenting a design, ROLA, that does exactly this for multi-partition transactions, is what we do in this chapter. As shown in Fig. 6.1, where we have added ROLA's *update atomic* (UA) consistency model to the hierarchy of consistency models in [29], UA is strictly stronger than RA, incomparable with CC, and strictly weaker than PSI (and NMSI).

The main idea of the ROLA algorithm is to extend the RAMP algorithm of Bailis et al. [14], that ensures read atomicity for partitioned data stores (i.e., data are partitioned across widely distributed data centers, but are not replicated) where a transaction can read and/or write data stored at different partitions, by adding mechanisms for preventing lost updates. Therefore, unlike Jessy and Walter, which support partially replicated data stores, ROLA, like RAMP, at the moment only targets partitioned data stores.

Two key questions about ROLA's design are:

1. Are there *natural applications* needing high performance where RA plus PLU provide a sufficient degree of consistency?

2. Can the new ROLA design meeting RA plus PLU *outperform* existing designs, like Walter and Jessy, that also guarantee RA and PLU?

Regarding question (a), an example of a transaction that requires RA and PLU but not CC is the "becoming friends" transaction on social media. Bailis et al. [14] point out that RA is crucial for this operation: If Edinson and Neymar become friends, then Thomas should not see a *fractured read* where Edinson is a friend of Neymar, but Neymar is not a friend of Edinson. An implementation of "becoming friends" must obviously guarantee PLU: the new friendship between Edinson and Neymar must not be lost. Finally, CC could be sacrificed for the sake of performance: Assume that Dani is a friend of Neymar. When Edinson becomes Neymar's friend, he sees that Dani is Neymar's friend, and therefore also becomes friend with Dani. The second friendship therefore causally depends on the first one. However, it does not seem crucial that others are aware of this causality: If Thomas sees that Edinson and Dani are friends, then it is not necessary that he knows that (this happened *because*) Edinson and Neymar are friends.

Regarding question (b), Section 6.5 shows that ROLA clearly outperforms both Walter and Jessy in all performance requirements for all read/write transaction rates. For a fair comparison, we have compared the performance of ROLA with those of Jessy and Walter without their replication features.

Main Contributions include: (1) the design, formal modeling, and model checking analysis of ROLA, a new transaction protocol having useful applications and meeting RA and PLU consistency properties with competitive performance; (2) a detailed performance comparison by statistical model checking between ROLA and the Walter and Jessy protocols showing that ROLA outperforms both Walter and Jessy in all such comparisons, including higher throughput and lower average latency; (3) to the best of our knowledge the first demonstration that, by a suitable use of formal methods, a completely new distributed transaction protocol can be designed and thoroughly analyzed, as well as be compared with other designs, very early on, *before* its implementation.

This chapter is structured as follows: Section 6.1 presents an informal overview of ROLA. Section 6.2 gives an informal correctness argument that ROLA satisfies RA and PLU. Section 6.3 presents our executable specification of ROLA in the CAT framework (Chapter 3). Section 6.4 explains how we can use Maude reachability analysis to automatically check whether ROLA satisfies the desired properties. Section 6.5 shows how we can estimate the performance of ROLA, Walter, and Jessy by using statistical model checking. Finally, Section 6.6 ends this chapter with some concluding remarks.

6.1 ROLA TRANSACTIONS

This section gives an informal overview of ROLA (Algorithm 6.1) that guarantees both RA and PLU, but not CC, for transactions accessing multiple partitions in a setting where the data are *partitioned* (but not replicated) across a number of widely distributed sites.

ROLA extends RAMP-Fast (see Chapter 4 and Fig.A.1 in Appendix A) to also ensure PLU. RAMP-Fast guarantees RA, but not PLU, since it allows a write to overwrite conflicting writes: When a partition commits a write, it only compares the write's timestamp t_1 with the local latest-committed timestamp t_2, and updates the latest-committed timestamp with t_1 or t_2. If the two timestamps are from two conflicting writes, then one of the writes is lost.

ROLA's key idea to prevent lost updates is to sequentially order writes on the same key from a partition's perspective by adding to each partition a map which maps each incoming version to an incremental sequence number. For example, suppose the transactions T_1 and T_2 both read version x_0 (with mapped sequence number 0, or the version 0) of the key x, and both try to write the respective versions x_1 and x_2; then if T_1 manages to write x_1 first (with mapped sequence number 1), T_2 is not allowed to overwrite x_1, since the local sequence number has increased by the time T_2 tries to write x_2. For write-only transactions the mapping can always be built; for a read-write transaction the mapping can only be built if there has not been a mapping built since the transaction fetched the value. This can be checked by comparing the last prepared version's timestamp's mapping on the partition with the fetched version's timestamp's mapping. In this way, ROLA prevents lost updates by allowing versions to be prepared only if no conflicting prepares occur concurrently.

More specifically, ROLA adds two partition-side data structures (Lines 3-4, Algorithm 6.1): *sqn*, denoting the local sequence number counter, and *seq*[*ts*], that maps a timestamp to a local sequence number. ROLA also changes the data structure of *versions* in RAMP from a set to a list (Line 1, Algorithm 6.1). ROLA then adds two methods to the existing RAMP-F functionality: the coordinator-side[3] method UPDATE (Lines 17-28, Algorithm 6.1) and the partition-side method PREPARE_UPDATE (Lines 5-10, Algorithm 6.1) for read-write transactions. Furthermore, ROLA modifies two partition-side methods in RAMP: PREPARE (Lines 11-12, Algorithm 6.1), besides adding the version to the local store, maps its timestamp

[3]The *coordinator*, or *client*, is the partition executing the transaction.

Algorithm 6.1 ROLA

```
    Server-side Data Structures
1:  $versions$: list of versions $\langle$item, value, timestamp $ts_v$, metadata $md\rangle$
2:  $latestCommit[i]$: last committed timestamp for item $i$
3:  $seq[ts]$: local sequence number mapped to timestamp $ts$
4:  $sqn$: local sequence counter

    Server-side Methods
    GET same as in RAMP-Fast (see Appendix A)

5:  procedure PREPARE_UPDATE($v$ : version, $ts_{prev}$ : timestamp)
6:      $latest \leftarrow$ last $w \in versions : w.item = v.item$
7:      if $latest =$ NULL or $ts_{prev} = latest.ts_v$ then
8:          $sqn \leftarrow sqn + 1$;  $seq[v.ts_v] \leftarrow sqn$;  $versions$.add($v$)
9:          return ACK
10:     else return $latest$

11: procedure PREPARE($v$ : version)
12:     $sqn \leftarrow sqn + 1$;  $seq[v.ts_v] \leftarrow sqn$;  $versions$.add($v$)

13: procedure COMMIT($ts_c$ : timestamp)
14:     $I_{ts} \leftarrow \{w.item \mid w \in versions \wedge w.ts_v = ts_c\}$
15:     for $i \in I_{ts}$ do
16:         if $seq[ts_c] > seq[latestCommit[i]]$ then $latestCommit[i] \leftarrow ts_c$
    ----------------------------------------------------------------
    Coordinator-side Methods
    PUT_ALL, GET_ALL same as in RAMP-Fast (see Appendix A)

17: procedure UPDATE($I$ : set of items, $OP$ : set of operations)
18:     $ret \leftarrow$ GET_ALL($I$); $ts_{tx} \leftarrow$ generate new timestamp
19:     parallel-for $i \in I$ do
20:         $ts_{prev} \leftarrow ret[i].ts_v$;  $v \leftarrow ret[i].value$
21:         $w \leftarrow \langle item = i, value = op_i(v), ts_v = ts_{tx}, md = (I - \{i\})\rangle$
22:         $p \leftarrow$ PREPARE_UPDATE($w$,$ts_{prev}$)
23:         if $p = latest$ then
24:             invoke application logic to, e.g., abort and/or retry the transaction
25:     end parallel-for
26:     parallel-for server $s$ : $s$ contains an item in $I$ do
27:         invoke COMMIT($ts_{tx}$) on $s$
28:     end parallel-for
```

to the increased local sequence number; and COMMIT (Lines 13-16, Algorithm 6.1) marks versions as committed and updates an index containing the highest-sequenced-timestamped committed version of each item. These two partition-side methods apply to both write-only and read-write transactions. ROLA invokes RAMP-Fast's PUT_ALL, GET_ALL and GET methods to deal with read-only and write-only transactions (see Fig.A.1 in Appendix A).

ROLA starts a read-write transaction with the UPDATE procedure. It invokes RAMP-Fast's GET_ALL method to retrieve the values of the items the client wants to update, as well as their corresponding timestamps (Line 18, Algorithm 6.1). ROLA writes then proceed in two phases: a first round of communication places each timestamped write on its respective partition. The timestamp of each version obtained previously from the GET_ALL call is also packaged in this *prepare* message (Lines 19-25, Algorithm 6.1). A second round of communication marks versions as committed (Lines 26-28, Algorithm 6.1).

At the partition's side, the partition begins the PREPARE_UPDATE routine by retrieving the last version in its *versions* list with the same item as the received version (Line 6, Algorithm 6.1). If such a version is not found, or if the version's timestamp ts_v matches the passed-in timestamp ts_{prev} (Line 7, Algorithm 6.1), then the version is deemed prepared. The partition keeps a record of this locally by incrementing a local sequence counter and mapping the received version's timestamp ts_v to the current value of the sequence counter (Line 8, Algorithm 6.1). Finally, the partition returns an ACK to the client (Line 9, Algorithm 6.1). If ts_{prev} does not match the timestamp of the last version in *versions* with the same item, then this *latest* timestamp is simply returned to the coordinator (Line 10, Algorithm 6.1).

If the coordinator receives an ACK from PREPARE_UPDATE, it immediately commits the version with the generated timestamp ts_{tx} (Line 27, Algorithm 6.1). If the returned value is instead a timestamp, the transaction is aborted (Lines 23-24, Algorithm 6.1).

Example 6.1. Assume that we have two data items, x and y, and two partitions, P_x and P_y, storing x and y, respectively. As depicted in Figure 6.2, two read-write transactions $T_1 : r(y); w(x_1); w(y_1)$ and $T_2 : r(y); w(y_2)$ are attempting concurrent writes, and a read-only transaction $T_3 : r(x); r(y)$ proceeds while T_1 is writing. T_1 and T_2 read the same version y_0. Both T_1 and T_2 perform the two-phase commit protocol on two partitions, P_x and P_y. However, T_2 fails to prepare y_2 after T_1 has prepared y_1, because when T_2's prepare arrives at P_y, the timestamp of the last version store on P_y is 1, which is not equal to $ts_{prev} = 0$ in T_2's prepare. T_2, upon receiving the returned version y_1, could abort the transaction or retry with a new transaction on y_1. Either way, the lost update problem is avoided. Regarding the case with T_1 and T_3, T_3 reads from P_x after P_x has committed T_1's write to x, but T_3 reads from P_y before P_y has committed T_1's write to y. Thus, T_3's first-round reads would

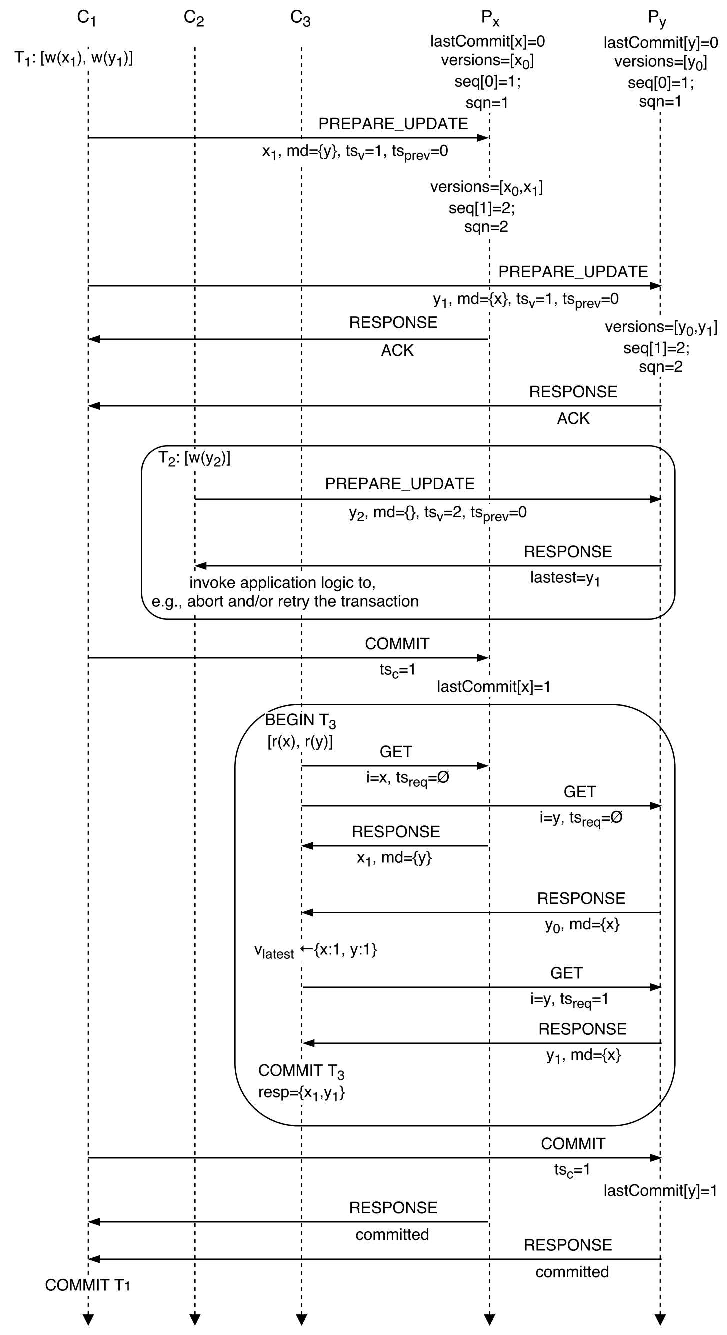


Figure 6.2: ROLA execution with three transactions in Example 6.1. For simplicity, we assume that T_1 and T_2 have fetched the same version y_0 (i.e., $ts_{prev} = 0$), when the sequence chart starts.

violate RA if it returns them. Using the metadata attached to its first-round reads, T_3 determines to issue a second-round read to fetch the missing data from P_y. After completing the second-round read, T_3 can safely return T_1's writes, not violating RA. Note that in this example RAMP would allow T_2 to commit, thus overwriting T_1's writes, which are then lost.

6.2 CORRECTNESS ARGUMENT FOR ROLA

In this section we give a somewhat informal correctness argument or proof sketch for ROLA. Since Section 6.3 defines a *formal* model of ROLA, we could—and should in the future—formally prove that our formal model of ROLA satisfies RA and PLU.[4]

6.2.1 Why ROLA Works

ROLA uses a two-phase commit protocol in order to detect concurrent writes. The first phase declares an intent to commit a write at the partition. Concurrent writes race to the partition without coordinating with each other. The partition can accept a preparation if there is no other prepared version after the latest commit associated with the incoming preparation. This in effect imposes a total order on the preparations, and thus on the commits, from the partition's perspective. In other words, the partition sees no logically concurrent updates. Our algorithm therefore provides read atomicity, and prevents updates from being lost, as concurrent updates are a necessary condition for lost updates.

By leveraging the partition-side sequence counter to commit, ROLA not only prevents lost updates, but also makes writes progress at the partition-side, and thus more recent prepared version can be reflected (we refer to this as ROLA's *progress property*). This is different from RAMP-Fast, where later prepared writes may never be fetched by reads as *latestCommit* only updates by simply comparing the coordinator-side timestamps.

6.2.2 Formalizing Consistency Models

We consider transactions to be ordered sequences of reads and writes to arbitrary sets of data items. Each data item has a single logical copy. We call the set of data items a transaction respectively reads or writes its *read set*, resp. *write set*. Each write creates a version of a data item. We identify versions of items by a timestamp from a totally ordered

[4]A fully mechanized proof of the correctness of ROLA (e.g., by using a theorem prover such as Coq [21], or the constructor-based reachability logic tool [84]) would be needed to claim that ROLA satisfies its correctness requirements, which is an important next step.

set (e.g., natural numbers) which is unique across all versions of each item. Thus, timestamps induce a total order on versions of each item. We denote version i of item x as x_i. Given two versions x_i and x_j, we write $x_i < x_j$ if x_j appears later than x_i in the version order, and write $x_i <_{next} x_j$ if x_j is x_i's next version. Each item x has an initial version x_0. Each transaction finishes with being either committed or aborted. A *history* consists of a set of transactions, together with the versions the transactions read and/or wrote.

Our correctness argument for ROLA, like that for RAMP in [14], is based on Adya's formalization of consistency models [4]. Following the formal reasoning about RAMP in [14], we also use Adya's formalization in the context of the above system model. We recall here the definitions from [4, 14] characterizing the various properties that ROLA (as well as RAMP [14]) should satisfy.

Between two transactions there may be three types of dependencies: read-dependencies, write-dependencies and anti-dependencies.

Definition 6.1. (*Read-Dependency*). Transaction T_j directly read-depends on T_i if transaction T_j reads the version x_i that T_i has written.

Definition 6.2. (*Write-Dependency*). Transaction T_j directly write-depends on T_i if transaction T_i writes a version x_i and T_j writes x_i's next version x_j.

Definition 6.3. (*Anti-dependency*). Transaction T_j directly anti-depends on T_i if transaction T_i reads some version x_h, and T_j writes x_h's next version x_j.

Definition 6.4. (*Direct Serialization Graph*). A direct serialization graph (DSG) w.r.t. a history H, denoted by $DSG(H)$, is defined as a directed graph such that:

- each node in the graph corresponds to a committed transaction;
- each directed edge corresponds to a type of direct dependency: there is a read-/write-/anti-dependency edge from T_i to T_j if T_j directly read-/write-/anti-depends on T_i.

In our model a transaction could also be a read-write transaction, in addition to read-only and write-only transactions, which are the only ones considered in [14].

We can formalize various anomalies for distributed transactions in terms of DSGs. These anomalies are then used to define consistency models.

