Bridging the Semantic Gap between Qualitative and Quantitative Models of Distributed Systems

SI LIU, ETH Zürich, Switzerland
JOSÉ MESEGUER, University of Illinois Urbana-Champaign, USA
PETER CSABA ÖLVECZKY, University of Oslo, Norway
MIN ZHANG, East China Normal University, China
DAVID BASIN, ETH Zürich, Switzerland

Today's distributed systems must satisfy both *qualitative* and *quantitative* properties. These properties are analyzed using very different formal frameworks: expressive untimed and non-probabilistic frameworks, such as TLA+ and Hoare/separation logics, for qualitative properties; and timed/probabilistic-automaton-based ones, such as Uppaal and Prism, for quantitative ones. This requires developing two quite different models of the same system, without guarantees of semantic consistency between them. Furthermore, it is very hard or impossible to *represent* intrinsic features of distributed object systems—such as unbounded data structures, dynamic object creation, and an unbounded number of messages—using finite automata.

In this paper we bridge this semantic gap, overcome the problem of manually having to develop two different models of a system, and solve the representation problem by: (i) defining a transformation from a very general class of distributed systems (a generalization of Agha's actor model) that maps an untimed non-probabilistic distributed system model suitable for qualitative analysis to a probabilistic timed model suitable for quantitative analysis; and (ii) proving the two models semantically consistent. We formalize our models in rewriting logic, and can therefore use the Maude tool to analyze qualitative properties, and statistical model checking with PVeStA to analyze quantitative properties. We have automated this transformation and integrated it, together with the PVeStA statistical model checker, into the *Actors2PMaude* tool. We illustrate the expressiveness of our framework and our tool's ease of use by automatically transforming untimed, qualitative models of numerous distributed system designs—including an industrial data store and a state-of-the-art transaction system—into quantitative models to analyze and compare the performance of different designs.

CCS Concepts: • **Software and its engineering → Formal methods**; **Model checking**; **Software performance**; • **Computing methodologies → Model verification and validation**.

Additional Key Words and Phrases: distributed systems, actors, formal model transformation, statistical model checking, rewriting logic, Maude

ACM Reference Format:
Si Liu, José Meseguer, Peter Csaba Ölveczky, Min Zhang, and David Basin. 2022. Bridging the Semantic Gap between Qualitative and Quantitative Models of Distributed Systems. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 136 (October 2022), 30 pages. https://doi.org/10.1145/3563299

1 INTRODUCTION

Problem Description. Virtually all distributed systems—from network protocols to distributed algorithms, and from cloud-based transaction systems to distributed cyber-physical systems—can

Authors' addresses: Si Liu, ETH Zürich, Switzerland; José Meseguer, University of Illinois Urbana-Champaign, USA; Peter Csaba Ölveczky, University of Oslo, Norway; Min Zhang, East China Normal University, China; David Basin, ETH Zürich, Switzerland.

2475-1421/2022/10-ART136
https://doi.org/10.1145/3563299

be naturally modeled and programmed as systems of *concurrent objects* that communicate through message passing. In distributed systems, logical correctness is necessary, but not sufficient, since *quantitative properties*, including *performance* properties, are equally important. The need for model-based analysis of both correctness and performance of distributed systems has been emphasized in academia and industry [Alur et al. 2015; Microsoft 2018; Newcombe et al. 2015]. Having a *unified way* of formally specifying and analyzing *both* qualitative and quantitative properties during distributed system design, and *automating* system analysis as much as possible are the problems we address in this work. By a "unified way" we mean avoiding a *modeling schizophrenia*, where very different, perhaps inconsistent, models are developed to analyze qualitative and quantitative properties. Avoiding such modeling schizophrenia also facilitates asking important *mixed property questions*, such as the following: "We know that the Cassandra data storage system only satisfies eventual consistency. But *how often* does it maintain stronger consistency properties in practice?"

