Generating Correct-by-Construction Distributed Implementations from Formal Maude Designs

Si Liu[1(✉)], Atul Sandur[2], José Meseguer[2], Peter Csaba Ölveczky[3], and Qi Wang[2]

[1] ETH Zürich, Zürich, Switzerland
si.liu@inf.ethz.ch
[2] University of Illinois, Urbana-Champaign, Champaign, USA
[3] University of Oslo, Oslo, Norway

Abstract. Developing a reliable distributed system meeting desired performance requirements is a hard and labor-intensive task. Formal specification and analysis of a system *design* can yield correct designs as well as reliable performance predictions. In this paper we present a correct-by-construction automatic transformation mapping such a verified formal specification of a system design in Maude to a *distributed implementation* satisfying the same safety and liveness properties. Two case studies 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 automatically generated. In this way, formal models of distributed systems analyzed within the same formal framework for both *logical* and *performance* properties are automatically transformed into correct-by-construction implementations for which similar performance trends can be shown.

1 Introduction

Designing and implementing high-performance distributed systems are complex tasks. Cloud-based systems, which typically rely on widely distributed data storage for scalability, availability, and disaster tolerance, have further increased this complexity. 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 important system requirements that should be analyzed.

Formal methods have been advocated to develop and analyze high-level models of distributed system designs. However, today's distributed systems present a number of challenges to formal methods: (i) the complexity and heterogeneity of such systems require a flexible and expressive formal framework [32]; (ii) the correctness properties that these systems must satisfy can be quite complex, and there is a desire in industry for *automatic* verification methods [32]; and (iii) both *correctness* and *performance* are, as mentioned, crucial requirements.

One formal framework that has shown promise in meeting the challenges (i)–(iii) is Maude [10], a high-performance formal framework for executable specification, verification, and programming of concurrent systems based on rewriting

R. Lee et al. (Eds.): NFM 2020, LNCS 12229, pp. 22–40, 2020.
https://doi.org/10.1007/978-3-030-55754-6_2

logic [8,28,29]. Maude meets challenge (i) by being based on a general and expressive, yet simple and intuitive, formalism. Regarding challenge (ii), Maude provides a range of automatic model checking methods, including reachability analysis and LTL and LTLR temporal logic model checking [2,10], which allows us to express and analyze complex properties (see, e.g., [25]). For challenge (iii), the PVeStA [1] *statistical model checker* can be used to statistically predict the performance of a design specified in Maude.

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 systems [29]. To cite just one 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 Apache Cassandra [18], ZooKeeper [19], Google's Megastore [4], P-Store [35], RAMP [3], and Walter [38]; and to design the entirely new system ROLA [22] (see [7,23,24,33]). Furthermore, model-based performance predictions using PVeStA have shown good correspondence with experimental evaluation of implementations of Cassandra, RAMP, and Walter.

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

Since Maude provides TCP/IP sockets as external objects which can interact with standard Maude objects by message passing [10], a Maude object system can be deployed as a *distributed system* across several machines. The goal of this paper is to *fully automate* the passage from an object-based Maude design M to a distributed Maude implementation $D(M)$, 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* [27,30] and therefore satisfy the same CTL^* properties for any formulas not using the "next" operator. Therefore, both *safety* and *liveness* properties are preserved by the transformation. Since both the formal specification and its distributed implementation are given in Maude, proving correctness of the generated code is quite straight-forward. This is in contrast to code generation frameworks that generate code in languages, such as C or Java, that are different from the formal specification language, and where proving correctness of the generated code is hard and typically not done.

We have developed a prototype that automates the $M \mapsto D(M)$ transformation, and have evaluated its effectiveness on two case studies. In the first one we compare the distributed Maude implementation $D(M)$ automatically generated from the Maude specification M of the NO_WAIT transaction protocol with a state-of-the-art conventional C++ implementation of NO_WAIT. In the second case study we compare the Maude design M of the new distributed transaction system ROLA with its *first ever* distributed implementation $D(M)$.