Definition 6.5. (*G0: Write Cycles*). A history H exhibits phenomenon *G0* if $DSG(H)$ contains a directed cycle consisting entirely of write-dependency edges.

Definition 6.6. (*G1a: Aborted Reads*). A history H exhibits phenomenon *G1a* if H contains an aborted transaction T_a and a committed transaction T_c such that T_c reads a version written by T_a.

Definition 6.7. (*G1b: Intermediate Reads*). A history H exhibits phenomenon *G1b* if H contains a committed transaction T_i that reads a version x_j written by T_j, and T_j also wrote a version x_k such that $j < k$.

Definition 6.8. (*G1c: Circular Information Flow*). A history H exhibits phenomenon $G1c$ if $DSG(H)$ contains a directed cycle that consists entirely of read-dependency and write-dependency edges.

Besides the above criteria, we need the definition of *fractured reads* to define RA.

Definition 6.9. (*Fractured Reads*). A transaction T_j exhibits the *fractured reads* phenomenon if some transaction T_i writes versions x_a and y_b (in any order, where x and y may or may not be distinct items), and some transactions T_j reads versions x_a and y_c, and $c < b$.

As defined in [14]: RA isolation prevents fractured reads, and transactions from reading uncommitted, aborted, or intermediate versions:

Definition 6.10. (*Read Atomicity*). A system provides RA isolation if it prevents the phenomena $G0$, $G1a$, $G1b$, $G1c$, and fractured reads.

Lost updates (LU) happen when two transactions simultaneously make conditional modifications to the same data item(s).

Definition 6.11. (*Lost Updates*). A history H exhibits the phenomenon LU if $DSG(H)$ contains a directed cycle that consists of one or more anti-dependency edges and all edges are by the same data item.

6.2.3 Proof Sketch of ROLA's RA and PLU Properties

We base our proof of ROLA satisfying RA and PLU on Definitions 6.10 and 6.11.

To prove that ROLA provides RA we must, according to Definition 6.10, prove that ROLA prevents the phenomena $G0$, $G1a$, $G1b$, $G1c$, and fractured reads. The proof is in general quite similar to that of RAMP providing RA (Appendix B in [14]), since ROLA reads are the same as RAMP reads, and ROLA writes are more restricted than RAMP's, thus decreasing the possibility of violating RA.

Lemma 6.1. ROLA prevents the phenomenon $G0$.

Proof Sketch. Each partition has a local sequence number that increases once a version is prepared; the increased sequence number is mapped to that version. Thus, versions (whether or not for the same item) on a partition are totally ordered. That is, there is no directed cycle consisting entirely of write-dependency edges. □

Lemma 6.2. ROLA prevents the phenomenon $G1a$.

Proof Sketch. ROLA first-round reads access *lastCommit*, so each version fetched by a first-round read is written by a committed transaction. ROLA second-round reads only access the versions in the same transaction as for the versions fetched by the first-round reads, which are also committed. Thus, ROLA never reads aborted writes. □

Lemma 6.3. ROLA prevents the phenomenon $G1b$.

Proof Sketch. The proof follows directly from Lemma 6.2. □

Lemma 6.4. ROLA prevents the phenomenon $G1c$.

Proof Sketch. Writes (on possibly different data items) in a transaction are assigned the same timestamp, which prevents read-dependency and write-dependency cycles. □

To prove ROLA preventing fractured reads, we first introduce the notions of *sibling versions*, *sibling item*, *companion version*, and *companion sets*.

Definition 6.12. (*Sibling Versions*). The set of versions produced by a transaction are called sibling versions.

Definition 6.13. (*Sibling Item*). Data item x is called a sibling item to a version y_j if there exists a version x_k written in the same transaction as y_j.

Definition 6.14. (*Companion Version*). Version x_i is a companion version of y_j if x_i is a sibling version of y_j or if the transaction that wrote y_j also wrote x_k and $i > k$.

Definition 6.15. (*Companion Sets*). A set of versions V is a companion set if, for every pair (x_i, y_j) of versions in V where x is a sibling item of y_j, x_i is a companion version of y_j.

Lemma 6.5. (*Atomicity of Companion Sets*). In the absense of $G1c$ phenomena, if the set of versions read by a transaction is a companion set, the transaction does not exhibit fractured reads.

Proof Sketch. If V is a companion set, then every version x_i in V is a companion version of every other version y_j in V that includes x in y_j's sibling items. Suppose V has fractured reads. According to Definition 6.9, there are two versions x_i and y_j such that the transaction that wrote y_j also wrote a version x_k with $i < k$. However, in this case x_i is not a companion version of y_j according to Definition 6.14. Therefore we reach a contradiction. □

Lemma 6.6. ROLA reads assemble a companion set.

Proof Sketch. Without loss of generality, suppose a transaction reads two versions x_i and y_j, and x is y_j's sibling item. The following continues the proof by comparing i and j:

- Case 1. If $i \geq j$, then x_i is already a companion version of y_j, and the set is therefore a companion set.
- Case 2. If $i < j$, then ROLA invokes RAMP-Fast's GET_ALL method to issue a second-round read to fetch the companion version x_j. Whether x_j has been committed or not by the time the second-round read reaches the partition, the ROLA partition invokes RAMP-Fast's GET method to return the prepared version x_j in *versions*.

Therefore, the resulting set of versions is a companion set. □

Theorem 6.7. ROLA guarantees RA.

Proof Sketch. The proof follows directly from Lemmas 6.1–6.6. □

To prove ROLA preventing LU (Definition 6.11), we must first prove some lemmas.

Lemma 6.8. Versions are ordered by the arrival order of the corresponding *prepare* messages.

Proof Sketch. The proof follows directly from Lemma 6.1. □

Lemma 6.9. Given a history H that is valid under ROLA, then each node in $DSG(H)$ directly read-depends on at most one other node with the same data item.

Proof Sketch. Suppose we have a transaction T_i in $DSG(H)$ that directly read-depends on two different transactions T_j and T_k with item x, namely $T_j \xrightarrow{r} T_i$ and $T_k \xrightarrow{r} T_i$. By Lemma 6.8, $x_j < x_k$ or $x_k < x_j$. In either case T_i exhibits fractured reads and H is not valid under RA (and thus under ROLA), a contradiction. □

Lemma 6.10. If a (read-write) transaction reads version x_i, and then writes version x_j, then $x_i <_{next} x_j$.

Proof Sketch. Since x_j has been prepared, it means that there was no prepared versions between x_i and x_j; otherwise the condition cannot be satisfied. Thus $x_i <_{next} x_j$. □

Lemma 6.11. Given a history H that is valid under ROLA, each node in $DSG(H)$ is then directly write-dependent on at most one other node with the same data item.

Proof Sketch. Suppose we have a transaction T_i in $DSG(H)$ that is directly write-dependent on two different transactions T_j and T_k with item x, namely, $T_i \xrightarrow{w} T_j$ and $T_i \xrightarrow{w} T_k$. We then have the following cases:

- Case 1. T_j and T_k are both write-only transactions. Say T_j prepares first, and we have $seq[x_j] < seq[x_k]$ because when prepared, the sequence number increases.
  - Case 1.1. If T_j also commits first, T_j overwrites T_i and we have $T_i \xrightarrow{w} T_j$. Currently, $latestCommit[x] = ts_j$. When T_k commits later, $latestCommit[x]$ is mapped to ts_k because $seq[ts_k] > seq[ts_j]$. Then we have $T_j \xrightarrow{w} T_k$, a contradiction.
  - Case 1.2. If T_k commits first, T_k overwrites T_i and we have $T_i \xrightarrow{w} T_k$. Currently, $latestCommit[x] = ts_k$. When T_j commits later, $latestCommit[x]$ is not updated because $seq[ts_j] < seq[ts_k]$. Then we only have $T_i \xrightarrow{w} T_k$, a contradiction.
- Case 2. T_j is a read-write transaction, while T_k is a write-only transaction. According to Lemma 6.10, T_j reads T_i. Thus T_k cannot prepare first because in that case T_j aborts. The proof then follows directly from the above Case 1.1.
- Case 3. T_j and T_k are both read-write transactions. According to Lemma 6.10, both T_j and T_k read T_i. Either of them prepares first, and then the other one has to abort (the later prepare's ts_{prev} does not match the latest version's, i.e., the other prepare's, timestamp).

□

Lemma 6.12. For any history H that is valid under ROLA, $DSG(H)$ does not contain a sequence $T_h \xrightarrow{anti} T_i \xrightarrow{anti} T_j$.

Proof Sketch. There is a transaction $T_{i'}$ whose version written $x_{i'}$ is read by T_i, namely $T_{i'} \xrightarrow{r} T_i$. T_j writes $x_{i'}$'s next version x_j, namely $T_{i'} \xrightarrow{w} T_j$. According to Lemma 6.10 and Lemma 6.11, we have $T_{i'} \neq T_h$. From $T_h \xrightarrow{anti} T_i$ we can build another two dependencies $T_{h'} \xrightarrow{r} T_h$ and $T_{h'} \xrightarrow{w} T_i$. We now have $x_{i'} <_{next} x_i$ and $x_{h'} <_{next} x_i$. According to Lemma 6.8 we must have $x_{i'} = x_{h'}$, and then we have $T_{i'} \xrightarrow{w} T_i$. Since we already have $T_{i'} \xrightarrow{w} T_j$, we reach a contradiction (Lemma 6.11). □

Lemma 6.13. For any history H that is valid under ROLA, $DSG(H)$ does not contain a sequence $T_h \xrightarrow{w} T_i \xrightarrow{anti} T_j$.

Proof Sketch. There is a transaction $T_{i'}$ whose version written $x_{i'}$ is read by T_i, namely $T_{i'} \xrightarrow{r} T_i$. T_j writes $x_{i'}$'s next version x_j, namely $T_{i'} \xrightarrow{w} T_j$. According to Lemma 6.11, $T_h \neq T_{i'}$. Now we have $x_h <_{next} x_i$ and $x_{i'} <_{next} x_j$. The following continues the proof by comparing the version order of x_h and $x_{i'}$:

- Case 1. If $x_h < x_{i'}$, we have $x_i < x_{i'}$ since x_h's next version is x_i. According to Lemma 6.10, we have $x_{i'} <_{next} x_i$. Therefore we reach a contradiction.
- Case 2. If $x_{i'} < x_h$, because of $x_{i'} <_{next} x_j$, we have $x_{i'} <_{next} x_j < x_h <_{next} x_i$. According to Lemma 6.10, we have $x_{i'} <_{next} x_i$. Therefore we reach a contradiction.

□

Theorem 6.14. ROLA prevents LU.

Proof Sketch. We prove this theorem by contradiction. Suppose there is a history H that is valid under ROLA, but $DSG(H)$ contains a directed cycle having one or more anti-dependency edges and all edges are labeled by the same data item x.

- Case 1. Suppose $DSG(H)$ contain a directed cycle having only one anti-dependency edge $T_a \xrightarrow{l1} ... \xrightarrow{l2} T_h \xrightarrow{l_{hi}} T_i \xrightarrow{l_{ij}} T_j \xrightarrow{l3} ... \xrightarrow{l4} T_a$. Let l_{ij} be the anti-dependency edge. Thus, there is a transaction $T_{i'}$ whose version written $x_{i'}$ is read by T_i, namely $T_{i'} \xrightarrow{r} T_i$; the transaction T_j writes $x_{i'}$'s next version x_j, namely $T_{i'} \xrightarrow{w} T_j$. According to Lemmas 6.13 and 6.9, we have $T_{i'} = T_h$. Thus, we have another directed cycle $T_a \xrightarrow{l1} ... \xrightarrow{l2} T_h \xrightarrow{w} T_j \xrightarrow{l3} ... \xrightarrow{l4} T_a$ consisting entirely of read-dependency and write-dependency edges ($G1c$), which contradicts Theorem 6.7.
- Case 2. Suppose $DSG(H)$ contain a directed cycle having at least two anti-dependency edges. According to Lemma 6.12, these anti-dependency edges are not consecutive. For each anti-dependency we directly follow Case 1, and eventually we are able to construct a directed cycle consisting entirely of read-dependency and write-dependency edges ($G1c$), which contradicts Theorem 6.7.

□

6.3 A FORMAL EXECUTABLE SPECIFICATION OF ROLA

This section presents an executable formal specification of ROLA in the CAT framework (Chapter 3). Since ROLA extends RAMP-Fast, we only show the basic common building blocks and major extensions, and refer to Chapter 4 for RAMP-Fast's formal specification.

6.3.1 Data Types, Classes, and Messages

We model ROLA in an object-oriented style: the state comprises a number of *replica* (or *site*) objects (each modeling a replica of the data store), and a number of messages (traveling between the objects). A *transaction* is defined as an object residing inside the replica object that executes it.

Some Data Types. Like in RAMP (Section 4.3.1), a *version* is a timestamped version of a data item (or key) and is modeled as a four-tuple `version(`*key*`,`*value*`,` *timestamp*`,`*metadata*`)` consisting of the key, its value, and the version's timestamp and metadata. A timestamp is modeled as a pair `ts(`*Oid*`,`*sqn*`)` consisting of a replica's identifier *Oid* and a local sequence number *sqn* that together uniquely identify a write transaction. Metadata are modeled as a set of keys, denoting, for each key, the other keys that are written in the same transaction.

Unlike RAMP (Section 4.3.1), a list, instead of a set, of versions of sort `Versions` is built from singleton lists (identified with versions of sort `Version` by means of a `subsort` declaration) with an associative concatenation operator __ with identity `nil`:

```
sorts Version Versions .
subsort Version < Versions .

op nil : -> Versions [ctor] .
op __ : Versions Versions -> Versions [ctor assoc id: nil] .
```

A list of read and write operations of sort `OperationList` is built from singleton lists (identified with operations of sort `Operation` by means of a `subsort` declaration) with an associative concatenation operator __ with identity element `nil`:

```
sorts Operation OperationList .
subsort Operation < OperationList .

op nil : -> OperationList [ctor] .
op __ : OperationList OperationList -> OperationList [ctor assoc id: nil] .
```

We also define a collection of votes of sort `Vote`, and the data type `TxnOidSet` for the situation when a replica is waiting for messages such as votes from other replicas w.r.t. some transaction, in the same way as in RAMP (Section 4.3.1).

Classes. ROLA transactions can be modeled as object instances of the subclass `ROLA-Txn` with four new attributes:

```
class ROLA-Txn | operations : OperationList,  localVars : LocalVars,
                 latest : KeyTimestamps,  txnSqn : Nat .
subclass ROLA-Txn < Txn .
```

The `operations` attribute denotes the transaction's operations. `localVars` maps the transaction's local variables to their current values. `latest` stores the local view as a mapping from keys to their respective latest committed timestamps. `txnSqn` stores the transaction's sequence number.

A ROLA replica (storing parts of the database) is modeled as an object instance of the following subclass `ROLA-Replica` of class `Replica` that adds ten new attributes:

```
class ROLA-Replica | datastore : Versions,         sqn : Nat,
                     gotTxns : ObjectList,         tsSqn : TimestampSqn,
                     latestCommit : KeyTimestamps, votes : Vote,
                     voteSites : TxnOidSet,       1stGetSites : TxnOidSet,
                     2ndGetSites : TxnOidSet,     commitSites : TxnOidSet .
subclass ROLA-Replica < Replica .
```

All attributes are defined in the same way as in the RAMP model (Section 4.3.1) except: (i) the `datastore` attribute represents the replica's local database as a *list*, instead of a *set*, of versions for each key stored at the replica; and (ii) the *new* attribute `tsSqn` maps each version's timestamp to a local sequence number `sqn`.

Like in the RAMP model, the state also contains a "table" object of class `Table` mapping each data item to the replica storing the item. Each mapping is a pair `sites(`***`key`***`,`***`part`***`)`.

Messages travel between replicas, and have the form:

```
msg msgContent from sender to receiver
```

where the message content ***`msgContent`*** is defined in our ROLA models as follows:

- `prepare(`*txn*`,`*version*`)` sends a version from a write-only transaction to its Replica;

- `prepare(`*txn*`,`*version*`,` *ts*`)` does the same thing for other transactions, with *ts* the timestamp of the version it has read;
- `prepare-reply(`*txn*`,`*vote*`)` is the reply to the corresponding prepare message, where *vote* tells whether this Replica can commit the transaction;
- `commit(`*txn*`,`*ts*`)` marks the versions with timestamp *ts* as committed;
- `get(`*txn*`,`*key*`,`*ts*`)` asks for the highest-timestamped committed version or a missing version for *key* by timestamp *ts*;
- `response1(`*txn*`,`*version*`)` responds to first-round `get` request;
- `response2(`*txn*`,`*version*`)` responds to second-round `get` requests.

Initial State. The following shows an automatically generated initial state (with some parts replaced by '...') with two replicas, `r1` and `r2`, that are coordinators for, respectively, transactions `t1`, `t2` and `t3`. `r1` stores the data items `x` and `z`, and `r2` stores `y`. Transaction `t1` is the read-only transaction `(xl :=read x) (yl :=read y)`, transaction `t2` is a write-only transaction `write(y,3) write(z,8)`, while transaction `t3` is a read-write transaction on data item `x`. The states also include a table which assigns to each data item the replica storing it. Initially, the value of each item is `[0]`; the version's timestamp is empty (`eptTS`), and metadata is an empty set:

```
eq init =
< tb : Table | table : [sites(x,r1) ;; sites(y,r2) ;; sites(z,r1)] >
< r1 : ROLA-Replica |
    gotTxns : < t1 : ROLA-Txn |
    operations : ((xl :=read x) (yl :=read y)), readSet : empty, writeSet : empty,
        latest : empty, localVars : (xl |-> [0], yl |-> [0]), txnSqn : 0 >,
    datastore : (version(x, [0],eptTS,empty) version(z, [0],eptTS,empty)),
    sqn : 1, executing : none, committed : none, aborted : none, tsSqn : empty,
    latestCommit : empty, votes : noVote, voteSites : empty, 1stGetSites : empty,
    2ndGetSites : empty, commitSites : empty >
< r2 : ROLA-Replica | gotTxns :
    < t2 : ROLA-Txn | operations : (write(y,3) write(z,8)), ... >
    < t3 : ROLA-Txn | operations : ((xl := read x) write(x, xl plus 1)), ... >
    datastore : version(y, [0],eptTS,empty), ... >
```

6.3.2 Formalizing ROLA's Behaviors

This section formalizes the dynamic behaviors of ROLA using rewrite rules, referring to the corresponding lines in Algorithm 6.1 in Section 6.1.[5] Since ROLA reads are the same with RAMP-Fast reads (Chapter 4.3.2), here we only illustrate ROLA writes. The entire specification is given at `https://github.com/siliunobi/cat`.