Challenges of Model Heterogeneity for Distributed Systems. Excellent methods and tools exist for analyzing both qualitative and quantitative properties. However, many of the best-known quantitative analysis tools are based on finite automata models enriched with probability information, and perhaps time. The problem is that object-based distributed systems have intrinsic features that are quite hard or impossible to represent in such models. For example: (i) object attributes may contain unbounded data structures; (ii) asynchronous message passing and dynamic object creation may increase the number of both messages and objects in an unbounded manner; and (iii) as we explain in this paper, the probability distributions suitable to model their behavior may not be fixed ones, but *parametric* families of *user-specified* distributions, whose parameter values may change *dynamically*. Issues like these make bridging the semantic gap between the different models used for qualitative and quantitative analysis highly non-trivial and practically important.

Our Approach. To avoid the schizophrenia of heterogeneous distributed system models and support both qualitative and quantitive analysis, we define automatable semantics-preserving transformations that turn a nondeterministic untimed system model suitable for qualitative analysis into a probabilistic timed model suitable for analyzing quantitative properties. We target a broad class of distributed systems: our qualitative models are *generalized actor systems*, which extend the already very large class of Agha's message-passing *actor systems* [Agha 1986] by allowing "active actors" that can change their state without receiving a message. Since performance properties intrinsically involve time, our main transformation *adds time* to a generalized actor model as well as system-specific and user-provided *probability distributions* on message delays. The result is a *real-time, probabilistic actor system* semantically consistent with its original non-deterministic version and *purely probabilistic*, i.e., at any time t, the state s_t reached at time t can perform *at most one* probabilistic transition; this is also called the *absence of nondeterminism* (AND) property. It greatly facilitates *statistical model checking* (SMC) analysis [Agha and Palmskog 2018], a formal method that scales well to large distributed systems. Two other transformations make this probabilisitic model *executable* by sampling, and specify the relevant *events to be analyzed*. These three transformations are proved correct, and automated in a tool supporting SMC-based quantitative analysis.

Methodologically, our approach makes it possible to *explore the design space and learn much about a distributed system's design before its implementation*. Many systems are developed without these benefits: it is often prohibitively expensive to explore *alternative implementations*; but quite easy to quickly explore alternative system designs formalized as *executable formal specifications* and analyze their respective qualitative and quantitative properties. Our approach is best exploited in the development of *new* distributed systems to arrive at a mature, thoroughly analyzed design *before* implementation. The resulting specification can then be used either as a prototype from which code is written, or, as demonstrated in [Liu et al. 2020], to automatically generate a, currently less

efficient, *correct by construction implementation*. Moreover, as several of our case studies in Section 9 show—for example, for Cassandra—by abstracting a formal model from a system implementation it is also possible to both analyze an existing system and to explore alternative designs for it.

Formalization and Main Contributions. Our approach is very general and formalism-independent. However, to mathematically define the theory transformations, prove their correctness, and obtain a correct-by-construction tool implementation, we *must formalize* our approach in an expressive formal framework that supports both qualitative and quantitative analysis. We use rewriting logic's natural way of modeling object-based distributed systems [Meseguer 1993], supported by the Maude rewriting logic language [Clavel et al. 2007], as follows. In Section 4 we define a transformation of rewrite theories $\mathcal{R} \mapsto \mathcal{R}_\Pi$, where: (1) $\mathcal{R}$ belongs to the class of *generalized actor rewrite theories* (*GARwTh*s), which formalize generalized actor systems and can naturally model most distributed systems. (2) $\mathcal{R}_\Pi$ is a *timed probabilisitic rewrite theory* [Agha et al. 2006] suitable for quantitative analysis by *statistical model checking* (SMC). (3) Π is a user-specified family of *parametric probability distributions* that model quantitatively the *message delays*.[1] In Section 5 we prove that: (a) for any $\mathcal{R} \in$ *GARwTh* and initial states satisfying natural requirements, all behaviors of $\mathcal{R}_\Pi$ are *purely probabilistic*; (b) $\mathcal{R}_\Pi$ is related to $\mathcal{R}$ by means of a *stuttering simulation*. The probabilistic rewrite theory $\mathcal{R}_\Pi$ is a *non-executable* mathematical model. In Section 6 we define a second theory transformation $\mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ that makes $\mathcal{R}_\Pi$ executable, and show that $\mathit{Sim}(\mathcal{R}_\Pi)$ simulates $\mathcal{R}_\Pi$. However, it is not always possible to directly express desired quantitative properties on either $\mathcal{R}$ or $\mathit{Sim}(\mathcal{R}_\Pi)$. To support the specification and SMC analysis of quantitative properties, Section 7 defines a third theory transformation $\mathit{Sim}(\mathcal{R}_\Pi) \mapsto M(\mathit{Sim}(\mathcal{R}_\Pi))$ that adds to $\mathit{Sim}(\mathcal{R}_\Pi)$ a monitor that "records" the *events* needed to measure quantitative properties during a run.