Main Contributions: (i) the formal definition of the $M \mapsto D(M)$ transformation; (ii) the proof that for any actor-like Maude specification M the system $D_0(M)$ and M are stuttering bisimilar; (iii) a "proof-of-concept" implementation 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 *foreign objects* (see Sect. 3.3) such as the YCSB workload generator [12]; (iv) two case studies using state-of-the-art distributed transaction systems 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 high-performance C++ implementation. 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 for which similar performance trends can be shown.

2 Preliminaries

Rewriting Logic and Maude. Maude [10] is a rewriting-logic-based executable formal specification language and high-performance analysis tool for distributed systems. Formal analysis methods include: simulation, reachability analysis, LTL model checking, theorem proving [34,37], and, for performance estimation purposes, statistical model checking with the PVeStA tool [1].

A Maude module specifies a *rewrite theory* $(\Sigma, E \cup B, R)$, where:

- Σ is an algebraic *signature*; i.e., a set of *sorts*, *subsorts*, and *function symbols*.
- $(\Sigma, E \cup B)$ is a *membership equational logic theory* specifying the system's data types, with E a set of conditional equations and membership axioms, and B a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms B.
- R is a collection of *labeled conditional rewrite rules* $[l] : t \longrightarrow t'$ **if** *cond*, specifying the system's local transitions.

We summarize the syntax of Maude and refer to [10] for details. Operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s and can have user-definable syntax. Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`. The mathematical variables in such statements are declared with the keywords `var` and `vars`.

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* of class C is represented as a term `<` o `:` C `|` 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 are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`. A system state is modeled as a term of the sort `Configuration`, and has the structure of a *multiset* made up of objects and messages. The dynamic behavior of a system is axiomatized by specifying its transition patterns as rewrite rules. For example, the rule

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

defines a family of transitions in which a message `m(O, w)` is read and consumed by an object `O` of class `C`, whose attribute `a1` 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` and `a2`, need not be mentioned in a rule.

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, here called a *session*, can communicate with *external objects* in the *same session* by message passing. One such external object is Maude's built-in *socket manager* object, with name `socketManager`, that supports communicating through TCP sockets with other *remote Maude objects* in other Maude sessions, as well as with *remote foreign objects* (see Sect. 3.3) in other processes. Some of the messages defining the interface between a Maude process and Maude's socket manager are the following: A message `send(`*socketName*`,`*myOid*`,`*string*`)` asks Maude to send *string* through the socket *socketName*, and `receive(`*socketName*`,`*myOid*`)` solicits its data through a socket. When some data (*string*) is received through a socket, the socket manager sends the message `received(`*myOid*`,`*socketName*`,`*string*`)`.

Stuttering Bisimulations. A *Kripke structure* $\mathcal{A}$ on a set AP of *atomic propositions* is a 4-tuple $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, a_0, L_{\mathcal{A}})$ where A is a set of *states*, $\rightarrow_{\mathcal{A}} \subseteq A \times A$ is the *total transition relation on states*, $a_0 \in A$ is the *initial state*, and $L_{\mathcal{A}}$, called the *labeling function*, is a function $L_{\mathcal{A}} : A \rightarrow \mathcal{P}(AP)$ 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 $\pi : \mathbb{N} \rightarrow A$ such that $\pi(0) = a_0$ and $\forall n \in \mathbb{N}\ \pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$.

Definition 1. *[30] Given Kripke structures* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, a_0, L_{\mathcal{A}})$ *and* $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, b_0, L_{\mathcal{B}})$*, a* stuttering bisimulation map*, denoted* $h : \mathcal{A} \rightarrow \mathcal{B}$*, is a function* $h : A \rightarrow B$ *such that: (1) given any path* π *in* $\mathcal{A}$ *there is a path* ρ *in* $\mathcal{B}$ *and a strictly monotonic function* $\kappa : \mathbb{N} \rightarrow \mathbb{N}$ *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* $\kappa : \mathbb{N} \rightarrow \mathbb{N}$ *satisfying (i)–(iii).*

The key property of a stuttering bisimulation map $h : \mathcal{A} \rightarrow \mathcal{B}$ 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^* and its LTL sublogic, see [9]):

Theorem 1. *[30] (Implementation Correctness). If* $h : \mathcal{A} \rightarrow \mathcal{B}$ *is a stuttering bisimulation map, for each* $\varphi \in CTL^* \setminus \bigcirc$ *we have:* $\mathcal{B} \models \varphi \Leftrightarrow \mathcal{A} \models \varphi$.

