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] Department of Computer Science, ETH Zürich, Zürich, Switzerland
[2] University of Illinois, Urbana-Champaign, USA
[3] University of Oslo, Oslo, Norway

Abstract. Developing a highly reliable distributed system meeting desired performance requirements is at present a hard and very labor-intensive task. Formal specification of a system design and formal verification and analysis can yield provably correct designs as well as reliable performance predictions. But there is still a *formality gap* between verified designs and distributed implementations. We present a correct-by-construction automatic transformation mapping a verified formal specification of a system design M in Maude to a distributed implementation $D(M)$ with the same safety and liveness properties as M. 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 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 for which similar performance trends can be shown.

1 Introduction

Designing and implementing correct high-performance distributed systems is a complex task. 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 critical system requirements.

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 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 [32]; (ii) the correctness properties that these systems must satisfy can be quite complex, and there is a desire in industry for *automatic* verification techniques [32]; and (iii) both *correctness* and *performance* are, as mentioned,

crucial requirements; a correct design that performs worse than similar designs is worthless.

Formal Design and Verification of Distributed Systems in Maude. The above criteria (i)–(iii) are challenging. One formal framework that has shown promise in meeting them is Maude [12], which is a high-performance language and formal framework for executable specification, verification and programming of concurrent systems based on rewriting logic [28,9,29]. 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 [12,3], which allows us to express and analyze complex properties (see, e.g., [25]). The Maude tool environment also provides theorem proving verification of invariants in the InvA tool [34], and of reachability logic properties in Maude's reachability logic prover [37]. For challenge (iii), the Maude tool environment includes the PVeStA [2] *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 [29]). 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 [18], ZooKeeper [19], Google's Megastore [5], P-Store [35], RAMP [4], and Walter [38]; and to design the entirely new system ROLA [22] (see the survey [8] and [33,24,23]). Furthermore, model-based performance predictions using PVeStA have shown good correspondence with experimental evaluation of implementations of systems such as 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.

From Distributed System Designs to Implementations. Maude provides TCP/IP sockets as Maude external objects so that it can interact with standard Maude objects by message passing [12], 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 [12] 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, [12] and [1].

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. The main goal of this paper 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* [30,27] 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 $(M, init, di)$, 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*[4] where each object in *init* will be located.

Prototype and Experimental Evaluation. We have developed a Maude prototype that automates the $M \mapsto D(M)$ transformation. To evaluate its effectiveness —as well as the predictive power of the statistical-model-checking-based performance predictions for a system design M when compared with performance measures experimentally obtained for $D(M)$— we have used two case studies. In the first one we compare the Maude specification M of the NO_WAIT transaction protocol with its distributed Maude implementation $D(M)$ and with a state-of-the-art conventional implementation of NO_WAIT. In the second case study we compare the Maude design M of the new distributed transaction system ROLA [22] with its *the first time ever* distributed implementation $D(M)$ and measure its performance.

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 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 *foreign objects* (see Section 3.3) such as YCSB [14]; (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 implementation: our unoptimized prototype $D(M)$

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

is only six times slower than a high-performance 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 [12] is a rewriting-logic-based high-performance executable formal specification language and analysis tool for object-based distributed systems. Formal analysis methods include: simulation, reachability analysis, LTL model checking, theorem proving, and, for performance estimation purposes, statistical model checking with the PVeStA tool [2].

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*, with E a set of possibly 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. $(\Sigma, E \cup B)$ specifies the system's states as members of the *initial algebra* $T_{\Sigma/E \cup B}$.
- 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 [12] for more details. Operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s. They can have user-definable syntax, with underbars '`_`' marking the argument positions, and equational *attributes*, such as `assoc` and `comm`, stating that the operator is associative and commutative. Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`. The mathematical variables in such statements are declared with the keywords `var` and `vars`, or can have the form *var*`:`*sort* and be introduced on the fly. Maude also provides standard parameterized data types (sets, maps, etc.) that can be instantiated (and renamed).