Starting a Transaction (Lines 17 – 22). A replica starts to execute a transaction by moving the first transaction (`TID`) in `gotTxns` to `executing`. If the new transaction is a write-only transaction (`write-only(OPS)`), the replica: (i) uses the function `genPuts` (defined in Section 4.3.2) to generate all `prepare` messages; (ii) uses a function `prepareSites` to remember the sites `RIDS` from which it awaits votes for transaction `TID` in the `voteSites` attribute; and (iii) increments its local sequence number by one:

```
crl [start-wo-txn] :
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : ROLA-Replica |
              gotTxns : (< TID : ROLA-Txn | operations : OPS, localVars : VARS,
                                                      txnSqn : N > ;; TXNS),
              executing : TRANSES,   sqn : SQN,   voteSites : VSTS >
   =>
    < TABLE : Table | table : REPLICA-TABLE >
    < RID : ROLA-Replica |
              gotTxns : TXNS,
              executing : < TID : ROLA-Txn | operations : OPS, localVars : VARS,
                                                       txnSqn : SQN' > TRANSES,
              sqn : SQN',   voteSites : (VSTS ; addrs(TID,RIDS)) >
    genPuts(OPS,RID,TID,SQN',VARS,REPLICA-TABLE)
    if SQN' := SQN + 1 /\  write-only(OPS) /\
       RIDS := prepareSites(OPS,RID,REPLICA-TABLE) .
```

Otherwise, if the first transaction in `gotTxns` is a read-only or read-write transaction, the replica updates `1stGetSites` instead to keep track of the replicas from which it receives the versions from the first-round gets. The function `genGets` generates all `get` messages for the keys concerned by `TID` (see the executable specification available online for the definition of this, and other, functions). The expression `1stSites` gives the corresponding replicas for those keys:

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

```
crl [start-ro-or-rw-txn] :
    < TABLE : Table | table : Replica-TABLE >
    < RID : ROLA-Replica | gotTxns :
                (< TID : ROLA-Txn | operations : OPS, latest : empty > ;; TXNS),
            executing : TRANSES, 1stGetSites : 1STGETS >
  =>
    < TABLE : Table | table : Replica-TABLE >
    < RID : ROLA-Replica | gotTxns : TXNS,
            executing : < TID : ROLA-Txn | operations : OPS,
                                           latest : vl(OPS) > TRANSES,
            1stGetSites : (1STGETS ; addrs(TID,RIDS)) >
    genGets(OPS,RID,TID,Replica-TABLE)
    if (not write-only(OPS)) /\
       RIDS := 1stSites(OPS,RID,Replica-TABLE) .
```

Receiving `prepare` Messages (Lines 5–10). When a replica receives a `prepare` message for a read-write transaction, the replica first determines whether the timestamp of the last version (`VERSION`) in its local version list `VS` matches the incoming timestamp `TS'` (which is the timestamp of the version read by the transaction). If so, the incoming version is added to the local store, the map `tsSqn` is updated, and a positive reply (`true`) to the prepare message is sent ("**return** *ack*" in our pseudo-code); otherwise, a negative reply (`false`, or "**return** *latest*" in the pseudo-code) is sent:

```
crl [receive-prepare-rw] :
    msg prepare(TID, version(K, V, TS, MD), TS') from RID' to RID
    < RID : ROLA-Replica | datastore : VS,  sqn: SQN,  tsSqn : TSSQN >
   =>
    if VERSION == eptVersion or tstamp(VERSION) == TS'
    then < RID : ROLA-Replica | datastore : (VS version(K,V,TS,MD)), sqn : SQN',
                         tsSqn : insert(TS,SQN',TSSQN) >
         msg prepare-reply(TID, true) from RID to RID'
    else < RID : ROLA-Replica | datastore : VS,  sqn : SQN,  tsSqn : TSSQN >
         msg prepare-reply(TID, false) from RID to RID'
    if SQN' := SQN + 1 /\ VERSION := latestPrepared(K,VS) .
```

If instead the received `prepare` message was for a write-only transaction, the replica simply adds the received version to its local datastore, and maps the associated timestamp to the incremented sequence number (`insert(TS,SQN',TSSQN)`):

```
 crl [receive-prepare-wo] :
     msg prepare(TID,version(K,V,TS,MD)) from RID' to RID
     < RID : ROLA-Replica | datastore : VS, sqn : SQN, tsSqn : TSSQN >
   =>
     < RID : ROLA-Replica | datastore : (VS version(K,V,TS,MD)),
                            sqn : SQN', tsSqn : insert(TS,SQN',TSSQN) >
     msg prepare-reply(TID,true) from RID to RID'
     if SQN' := SQN + 1 .
```

Receiving Negative Replies (Lines 23–24). When a replica receives a `prepare-reply` message with `false` vote, it aborts the transaction by moving it to `aborted`, and removes `RID'` from the "vote waiting list" for this transaction. If the transaction has been aborted, the incoming `prepare-reply` message is simply consumed by the replica:

```
rl [receive-prepare-reply-false-executing] :
   msg prepare-reply(TID,false) from RID' to RID
   < RID : ROLA-Replica | executing : < TID : ROLA-Txn | > TRANSES,
                          aborted : TRANSES',
                          voteSites : VSTS addrs(TID,(RID',RIDS)) >
 =>
   < RID : ROLA-Replica | executing : TRANSES,
                          aborted : TRANSES' < TID : ROLA-Txn | >,
                          voteSites : VSTS addrs(TID,RIDS) > .

rl [receive-prepare-reply-aborted] :
   msg prepare-reply(TID,FLAG) from RID' to RID
   < RID : ROLA-Replica | aborted : TRANSES < TID : ROLA-Txn | >,
                          voteSites : VSTS  >
 =>
   < RID : ROLA-Replica | aborted : TRANSES < TID : ROLA-Txn | >,
                          voteSites : remove(TID,RID',VSTS) > .
```

Receiving Acks (Lines 26–28). Upon receiving a "true" vote, the replica first checks whether all votes have now been collected. The expression `VSTS'[TID]` extracts for `TID` the remaining replicas from which it is awaiting votes. If all received votes are "yes," the replica starts to commit `TID` at the associated replicas by invoking `genCommits` to generate all commit messages with the commit timestamp including the current sequence number `SQN`. The replica also adds to `commitSites` the replicas from which it is awaiting `committed` messages to commit the transaction:

```
crl [receive-prepare-reply-true-executing] :
    msg prepare-reply(TID,true) from RID’ to RID
    < TABLE : Table | table : Replica-TABLE >
    < RID : ROLA-Replica |
            executing : < TID : ROLA-Txn | operations : OPS > TRANSES,
            voteSites : VSTS, sqn : SQN, commitSites : CMTS >
  =>
    < TABLE : Table | table : Replica-TABLE >
    if VSTS’[TID] == empty  --- all votes received and all yes!
    then < RID : ROLA-Replica |
            executing : < TID : ROLA-Txn | operations : OPS > TRANSES,
            voteSites : VSTS’, sqn : SQN,
            commitSites : (CMTS ; addrs(TID,RIDS)) >
         genCommits(TID,SQN,RIDS,RID)
    else < RID : ROLA-Replica |
            executing : < TID : ROLA-Txn | operations : OPS > TRANSES,
            voteSites : VSTS’, sqn : SQN, commitSites : CMTS >
    fi
    if VSTS’ := remove(TID,RID’,VSTS) /\
       RIDS := commitSites(OPS,RID,Replica-TABLE) .
```

Receiving `commit` Messages (Lines 13–16). Upon receiving a `commit` message, the Replica invokes the function `cmt` to commit the transaction. `cmt` looks up `tsSqn` for the commit timestamp `TS` and the latest committed version’s timestamp in `LC`, and updates the latest committed version if `TS`’s local sequence number is higher. A `committed` message is then sent back to confirm the commit:

```
rl [receive-commit] :
   msg commit(TID, TS) from RID’ to RID
   < RID : ROLA-Replica | tsSqn : TSSQN, datastore : VS, latestCommit : LC >
  =>
   < RID : ROLA-Replica | tsSqn : TSSQN, datastore : VS,
                          latestCommit : cmt(LC, VS, TSSQN, TS) >
   msg committed(TID) from RID to RID’ .
```

Receiving `committed` Messages. (For replicas to commit transactions locally). Upon receiving a `committed` message, the replica first checks if all committed messages have now been collected. The expression `CMTS’[TID]` projects for `TID` the remaining replicas from which it is awaiting `committed` messages. If the extraction is `empty`, the replica commits the transaction:

```
crl [receive-committed] :
    msg committed(TID) from RID' to RID
    < RID : ROLA-Replica | executing : < TID : ROLA-Txn | > TRANSES,
                           committed : TRANSES', commitSites : CMTS >
  =>
    if CMTS'[TID] == empty  --- all "committed" received
    then < RID : ROLA-Replica | executing : TRANSES,
                                committed : TRANSES' < TID : ROLA-Txn | >,
                                commitSites : CMTS' >
    else < RID : ROLA-Replica | executing : < TID : ROLA-Txn | > TRANSES,
                                committed : TRANSES', commitSites : CMTS' >
    fi
    if CMTS' := remove(TID,RID',CMTS) .
```

6.4 MODEL CHECKING CONSISTENCY PROPERTIES OF ROLA

Section 6.2 gave a proof sketch that ROLA guarantees RA and PLU, i.e., update atomicity (UA). However, it is well known that hand proofs may be erroneous or may make crucial assumptions that may not have been made explicit in a formal model. Indeed, we have experienced that Maude model checking can uncover nontrivial errors as well as both missing and unclear assumption in a supposedly verified distributed transaction system that is less complex than ROLA [74].

We therefore have used the CAT tool (Chapter 3) to model check the correctness of ROLA's formal model. As shown in Table 3.1, all model checking results are as expected. In particular, the ROLA model satisfies consistency properties weaker than or equal to UA, and violates SI and any stronger consistency properties.

Below we show an example model checking experiment, where we provide the tool CAT with one read-only transaction (of two operations), two read-write transactions (four operations per each), two replicas, and two keys. Under the hood, CAT executes the following command to search, from all generated initial states, for one reachable final state where UA does not hold:

```
search [1] init(2,0,1,2,0,4,0,2,2,1) =>! C:Configuration
    < M:Oid : Monitor | log: LOG:Log  clock: N:Nat > such that not ua(LOG:Log) .
```

CAT outputs “`No solution`,” meaning that all runs from all the given initial states satisfy UA, for our ROLA model.

6.5 STATISTICAL PERFORMANCE COMPARISON

The weakest consistency models in [29, 9] guaranteeing RA and PLU are PSI and NMSI, and the main systems providing PSI and NMSI are, respectively, Walter [85] and Jessy [9]. To be an attractive design option, ROLA should outperform both Walter and Jessy. To quickly check whether ROLA indeed does so, we have also modeled Walter and Jessy—without their data replication features—in Maude (see `https://github.com/siliunobi/cat`), and have used statistical model checking with PVeStA to compare the performance of ROLA, Walter, and Jessy in terms of throughput and average transaction latency.

6.5.1 A Probabilistic Model of ROLA

Following the approach in Section 4.5, to introduce both time and probabilities for *performance* estimation, each message gets assigned a *message delay* that is sampled probabilistically from a dense time interval according to a certain probability distribution.

Example 6.2. In the transformed rule below (the original rule is in black), the incoming message `prepare` is equipped with the current global time `GT`, and the outgoing message `prepare-reply` is equipped with a `delay`:

```
crl [receive-prepare-rw-prob] :
    { GT, msg prepare(TID,version(K,V,TS,MD),TS') from RID' to RID }
    < RID : ROLA-Replica | datastore : VS,  sqn: SQN,  tsSqn : TSSQN >
   =>
    if VERSION == eptVersion or tstamp(VERSION) == TS'
    then < RID : ROLA-Replica | datastore : (VS version(K,V,TS,MD)), sqn : SQN',
                              tsSqn : insert(TS,SQN',TSSQN) >
         [ delay, msg prepare-reply(TID,true) from RID to RID' ]
    else < RID : ROLA-Replica | datastore : VS,  sqn : SQN,  tsSqn : TSSQN >
         [ delay, msg prepare-reply(TID,false) from RID to RID' ]
    if SQN' := SQN + 1 /\ VERSION := latestPrepared(K,VS) .
```

where the `prepare` message is consumed by `RID` at the global time `GT`, and the `prepare-reply` message will be consumed by `RID'` after `delay` time units. `delay` of sort `Float` is a parameter instantiated with a certain probability distribution (see Section 4.5).

6.5.2 Recording Executions

Following the monitoring mechanism in Section 4.6, we also equip the transformed probabilistic model with an execution log recording the history of relevant events during a system execution. The log is updated each time an interesting event happens (i.e., the start, commit, and abort of a transaction).[6] We (manually) identify those events in the model, and transform the corresponding rules by adding and updating the monitor object.

Example 6.3. *Abort a Transaction.* When the replica receives a `false` vote, it aborts the transaction. The monitor records the abort/finish time `GT` for that transaction (and the "committed" flag remains `false`):

```
rl [receive-prepare-reply-false-executing-monitor] :
   < O@M : Monitor | log : LOG@M, (TID |->
        < RID, T@M, VTS@M, FLAG@M, READS@M, WRITES@M)) >
   { GT, msg prepare-reply(TID,false) from RID' to RID }
   < RID : ROLA-Replica |
            executing : < TID : ROLA-Txn | writeSet : WS, readSet : RS > TRANSES,
            aborted : TRANSES',
            voteSites : VSTS addrs(TID,(RID' ,RIDS)) >
 =>
   < O@M : Monitor | log : LOG@M, (TID |->
         < RID, T@M, insert(RID,GT,VTS@M), false, RS, WS >)
   < RID : ROLA-Replica |
            executing : TRANSES,
            aborted : TRANSES' < TID : ROLA-Txn | writeSet : WS, readSet : RS >,
            voteSites : VSTS addrs(TID,RIDS) > .
```

6.5.3 Performance Comparison

Experimental Setup. We performed our PVeStA experiments with many different probabilistically generated configurations.[7] Each configuration had, 200 transactions, 2 or 4 operations per read-only or read-write transaction, up to 150 data items and up to 50 partitions, with lognormal message delay distributions (obtained by characterizing real-world

[6]We do not consider replication in our ROLA model, and therefore interesting events exclude transaction commits on remote sites.

[7]The number of probabilistically generated configuration for analyzing a given performance property depends on the chosen level of statistical confidence (in our case 95%), and is determined by the PVeStA tool, which stops performing more simulations when such a confidence level is reached. Computing the probabilities took a day (in the worst case) on 20 servers, each with a 64-bit Intel Quad Core Xeon E5530 CPU with 12 GB of memory.

data centers [17]), and with uniform and Zipfian data item access distributions (used by Yahoo! Cloud Serving Benchmark (YCSB) [36], the open standard for comparative performance evaluation of data stores). Regarding lognormal's parameters, local delays use $\mu = 0$ and $\sigma = 1$, while remote delays use $\mu = 3$ and $\sigma = 2$.

The plots in Fig. 6.3 show the *throughput* as a function of the percentage of read-only transactions, and number of keys (data items), sometimes with both uniform and Zipfian distributions. The plots show that ROLA outperforms Jessy, which itself outperforms Walter, for all parameter combinations. As the number of keys increases, the throughput of all three protocols increases. In particular, with 100 or more keys, ROLA with uniform distribution has significantly higher throughput than Walter and Jessy. We also learn from the plots that more reads give higher throughput, since read-only transactions in all three protocols can commit locally without certification. We only plot the results under uniform key access distribution for the top plot; these results are consistent with the results using Zipfian distributions.

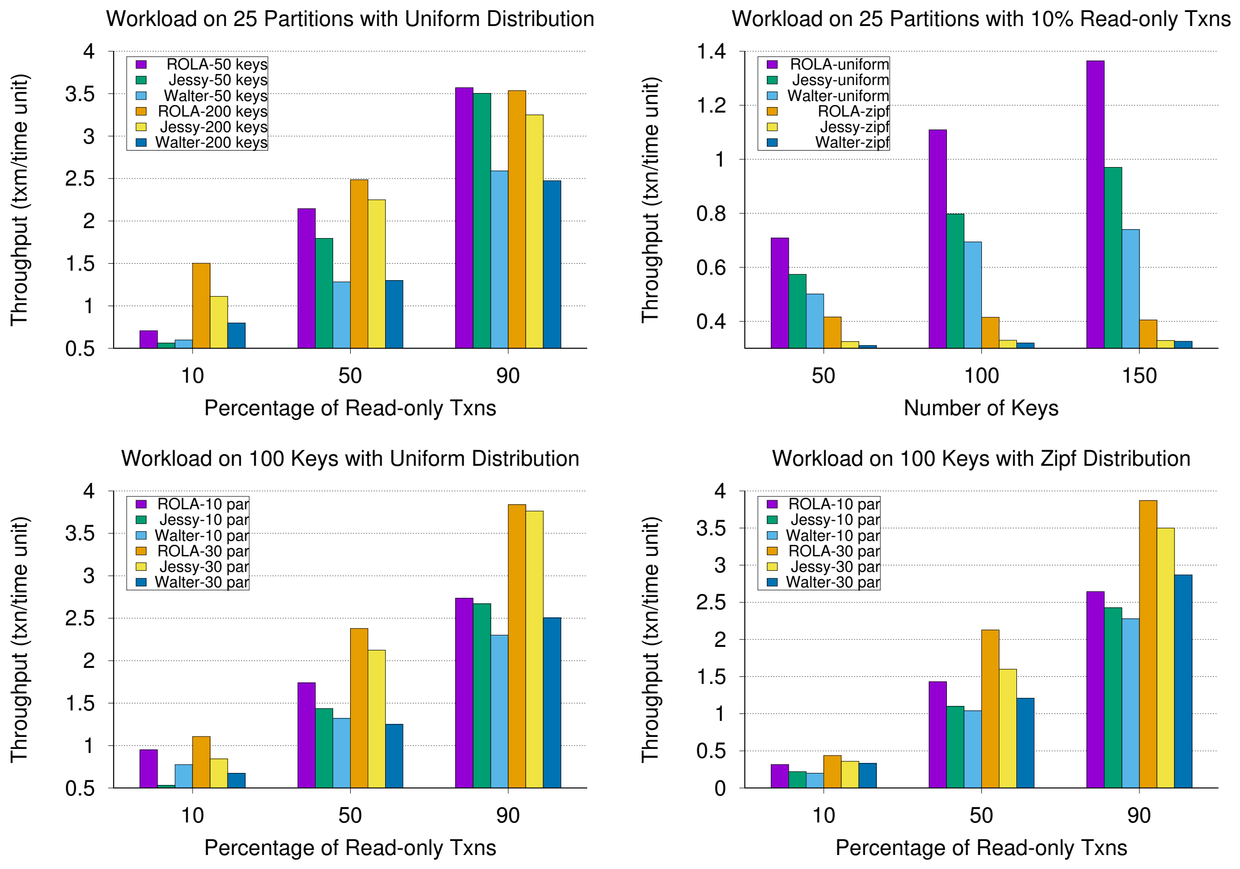


Figure 6.3: Throughput comparison under different workload conditions.