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* $\mathcal{K}(\mathcal{R}, init) = (Reach(init), \longrightarrow^{\bullet}_{R/E}, init, L)$ where $Reach(init)$ is the set of all states $[u] \in T_{\Sigma/E}$ *reachable* from $init$, $\longrightarrow^{\bullet}_{R/E}$ is the (totalization of) the one-step rewrite relation $\longrightarrow_{R/E}$, and L maps each reachable state $[u]$ to the set $L([u]) = \{p \in AP \mid u \models p =_E true\}$.

3 The D Transformation

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

The transformation D takes as input:

- an object-oriented Maude module M defining an actor system (see below);
- an initial state `init` of sort `Configuration`, which is a set of objects

  $\texttt{<}\ o_1 : C_1 \mid atts_1\ \texttt{>} \quad \ldots \quad \texttt{<}\ o_n : C_n \mid atts_n\ \texttt{>}$ with distinct names o_i;

- a *distribution information* function $di : \{o_1, \ldots, o_n\} \to \texttt{String} \times \mathbb{N}$ assigning to each (top-level) object o_j in `init` a pair (ip, i), where ip is the IP address of the machine in which o_j resides, and i is a session number.

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).

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

$$\texttt{(to}\ o\ \texttt{from}\ o'\ \texttt{:}\ mc\texttt{)}\ \texttt{<}\ o : C \mid \ldots\ \texttt{>}\ \texttt{=>}\ \texttt{<}\ o : C \mid \ldots\ \texttt{>}\ msgs\ [\texttt{if}\ \ldots] \qquad (\dagger)$$

or

$$\texttt{<}\ o : C \mid \ldots\ \texttt{>}\ \texttt{=>}\ \texttt{<}\ o : C \mid \ldots\ \texttt{>}\ msgs\ [\texttt{if}\ \ldots] \qquad (\ddagger)$$

where $msgs$ is a term of sort `Configuration` which, applying the equations in the module, reduces to a multiset of *messages*

$$\texttt{(to}\ o_1\ \texttt{from}\ o\theta\ \texttt{:}\ mc_1\texttt{)} \quad \ldots \quad \texttt{(to}\ o_k\ \texttt{from}\ o\theta\ \texttt{:}\ mc_k\texttt{)}$$

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 $o\theta$.

3.1 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 non-local objects, as illustrated in Fig. 1.

This mediator object opens and maintains sockets for communication between 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 without using the mediator.

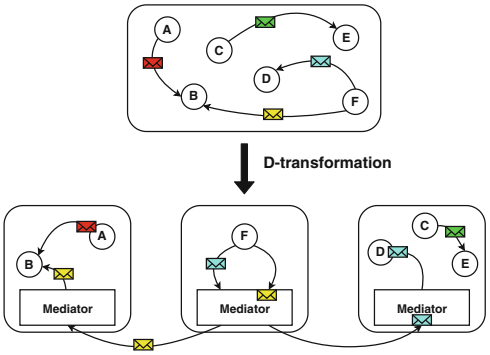

Fig. 1. Visualization of the D-Transformation

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 (i) establishes the required socket between the pair of objects if not already established; (ii) transforms the original message into a string with an "end-of-message" marker; and (iii) 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* as a map from `Oid` to `Pair{String,Nat}` using an equation `eq di = ....`
- The module *filter*(M), which transforms M as described below.
- Declarations and rewrite rules defining the mediator objects and their behaviors (which import the `SOCKET` module).

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 an object identifier `<` *ip* `;` *session* `>` for each mediator object, 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 rules of the form (‡)), where `filter` redirects the messages going to remote objects to the mediator and leaves the other messages unchanged[1]:

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

```
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 of class

```
class Med | sockets : Sockets,
            contacts : Contacts,
            bufferedMsgs : Configuration .
```

- `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/siliunobi/d-transformation for a complete specification of the mediator object, where most of the rewrite rules deal with establishing Maude sockets along the lines explained in [10, Chapter 11]. In this paper we just show the following two rewrite rules for the mediator.