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 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 by 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.

Example 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. 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:

```
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 .
```

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 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 `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*`)`.

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 $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, a_0, L_{\mathcal{A}})$ where A is a set of *states*, $\rightarrow_{\mathcal{A}} \subseteq A \times A$ is the *total transition relation on states* (total means that $\forall a \in A \; \exists a' \in A$ s.t. $a \rightarrow_{\mathcal{A}} a'$), $a_0 \in A$ is the *initial state*, and $L_{\mathcal{A}}$, called the *labeling function*, is a function $L_{\mathcal{A}} : A \rightarrow \mathcal{P}(AP)$ 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 $\forall n \in \mathbb{N} \; \pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$.

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 paper, $\mathcal{B}$ 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 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 $\kappa : \mathbb{N} \to \mathbb{N}$ 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 [11]). That is, we have:

Theorem 1. *[30], 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 1 is conceptually appealing but hard to check directly. As explained in [30], a more easily checkable characterization by Manolios [27] can be adapted to our setting as the following theorem:

Theorem 2. *(Adapted from [27,30]). 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 [12]: (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 $\models$.

3 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

  $\texttt{<}\ o_1\ \texttt{:}\ C_1\ \texttt{|}\ atts_1\ \texttt{>}\ \ \ldots\ \ \texttt{<}\ o_n\ \texttt{:}\ C_n\ \texttt{|}\ atts_n\ \texttt{>}$

  in $\mathcal{T}_{M,\texttt{Configuration}}$, with distinct object 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[5] 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

$$\lambda M \in \mathit{OModule}\ .\ \lambda \texttt{init} \in \mathcal{T}_{M,\texttt{Configuration}}\ \cdot \\ \lambda di \in [\mathit{oids}(\texttt{init}) \to \texttt{String} \times \mathbb{N})]\ .\ D(M, \texttt{init}, di) \in \mathit{OModule}.$$

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

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

or

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

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

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

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 $o\theta$. Although no "top-level" objects are created or deleted by the (†), (‡) rules, object-creating rules can also be added.[6]

[5] 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.

[6] 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.

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 objects that are not local, as illustrated in Fig. 1.

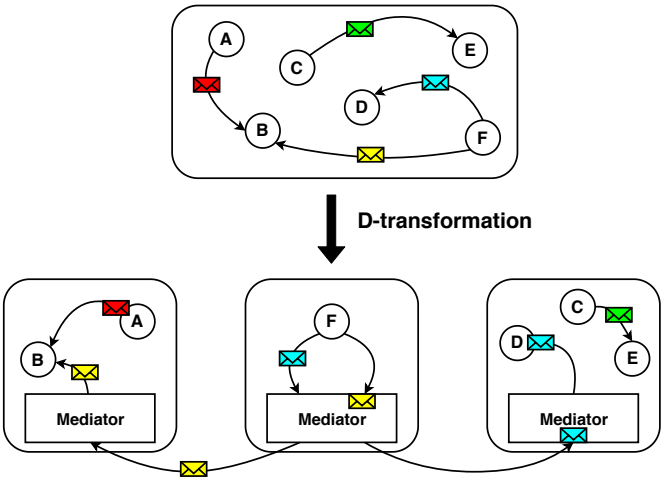

Fig. 1. Visualization of the D Transformation

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:

- 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 2. If the objects in state `init` in Example 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 |-> <"ip1";1>,   c2 |-> <"ip2";1>,
        db1|-> <"ip3";1>,   db2|-> <"ip4";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[7]:

```
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 .
```