These transformations allow automatic generation of correct-by-construction, executable, purely probabilistic, quantitative models from nondeterministic ones for a large class of distributed systems. This enables automatic SMC analysis of quantitative system properties, which we accomplish by:

- automating the transformations $\mathcal{R} \mapsto \mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi) \mapsto M(\mathit{Sim}(\mathcal{R}_\Pi))$ in Maude; and
- automating within the same environment the quantitative analysis of $M(\mathit{Sim}(\mathcal{R}_\Pi))$ in the PVeStA SMC tool [AlTurki and Meseguer 2011], a parallelized tool for SMC analysis.

This automation of theory transformations and SMC analysis is supported by the *Actors2PMaude* tool described in Section 8. We present in Section 9 a collection of case studies that apply the Actors2PMaude tool to different kinds of distributed system designs. Our case studies focus on the SMC analysis of *quantitative* and *mixed* properties. However, we show in Section 8 how *qualitative* safety or liveness properties expressed in linear temporal logic (LTL) can also be verified on the *same* distributed system model $\mathcal{R}$ (no model schizophrenia!) using other tools.

Prior Maude-Based Work and Missing Links. We discuss related work in Section 10. Here we focus on previous work using Maude and probabilistic rewrite theories for the quantitative analysis of distributed systems. The paper [Agha et al. 2006] introduced the notion of a probabilistic

[1] Quantitative properties of distributed systems often involve *time*. For example, key performance metrics in transaction systems include *throughput* (completed transactions per second) and the average *latency* of each transaction. For analysis purposes, message delays can be modeled as following certain *probability distributions* (see, e.g., [Benson et al. 2010]). However, care must be taken to choose probability distributions that approximate well the communication delays in actual systems. Furthermore, as shown in Section 9.2, the *same implementation* can have quite different performance results on different execution platforms. Message delays on different execution platforms therefore need to be approximated by different probability distributions. This is one reason for the expressiveness of our framework: In most simulation and formal (performance) analysis tools, the designer is given a fixed set of (parametric) probability distributions. Instead, while we provide a library of predefined parametric probability distributions, our framework and tool allow the user to define her own distributions, to best approximate the message delay distribution on her execution platform. We also explain (this is the contribution of Section 6) how the user can obtain a correct simulation model based on her user-defined delay distributions.

rewrite theory and the QuaTEx quantitative probabilistic temporal logic, and used the VeStA tool [Sen et al. 2005b] for SMC verification of QuaTEx properties. Follow-up work, using VeStA or its PVeStA parallelization, includes applications to the quantitative analysis of, e.g., sensor networks [Katelman et al. 2008], protocols protecting systems against distributed denial of service (DDoS) attacks [Agha et al. 2005; AlTurki et al. 2009; Eckhardt et al. 2012], stochastic hybrid systems [Meseguer and Sharykin 2006], inter-domain bandwidth reservation infrastructure [Weghorn et al. 2022], cloud-based data storage systems [Bobba et al. 2018; Liu et al. 2019a,b, 2020], and blockchain algorithms [Alturki and Rosu 2019]. In comparison to that prior work, our work solves important problems associated with critical *missing links* between qualitative and quantitative models:

(1) The enrichment process of an object-based rewrite theory into a probabilistic one was previously an awkward, time-consuming, and error-prone manual process.
(2) The passage from a non-executable probabilistic rewrite theory to its executable version for simulation and SMC analysis was also performed by hand, raising similar concerns.
(3) The specification of the measurable events of interest needed to express relevant quantitative properties likewise had to be added by hand.
(4) Even if all these manual transformations were correct, for SMC verification in VeStA and PVeStA, the rewrite theory had to be *purely probabilisitic*; however, no *theory-generic* meta-theorems of the kind proved in Sections 5–7 were available.
(5) The *combined effect* of all these *missing links* made quantitative modeling and analysis difficult and error prone, requiring substantial user expertise. Furthermore, there was no automated support at any stage, except for the last, SMC verification step.

This work fills all these foundational and automation gaps in the passage from qualitative to quantitative models. Moreover, by automating most of the steps, it greatly simplifies the quantitative analysis of distributed systems, supporting the use of advanced verification tools by non-experts.

2 PRELIMINARIES

Maude [Clavel et al. 2007] is a formal specification language and analysis tool for distributed systems. A Maude module specifies a *rewrite theory* [Meseguer 1992] $\mathcal{R} = (\Sigma, E, L, R)$, where:

- Σ is an algebraic *signature*, i.e., a set of *sorts*, *subsorts*, and *function symbols*.
- (Σ, E) is an *order-sorted equational logic theory* [Goguen and Meseguer 1992] specifying the system's data types, with E a set of (possibly conditional) equations and axioms.
- L is a set of rule *labels*.
- R is a collection of *labeled conditional rewrite rules* $[l] : t \longrightarrow t'$ **if** *cond*, with t, t' Σ-terms and $l \in L$, that specify the system's local transitions.

We summarize Maude's syntax and refer to [Clavel et al. 2007] for details. Operators are declared **op** $f : s_1 \ldots s_n$ -> s and can have user-definable syntax, with '_' denoting argument positions, as in _+_. (Unconditional and conditional) equations and rewrite rules are introduced with, resp., the keywords **eq** and **ceq**, and **rl** and **crl** . Mathematical variables are declared with the keywords **var** and **vars**. Comments start with '***'. The following example illustrates how a Maude specification is just a *mathematical definition* of a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, given in typewriter notation.

Example 2.1. The rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ specifying an actor system of *bank accounts* imports the module NAT of natural numbers of sort *Nat*. Its signature Σ extends that of NAT by adding sorts *Oid* of object identifiers, *Accnt*, of bank account objects, *Msg* of messages, and *Configuration* of distributed states, called *configurations*, with subsort inclusions *Accnt*, *Msg* < *Configuration* and: (i) an account-building operator $\langle \_ : Accnt \mid bal : \_ \rangle : Oid\ Nat \rightarrow Accnt$, (ii) a configuration union operator with empty syntax (juxtaposition) $\_\ \_ : Configuration\ Configuration \rightarrow Configuration$,

with a constant *none* of sort *Configuration* (the empty configuration), and (iii) message operators $credit : Oid\ Nat \longrightarrow Msg$, $debit : Oid\ Nat \longrightarrow Msg$. Equations E are the equations of `NAT` as well as the associativity $(X\ Y)\ Z = X\ (Y\ X)$, commutativity $X\ Y = Y\ X$, and identity $X\ none = X$ axioms, with X, Y, Z of sort *Configuration*, making configurations into *multisets* of messages and bank account objects. The set L of labels is $L = \{cred, deb\}$. The rewrite rules R are

$$[cred] : credit(A, N)\ \langle\ A : Accnt \mid bal : M\rangle \longrightarrow \langle\ A : Accnt \mid bal : M + N\rangle$$
$$[deb] : debit(A, N)\ \langle\ A : Accnt \mid bal : M\rangle \longrightarrow \langle\ A : Accnt \mid bal : M - N\rangle\ \textbf{if}\ M \geq N$$

with A of sort *Qid* and N, M of sort *Nat*. In Maude, all these declarations are specified with *isomorphic typewriter notation*. For example, $credit : Oid\ Nat \longrightarrow Msg$ is declared as **op** `credit : Oid Nat -> Msg .` Likewise, the two rules are declared as:

```
rl [cred] : credit(A,N) < A : Accnt | bal : M > => < A : Accnt | bal : M + N > .
crl [deb] : debit(A,N) < A : Accnt | bal : M > => < A : Accnt | bal : M - N > if M >= N .
```

We will use Maude notation in the remainder of the paper.