The plots in Fig. 6.4 show the *average transaction latency* as a function of the same

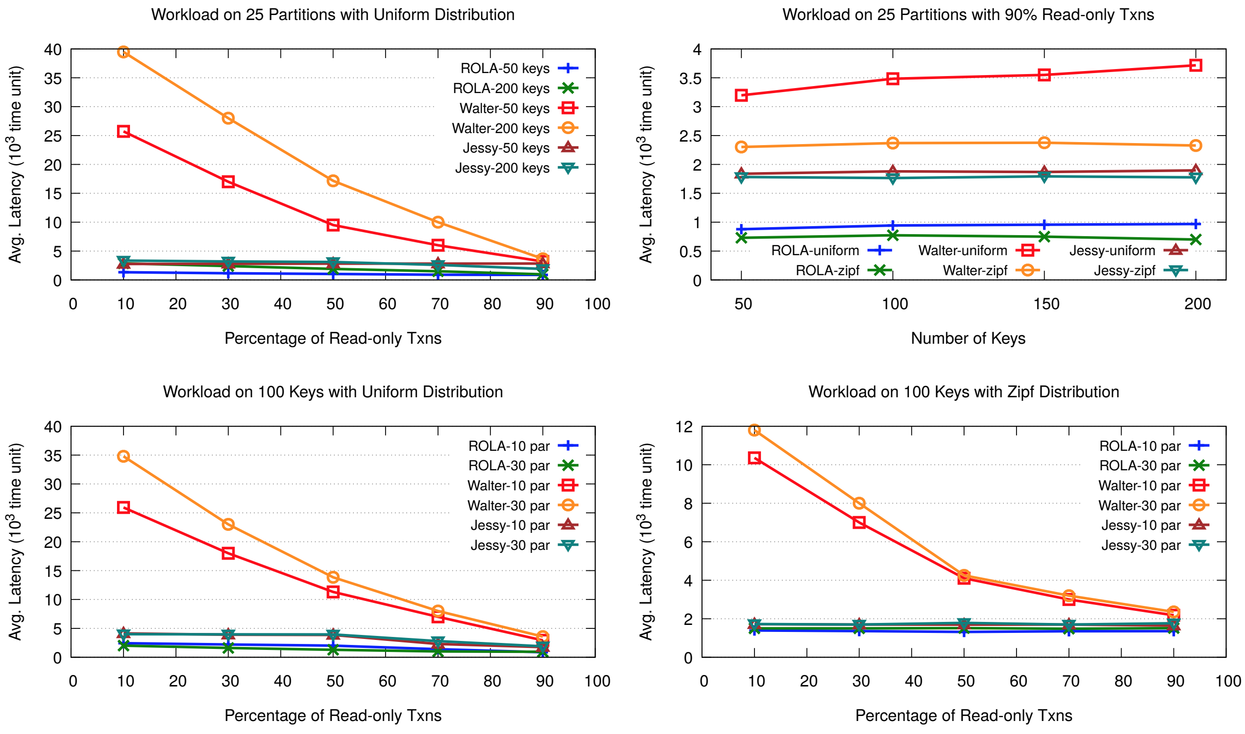
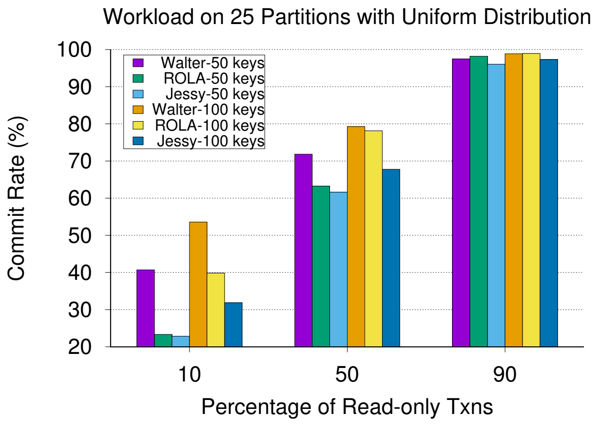
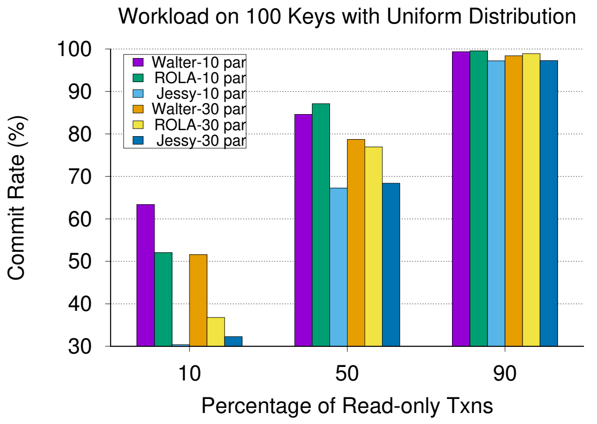


Figure 6.4: Average latency comparison across varying workload conditions.

parameters as the plots for throughput. Again, we see that ROLA outperforms Jessy and Walter in all settings. In particular, the difference between ROLA/Jessy and Walter is quite large for write-heavy workloads; the reason is that Walter incurs a high overhead for ensuring causal consistency, which requires background propagation to advance the vector timestamp. The latency tends to converge under read-heavy workload (because reads in all three protocols can commit locally without certification), but ROLA still has noticeably lower latency than the other two protocols.

The plots in Fig. 6.5 present the *transaction commit rate* as a function of the same parameters as for the other two properties. The plots show that Walter has overall higher commit rate than ROLA, which itself has higher commit rate than Jessy, because Walter trades latency for more committed transactions. As the number of keys increases, the commit rate of all three protocols increases. In particular, with 100 or more keys, ROLA has significantly higher commit rate than Jessy. We also learn from the plots that more reads give higher commit rate, as read-only transactions in all three protocols can commit directly. We only plot the results under uniform key access distribution for some parameter combinations, which are consistent with the results using Zipfian distributions.





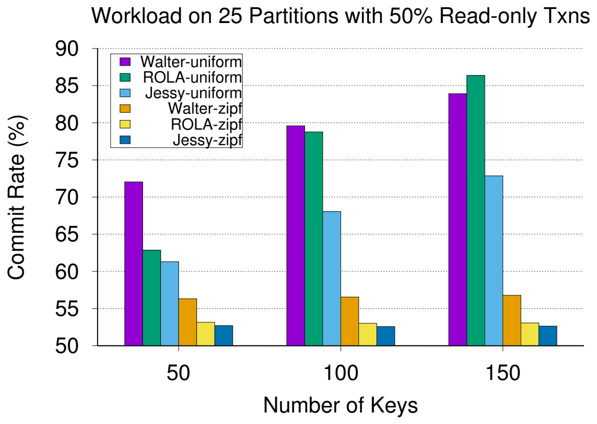


Figure 6.5: Transaction commit rate comparison across varying workload conditions.

6.6 CONCLUDING REMARKS

We have presented the formal design and analysis of ROLA, a distributed transaction protocol that supports a new consistency model not present in the survey by Cerone et al. [29]. Using formal modeling and both standard and statistical model checking analyses we have: (i) validated ROLA's RA and PLU consistency requirements; and (ii) analyzed its performance requirements, showing that ROLA outperforms Walter and Jessy in all performance measures.

This work has shown, to the best of our knowledge for the first time, that the design and validation of the consistency and performance properties of a *new* distributed transaction protocol can be achieved relatively quickly *before* its implementation by the use of formal methods. It also shows that, if Maude specifications of other system designs are available, it is possible to compare the advantages and disadvantages of the new design w.r.t. those alternative designs also before their implementations. This of course does not exclude the additional information and improvements that will be gained by *implementing* ROLA; but it substantially reduces the effort required in reaching a mature design.

CHAPTER 7: AUTOMATIC TRANSFORMATION OF MAUDE DESIGNS INTO CORRECT-BY-CONSTRUCTION DISTRIBUTED IMPLEMENTATIONS

Developing a highly reliable distributed system meeting desired performance requirements is at present a hard and very labor-intensive task. As shown in the previous chapters in this dissertation, formal specification of a system design and formal verification and analysis can yield formally verified designs as well as reliable performance predictions about their behaviors. But there is still a *formality gap* between verified designs and distributed implementations. Suppose that a given property has been verified for a given system design. Does the property still hold true for its distributed implementation?

In this chapter we bridge this formality gap by presenting a correct-by-construction automatic transformation (called the D transformation) mapping a verified formal specification of a system design in Maude to a distributed implementation with the same safety and liveness properties as the formal design. Two case studies, namely the NO_WAIT transaction protocol [47] and the ROLA transaction system (Chapter 6), applying this transformation to state-of-the-art distributed transaction systems show that high-quality implementations with acceptable performance and meeting performance predictions can be obtained in this way. To the best of our knowledge, this is the first time that formal models of distributed systems analyzed within the *same formal framework* for *both* logical and performance properties are automatically transformed into logically correct-by-construction implementations in the *same programming language* for which similar performance trends can be shown.

This chapter is organized as follows: Section 7.1 describes the formal definition of the D transformation. Section 7.2 shows the proof that the obtained distributed implementation of a Maude specification is correct by construction. Section 7.3 presents a Maude prototype automation of the D transformation, and two case studies using state-of-the-art distributed transaction systems evaluating the implementations obtained by the D transformation. Finally, Section 7.4 discusses related work and Section 7.5 gives some concluding remarks.

7.1 THE D TRANSFORMATION

We define the transformation $M \mapsto D(M)$, mapping a Maude formal design M of a distributed system to a distributed Maude program $D(M)$ deployed on different machines. We allow multiple concurrent Maude sessions to run on the same machine.

The transformation D takes as input:

- an object-oriented Maude module M defining an actor system as explained below;

- an initial state `init` of sort `Configuration`, which is a set of objects

  `<` o_1 `:` C_1 `|` $atts_1$ `>` `...` `<` o_n `:` C_n `|` $atts_n$ `>`

  in $\mathcal{T}_{M,\texttt{Configuration}}$, with distinct object names o_i;

- a *distribution information* function

$$di : \{o_1, \dots, o_n\} \to \texttt{String} \times \mathbb{N}$$

  assigning to each "top-level" object[1] o_j in `init` a pair (ip, i), where ip is the IP address (given as a string) of the machine in which the object should reside, and i denotes the ith Maude session on that machine.

The transformation D then gives us:

- A Maude program $M_{D_{di}}$ that runs on each distributed Maude session; and
- an initial state $\texttt{init}_{D_{di}}(ip, i)$ for each Maude session (ip, i).

The transformation D is then a function

$$\begin{aligned}&\lambda M \in \mathit{OModule} \,.\, \lambda\texttt{init} \in \mathcal{T}_{M,\texttt{Configuration}} \,.\\ &\quad \lambda di \in [\mathit{oids}(\texttt{init}) \to \texttt{String} \times \mathbb{N})] \,.\, D(M, \texttt{init}, di) \in \mathit{OModule}.\end{aligned}$$

Notation. We write $M_{D_{di}}$ for $D(M, \texttt{init}, di)$.

The object-oriented module M should model an "actor" system, so that its rewrite rules must have the form

$(\texttt{to}\ o\ \texttt{from}\ o'\ \texttt{:}\ mc)\ \texttt{<}\ o : C\ \texttt{|}\ \texttt{...}\ \texttt{>}\ \texttt{=>}\ \texttt{<}\ o : C\ \texttt{|}\ \texttt{...}\ \texttt{>}\ msgs\ [\texttt{if}\ \texttt{...}]$ (†)

or

$\texttt{<}\ o : C\ \texttt{|}\ \texttt{...}\ \texttt{>}\ \texttt{=>}\ \texttt{<}\ o : C\ \texttt{|}\ \texttt{...}\ \texttt{>}\ msgs\ [\texttt{if}\ \texttt{...}]$ (‡)

where *msgs* is a term of sort `Configuration` which, applying the equations in the module, reduces to a multiset of *messages*

$(\texttt{to}\ o_1\ \texttt{from}\ o\theta\ \texttt{:}\ mc_1)\ \ \texttt{...}\ \ (\texttt{to}\ o_k\ \texttt{from}\ o\theta\ \texttt{:}\ mc_k)$

[1]Such "top-level" objects may be hierarchical; i.e., they may have attributes whose values contain other objects or entire configurations. Such "inner" objects often represent structured data rather than computational actors.

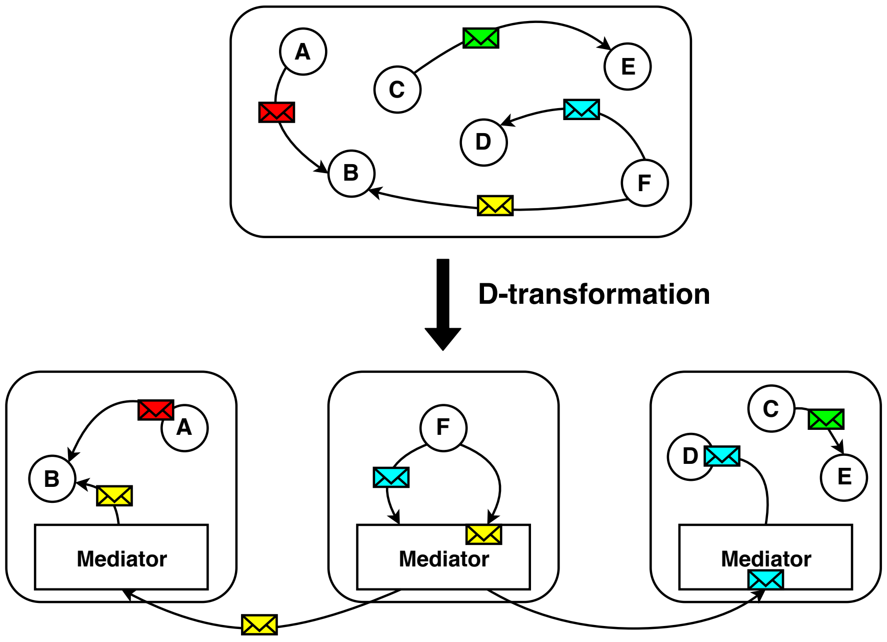


Figure 7.1: Visualization of the D Transformation

for $k \geq 0$, where θ is the substitution used when applying the rule. In such a message, mc_i is the message content (or payload) of the message being sent to the object named o_i from the object named $o\theta$. Although no "top-level" objects are created or deleted by the (†), (‡) rules, object-creating rules can also be added.[2]

7.1.1 The $M \mapsto M_{D_{di}}$ Transformation

The main idea for defining the distributed Maude program $M_{D_{di}}$ is to add middleware for communication between Maude sessions and with external objects. This is done by adding to *each* Maude session a *communication mediator* object that takes care of communication with objects that are not local, as illustrated in Fig. 7.1.

This mediator object opens and maintains sockets for communication between pairs of objects; there is in general one socket for each pair of objects that communicate remotely (across machine/session boundaries). Objects in the same Maude session communicate with each other without going through the mediator object.

The only modification of the rewrite rules in M is that a message addressed to a remote object is "redirected" to the local mediator, which:

[2]Our framework also allows us to have rules that create new "top-level" objects in their right-hand sides. A new "child" object will run in the same Maude session as its parent. Furthermore, the name of the child object must be chosen such that the `di` function can determine its Maude session. This can, e.g., be achieved by letting the child's identifier be a string of which the parent's is a prefix.

- establishes the required socket between the pair of objects if not already established;
- transforms the original message into a string with an "end-of-message" marker; and
- sends the resulting string through the appropriate socket.

For receiving, the mediator object receives external messages through sockets associated to "its" objects. Since TCP sockets do not preserve message boundaries, the mediator has to buffer the messages received in each socket. When the buffered string contains the "end-of-message" string, the mediator extracts the string representing the message, transforms it to a message, and leaves the message (having a local addressee) in the local configuration.

The distributed program $M_{D_{di}}$ consists of:

- A constant `di` of sort `Map{Oid,Pair{String,Nat}}` which specifies *di* in Maude as a map from `Oid` to `Pair{String,Nat}` using an equation `eq di = ...`.
- The module *filter*(M), which transforms M with only a minor change in its rules as described below.
- Declarations and rewrite rules defining the mediator objects and their behaviors (which import the `SOCKET` module).