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

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 `github.com/siliunobi/d-transformation` 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 [12, Chapter 11]. In this paper we just show the following three 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 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 [receiveData] :
    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 $\mathit{filter}(M)$ and the mediator specification:

```
mod M_Ddi 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 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; and
- one message

  `createServerTcpSocket(socketManager,` o`,` $port\#$`, 5)`

  for each top-level (non-mediator) object o in the configuration.

3.3 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 paper 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 3.1, 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 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 5, in this paper we have used some of the steps (a)–(c) to allow communication of a YCSB [14] 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 foreign objects. Furthermore, the D transformation explained in Section 3.4 can be easily extended to initial configurations where some of their objects are foreign objects.

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 [14] as explained in Section 3.3. 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.

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 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 paper. Instead, we adopt here the approach followed in other proofs of correctness of distributed systems obtained by transformation from formal specifications, e.g., [40,36], 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.

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 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 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)) .

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 `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 $\texttt{init}_{D_0}$. (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 $D_0(M, \texttt{init}, di)$).

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 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 $h : \mathit{Reach}(\texttt{init}_{D_0}) \to \mathit{Reach}(\texttt{init})$ 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 3. *h is a stuttering bisimulation map*

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

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

Proof. According to Theorem 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 4.1, 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

m `<` o `:` C `|` $atts$ `>  =>  <` o `:` C `|` $atts'$ `>  filter(`$msgs$`)` [`if` ...]

(the case with rules (‡) is easier), then there is a substitution θ such that $t = c_0\ m\theta\ (\texttt{<}\ o : C \mid atts\ \texttt{>})\theta$ and $t' = c_0\ (\texttt{<}\ o : C \mid atts'\ \texttt{>})\theta)\ \texttt{filter}(msgs\theta)$. Since $h(\texttt{filter}(msgs\theta)) = msgs\theta$, and $h(t) = h(c_0)\ m\theta\ (\texttt{<}\ o : C \mid atts\ \texttt{>})\theta$, 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 $(MsgConf, >_{mul})$ 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 $\texttt{(to}\ o\ \texttt{transfer}\ mc\ \texttt{from}\ o'\ \texttt{to}\ o''\texttt{)} > \texttt{transfer(}o'''\texttt{,}o\texttt{,}\ \texttt{msg2string(to}\ o''\ \texttt{from}\ o'\ :\ mc\texttt{))} > \texttt{(to}\ o''\ \texttt{from}\ o'\ :\ mc\texttt{)}$. 