Maude supports the following *syntactic sugar* and translates it into standard $\mathcal{R} = (\Sigma, E, L, R)$ notation: A declaration **class** C | $att_1 : s_1, \ldots, att_n : s_n$ declares an *object class* C with attributes att_1 to att_n of sorts s_1 to s_n. For example, **class** `Accnt | bal : Nat .` An *object* of class C is 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 val_1 to val_n are the current values of the attributes att_1 to att_n. *Messages* are terms of sort `Msg`.

Maude's built-in function `random(`k`)` returns the k-th pseudo-random number as a number between 0 and $2^{32} - 1$, and a built-in constant `counter` with a rewrite rule `counter => N:Nat.` which rewrites `counter` to a different natural number each time it is rewritten. The rule

```
rl [rnd] : rand => real(random(counter + 1) / 4294967295) .
```

rewrites the constant `rand` (used in Section 8) to a real number between 0 and 1, pseudo-randomly chosen according to the uniform distribution.

Probabilistic Rewrite Theories [Agha et al. 2006] can express a wide range of probabilistic systems, including discrete- and continuous-time Markov chains, Markov decision processes (MDPs), probabilistic (timed) automata [Bentea and Ölveczky 2011], probabilistic Petri nets, object-based probabilistic real-time systems [Bobba et al. 2018; Eckhardt et al. 2012; Katelman et al. 2008], and object-based stochastic hybrid systems [Meseguer and Sharykin 2006]. They have rules of the form

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

where the term t' has new variables $\vec{y}$ disjoint from the variables $\vec{x}$ in t. The concrete values of $\vec{y}$ are chosen probabilistically according to the *parametric* probability distribution $\pi(\vec{x})$: each matching substitution $\{\vec{x} \mapsto \vec{a}\}$ of the left-hand side's variables defines a concrete distribution $\pi(\vec{a})$.

Statistical Model Checking (SMC) [Agha and Palmskog 2018; Sen et al. 2005a; Younes and Simmons 2006] trades off the exactness of probabilistic model checking [Baier et al. 2018] in exchange for greater performance and scalability. Quantitative system properties expressed in a probabilistic temporal logic are evaluated by Monte Carlo simulation up to a chosen level of statistical confidence. Although the model was originally required to be *purely probabilistic*, this requirement has been relaxed to allow SMC of MDP models in, e.g., [Ashok et al. 2019; Bogdoll et al. 2011; Henriques et al. 2012; Lassaigne and Peyronnet 2012; Wang et al. 2020]. In this work, only purely probabilistic models will be used.

Quantitative Temporal Expressions (QuaTEx) [Agha et al. 2006] is a quantitative temporal logic that extends *probabilistic computation tree logic* [Hansson and Jonsson 1994] by supporting real-valued expressions. A QuaTEx query Q consists of a set of definitions D followed by a query of the expected value of a *path expression PExp* interpreted over an execution path. A *state expression SExp* is interpreted over a state.

$$\begin{array}{rcl@{\qquad}rcl} Q & ::= & D\ \texttt{eval E[}\ PExp\ \texttt{]}\ ; & D & ::= & set\ of\ Def \\ Def & ::= & N(x_1, ..., x_m) = PExp\ ; & SExp & ::= & c \mid f \mid F(SExp_1, ..., SExp_k) \mid x_i \\ PExp & ::= & SExp \mid \bigcirc N(SExp_1, ..., SExp_n) \mid \texttt{if}\ SExp\ \texttt{then}\ PExp_1\ \texttt{else}\ PExp_2\ \texttt{fi} & & & \end{array}$$

A definition *Def* of a *temporal operator* consists of a name N and a set of (freeze) formal parameters on the left-hand side, and a path expression on the right-hand side. A *state expression* can be a constant c, a function f mapping a state to a concrete value, a k-ary function F mapping k state expressions to a state expression, or a formal parameter x. A *path expression* can be a state expression, a next operator $\bigcirc$ followed by an application of a temporal operator defined in D, where the formal parameters are replaced by state expressions or a conditional expression. The $\bigcirc$ operator takes an expression at the next state and makes it an expression for the current state.