```
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 applies when a configuration receives a message `received(S, SKT, DATA)`, denoting that a string `DATA` has been received through socket `SKT`. The mediator adds `DATA` to the string `STR` that it has buffered for socket `SKT`:

```
rl [receiveData] :
    received(S, SKT, DATA)
    < O : Med | sockets : SKTS [SKT, STR] >
  =>
    < O : Med | sockets : SKTS [SKT, STR + DATA] >
    receive(SKT, S) .
```

See our report [26] for the rule where the mediator extracts a message from a socket and adds it to the local configuration. Objects in the same Maude session communicate without going through sockets or mediators.

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`

3.2 Distributed Initial States

The initial state $\texttt{init}_{D_{di}}(ip, n)$ at Maude session (ip, n) is a configuration with:

- 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; and
- for each top-level (non-mediator) object *o* in the configuration, a message

  `createServerTcpSocket(socketManager,` *o*`,` *port#*`, 5)`

3.3 Communicating with Foreign Objects

A socket-based distributed Maude object system can easily be 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.

Suppose that `C` is a class of Maude objects that needs to communicate with foreign objects. All we need are three things: (a) a *signature* of messages sent by objects in `C` to such foreign object and by foreign objects to objects in `C`; (b) *rewrite rules* for the objects of class `C` specifying how messages to foreign objects are generated and how objects of class `C` react to messages sent by foreign objects; and (c) a *wrapper* encapsulating a foreign object that can transform the *string representation* of a message from a `C` 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 `C` object. In this work we have used the steps (a)–(c) to allow communication between a YCSB [12] foreign object and standard Maude objects to carry out system evaluations on realistic workloads.

3.4 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 [12] as explained in Sect. 3.3. Our tool also invokes the workload generator to initialize and to load data into the database, and 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.

4 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 initial 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 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, which is beyond the scope of this paper. Instead, we use the approach followed in other proofs of correctness of distributed systems obtained by transformation from formal specifications, e.g., [36,40], where network communication is abstracted away. Therefore, we present below a proof of correctness 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.

4.1 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 some socket was not established, is not 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 those in $M_{D_{di}}$ as follows:

– Since we abstract from the establishment of sockets, the rules in $M_{D_{di}}$ dealing with this issue (not shown in this paper) 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)) .
```

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. Rules `receiveData` 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 $\texttt{init}_{D_0}$.

4.2 $D_0(M, \texttt{init}, di)$ 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 their respective labeling functions $L \circ h$ and L.

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 2. *h is a stuttering bisimulation map*

$$h : \mathcal{K}(D_0(M, \mathit{init}, di), \mathit{init}_{D_0}) \rightarrow \mathcal{K}(M, \mathit{init})$$

with corresponding labeling functions $L \circ h$ and L.

The proof of Theorem 2 is given in our longer report [26]. The following main correctness-preservation result follows immediately from Theorems 1 and 2:

Theorem 3. *Given a rewrite theory M specifying a distributed system and an initial state init as described in Sect. 3, 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*

$$\mathcal{K}(M, \mathit{init}) \models \varphi \text{ if and only if } \mathcal{K}(D_0(M, \mathit{init}, di), \mathit{init}_{D_0}) \models \varphi$$

for the labeling function $L \circ h$ in $\mathcal{K}(D_0(M, \mathit{init}, di), \mathit{init}_{D_0})$.

5 Prototype and Experiments

We have implemented, in around 300 LOC, a "proof-of-concept" 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 lock-based distributed transaction protocol which has been implemented in C++ and evaluated in [16]; and (ii) the ROLA transaction system design. ROLA [22] is a new design 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 conventional distributed implementations of the same designs (if available) and with the model-based performance predictions obtained by statistical model checking of the original Maude designs?
Q2: How does the performance of a distributed Maude implementation automatically generated by our unoptimized prototype compare with that of a state-of-the-art distributed implementation in C++ of the same design?

Answers to Q1 cannot be an agreement between the performance *values predicted* by statistically 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 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. Therefore, the desired consistency between the performance predicted by statistically model checking a model and the performance obtained by experimentally evaluating an implementation is an agreement between predicted and measured *trends*: If, e.g., throughput increases as a function of the proportion of read transactions, then consistency means that it should do so along curves that are *similar* up to a change of scale.

5.1 Experimental Setup

Implementation-Based Evaluation. We evaluated the two case studies using the Yahoo! Cloud Serving Benchmark (YCSB) [12], which is the open standard for performance evaluation of data stores. We used the built-in C++ implementation of YCSB in [16] in our first case study. For ROLA, we used a variant of the original Java implementation of YCSB adapted to transaction systems [3]. We deployed the two systems on a cluster of d430 Emulab machines, with ping time between machines approximately 0.13 ms. In both cases, we considered 5 partitions (of the database) on 5 machines, and client processes split across another 5 separate machines; we considered the same mix of read-only, write-only, and read-write transactions, with each transaction accessing up to 8 keys. We used Zipfian distribution for key accesses with parametric skew factor *theta*.

Statistical Model Checking (SMC). By running Monte-Carlo simulations from a given initial state, SMC 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 state. To mimic the real-world network environment, we used lognormal distribution for message delays [5]. 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 the CAT tool [25] for model checking consistency properties of our Maude models. The analysis was performed with all initial states up to 4 transactions, 2 keys, 2 clients, and 2 servers.

5.2 Case Study I: Lock-Based Distributed Transactions