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 (‡) is used. We show the case for rule (†); the (‡) case is trivial. For a (†) rule, u must have the form $u = h(c_0)\ h(m')\ (\texttt{<}\ o : C \mid atts\ \texttt{>})\theta$ with $h(m') = m\theta$ and $u' = h(c_0)\ (\texttt{<}\ o : C \mid atts'\ \texttt{>})\theta\ msgs\theta$. We can distinguish two cases: (i) if $m' = m\theta$ then $t = c_0\ m\theta\ (\texttt{<}\ o : C \mid atts\ \texttt{>})\theta$ and can be rewritten to $t' = c_0\ (\texttt{<}\ o : C \mid atts'\ \texttt{>})\theta\ \texttt{filter}(msgs\theta)$, so that $h(t') = u'$, as desired. Otherwise, $m\theta$ must be of the form $\texttt{(to}\ b\ \texttt{from}\ a\ :\ x\texttt{)}$, with $o\theta = b$, and m' is either (i) $\texttt{(to}\ di(a)\ \texttt{transfer}\ x\ \texttt{from}\ a\ \texttt{to}\ b\texttt{)}$, or (ii) $\texttt{transfer(}di(b)\texttt{,}di(a)\texttt{,}\ \texttt{msg2string(to}\ b\ \texttt{from}\ a : x\texttt{))}$. We do case (i) and leave case (ii) (requiring fewer steps) to the reader. The c_0 has the form $c_0 = \texttt{<}\ di(a) : \texttt{Med | >}\ \texttt{<}\ di(b) : \texttt{Med | >}\ c_0'$ and we have rewrites $t \longrightarrow t_1 = \texttt{<}\ di(a) : \texttt{Med | >}\ \texttt{transfer(}di(b)\texttt{,}\ di(a)\texttt{,}\ \texttt{msg2string(to}\ b\ \texttt{from}\ a\ :\ \texttt{x))}$ $\texttt{<}\ di(b) : \texttt{Med | >}\ \texttt{<}\ b : C \mid atts\theta\ \texttt{>}\ c_0' \longrightarrow t_2 = \texttt{<}\ di(a) : \texttt{Med | >}\ \texttt{(to}\ b\ \texttt{from}\ a\ :\ \texttt{x)}\ \texttt{<}\ di(b) : \texttt{Med | >}\ \texttt{<}\ b : C \mid atts\theta\ \texttt{>}\ c_0'$ $\longrightarrow\ t_3\ =\ \texttt{<}\ di(a) : \texttt{Med | >}\ \texttt{<}\ di(b) : \texttt{Med | >}\ \texttt{<}\ b : C \mid atts'\theta\ \texttt{>}\ \texttt{filter}(msgs\theta)\ c_0'$. 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 1 and 3:

Theorem 4. *Given a rewrite theory M specifying a distributed system and an initial state* ***init*** *as described in Section 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*

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

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

5 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 [16]; and (ii) the ROLA transaction system design. ROLA [22] is a new design with attractive features 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.

5.1 Experimental Setup

Implementation-Based Evaluation We have evaluated the two case studies using the Yahoo! Cloud Serving Benchmark (YCSB) [14], which is the open standard

for comparative 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 [4]. We deployed the two case studies on a cluster of d430 Emulab machines [39], 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 [6]. 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 integrated our Maude models into the CAT framework [25] for model checking consistency properties of distributed transaction systems. The analysis was performed with exhaustively generated initial states with 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.

5.2 Lock-Based Distributed Transactions

This case study considers the protocol NO_WAIT implemented in the Deneva framework [16] 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.