For example, a QuaTEx query on *"the number of clients connected to the server within 10 time units"* can be given as follows:

```
numberOfConnected(t, cls) = if t < currentTime() then cls
        else if connected() then ○(numberOfConnected(t, cls + 1))
                else ○(numberOfConnected(t, cls)) fi fi ;
eval E[numberOfConnected(currentTime() + 10, 0)]
```

The first three lines define the operator `numberOfConnected(`t, cls`)`, which increments the count of the number of connected clients if along an execution path any client is connected to the server in the state[2] (state function *connected*()) within time t, and returns the current number of connected clients cls otherwise. Function *currentTime*() returns the state's global time. The fourth line returns the expected number of connected clients within 10 time units from a given initial state.

The PVeStA Statistical Model Checker. Since QuaTEx path expressions are *real-valued*, they can naturally express quantitative properties. For a *purely probabilistic* Maude model they can be analyzed by SMC using the VeStA family of tools [AlTurki and Meseguer 2011; Sebastio and Vandin 2013; Sen et al. 2005b]. They are evaluated by Monte Carlo simulation up to a given statistical confidence level. The expected value of the path expression is iteratively evaluated w.r.t. two parameters α and δ until a value $\bar{v}$ is obtained such that with $(1-\alpha)$ statistical confidence, the expected value lies in the interval $[\bar{v} - \frac{\delta}{2}, \bar{v} + \frac{\delta}{2}]$. Our Actors2PMaude tool incorporates PVeStA [AlTurki and Meseguer 2011]. PVeStA invokes the Maude interpreter to execute purely probabilistic Maude models, and verifies QuaTEx formulas on model simulations by parallel SMC.

Transition Systems, Stuttering Simulations, and Bisimulations. A *transition system* $\mathcal{A}$ is a triple $(A, \rightarrow_{\mathcal{A}}, a_0)$, where A is a set of *states*, $\rightarrow_{\mathcal{A}} \subseteq A \times A$ is a *transition relation on states*, and $a_0 \in A$ is the *initial state*. $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, a_0)$ is called *total* (or *deadlock-free*) iff its transition relation $\rightarrow_{\mathcal{A}}$ is so, i.e., iff $\forall a \in A\ \exists a' \in A$ s.t. $a \rightarrow_{\mathcal{A}} a'$. We use $Reach(a)$ to denote the set of states *reachable* from $a \in A$ in $\mathcal{A}$, i.e., $Reach(a) = \{a' \in A \mid a \rightarrow^*_{\mathcal{A}} a'\}$, where $\rightarrow^*_{\mathcal{A}}$ denotes the reflexive-transitive closure of $\rightarrow_{\mathcal{A}}$. Any transition system $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, a_0)$ can be totalized as $\mathcal{A}^\bullet = (A, \rightarrow^\bullet_{\mathcal{A}}, a_0)$, with $(\rightarrow^\bullet_{\mathcal{A}}) = (\rightarrow_{\mathcal{A}}) \cup \{(a, a) \mid \neg\exists a' \in A\ \ s.t.\ \ a \rightarrow_{\mathcal{A}} a'\}$. A *path* π in a total transition system $\mathcal{A}$ is a function $\pi : \mathbb{N} \rightarrow A$ such that $\pi(0) = a_0$ and $\forall n \in \mathbb{N},\ \pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$.

[2]In an execution path, we assume that at most one client is connected to the server at any state.