Example 7.1. If the objects in state `init` in Example 2.1 (Chapter 2.1) are executed on different machines, say, ip_1, ip_2, ip_3, and ip_4, respectively, then the map `di` can be given in Maude as follows:

```
eq di = c1  |-> < "ip_1" ; 1 >,    c2  |-> < "ip_2" ; 1 >,
        db1 |-> < "ip_3" ; 1 >,    db2 |-> < "ip_4" ; 1 > .
```

The module *filter*(M). The only change made by *filter*(M) to the rewrite rules in M is that any message `(to` o' `from` o `:` mc`)` generated by a rule in M is replaced by a message

```
(to di(o') transfer mc from o to o')
```

if o' and o reside in different Maude sessions. Formally, this is done by adding a subsort declaration

```
subsort Pair{String,Nat} < Oid .
```

stating that a `<` *ip* ; *session* `>` pair is an object identifier (for the mediator objects), adding a message constructor

```
op to_transfer_from_to_ : Oid MsgContent Oid Oid -> Msg [ctor] .
```

and changing each rewrite rule in M of the form (†) to

(to o from o' : mc) < o : C | ... > => < o : C | ... > filter($msgs$) [if ...]

(and similar with rewrite rules of the form (‡)), where `filter` redirects the messages going to remote objects to the mediator and leaves the other (internal) messages unchanged[3]:

```
op filter : Configuration -> Configuration .
eq filter(none) = none .
eq filter((to O from O' : MC) CONF)
 = if di[O] =/= di[O'] then
      (to di[O'] transfer MC from O' to O)  filter(CONF)
   else (to O from O' : MC)  filter(CONF) fi .
```

Specifying the Mediator Each mediator is defined as an object instance of the class

```
class Med | sockets : Sockets,
            contacts : Contacts,
            bufferedMsgs : Configuration .
```

where:

- `sockets` values are terms [$socket_1, str_1$] ... [$socket_k, str_k$], denoting that the string str_j has been received through socket $socket_j$ (and then buffered) since the last time a message was extracted from this buffer;
- `contacts` is a set of triples < $localObjId$, $socket$, $remoteObjId$ >, denoting the socket used to communicate between two objects; and
- `bufferedMsgs` contains the outgoing messages when the appropriate sockets have not yet been established.

We refer to `https://github.com/anonymous-yokai/fse19` for a complete executable specification of the mediator object, where most of the rewrite rules deal with establishing Maude sockets along the lines explained in [32, Chapter 11]. In this chapter we just show the following three rewrite rules for the mediator.

[3]We do not show variable declarations in the rest of this chapter, but follow the convention that variables are written in (all) capital letters.

```
rl [sendRemote] :
   (to O transfer MC from O' to O'')
   < O : Med | contacts : CONTACTS ; < O', SOCKET, O'' > >
 =>
   < O : Med | >
   send(SOCKET,O',
        msg2string(to O'' from O' : MC) + "[msep]") .
```

In this rule, the mediator is tasked with transferring the message content `MC` from the local object `O'` to the remote object `O''`. The rule uses Maude's built-in message `send` to send the message through the socket `SOCKET`, which has already been established between `O'` and `O''`. Since sockets transport strings, the function `msg2string` is used to transform the message into a string; the end-of-message separator `"[msep]"` is then appended to the string.

The following rule shows the case when a configuration receives a message `received(S, SKT, DATA)`. This message denotes that the string `DATA` has been received through socket `SKT`. The mediator object just adds this string `DATA` to the string `STR` that it has already buffered for socket `SKT`:

```
rl [receive-data] :
    received(S, SKT, DATA)
    < O : Med | sockets : SKTS [SKT,STR] >
  =>
    < O : Med | sockets : SKTS [SKT, STR + DATA] >
    receive(SKT, S) .
```

Finally, when enough data has been received through a socket `SKT` so that a message `MSG` can be extracted from it, the message is extracted from the string, converted into a message which is added to the configuration to be consumed by a local object, and the remaining string (after the message and the end-of-message separator have been removed) is buffered with `SKT`:

```
crl [extractRemoteMsg] :
    < O : Med | sockets : SKTS [SKT,STR] >
  =>
    < O : Med | sockets : SKTS [SKT,
        substr(STR,find(STR, "[msep]", 0) + 6,length(STR))] >
    MSG
    if  MSG := string2msg(substr(STR,0,find(STR,"[msep]",0))) .
```

Communication between objects in the same Maude session takes place without going through sockets or mediators. $M_{D_{di}}$ also adds functions `string2msg` and `msg2string`, converting between messages and strings and satisfying `string2msg(msg2string(M)) = M`.

The Module $M_{D_{di}}$. To summarize, the distributed Maude program $M_{D_{di}}$ executed at each local host consists of the definition of *di* and the union of the module *filter*(M) and the mediator specification:

```
mod $M_{D_{di}}$ is
  including filter(M) + MEDIATOR .
  eq di = ... .
endm
```

7.1.2 Distributed Initial States

The initial state $\mathtt{init}_{D_{di}}(ip, n)$ at Maude session (ip, n) is a configuration containing:

- the objects in `init` mapped to (ip, n) by *di*;
- one mediator object

  ```
  < < ip ; n > : Med | sockets : empty, contacts : empty,
                       bufferedMsgs : none >
  ```

- one occurrence of the built-in "portal" object `<>` denoting that we rewrite with external objects, such as Maude's built-in socket manager (see Chapter 11 in [32] for more details about the portal object and external rewriting); and
- one message

  ```
  createServerTcpSocket(socketManager, o, port#, 5)
  ```

  for each top-level (non-mediator) object o in the configuration.

7.1.3 Adding Foreign Objects

A distributed Maude object-based system can be *easily extended to interact with objects foreign to it* with no changes to the existing rewrite rules: only the new messages and rules defining the interaction with new foreign objects —databases, web sites, display devices, and so on— need to be specified. This is easy to achieve thanks to the message-passing abstraction: an object of some class needs no information at all about the *internal representation* of objects of other classes which with it communicates. Only the *interfaces* specifying the messages are needed.

Within Maude itself, two kinds of objects are supported: (i) *standard Maude objects*, which are terms in an object-based rewrite theory, and (ii) *external Maude objects*. In this chapter it suffices to focus on *socket external Maude objects* already described in Section 2. If the only objects involved in our distributed Maude system are *standard* Maude objects, only socket external Maude objects, opened and closed by the *communication mediator objects* described in Section 7.1.1, are needed. But how can such a distributed Maude system communicate with *foreign objects*, that is, objects such as a display or a database completely outside Maude? The simple answer has been already hinted at above. Suppose `Cl1` is a class of Maude objects that needs to communicate with, say, database foreign objects. All we need are three things: (a) a *signature* of messages sent by objects in `Cl1` to such foreign objects and by foreign objects to objects in `Cl1`; (b) *rewrite rules* for the objects of class `Cl1` specifying how messages to foreign objects are generated and how objects of class `Cl1` react to messages sent to them by foreign objects; and (c) a *wrapper* encapsulating a foreign object that can transform the *string representation* of a message from a `Cl1` object into an internal command to the foreign object, and a reply from the foreign object into the *string representation* of a message to a `Cl1` object. Once items (a)—(c) are specified, socket-based communication can proceed as before: messages represented as strings will travel though the sockets communicating Maude standard objects with foreign objects and vice versa. As explained in Section 7.3, in this chapter we have used some of the steps (a)–(c) to allow communication of a YCSB [36] foreign object with standard Maude objects to carry out system evaluations in the two case studies we present. The same methodology can be used to allow communication of distributed Maude objects with any other foreign objects. Furthermore, the D transformation explained in Section 7.1.4 can be easily extended to initial configurations where some of their objects are foreign objects.

7.1.4 Deployment

We have built a simple Python-based prototype that automates the process of deploying and running the distributed Maude model on distributed machines. The tool takes as input the IP addresses of the distributed machines and the number of Maude sessions on each machine.

We have run distributed Maude deployments to perform large-scale experiments on distributed transaction systems. To experiment with realistic workloads, we have connected our distributed implementation to the well-known YCSB workload generator [36] as explained in Section 7.1.3. Our deployment tool also invokes the workload generator (e.g., YCSB) to initialize and to load data into the database, and then invokes the workload generator to generate transactions for the different Maude instances to execute.

To measure the performance of our distributed implementation, we have added a "log" attribute to each mediator object that records relevant data during the distributed execution. A Python script then inspects and aggregates these logs after execution to compute the overall performance metric of the system.

7.2 CORRECTNESS PRESERVATION

Our goal is to obtain a distributed implementation of a Maude specification that is correct by construction: If the original Maude model M, with intial state `init`, satisfies a CTL^* temporal logic property ϕ that does not contain the "next" operator $\bigcirc$, then ϕ should also hold in the distributed implementation $M_{D_{di}}$ when started with corresponding distributed initial state(s), and vice versa.

Since $M_{D_{di}}$ uses Maude external TCP/IP socket objects for communication between different Maude sessions, a full proof of correctness of the $M \mapsto M_{D_{di}}$ transformation would require modeling the TCP/IP protocol and its associated network failure model. This is possible, but is beyond the scope of this chapter. Instead, we adopt here the approach followed in other proofs of correctness of distributed systems obtained by transformation from formal specifications, e.g., [91, 82], where network communication is delegated to a trusted shim and is abstracted away in correctness proofs. In our case, the Maude TCP/IP socket objects invoked by the communication mediator objects play the role of such a trusted shim.

Therefore, we present below a proof of correctness, in the form of a stuttering bisimulation, which uses an intermediate formal model $D_0(M, \texttt{init}, di)$ which abstracts away the network communication details by providing a high-level abstraction of it.

7.2.1 The Model $D_0(M, \texttt{init}, di)$

The rewrite theory $D_0(M, \texttt{init}, di)$ is essentially as $M_{D_{di}}$, except that it abstracts away the establishment of the appropriate sockets, and models the effect of socket communication in rewriting logic at a higher level of abstraction. The model $D_0(M, \texttt{init}, di)$ therefore simplifies $M_{D_{di}}$ as follows.

Concerning the *mediator* class:

- Since we no longer have explicit sockets, the `contacts` attribute of `Med` is no longer needed.
- Since we assume that the sockets have been successfully established, the attribute `bufferedMsgs`, used to buffer outgoing messages that could not yet be transmitted since the appropriate socket was not established, is no longer needed.
- Since we abstract away the fact that TCP sockets do not preserve message boundaries, we do not need to buffer messages at the receiving end, and therefore the attribute `sockets` is no longer needed.

The mediator class therefore no longer needs any attributes, and is declared as follows in $D_0(M, \texttt{init}, di)$:

```
class Med .
```

The *rewrite rules* in $D_0(M, \texttt{init}, di)$ differ from the rewrite rules in $M_{D_{di}}$ as follows:

- Since we abstract from the establishment of sockets, the rewrite rules in $M_{D_{di}}$ dealing with this issue (not shown in this chapter) are omitted from $D_0(M, \texttt{init}, di)$.
- The rule `sendRemote` in $M_{D_{di}}$ is replaced by the rule

```
rl [sendRemote] :
   (to O transfer MC from O' to O'')
   < O : Med | >
 =>
   < O : Med | >
   transfer(di[O''], O, msg2string(to O'' from O' : MC)) .

op transfer : Oid Oid String -> Msg [ctor] .
```

  where a "transfer" message models socket communication.

- When a mediator receives such a transfer message (modeling socket communication), it transforms the received string into a message, which is then released into the configuration. The rewrite rules `receive-data` and `extractRemoteMsg` in $M_{D_{di}}$ are therefore replaced by the following rewrite rule in $D_0(M, \texttt{init}, di)$:

```
    crl [receiveRemoteMsg] :
        transfer(O, O', STRING)
        < O : Med | >
      =>
        < O : Med | >
        string2msg(STRING) .
```

Initial States The initial state in $D_0(M, \texttt{init}, di)$ corresponding to the state init in M is just init with an additional mediator object `< <` ip `;` n `> : Med | >` for each $(ip, n) \in image(di)$. We call this initial state $\texttt{init}_{D_0}$. (Compared to the distributed initial states in $M_{D_{di}}$, $\texttt{init}_{D_0}$ is the multiset union of all those distributed states, minus `<>`, where the Med objects no longer have attributes, and without the messages used to establish sockets.)

It is worth remarking that, although the distributed state is represented as a single flat configuration of objects in $\texttt{init}_{D_0}$, "direct" message communication between two objects assigned to different Maude sessions in di cannot take place due to the "filtering" of generated messages in $M_{D_{di}}$ (and hence also in $D_0(M, \texttt{init}, di)$).

7.2.2 $D_0(M, \texttt{init}, di)$ and M are Stuttering Bisimilar

We show that the Kripke structures $\mathcal{K}(D_0(M, \texttt{init}, di), \texttt{init}_{D_0})$ and $\mathcal{K}(M, \texttt{init})$ are stuttering bisimilar for the labeling functions L in $\mathcal{K}(M, \texttt{init})$ and $L \circ h$ in $\mathcal{K}(D_0(M, \texttt{init}, di), \texttt{init}_{D_0})$.

We define the map $h : Reach(\texttt{init}_{D_0}) \to Reach(\texttt{init})$ as follows:

```
eq h(none) = none .
eq h(< O : Med | >  CONF) = h(CONF) .
ceq h(< O : C | >  CONF) = < O : C | >  h(CONF) if C =/= Med .
eq h((to O transfer MC from O' to O'') CONF)
 = (to O'' from O' : MC) h(CONF) .
eq h((transfer(O,O',STRING)) CONF)
 = string2msg(STRING) h(CONF) .
eq h((to O from O' : MC) CONF) = (to O from O' : MC) h(CONF) .
```

That is, h maps a configuration in $D_0(M, \texttt{init}, di)$ to a similar configuration in M with the following modifications: (i) the mediator objects are forgotten, and (ii) the three intermediate messages involved in transferring a message content mc from o to a remote o' are all mapped to the message (`to` o' `from` o `:` mc).

Theorem 7.1. h is a stuttering bisimulation map

$$h : \mathcal{K}(D_0(M, \texttt{init}, di), \texttt{init}_{D_0}) \to \mathcal{K}(M, \texttt{init})$$

with corresponding labeling functions $L \circ h$ and L.

Proof. According to Theorem 2.2, h is such a stuttering bisimulation map if there is a well-founded domain $(W, >)$ and a function $\mu : \mathit{Reach}(\texttt{init}_{D_0}) \times \mathit{Reach}(\texttt{init}) \to W$ so that:

1. $h(\texttt{init}_{D_0}) = \texttt{init}$.

2. If $h(t) = u$ and $t \longrightarrow t'$ then either $u \longrightarrow u'$ for some $u' = h(t')$, or $h(t') = u$ and $\mu(t, u) > \mu(t', u)$.

3. If $h(t) = u$ and $u \longrightarrow u'$, then there is a sequence of transitions $t \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$, with $n \geq 1$, such that $h(t_n) = u'$ and $h(t_1) = \cdots = h(t_{n-1}) = u$.

4. $h(t)$ and t satisfy the same atomic propositions, which holds since $L \circ h$ is $\mathcal{K}(D_0(M, \texttt{init}, di), \texttt{init}_{D_0})$'s labeling function.

(In this proof, terms $t, t', t_1, \ldots$ and rewrites between such terms denote states and transitions in $\mathcal{K}(D_0(M, \texttt{init}, di), \texttt{init}_{D_0})$, and the u's denote states and transitions in $\mathcal{K}(M, \texttt{init})$.)
1. As explained in Section 7.2.1, the initial state $\texttt{init}_{D_0}$ just adds a number of `Med` objects to the initial state `init` of M. Since h forgets all `Med` objects, we have the desired $h(\texttt{init}_{D_0}) = \texttt{init}$.
2. Assume that $t \longrightarrow t'$ is a rewrite in $D_0(M, \texttt{init}, di)$ and $h(t) = u$.

- If the rule used in the rewrite above is of the form (†):

  m `<` $o : C$ `|` atts `> => <` $o : C$ `|` atts' `> filter(`msgs`)` [`if ...`]

  (the case with rules of the form (‡) is easier), then there is a substitution θ such that $t = c_0\ m\theta\ \ (\texttt{<}\, o\ C \mid \mathit{atts}\,\texttt{>})\theta$ and $t' = c_0\ \ (\texttt{<}\, o : C \mid \mathit{atts}'\,\texttt{>})\theta)\ \texttt{filter}(\mathit{msgs}\theta)$. Since $h(\texttt{filter}(\mathit{msgs}\theta)) = \mathit{msgs}\theta$, and $h(t) = h(c_0)\ m\theta\ (\texttt{<}\, o : C \mid \mathit{atts}\,\texttt{>})\theta$, it follows that $h(t) \longrightarrow h(t')$ with the rule

  m `<` $o : C$ `|` atts `> => <` $o : C$ `|` atts' `>` msgs [`if ...`].

- For the additional rewrite rules in $D_0(M, \texttt{init}, di)$, we use $(\mathit{MsgConf}, >_{mul})$ as the well-founded order, where *MsgConf* denotes finite multisets of $D_0(M, \texttt{init}, di)$-messages, and $>_{mul}$ is the multiset extension induced by the order $>$ given by $(\texttt{to } o \texttt{ transfer } mc \texttt{ from } o' \texttt{ to } o'') > \texttt{transfer(}o''', o\texttt{, msg2string(to } o'' \texttt{ from } o' \texttt{ : } mc\texttt{))} > (\texttt{to } o'' \texttt{ from } o' \texttt{ : } mc)$. We define $\mu(t,u)$ to be the multisets of messages in t.

  Since the rules `sendRemote` and `receiveRemoteMsg` only replace a message with the corresponding h-equivalent and $>$-decreasing message, $t \longrightarrow t'$ using one of these rules means that $h(t) = h(t')$ and $\mu(t,u) >_{mul} \mu(t',u)$ (for any u).