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 [16] in our experiments with NO_WAIT in [16]. Our Maude model of NO_WAIT is 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 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 of 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 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 [25] 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 assumption that our trusted code base executes correctly, Theorem 4 ensures that our distributed Maude implementation of NO_WAIT satisfies the same consistency properties for the corresponding initial states.

5.3 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 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

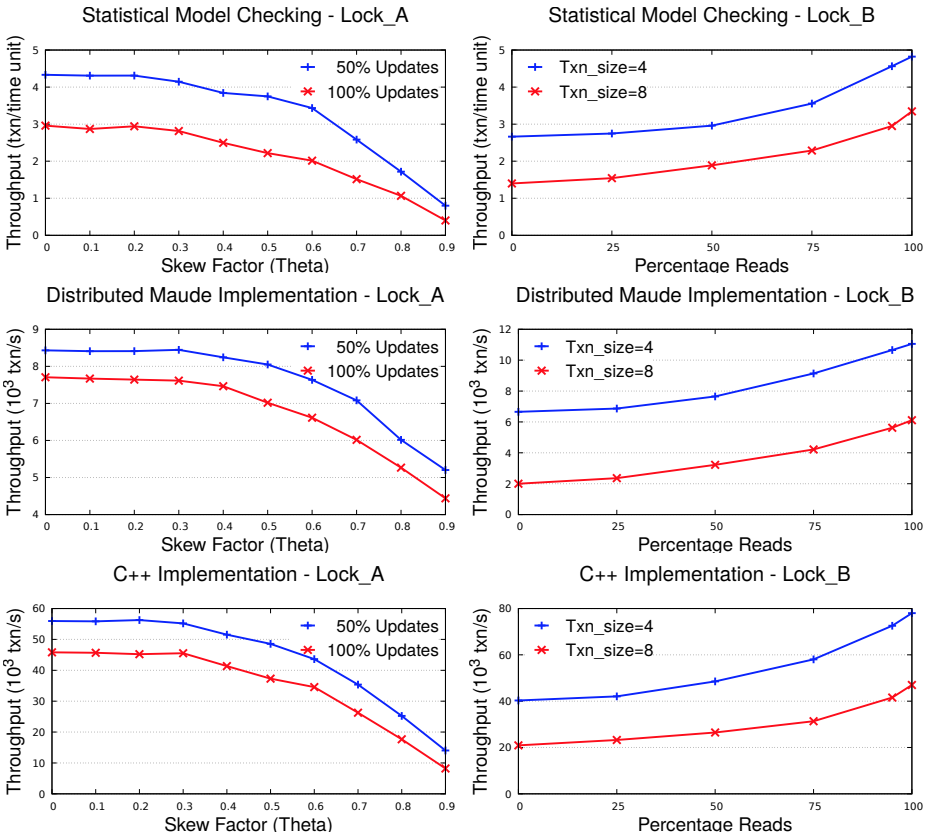

Fig. 2. NO_WAIT: Throughput comparison between statistical model checking (top), distributed Maude implementation (middle), and C++ implementation (bottom). Experiments Lock_A (left) and Lock_B (right) measure throughput of different ratios of updates and transaction sizes when varying skew factors and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

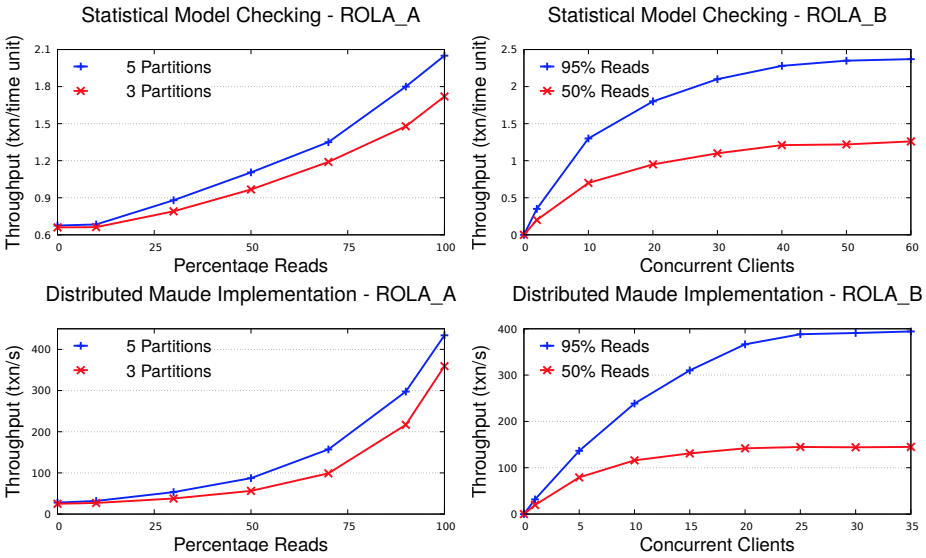

Fig. 3. ROLA: Comparison between statistical model checking (top) and distributed Maude implementation (bottom). Experiments ROLA_A (left) and ROLA_B (right) measure throughput for different number of partitions and different ratios of reads when varying ratios of reads and concurrent clients, respectively.

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 4) assuming correct execution of the trusted code base.

Data Availability. The system models, properties specifications, and distributed Maude implementations are available at `github.com/siliunobi/d-transformation`.

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. Specification, Verification, and Compilation to Imperative Implementation. The IOA formal framework [26,15] 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 [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 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 [13] and Coq [7] 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 [40] 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 [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 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,[8] whereas Maude does so through the PVeSta tool [2]. 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-

[8] Probabilistic system behaviors can be specified using probabilistic IOA [10]. However, we are not aware of tools supporting statistical model checking analysis of performance properties for distributed system designs in the IOA framework.

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 [34] and reachability logic properties [37] 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.

7 Conclusions

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.

The current Maude implementation of the $M \mapsto D(M)$ transformation is an unoptimized prototype with ample room for improvement. The next obvious step is to arrive at a mature Maude implementation of the $M \mapsto D(M)$ transformation.

References

1. AlTurki, M., Meseguer, J.: Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In: RTRTS'10. EPTCS, vol. 36, pp. 26–45 (2010)
2. AlTurki, M., Meseguer, J.: PVeStA: A parallel statistical model checking and quantitative analysis tool. In: CALCO'11. LNCS, vol. 6859, pp. 386–392. Springer (2011)
3. Bae, K., Meseguer, J.: Model checking linear temporal logic of rewriting formulas under localized fairness. Sci. Comput. Program. 99, 193–234 (2015)

4. 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)
5. Baker, J., Bond, C., Corbett, J.C., Furman, J.J., Khorlin, A., Larson, J., Leon, J., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: CIDR'11. pp. 223–234 (2011)