Definition 2.2. [Meseguer et al. 2010] Given total transition systems $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, a_0)$ and $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, b_0)$, a *stuttering simulation map*, denoted $h : \mathcal{A} \rightarrow \mathcal{B}$, is a function $h : Reach(a_0) \rightarrow Reach(b_0)$ such that: (1) $h(a_0) = b_0$, (2) given any path π in $\mathcal{A}$ starting at a_0 (i.e., $\pi(0) = a_0$), there is a path ρ in $\mathcal{B}$ starting at b_0 and a strictly monotonic function $\kappa : \mathbb{N} \rightarrow \mathbb{N}$ such that, for each $n \in \mathbb{N}$ and each i with $\kappa(n) \leq i < \kappa(n+1)$, $h(\pi(\kappa(n))) = h(\pi(\kappa(i))) = \rho(n)$.

A function $h : Reach(a_0) \rightarrow Reach(b_0)$ is called a *bisimulation map* $h : \mathcal{A} \rightarrow \mathcal{B}$ from $\mathcal{A}$ to $\mathcal{B}$ iff: (i) $h(a_0) = b_0$; and (ii) for any $a \in Reach(a_0)$, if $a \rightarrow_{\mathcal{A}} a'$ then $h(a) \rightarrow_{\mathcal{B}} h(a')$, and if $h(a) \rightarrow_{\mathcal{B}} b$ then there exists $a'' \in Reach(a_0)$ with $a \rightarrow_{\mathcal{A}} a''$ and $h(a'') = b$.

We can associate to a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ and an initial state $init \in T_{\Sigma/E,k}$[3] a total transition system $(\mathcal{R}, init)^\bullet$ defined by $(\mathcal{R}, init)^\bullet = (T_{\Sigma/E,k}, \rightarrow^\bullet_{\mathcal{R}}, init)$, where $\rightarrow^\bullet_{\mathcal{R}}$ is the totalization of the one-step rewrite relation on states $\rightarrow_{\mathcal{R}}$ defined by $\mathcal{R}$.

3 GENERALIZED ACTOR SYSTEMS AND THEIR FORMALIZATION

The goal of this section is to develop a general and widely applicable model of distributed systems, called *generalized actor systems*, which is formalized in rewriting logic in Section 3.2.

3.1 Generalized Actor Systems

Actors [Agha 1986] are a popular model for distributed systems, where distributed objects communicate through asynchronous message passing. When an actor receives a message it can change its state, send messages, and create new actors. Furthermore, these actions are *deterministic* [Agha 1986]: the actor's new state, the generated messages, and the newly created actors are uniquely determined by the received message and the actor's internal state. This captures *local determinism* in distributed systems: the state change of a node in response to an event is typically deterministic.

In practice it is convenient (or even necessary) to allow nodes in a distributed system to exhibit "internal actions" that are not triggered by messages. We therefore introduce *generalized actor systems* (GASs), which extend (Agha's) actors by allowing actors to exhibit autonomous behaviors ("internal actions") that are uniquely determined by the actor's state. Furthermore, in a GAS, no actor can perform an infinite sequence of such internal actions, and an actor can always read a message addressed to it, unless it (first) can perform an internal action.

We also assume that at most one actor in a GAS can perform an internal action in the initial state. This does not restrict the systems that can be seen as GASs, since we can trigger an actor's initial action by adding to the initial state an "initialization message" for that actor.

3.2 Formalizing GASs: Generalized Actor Rewrite Theories

We formalize generalized actor systems as *generalized actor rewrite theories* (*GARwTh*s). These are object-oriented rewrite theories that (together with an initial state) satisfy natural requirements.

The distributed states of a *GARwTh* (terms of sort `Configuration`) are multisets of *objects* (terms of sort `Object`) and *messages* (terms of sort `Msg`). Multiset union is modeled by an associative and commutative operator `_ _` (juxtaposition), where `null` is the empty multiset.