3. Any step in M can be simulated by a sequence of steps in $D_0(M, \texttt{init}, di)$. Suppose that $u \longrightarrow u'$ in M and $h(t) = u$. Then either a rule of the form (†), say,

m `<` o `:` C `|` *atts* `> => <` o `:` C `|` *atts'* `>` *msgs* `[if ...]`,

or of the form (‡) is used. We show the case for rule (†); the (‡) case is trivial. For a (†) rule, u must have the form $u = h(c_0)\ h(m')\ (\texttt{<}\, o : C \mid \mathit{atts}\, \texttt{>})\theta$ with $h(m') = m\theta$ and $u' = h(c_0)\ (\texttt{<}\, o : C \mid \mathit{atts}'\, \texttt{>})\theta\ \mathit{msgs}\theta$. We can distinguish two cases: (i) if $m' = m\theta$ then $t = c_0\ m\theta\ (\texttt{<}\, o : C \mid \mathit{atts}\, \texttt{>})\theta$ and can be rewritten to $t' = c_0\ (\texttt{<}\, o : C \mid \mathit{atts}'\, \texttt{>})\theta\ \texttt{filter}(\mathit{msgs}\theta)$, so that $h(t') = u'$, as desired. Otherwise, $m\theta$ must be of the form $(\texttt{to } b \texttt{ from } a \texttt{ : } x)$, with $o\theta = b$, and m' is either (i) $(\texttt{to } di(a) \texttt{ transfer } x \texttt{ from } a \texttt{ to } b)$, or (ii) $\texttt{transfer(}di(b)\texttt{,}di(a)\texttt{, msg2string(to } b \texttt{ from } a \texttt{:} x\texttt{))}$. We do case (i) and leave case (ii) (requiring fewer steps) to the reader. The c_0 has the form $c_0 = \texttt{<}\, di(a) : \texttt{Med} \mid \texttt{>}\ \texttt{<}\, di(b) : \texttt{Med} \mid \texttt{>}\ c_0'$ and we have rewrites $t \longrightarrow t_1 = \texttt{<}\, di(a) : \texttt{Med} \mid \texttt{>}\ \texttt{transfer(}di(b), di(a)\texttt{,msg2string(to } b \texttt{ from } a \texttt{ : x))}$ $\texttt{<}\, di(b) : \texttt{Med} \mid \texttt{>}\ \texttt{<}\, b : C \mid \mathit{atts}\theta\, \texttt{>}\ c_0' \longrightarrow t_2 = \texttt{<}\, di(a) : \texttt{Med} \mid \texttt{>}\ (\texttt{to } b \texttt{ from } a \texttt{ : x})\ \texttt{<}\, di(b) : \texttt{Med} \mid \texttt{>}$ $\texttt{<}\, b : C \mid \mathit{atts}\theta\, \texttt{>}\ c_0' \longrightarrow t_3 = \texttt{<}\, di(a) : \texttt{Med} \mid \texttt{>}\ \texttt{<}\, di(b) : \texttt{Med} \mid \texttt{>}\ \texttt{<}\, b : C \mid \mathit{atts}'\theta\, \texttt{>}\ \texttt{filter}(\mathit{msgs}\theta)\ c_0'$. But then $h(t) = h(t_1) = h(t_2) = t$, and $h(t_3) = u'$, as desired. □

The main result immediately follows from Theorems 2.1 and 7.1:

Theorem 7.2. Given a rewrite theory M specifying a distributed system and an initial state `init` as described in Section 7.1, a distribution information function di mapping the top-level objects in `init` to different machines/Maude sessions, a labeling function L over a set AP of atomic propositions, and a CTL^* formula φ over AP not containing the "next" operator, then

$$\mathcal{K}(M, \texttt{init}) \models \varphi \text{ if and only if } \mathcal{K}(D_0(M, \texttt{init}, di), \texttt{init}_{D_0}) \models \varphi$$

for the labeling function $L \circ h$ in $\mathcal{K}(D_0(M, \texttt{init}, di), \texttt{init}_{D_0})$.

7.3 PROTOTYPE AND EXPERIMENTS

We have implemented, in around 300 LOC, a prototype of the D transformation that automatically transforms a Maude model of a distributed system into a distributed Maude implementation. We have applied our prototype to the Maude specification of: (i) a well-known lock-based distributed transaction protocol which has been implemented in C++ and evaluated in [47]; and (ii) the ROLA transaction system design (Chapter 6), whose correctness and performance have been analyzed using Maude and PVeStA, but which has never been implemented. Using our prototype and the Maude specification of ROLA we obtain the first distributed implementation of ROLA for free.

We have subjected our two distributed Maude implementations so obtained to realistic workloads generated by YCSB to answer to the following questions:

Q1: Are the performance evaluations obtained for the distributed Maude implementations consistent with the performance predictions obtained by statistical model checking for the original Maude designs? If a conventional distributed implementation of the design is also available, is its performance consistent with the distributed Maude one and with the model-based predictions?

Q2: How does the performance of a distributed Maude implementation $D(M)$ automatically generated by the unoptimized prototype transformation D from a Maude design M compare with that of an available state-of-the-art distributed implementation in C++ of such a design?

Note that answers to Q1 cannot take the form of an exact or approximate agreement between the performance *values predicted* by statistical model checking a Maude model and the *values measured* in an experimental evaluation. This is impossible because: (i) measured values depend on the experimental platform used; (ii) the probability distributions used in statistical model checking are only approximations of the expected behavior; and (iii) the sizes (e.g., number of objects) of initial states used in statistical model checking and in experimental evaluations are typically quite different, due to feasibility restrictions placed by statistical model checking.

For the above reasons (i)-(iii), the consistency to be expected between the performance predicted by statistical model checking a model and those obtained by experimentally evaluating an implementation is not an agreement between predicted and measured *values*, but between predicted and measured *trends*. For example, if throughput increases as a function of the proportion of read and write transactions, then consistency means that it should do so along curves that are *similar* up to some change of scale.

7.3.1 Experimental Setup

Implementation-Based Evaluation. We have evaluated the two case studies using the Yahoo! Cloud Serving Benchmark (YCSB) [36], which is the open standard for comparative performance evaluation of data stores. We used the built-in C++ implementation of YCSB in [47] in our first case study. For ROLA, we used a variant of the original Java implementation of YCSB adapted to transaction systems [14]. We deployed the two case studies on a cluster of d430 Emulab machines [90], each with two 2.4 GHz 8-Core Intel Xeon processors and 64 GB RAM. The ping time between machines is approximately 0.13 ms. We also set the same system and workload configuration. In both cases, we considered 5 partitions (of the entire database) on 5 machines, and all client processes split across another 5 separate machines; we considered the same mixture of read-only, write-only, and read-write transactions, with each transaction accessing up to 8 keys; and we used Zipfian distribution for key accesses with the parametric skew factor *theta*.

Statistical Model Checking (SMC). By running Monte-Carlo simulations from a given initial state, SMC verifies a property (or estimates the expected value of an expression) up to a user-specified level of confidence. We probabilistically generated initial states so that each PVeStA simulation starts from a different initial state. To mimic the real-world network environment, we used the lognormal distribution for message delays [17]. We used 10 machines of the above type to perform statistical model checking with PVeStA. The confidence level for all our statistical experiments is 95%.

Standard Model Checking. We used our Maude models of NO_WAIT and ROLA specified in the CAT framework (Chapter 3) for model checking consistency properties of distributed transaction systems. The analysis was performed with exhaustively generated initial states for a size bound.

Trusted Code Base. Our trusted code base includes the Maude implementation (including the implementation of TCP/IP external socket objects) as well as the Python-based tool used for deploying and initializing the D-transformed distributed Maude system.

7.3.2 Lock-Based Distributed Transactions

This case study considers the protocol NO_WAIT implemented in the Deneva framework [47] using C++. NO_WAIT is a strict two-phase locking (strict 2PL)-based distributed transaction system with two-phase commit (2PC) as its atomic commitment protocol.

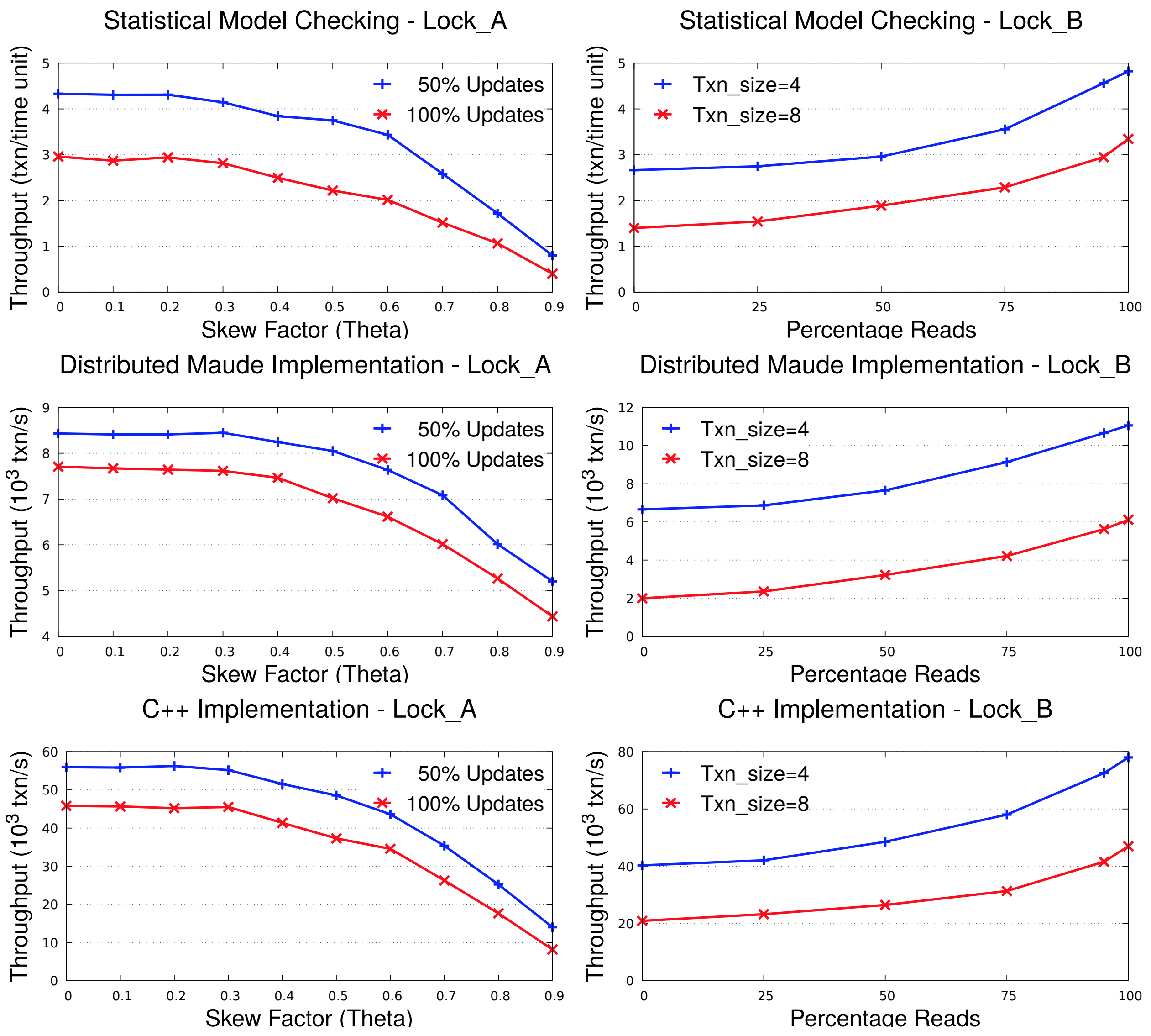


Figure 7.2: NO_WAIT: Throughput comparison between statistical model checking (top), distributed Maude implementation (middle), and C++ implementation (bottom). Experiments Lock_A (left) and Lock_B (right) measure throughput of different ratios of updates and transaction sizes when varying skew factors and ratios of reads, respectively.

We formally specified NO_WAIT in Maude, and then automatically D-transformed the Maude specification to its corresponding distributed Maude implementation. We used the C++ implementation in [47] in our experiments with NO_WAIT in [47]. Our Maude model of NO_WAIT is around 600 LOC, whereas the C++ implementation in [47] has approximately 12K LOC.

We performed two sets of experiments (Lock_A and Lock_B in Fig. 7.2), focusing on the effect of varying amounts of contention in the system. For each set of experiments we plot the experimental results of statistical model checking of our Maude model, and show the corresponding plots for the measurements of the distributed Maude and C++ implementations.

In Lock_A we vary the contention by tuning the skew *theta*, and compare two workloads with 50% and 100% update transactions, respectively. In Lock_B we analyze the throughput as a function of the percentage of read-only transactions with skew $theta = 0.5$, and focus on the impact of transaction sizes (i.e., number of operations in a transaction). Regarding Q1, all three plots in each experiment show similar trends for the model- and implementation-based evaluations. That is, our distributed Maude implementation-based evaluation not only confirms the statistical predictions, but also agrees with the state-of-the-art implementation-based results.

Regarding Q2, our correct-by-construction lock-based distributed transaction system achieves lower peak throughput, but only by a factor of 6, than the corresponding C++ implementation. Some reasons for this lower performance are: (i) the $M \mapsto D(M)$ transformation is an unoptimized prototype; instead, the C++ implementation of NO_WAIT is optimized for high performance (e.g., the socket library *nanomsg* provides a fast and scalable networking layer); and (ii) the $M \mapsto D(M)$ transformation allows adding any benchmarking tool as a *foreign object*, wich is very flexible but adds an extra layer of communication; instead, in the C++ implementation YCSB and the protocol clients are directly integrated.

Model Checking Consistency Properties. We have used the tool CAT (Chapter 3) to model check our Maude model of NO_WAIT against 6 consistency properties (*read committed*, *read atomicity*, *cursor stability*, *update atomicity*, *snapshot isolation*, and *serializability*), without finding a violation of any of them. Under the assumption that our trusted code base executes correctly, Theorem 7.2 ensures that our distributed Maude implementation of NO_WAIT satisfies the same consistency properties for the corresponding initial states.

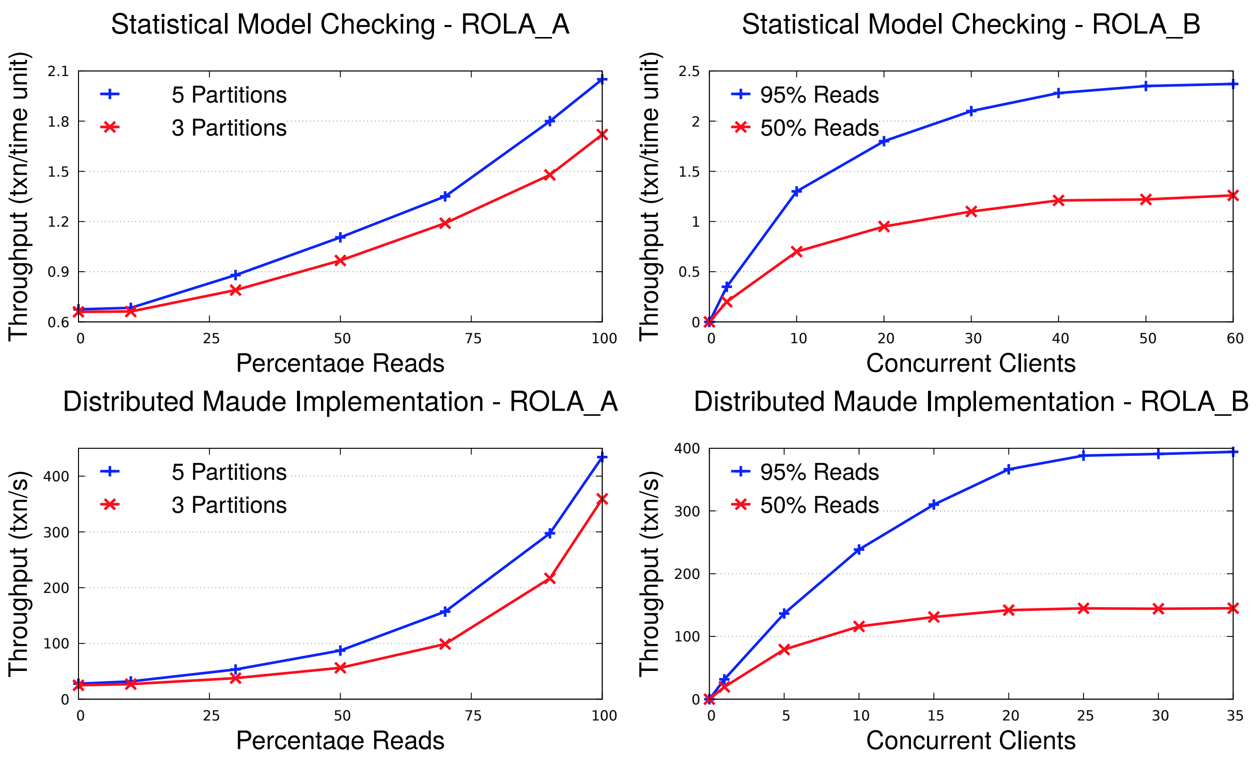


Figure 7.3: ROLA: Comparison between statistical model checking (top) and distributed Maude implementation (bottom). Experiments ROLA_A (left) and ROLA_B (right) measure throughput for different number of partitions and different ratios of reads when varying ratios of reads and concurrent clients, respectively.

7.3.3 The ROLA Transaction System

ROLA is a recent distributed transaction protocol design that guarantees read atomicity (RA) and prevents lost updates (PLU). As shown in Chapter 6, ROLA has been formalized in the CAT framework, and model checked against the above consistency properties. Statistical model checking performance estimation has also showed that ROLA outperforms well-known distributed transaction system designs guaranteeing RA and PLU (Chapter 6). However, up to now there was no distributed implementation of ROLA. Using our tool and the Maude specification of ROLA in the CAT framework (which consists of approximately 850 LOC), we obtain such a correct-by-construction distributed implementation *for free*.

We have performed statistical model checking of the Maude specification, and have run our distributed Maude implementation on YCSB-generated workloads, on two groups of experiments (see Fig. 7.3). In ROLA_A we increase the amount of reads, and compare throughput with various partitions of the entire database (5 partitions against 3 partitions). In ROLA_B we plot throughput as a function of the number of concurrent clients, and focus on the effect of increasing the amount of contention (95% reads against 50% reads). Both plots in each experiment agree reasonably well.