NO_WAIT [13] is a strict two-phase-locking-based distributed transaction system with two-phase commit (2PC) as its atomic commitment protocol, and has been implemented in the Deneva framework [16] using C++. We formally specified NO_WAIT in Maude, and then automatically generated the corresponding distributed Maude implementation. We used the C++ implementation in [16] in our experiments with NO_WAIT. Our Maude model of NO_WAIT has around 600 LOC, whereas the C++ implementation in [16] has approximately 12K LOC.

We performed two sets of experiments (Lock_A and Lock_B in Fig. 2), focusing on the effect of varying the contention in the system. For each set of experiments, we plot the results of statistical model checking of our Maude model, and of measurements of the distributed Maude and C++ implementations.

Regarding Q1, in Lock_A we vary the contention by tuning the skew *theta*, and compare two workloads, with 50% and 100% update transactions. 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 (number of operations in a transaction). 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 agrees with statistical predictions, but also with state-of-the-art implementation-based results.

Regarding Q2, our distributed system achieves lower peak throughput, by a factor of 6, than the C++ implementation. Some reasons for this lower performance are: (i) our tool is an unoptimized prototype, whereas the C++ implementation of NO_WAIT is optimized for 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*, which is flexible but adds an extra layer of communication, whereas YCSB and the protocol clients are directly integrated in the C++ implementation.

We have also used the CAT tool [25] to model check our Maude model of NO_WAIT against 6 consistency properties, without finding any violation. If our trusted code base executes correctly, Theorem 3 ensures that our distributed Maude implementation of NO_WAIT satisfies the same consistency properties for the corresponding initial states.

5.3 Case Study II: The ROLA Transaction System

ROLA [22] is a recent distributed transaction protocol design that guarantees read atomicity (RA) and prevents lost updates (PLU). In [22], ROLA was formalized in Maude, model checked for the above consistency properties, and statistical model checking performance estimation showed that ROLA outperforms well-known distributed transaction system designs guaranteeing RA and PLU. However, up to now there was no distributed implementation of ROLA. Using our tool and the Maude specification of ROLA in [22] (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. 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 consistency properties model checked in [22] are preserved (Theorem 3) assuming correct execution of the trusted code base.

All system models, property specifications, and distributed Maude implementations are available at https://github.com/siliunobi/d-transformation.

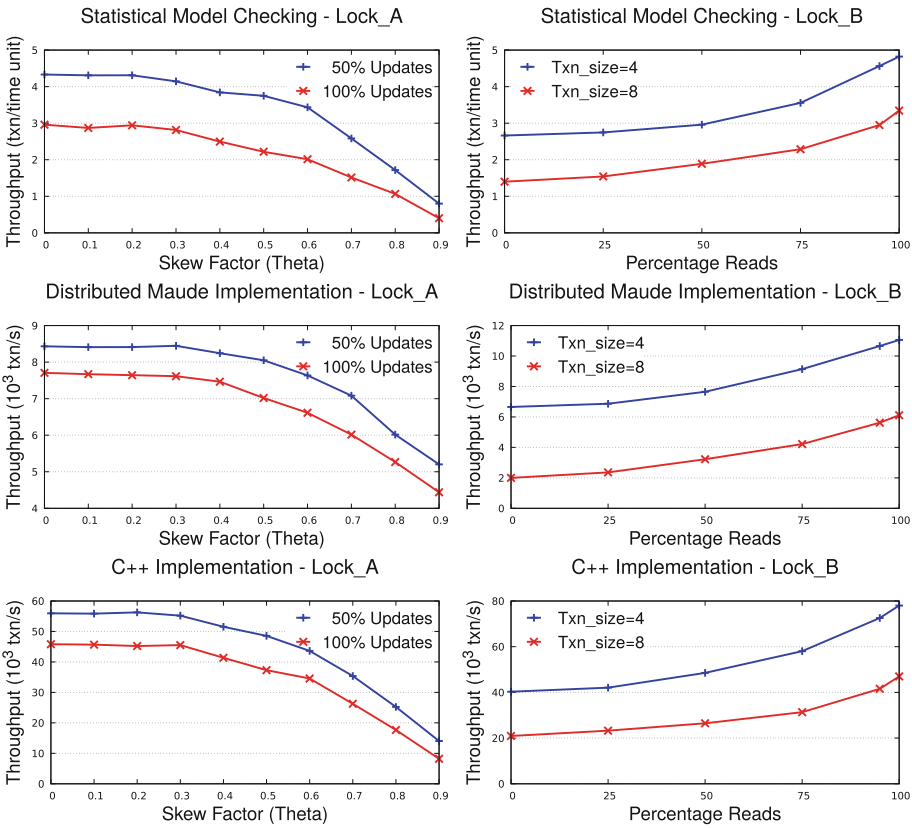

Fig. 2. NO_WAIT: Throughput obtained from statistical model checking (top), distributed Maude implementation (middle), and C++ implementation (bottom). Experiments Lock_A (left) and Lock_B (right) measure throughput for different ratios of updates and transaction sizes when varying skew factors and ratios of reads, resp.

6 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* [41] 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. Generating Imperative Implementation from Formal Models. Formal frameworks such as those in, e.g., [14,15,39], offer the possibility of generating distributed Java or C implementations from formal models.

2. Specification, Verification, and Proof of Imperative Implementation. A good example of state-of-the art recent work in this category is the IronFleet frame-

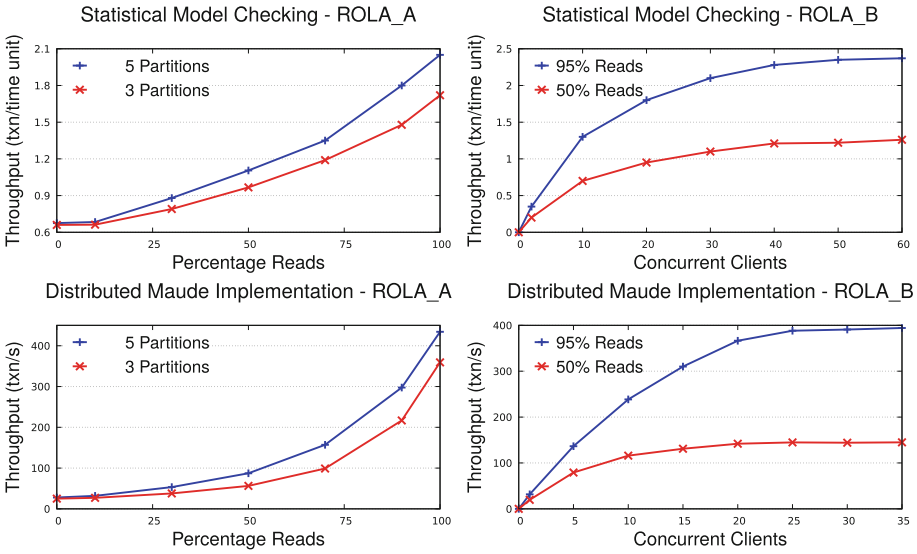

Fig. 3. ROLA: Comparison between statistical model checking (top) and distributed Maude implementation (bottom). Experiments ROLA_A (left) and ROLA_B (right) measure throughput for different number of partitions and different ratios of reads when varying ratios of reads and concurrent clients, respectively.

work [17]. Distributed systems are specified in a mixture of Lamport's TLA and Hoare logic assertions for imperative sequential code in Leino's Dafny language [20]. They are then formally verified with various tools, including Z3 [31] and the Dafny prover. Dafny code is then compiled 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 Nuprl [11] and Coq [6]. 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 [40] begins with a distributed system design and a set of safety properties, both specified in Coq; 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 [21] 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 [36] specifies both distributed system designs and their desired properties in Coq, uses Coq to prove the desired properties, and extracts correct-by construction OCaml code.