6. Benson, T., Akella, A., Maltz, D.A.: Network traffic characteristics of data centers in the wild. In: IMC'10. pp. 267–280. ACM (2010)
7. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
8. Bobba, R., Grov, J., Gupta, I., Liu, S., Meseguer, J., Ölveczky, P.C., Skeirik, S.: 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)
9. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theor. Comput. Sci. 360(1-3), 386–414 (2006)
10. Canetti, R., Cheung, L., Kaynar, D.K., Liskov, M., Lynch, N.A., Pereira, O., Segala, R.: Task-structured probabilistic I/O automata. J. Comput. Syst. Sci. 94, 63–97 (2018)
11. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2001)
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: All About Maude, LNCS, vol. 4350. Springer (2007)
13. Constable, R.L.: Implementing Mathematics with the Nuprl Proof Development System. Prentice Hall (1987)
14. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SOCC'10. pp. 143–154. ACM (2010)
15. 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)
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., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., Zill, B.: 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 (2010)
19. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX ATC'10. USENIX Association (2010)
20. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR'10. LNCS, vol. 6355, pp. 348–370. Springer (2010)
21. Lesani, M., Bell, C.J., Chlipala, A.: Chapar: certified causally consistent distributed key-value stores. In: POPL'16. 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: FASE. LNCS, vol. 10802, pp. 77–93. Springer (2018)
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 Aspects of Computing (2019)
24. Liu, S., Ölveczky, P.C., Wang, Q., Meseguer, J.: Formal modeling and analysis of the Walter transactional data store. In: WRLA. LNCS, vol. 11152, pp. 136–152. Springer (2018)

25. Liu, S., Ölveczky, P.C., Zhang, M., Wang, Q., Meseguer, J.: Automatic analysis of consistency properties of distributed transaction systems in Maude. In: TACAS'19. LNCS, vol. 11428, pp. 40–57. Springer (2019)
26. Lynch, N.: Distributed Algorithms. Morgan Kaufmann (1996)
27. Manolios, P.: A compositional theory of refinement for branching time. In: CHARME'03. LNCS, vol. 2860, pp. 304–318. Springer (2003)
28. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
29. Meseguer, J.: Twenty years of rewriting logic. J. Algebraic and Logic Programming 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.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008)
32. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Communications of the ACM 58(4), 66–73 (April 2015)
33. Ölveczky, P.C.: Formalizing and validating the p-store replicated data store in maude. In: WADT'16. LNCS, vol. 10644, pp. 189–207. Springer (2016)
34. Rocha, C., Meseguer, J.: Proving safety properties of rewrite theories. In: CALCO'11. LNCS, vol. 6859, pp. 314–328. Springer (2011)
35. Schiper, N., Sutra, P., Pedone, F.: P-store: Genuine partial replication in wide area networks. In: SRDS'10. 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: LOPSTR. LNCS, vol. 10855, pp. 201–217. Springer (2017)
38. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: SOSP'11. pp. 385–400. ACM (2011)
39. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: OSDI. USENIX Association (2002)
40. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: A framework for implementing and formally verifying distributed systems. In: PLDI'15. 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'16. pp. 154–165. ACM (2016)