All ROLA's consistency properties model checked in Chapters 3 and 6 are preserved (Theorem 7.2) assuming correct execution of the trusted code base.

7.4 RELATED WORK

Our work is related to various formal frameworks for specification, verification, and implementation of distributed systems that try to reduce the *formality gap* [92] between the formal specification of a distributed system's *design* and its *implementation.* They can be roughly classified in three categories (only some example frameworks in each category are discussed):

1. Specification, Verification, and Compilation to Imperative Implementation. The IOA formal framework [65, 42] formalizes distributed system designs as IO-automata, provides a toolset for both model checking and theorem proving verification of IOA designs, and offers also the possibility of generating Java distributed implementations of IO designs by compilation.

2. Specification, Verification, and Proof of Imperative Implementation. A good example of state-of-the art recent work in this category is the IronFleet framework [48]. Distributed systems are specified in a mixture of Lamport's TLA and Hoare logic assertions for imperative sequential code in Leino's Dafny language [52]. They are then formally verified with various tools, including Z3 [39] and the Dafny prover. Dafny code is then complied into C# code.

3. Specification, Verification, and Transformation into Correct Distributed Implementation. Work in this category has for the most part been based on constructive logical frameworks such as those of Nuprl [35] and Coq [21] and has been shown effective in generating sophisticated system implementations. In particular: (i) the *Event-ML* framework begins with an Event-ML specification and the desired properties both expressed in Nuprl and extracts a GPM program implementation; (ii) the *Verdi* framework [91] begins with a distributed system design and a set of safety properties, both specified in Coq; it offers the important advantage of allowing the specifier to ignore various network failures and replication issues: they are delegated to so-called *verified system transformers* which automatically transform the design and ensure correct execution of the transformed design under such failure scenarios. After desired properties are verified in Coq, the OCaml code of a correct implementation is extracted and deployed using a trusted shim; (iii) the *Chapar* framework [53] is specialized to extract correct-by-construction implementations of key-value stores in OCaml from formal specifications of such stores and of their consistency properties expressed and verified in Coq; and (iv) the *Disel* modular framework [82] specifies both distributed system designs and their desired properties in separation logic, it expresses both the system and property specifications in Coq, uses Coq to prove the desired properties, and extracts correct-by construction OCaml code, which is then deployed using a trusted shim.

Discussion and Comparison with the Maude Framework. To the best of our knowledge, none of the above frameworks provide support for prediction of performance properties by statistical model checking,[4] whereas Maude does so through the PVeSta tool [7]. Regarding work in category (1), the Maude framework shares the use of *executable specifications* and the availability of a formal environment of model checking and theorem proving tools with IOA; but in comparison with IOA's automatic generation of Java distributed implementations from IOA specifications, the Maude approach substantially reduces the "formality gap" by avoiding compilation into a complex imperative language. The main difference with the IronFleet framework in category (2) is that imperative programs are a problematic, low level choice for expressing formal design specifications. Furthermore, system properties can be considerably harder to prove at that level. Regarding frameworks in category (3), the present work within the Maude framework shares with them the possibility of generating correct-by-construction distributed implementations from designs; but adds to them the following additional possibilities: (i) rapid exploration of different design alternatives by testing and by automatic breadth first search, LTL and statistical model checking analysis of such designs; (ii) prediction of system performance properties before implementation; and (iii) flexible range of properties that can be verified of a design: theorem proving verification of both invariants [77] and reachability logic properties [84] is supported but is not *required*: LTL and statistical model checking verification can already yield systems with considerably higher quality than those developed by conventional methods. The main point is that, for an entirely new system never specified or built before, beginning with a human-intensive theorem proving verification effort may be both premature and costly. Instead, in the Maude framework designs can be thoroughly analyzed and improved by fully automated methods *before* a mature design is fully verified using theorem proving tools.

[4]Probabilistic system behaviors can be specified using probabilistic IOA [28]. However, we are not aware of tools supporting statistical model checking analysis of performance properties for distributed system designs in the IOA framework.

7.5 CONCLUDING REMARKS

In this chapter we have presented the $M \mapsto D(M)$ transformation and proved that M and a model $D_0(M)$ of $D(M)$ abstracting network communication details are *stuttering bisimilar* and therefore satisfy the same safety and liveness properties. We have also presented two case studies evaluating the performance of $D(M)$ for designs M of two state-of-the-art distributed transaction systems, and that of a high-perfomance conventional implementation. These case studies have also confirmed that the statistical-model-checking-based performance predictions obtained from a design M before implementation are similar to the performance measures for $D(M)$ and a conventional implementation. This work shows that it is possible to automatically generate reasonable, but not yet optimal, correct-by-construction distributed implementations from very high level and easy to understand executable formal specifications of state-of-the-art system designs which are much shorter (a factor of 20 for the C++ implementation of NO_WAIT) than conventional implementations.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

This chapter presents the conclusions of this dissertation, followed by a discussion on future research directions.

8.1 CONCLUSIONS

Designing, verifying, and implementing highly reliable high-performance distributed systems is at present a hard and very labor-intensive task. Cloud-based systems have further increased this complexity due to the desired consistency, availability, scalability, and disaster tolerance. This dissertation has addressed this challenge in the context of distributed transaction systems (DTSs) from two complementary perspectives: (i) designing cloud storage systems with high assurance such that they satisfy desired *correctness* and *performance* requirements; and (ii) transforming *verified system designs* into *correct-by-construction distributed implementations.*

Regarding the *correctness* requirements mentioned in (i), we have provided an object-based framework (called CAT) for formally modeling DTSs in Maude, have explained how such models can be automatically instrumented to record relevant events during a run, and have formally defined a wide range of consistency properties on such histories of events. We have implemented a tool which automates the entire specification instrumentation and model checking process.

Regarding the *performance* requirements mentioned in (i), we have proposed a general, though not yet automated, method that transforms the untimed, non-probabilistic, and nondeterministic formal Maude models of DTSs in the CAT framework into probabilistic rewrite theories, have explained how we can monitor the system executions of such probabilistic theories, and have showed how we can evaluate the performance of the DTS designs based on the recorded log for different performance parameters and workloads using the PVeStA statistical model checker.

Three DTS case studies, namely, RAMP, Walter, and ROLA, have been presented in detail to demonstrate the applicability of the above two methodologies. We have shown *new* and *promising* results for each of the three case studies. In particular, we have identified promising new design alternatives for given classes of applications relatively easily *before* they are implemented:

RAMP. We have used our models to analyze RAMP's extensions against a wide range of consistency properties, which the original RAMP paper did not do. Besides, we have explored various design alternatives for RAMP transactions, and have been able to make rigorous comparisons between them in terms of their consistency and performance properties by means of logical and statistical model checking. In this way, we have gained substantial knowledge that can help find the best match between a RAMP version and a given class of applications.

Walter. Our statistical model checking analysis has offered new insights about Walter's performance for a wider range of workloads than those evaluated experimentally by the developers [85], while at the same time showing agreement between the performance predictions obtained from a Maude model of Walter by statistical model checking, and the experimental performance evaluations in [85].

ROLA. Our work on ROLA has shown, to the best of our knowledge for the first time, that the design and validation of a *new* distributed transaction protocol can be achieved relatively quickly *before* its implementation by the use of formal methods.

Regarding challenge (ii), to bridge the formality gap between verified designs of actor-like distributed systems and their distributed implementations we have presented a correct-by-construction automatic transformation mapping a verified formal specification of an actor-based distributed system design in Maude to a distributed implementation enjoying the same safety and liveness properties as the original formal design. Two case studies, applying this automatic transformation to state-of-the-art distributed transaction systems analyzed within the same formal framework for *both* logical and performance properties, show that high-quality implementations with acceptable performance and meeting performance predictions can be automatically generated in this way.

8.2 FUTURE WORK

Extending and Optimizing the CAT framework. We have formalized 9 transactional consistency properties in the CAT framework, and modeled checked whether they are satisfied or not for 13 DTS models. In future work we should verify the correctness of our formalization of those consistency properties by formally relating our definitions in Maude to other (non-executable) formalizations (e.g., Adya's formalization of transactional consistency models [4]).

Although the model checking verification has been made systematic by generating all possible initial states with certain bounds, there is still room for obtaining higher coverage and assurance in the future to arrive at stronger claims for our model checking analysis. In particular, we should (automatically) explore all possible combinations of the user-provided parameters, instead of (manually) choosing some of them. For example, in Section 5.3 we have model checked our Walter model against SI & PSI for only 5 (out of 10) combinations with 3 types of transactions, but of course a full coverage of all combinations would provide fuller assurance.

The CAT framework focuses only on distributed *transaction* systems and their *transactional* consistency properties (read atomiciy, serializability, etc.). In future work we should extend the CAT framework to formalizing and model checking NoSQL key-value stores[1] (e.g., Cassandra [2] and Riak [1]) and their *non-transactional* consistency properties [89] (e.g., eventual consistency and strong consistency). Previous work [63, 56, 55] on formal modeling of Cassandra and formal analysis of its consistency models can be a good basis for this future research direction.

Automatic Performance Estimation of DTSs. Regarding design and verification of DTSs, the CAT framework currently focuses only on *correctness* requirements (i.e., model checking consistency properties). To meet performance requirements, an automated approach to performance estimation by statistical model checking is also highly needed. We plan to integrate the consistency and performance measures formally defined in Chapter 4 into the CAT framework, and to automate the transformation (manually done for RAMP, Walter, and ROLA in Chapter 4, 5, and 6, respectively) from nondeterministic models into probabilistic ones that can be subjected to statistical model checking analysis using a tool such as PVeStA [7].

Toolkit for Automatic Analysis of Cloud Storage Systems. The transformation in the CAT framework is from nondeterministic models to probabilistic rewrite theories. Alternatively, we can design a new tool P2ND based on exploring the opposite transformation direction (from a probabilistic model into a nondeterministic one), which, together with the CAT tool, provides users with flexible choices in formal design and analysis of DTSs in Maude. For example, if a user's first attempt at a formal model is a probabilistic one (resp. a nondeterministic one), he can choose P2ND (resp. CAT) to perform automatic analysis of both *correctness* and *performance* properties. Both tools should be integrated into a toolkit,

[1]We classify cloud storage systems into two categories: distributed transactional systems and NoSQL key-value stores.

along with the above feature of also supporting NoSQL data stores, which offers users a generic framework for modeling and analysis of cloud storage systems.

Infinite-State Model Checking. CAT uses explicit-state model checking to explore the system behaviors from many initial states generated with user-provided bounds, and therefore cannot be used to verify that an algorithm is correct for all possible initial system configurations, which can be infinite. One promising technique to addressing this limitation is *symbolic model checking*, 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* [78], which has already been applied to various distributed real-time systems, and by *narrowing-based symbolic model checking* [11, 12]. 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.

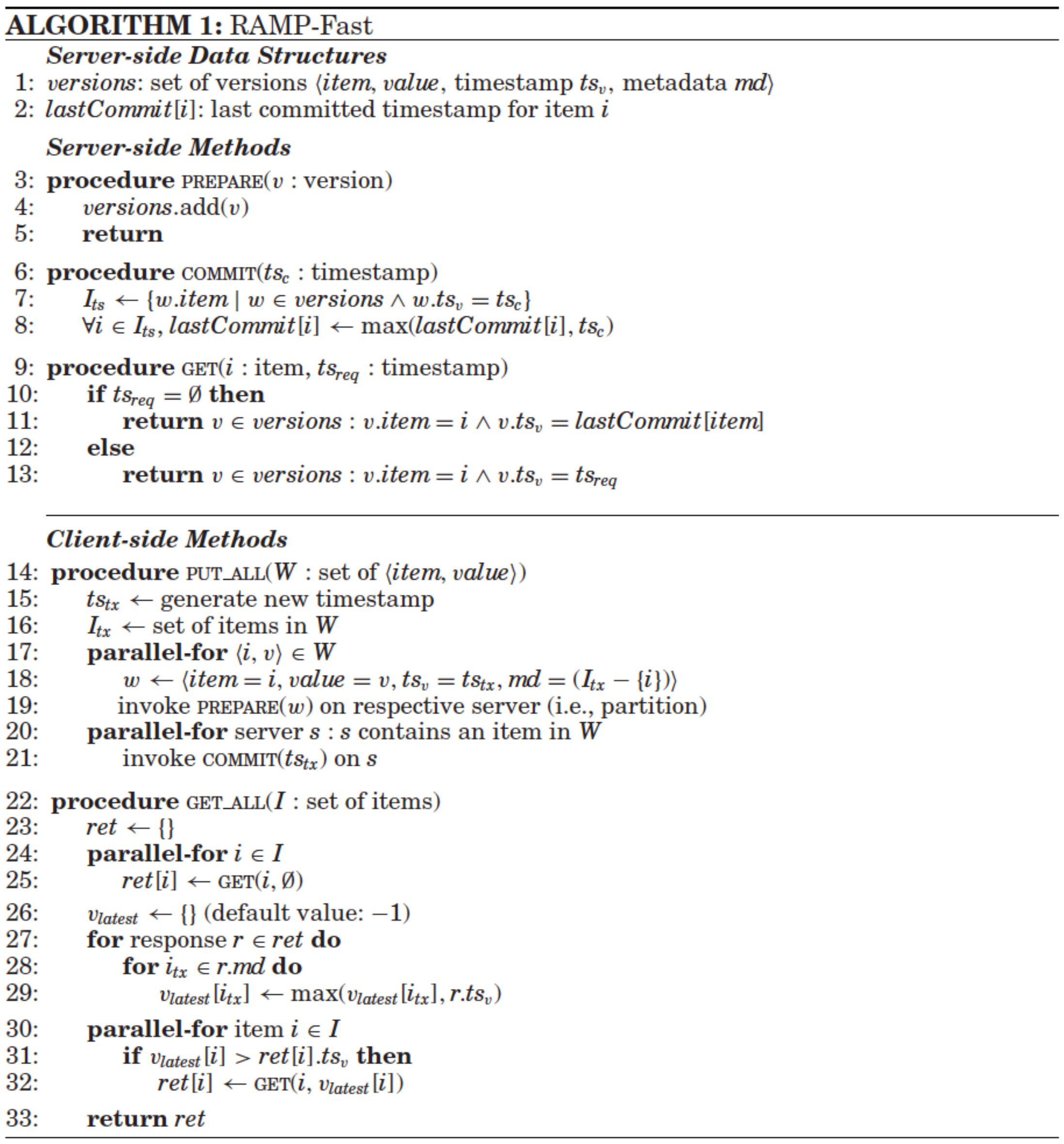
Deductive Verification in Reachability Logic. In the correct-by-construction transformation from distributed system designs to distributed implementations, the stronger the properties that have been verified of the design, the stronger the correctness guarantees that can be claimed for the implementation. Since symbolic model checking does not always terminate in general, it may not be possible to verify some properties this way. Abstraction methods may sometimes make a symbolic state space finite, but finding such an abstraction may not always be possible. A natural next step after infinite-state symbolic model checking verification is *deductive verification*. A promising approach available for Maude-based specification is the constructor-based reachability logic and tool reported in [84] and currently under active development and experimentation.

Modular Design and Analysis of Cloud Storage Systems. Our work in this dissertation is part of a long-term research effort in which we have been using Maude to both meet the challenges and exploit the opportunities of modular design and analysis for cloud storage systems (see [22, 70] for surveys). As part of this effort, we have formally specified 13 Maude models of state-of-the-art DTSs in the CAT framework, as well as Cassandra in [63, 56, 55], and have formally analyzed both the consistency and the performance properties of these systems. From the analysis of all these systems, which span different points in the consistency vs. performance spectrum for cloud storage systems, a better, more modular understanding of the different algorithms that need to be combined to achieve the different designs and their relationships has been gained. An important next step in the near future is to develop a library of formally specified components and show how system designs such as those for RAMP, Walter, ROLA, Cassandra, and other future systems can be obtained as *modular compositions* out of such a library of components.

Optimizing the D Transformation. Despite of the promising results (i.e., our unoptimized prototype is only six times slower than a high-performance implementation), there is still ample room for improvement in the future to arrive at a mature and highly optimized Maude implementation of the D transformation. Extensive experimentation is also needed to more fully demonstrate the effectiveness and performance of the D transformation methodology and to optimize the transformation itself.

APPENDIX A: THE RAMP-FAST ALGORITHM

Figure A.1 shows the RAMP-Fast algorithm as it is described in [14].



```
ALGORITHM 1: RAMP-Fast
    Server-side Data Structures
 1: versions: set of versions ⟨item, value, timestamp $ts_v$, metadata md⟩
 2: lastCommit[i]: last committed timestamp for item i
    Server-side Methods
 3: procedure PREPARE(v : version)
 4:     versions.add(v)
 5:     return
 6: procedure COMMIT($ts_c$ : timestamp)
 7:     $I_{ts} \leftarrow \{w.item \mid w \in versions \wedge w.ts_v = ts_c\}$
 8:     $\forall i \in I_{ts}, lastCommit[i] \leftarrow \max(lastCommit[i], ts_c)$
 9: procedure GET(i : item, $ts_{req}$ : timestamp)
10:     if $ts_{req} = \emptyset$ then
11:         return $v \in versions : v.item = i \wedge v.ts_v = lastCommit[item]$
12:     else
13:         return $v \in versions : v.item = i \wedge v.ts_v = ts_{req}$
    Client-side Methods
14: procedure PUT_ALL(W : set of ⟨item, value⟩)
15:     $ts_{tx} \leftarrow$ generate new timestamp
16:     $I_{tx} \leftarrow$ set of items in W
17:     parallel-for $\langle i, v \rangle \in W$
18:         $w \leftarrow \langle item = i, value = v, ts_v = ts_{tx}, md = (I_{tx} - \{i\}) \rangle$
19:         invoke PREPARE(w) on respective server (i.e., partition)
20:     parallel-for server s : s contains an item in W
21:         invoke COMMIT($ts_{tx}$) on s
22: procedure GET_ALL(I : set of items)
23:     $ret \leftarrow \{\}$
24:     parallel-for $i \in I$
25:         $ret[i] \leftarrow$ GET($i, \emptyset$)
26:     $v_{latest} \leftarrow \{\}$ (default value: −1)
27:     for response $r \in ret$ do
28:         for $i_{tx} \in r.md$ do
29:             $v_{latest}[i_{tx}] \leftarrow \max(v_{latest}[i_{tx}], r.ts_v)$
30:     parallel-for item $i \in I$
31:         if $v_{latest}[i] > ret[i].ts_v$ then
32:             $ret[i] \leftarrow$ GET($i, v_{latest}[i]$)
33:     return ret
```

Figure A.1: The RAMP-Fast algorithm as described in [14].