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, whereas Maude does so through PVeStA. In con-

trast to related work in category (1), where the correctness of the generated Java or C code is not proved (e.g., [15]), we prove the correctness of the generated distributed implementation. A possible exception is the effort in [14,39] which "argues the correctness" of their compilation from I/O automata to Java by modeling the compiled code as I/O automata. They also assume correctness of data type implementations, and only claim preservation of safety properties, whereas we also prove preservation of liveness properties. 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), our work within the Maude framework shares with them the possibility of generating correct-by-construction distributed implementations from designs and of verifying such designs using theorem proving [34,37], but also adds the possibility of rapid exploration of different design alternatives by testing and by automatic model checking analysis, and the prediction of system performance before implementation. The point is that beginning with a human-intensive theorem proving verification effort may be both premature and costly. Instead, in Maude, designs can be analyzed and improved by fully automated methods *before* a mature design is fully verified by theorem proving.

7 Conclusions

We have presented and implemented a "proof-of-concept" prototype of the D transformation taking a Maude model M of a distributed system design and automatically generating the distributed Maude implementation $D(M)$. We have 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 applied our method to automatically obtain distributed implementations of two state-of-the-art distributed transaction system designs—and have executed them on YCSB workloads. We have also compared the performance of $D(M)$ and a high-performance conventional C++ implementation, which outperforms our prototype by a factor of six. 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.

Our Maude implementation of the $M \mapsto D(M)$ transformation is a proof-of-concept prototype with ample room for improvement. The obvious next step is to arrive at an efficient Maude implementation of the $M \mapsto D(M)$ transformation.

Acknowledgments. We thank the anonymous reviewers for helpful comments on a previous version of this paper. This work has been partially supported by NRL under contract N00173-17-1-G002, and the National Science Foundation under grant NSF CCF 16-17401.

References

1. AlTurki, M., Meseguer, J.: PVeStA: a parallel statistical model checking and quantitative analysis tool. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 386–392. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_28
2. Bae, K., Meseguer, J.: Model checking linear temporal logic of rewriting formulas under localized fairness. Sci. Comput. Program. **99**, 193–234 (2015)
3. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Scalable atomic visibility with RAMP transactions. ACM Trans. Database Syst. **41**(3), 15:1–15:45 (2016)
4. Baker, J., et al.: Megastore: providing scalable, highly available storage for interactive services. In: CIDR 2011, pp. 223–234 (2011)
5. Benson, T., Akella, A., Maltz, D.A.: Network traffic characteristics of data centers in the wild. In: IMC 2010, pp. 267–280. ACM (2010)
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
7. Bobba, R., et al.: Survivability: design, formal modeling, and validation of cloud storage systems using Maude. In: Assured Cloud Computing, chap. 2, pp. 10–48. Wiley-IEEE Computer Society Press (2018)
8. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. **360**(1–3), 386–414 (2006)
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2001)
10. Clavel, M., et al.: All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
11. Constable, R.L.: Implementing Mathematics with the Nuprl Proof Development System. Prentice Hall, Englewood Cliffs (1987)
12. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SOCC 2010, pp. 143–154. ACM (2010)
13. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM **19**(11), 624–633 (1976)
14. Georgiou, C., Lynch, N.A., Mavrommatis, P., Tauber, J.A.: Automated implementation of complex distributed algorithms specified in the IOA language. STTT **11**(2), 153–171 (2009)
15. Haberl, W.: Code generation and system integration of distributed automotive applications. Ph.D. thesis, Technical University Munich (2011)
16. Harding, R., Van Aken, D., Pavlo, A., Stonebraker, M.: An evaluation of distributed concurrency control. Proc. VLDB Endow. **10**(5), 553–564 (2017)
17. Hawblitzel, C., et al.: IronFleet: proving safety and liveness of practical distributed systems. Commun. ACM **60**(7), 83–92 (2017)
18. Hewitt, E.: Cassandra: The Definitive Guide. O'Reilly Media, Sebastopol (2010)
19. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: USENIX ATC 2010. USENIX Association (2010)
20. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
21. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: certified causally consistent distributed key-value stores. In: POPL 2016, pp. 357–370. ACM (2016)

22. Liu, S., Ölveczky, P.C., Santhanam, K., Wang, Q., Gupta, I., Meseguer, J.: ROLA: a new distributed transaction protocol and its formal analysis. In: Russo, A., Schürr, A. (eds.) FASE 2018. LNCS, vol. 10802, pp. 77–93. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89363-1_5
23. Liu, S., Ölveczky, P.C., Wang, Q., Gupta, I., Meseguer, J.: Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. Formal Asp. Comput. **31**(5), 503–540 (2019)
24. Liu, S., Ölveczky, P.C., Wang, Q., Meseguer, J.: Formal modeling and analysis of the Walter transactional data store. In: Rusu, V. (ed.) WRLA 2018. LNCS, vol. 11152, pp. 136–152. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99840-4_8
25. Liu, S., Ölveczky, P.C., Zhang, M., Wang, Q., Meseguer, J.: Automatic analysis of consistency properties of distributed transaction systems in Maude. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 40–57. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_3
26. Liu, S., Sandur, A., Meseguer, J., Ölveczky, P.C., Wang, Q.: Generating correct-by-construction distributed implementations from formal Maude designs. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign (2019). http://hdl.handle.net/2142/106018
27. Manolios, P.: A compositional theory of refinement for branching time. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 304–318. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39724-3_28
28. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992)
29. Meseguer, J.: Twenty years of rewriting logic. J. Algebr. Log. Program. **81**, 721–781 (2012)
30. Meseguer, J., Palomino, M., Martí-Oliet, N.: Algebraic simulations. J. Log. Algebr. Program. **79**(2), 103–143 (2010)
31. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
32. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Commun. ACM **58**(4), 66–73 (2015)
33. Ölveczky, P.C.: Formalizing and validating the P-Store replicated data store in Maude. In: James, P., Roggenbach, M. (eds.) WADT 2016. LNCS, vol. 10644, pp. 189–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72044-9_13
34. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 314–328. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_22
35. Schiper, N., Sutra, P., Pedone, F.: P-store: genuine partial replication in wide area networks. In: SRDS 2010, pp. 214–224. IEEE Computer Society (2010)
36. Sergey, I., Wilcox, J.R., Tatlock, Z.: Programming and proving with distributed protocols. PACMPL **2**(POPL), 28:1–28:30 (2018)
37. Skeirik, S., Stefanescu, A., Meseguer, J.: A constructor-based reachability logic for rewrite theories. In: Fioravanti, F., Gallagher, J.P. (eds.) LOPSTR 2017. LNCS, vol. 10855, pp. 201–217. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94460-9_12
38. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP 2011, pp. 385–400. ACM (2011)

39. Tauber, J.A.: Verifiable compilation of I/O automata without global synchronization. Ph.D. thesis, Massachusetts Institute of Technology (2005)
40. Wilcox, J.R., et al.: Verdi: a framework for implementing and formally verifying distributed systems. In: PLDI 2015, pp. 357–368. ACM (2015)
41. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.E.: Planning for change in a formal verification of the Raft consensus protocol. In: CPP 2016, pp. 154–165. ACM (2016)