APPENDIX B: THE ORIGINAL PERFORMANCE EVALUATION RESULTS FOR RAMP

Figures B.1 and B.2 show the RAMP performance w.r.t. throughput and average latency as plotted in [14]. The relative performance for RAMP-Fast and RAMP-Small is indeed similar to the predictions by our statistical model checking analysis in Chapter 4.7.

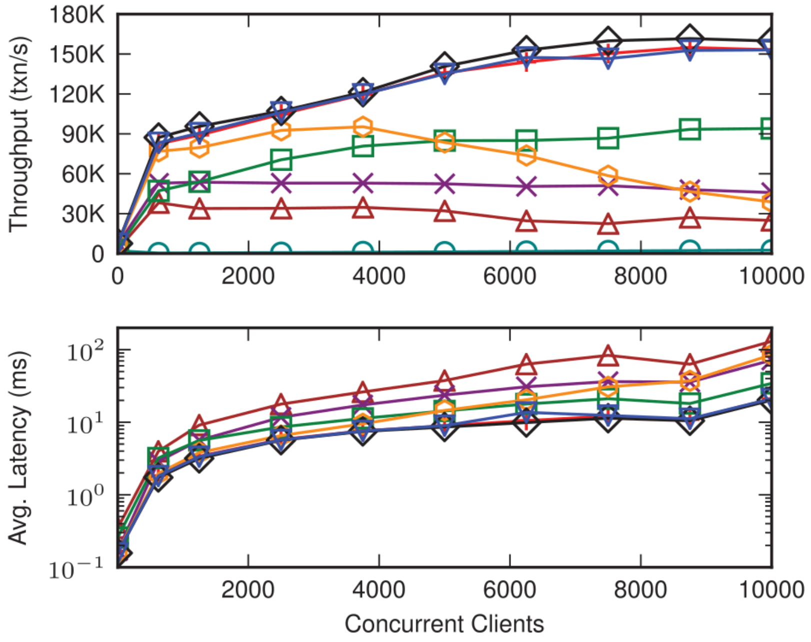


Figure B.1: Throughput and latency under varying client load as shown in [14]. Blue and green curves refer to RAMP-Fast and RAMP-Small, respectively.

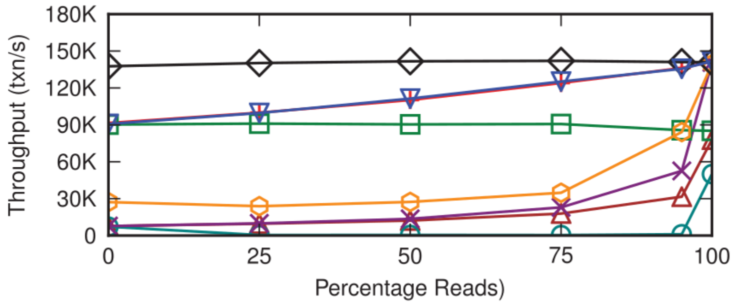


Figure B.2: Throughput under varying percentage reads as shown in [14]. Blue and green curves refer to RAMP-Fast and RAMP-Small, respectively.

APPENDIX C: THE ORIGINAL PERFORMANCE EVALUATION RESULTS FOR WALTER

Figure C.1 shows the Walter throughput under varying workload as plotted in [85], which is consistent with our statistical model checking results in Chapter 5.4.

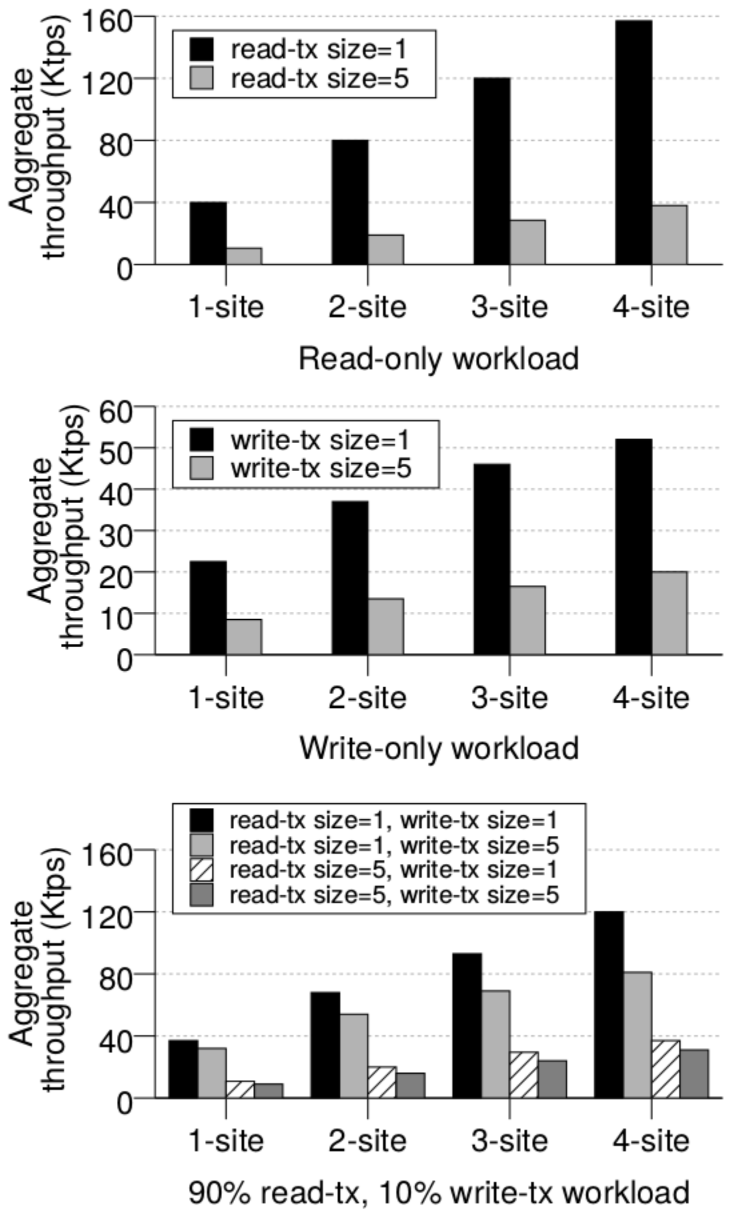


Figure C.1: Throughput under varying workload as shown in [85]

REFERENCES

[1] Basho Riak. `http://basho.com/riak/`.

[2] Cassandra. `http://cassandra.apache.org`.

[3] IBM DB2. `https://www.ibm.com/analytics/us/en/db2/`.

[4] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* PhD thesis, MIT, 1999.

[5] Gul A. Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. *Electr. Notes Theor. Comput. Sci.*, 153(2), 2006.

[6] Musab AlTurki and José Meseguer. Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In *RTRTS'10*, volume 36 of *EPTCS*, pages 26–45, 2010.

[7] Musab AlTurki and José Meseguer. PVeStA: A parallel statistical model checking and quantitative analysis tool. In *CALCO'11*, volume 6859 of *LNCS*, pages 386–392. Springer, 2011.

[8] Masoud Saeida Ardekani, Pierre Sutra, Nuno M. Preguiça, and Marc Shapiro. Non-monotonic snapshot isolation. *CoRR*, abs/1306.3906, 2013.

[9] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, pages 163–172. IEEE Computer Society, 2013.

[10] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *OSDI'14*, pages 367–381. USENIX Association, 2014.

[11] Kyungmin Bae, Santiago Escobar, and José Meseguer. Abstract Logical Model Checking of Infinite-State Systems Using Narrowing. In *Rewriting Techniques and Applications (RTA'13)*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013.

[12] Kyungmin Bae and José Meseguer. Infinite-state model checking of LTLR formulas using narrowing. In *Proc. WRLA 2014*, volume 8663 of *LNCS*, pages 113–129. Springer, 2014.

[13] Kyungmin Bae and José Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. *Sci. Comput. Program.*, 99:193–234, 2015.

[14] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP transactions. *ACM Trans. Database Syst.*, 41(3):15:1–15:45, 2016.

[15] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. Scalable atomic visibility with RAMP transactions. In *Proc. SIGMOD'14*. ACM, 2014.

[16] Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR'11*, pages 223–234, 2011.

[17] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC'10*, pages 267–280. ACM, 2010.

[18] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10. ACM, 1995.

[19] Giovanni Bernardi and Alexey Gotsman. Robustness against consistency models with atomic visibility. In *CONCUR*, volume 59 of *LIPIcs*, pages 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[20] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[21] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[22] Rakesh Bobba, Jon Grov, Indranil Gupta, Si Liu, José Meseguer, Peter Csaba Ölveczky, and Stephen Skeirik. Survivability: Design, formal modeling, and validation of cloud storage systems using Maude. In *Assured Cloud Computing*, chapter 2, pages 10–48. Wiley-IEEE Computer Society Press, 2018.

[23] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying eventual consistency of optimistic replication systems. In *POPL*, pages 285–296. ACM, 2014.

[24] Roberto Bruni and José Meseguer. Semantic foundations for generalized rewrite theories. *Theor. Comput. Sci.*, 360(1-3):386–414, 2006.

[25] Sebastian Burckhardt. *Principles of Eventual Consistency*, volume 1 of *Foundations and Trends in Programming Languages*. Now Publishers, 2014.

[26] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *ESOP*, volume 7211 of *LNCS*, pages 67–86. Springer, 2012.

[27] Simin Cai. Modeling real-time transactions in UPPAAL. Technical report, Mälardalen University, 2015. http://www.es.mdh.se/publications/3911.

[28] Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Moses Liskov, Nancy A. Lynch, Olivier Pereira, and Roberto Segala. Task-structured probabilistic I/O automata. *J. Comput. Syst. Sci.*, 94:63–97, 2018.

[29] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[30] Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. In *PODC*, pages 55–64. ACM, 2016.

[31] Edmund M. Clarke, Orna. Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2001.

[32] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[33] Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285:245–288, 2002.

[34] Manuel Clavel, José Meseguer, and Miguel Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373:70–91, 2007.

[35] Robert Lee Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1987.

[36] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SOCC'10*, pages 143–154. ACM, 2010.

[37] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is believing: A client-centric specification of database isolation. In *PODC*, pages 73–82. ACM, 2017.

[38] C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, 5 edition, 1990.

[39] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[40] Ha Thi Thu Doan, Kazuhiro Ogata, and François Bonnet. Specifying a distributed snapshot algorithm as a meta-program and model checking it at meta-level. In *ICDCS*. IEEE Computer Society, 2017.

[41] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. Statistical model checking for composite actor systems. In *WADT'12*, volume 7841 of *LNCS*. Springer, 2013.

[42] Chryssis Georgiou, Nancy A. Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber. Automated implementation of complex distributed algorithms specified in the IOA language. *STTT*, 11(2):153–171, 2009.

[43] Wojciech M. Golab, Muntasir Raihan Rahman, Alvin AuYoung, Kimberly Keeton, and Indranil Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *ICDCS*, pages 493–502. IEEE Computer Society, 2014.

[44] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In *POPL*, pages 371–384. ACM, 2016.

[45] Jon Grov and Peter Csaba Ölveczky. Formal modeling and analysis of Google's Megastore in Real-Time Maude. In *Specification, Algebra, and Software*, volume 8373 of *LNCS*. Springer, 2014.

[46] Jon Grov and Peter Csaba Ölveczky. Increasing consistency in multi-site data stores: Megastore-CGC and its formal analysis. In *SEFM*, volume 8702 of *LNCS*. Springer, 2014.

[47] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, 2017.

[48] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *SOSP*. ACM, 2015.

[49] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC'10*. USENIX Association, 2010.

[50] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: multi-data center consistency. In *EuroSys*, pages 113–126. ACM, 2013.

[51] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *OSDI*. USENIX Association, 2014.

[52] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR'10*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[53] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL'16*, pages 357–370. ACM, 2016.

[54] Jinle Li. Model checking transaction properties for concurrent real-time transactions in UPPAAL. Master's thesis, Mälardalen University, 2016.

[55] Si Liu, Jatin Ganhotra, Muntasir Rahman, Son Nguyen, Indranil Gupta, and José Meseguer. Quantitative analysis of consistency in NoSQL key-value stores. *Leibniz Transactions on Embedded Systems*, 4(1):03:1–03:26, 2017.

[56] Si Liu, Son Nguyen, Jatin Ganhotra, Muntasir Raihan Rahman, Indranil Gupta, and José Mesegue. Quantitative analysis of consistency in NoSQL key-value stores. In *QEST 2015*, pages 228–243, 2015.

[57] Si Liu, Peter Csaba Ölveczky, Jatin Ganhotra, Indranil Gupta, and José Meseguer. Exploring design alternatives for RAMP transactions through statistical model checking. In *ICFEM*, LNCS. Springer, 2017.

[58] Si Liu, Peter Csaba Ölveczky, Muntasir Raihan Rahman, Jatin Ganhotra, Indranil Gupta, and José Meseguer. Formal modeling and analysis of RAMP transaction systems. In *SAC*. ACM, 2016.

[59] Si Liu, Peter Csaba Ölveczky, Keshav Santhanam, Qi Wang, Indranil Gupta, and José Meseguer. ROLA: A new distributed transaction protocol and its formal analysis. In *FASE*, volume 10802 of *LNCS*, pages 77–93. Springer, 2018.

[60] Si Liu, Peter Csaba Ölveczky, Qi Wang, Indranil Gupta, and José Meseguer. Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. Technical report, University of Illinois at Urbana-Champaign, 2018. http://hdl.handle.net/2142/101836.

[61] Si Liu, Peter Csaba Ölveczky, Qi Wang, and José Meseguer. Formal modeling and analysis of the Walter transactional data store. In *WRLA*, volume 11152 of *LNCS*, pages 136–152. Springer, 2018.

[62] Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. Automatic analysis of consistency properties of distributed transaction systems in Maude. In *TACAS'19*, volume 11428 of *LNCS*. Springer, 2019.

[63] Si Liu, Muntasir Raihan Rahman, Stephen Skeirik, Indranil Gupta, and José Meseguer. Formal modeling and analysis of Cassandra in Maude. In *ICFEM*, volume 8829 of *LNCS*. Springer, 2014.

[64] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *SOSP*, pages 295–310. ACM, 2015.

[65] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[66] Panagiotis Manolios. A compositional theory of refinement for branching time. In *CHARME'03*, volume 2860 of *LNCS*, pages 304–318. Springer, 2003.

[67] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[68] José Meseguer. Membership algebra as a logical framework for equational specification. In *Proc. WADT'97*, volume 1376 of *LNCS*. Springer, 1998.

[69] José Meseguer. Twenty years of rewriting logic. *J. Algebraic and Logic Programming*, 81:721–781, 2012.

[70] José Meseguer. Formal design of cloud computing systems in Maude. Technical report, University of Illinois at Urbana-Champaign, 2018.

[71] José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Algebraic simulations. *J. Log. Algebr. Program.*, 79(2):103–143, 2010.

[72] Hussan Munir, Misagh Moayyed, and Kai Petersen. Considering rigor and relevance when evaluating test driven development: A systematic review. *Inf. Softw. Technol.*, 56(4):375–394, April 2014.

[73] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, April 2015.

[74] Peter Csaba Ölveczky. Formalizing and validating the P-Store replicated data store in maude. In *WADT'16*, volume 10644 of *LNCS*, pages 189–207. Springer, 2016.

[75] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[76] Muntasir Raihan Rahman, Lewis Tseng, Son Nguyen, Indranil Gupta, and Nitin Vaidya. Characterizing and adapting the consistency-latency tradeoff in distributed key-value stores. *ACM Trans. Auton. Adapt. Syst.*, 11(4):20:1–20:36, 2017.

[77] Camilo Rocha and José Meseguer. Proving safety properties of rewrite theories. In *CALCO'11*, volume 6859 of *LNCS*, pages 314–328. Springer, 2011.

[78] Camilo Rocha, José Meseguer, and César A. Muñoz. Rewriting modulo SMT and open system analysis. *J. Log. Algebr. Meth. Program.*, 86(1):269–297, 2017.

[79] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *SRDS'10*, pages 214–224. IEEE Computer Society, 2010.

[80] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In *CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

[81] Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. VESTA: A statistical model-checker and analyzer for probabilistic systems. In *QEST'05*. IEEE Computer Society, 2005.

[82] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, 2017.

[83] Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. Consistency in 3D. In *CONCUR*, volume 59 of *LIPIcs*, pages 3:1–3:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[84] Stephen Skeirik, Andrei Stefanescu, and José Meseguer. A constructor-based reachability logic for rewrite theories. In *LOPSTR'17*, volume 10855 of *LNCS*, pages 201–217. Springer, 2017.

[85] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *SOSP'11*, pages 385–400. ACM, 2011.

[86] Adriana Szekeres and Irene Zhang. Making consistency more consistent: A unified model for coherence, consistency and isolation. In *PaPoC*. ACM, 2018.

[87] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, 2013.

[88] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, pages 309–324. ACM, 2013.

[89] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, 2016.

[90] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*. USENIX Association, 2002.

[91] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *PLDI'15*, pages 357–368. ACM, 2015.

[92] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *CPP'16*, pages 154–165. ACM, 2016.

[93] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: transparent model checking of unmodified distributed systems. In *NSDI*, pages 213–228. USENIX Association, 2009.

[94] Håkan L. S. Younes and Reid G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.*, 204(9):1368–1409, 2006.

[95] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. In *SOSP 2015*, pages 263–278. ACM, 2015.