The rewrite rules in a *GARwTh* have the form[4] (where '[...]' are optional parts)

`[`*l*`]` : `(to` *o* `[from` *o'*`]` : *mp*) `<` *o* : *C* `|` *atts* `>` `=>` `<` *o* : *C* `|` *atts'* `>` *msgs* *newobjs* [**if** *cond*] (†)

or

`[`*l*`]` : `<` *o* : *C* `|` *atts* `>` `=>` `<` *o* : *C* `|` *atts'* `>` *msgs* *newobjs* [**if** *cond*] (‡)

where:

[3] $T_{\Sigma/E,k}$ denotes the E-equivalence classes of ground Σ-terms of kind k [Clavel et al. 2007].

[4] Logical *variables* appearing in rules are written in capital letters, while *terms* are written in italics.

- *msgs* is a (possibly `null`) term of sort `Configuration` which, applying the equations, reduces to a set of messages ($n \geq 0$), each of which is a term of sort `Msg`, of the form:

  $$\texttt{(to } o_1 \texttt{ from } o\theta \texttt{ : } mp_1\texttt{)} \quad \ldots \quad \texttt{(to } o_n \texttt{ from } o\theta \texttt{ : } mp_n\texttt{)}$$

  where θ is the matching substitution used when applying the rule; the term mp_i is the payload of the message sent to the receiver o_i from the sender $o\theta$.
- *newobjs* is a (possibly `null`) term of sort `Configuration` which, applying the equations, reduces, for each matching substitution θ, to a set of *new objects* which are added to the configuration. The *names* of the new objects must all be distinct and different from the object names in the current configuration; this can be achieved as described in [Meseguer 1993].

We call (†) and (‡) *message-triggered* and *object-triggered* rules, respectively.

An initial state *initconf* (of sort `Configuration`) of a generalized actor rewrite theory consists of a set of objects (with distinct names) and messages of the form:

$$\texttt{<}\ o_1 : C_1 \ \texttt{|}\ atts_1\ \texttt{>} \quad \ldots \quad \texttt{<}\ o_n : C_n \ \texttt{|}\ atts_n\ \texttt{>} \quad \texttt{(to } o_{i_1} \texttt{ [from } o_{l_1}\texttt{]} \texttt{ : } mp_{i_1}\texttt{)} \quad \ldots \quad \texttt{(to } o_{i_k} \texttt{ [from } o_{l_k}\texttt{]} \texttt{ : } mp_{i_k}\texttt{)}$$

with $1 \leq i_1 < \ldots < i_k \leq n$, and $\{l_1, \ldots, l_k\} \subseteq \{1, \ldots, n\}$, so that the rewrite theory and the initial state *initconf* together satisfy the following requirements:

(1) In any concrete configuration of objects and messages, any object *enabled* by a message-triggered rule to receive a given message addressed to it, is enabled to receive that message by the application of a *unique* message-triggered rule with a *unique* substitution.
(2) In any concrete configuration, any object *enabled* to perform a transition by an object-triggered rule, is so enabled by a *unique* object-triggered rule with a *unique* substitution.
(3) *At most one* object in *initconf* is enabled to be rewritten by an object-triggered rule.
(4) In *initconf*, if an object has a message addressed to it, then it is enabled to receive it, and it is *not* enabled to be rewritten by an object-triggered rule.
(5) In any configuration reachable from *initconf*, if an object is in a state *not* enabled by any object-triggered rule and the configuration contains a message addressed to it, then it is enabled to receive such a message.
(6) In any configuration reachable from *initconf*, if a (†) or (‡) rule applies to an object with the ground substitution θ, then all addressees in the (normal form of the) ground set of messages $msgs\theta$ in the instance of the applied rewrite rule are objects that either: (i) belong to the current configuration, or (ii) are among the new objects in the set $newobjs\theta$ introduced in the instance of the applied rewrite rule.
(7) Any rewrite sequence where in each step some object-triggered rule is applied to the same object must be finite.

By requirements **1** and **2** an actor's message-reception actions and internal actions are *locally deterministic*.[5] By requirements **3** and **4** at most one actor can perform an internal action in the initial state and the other actors can receive initialization messages in their initial states. By requirement **5** sooner or later any message in the system *can* be received. By requirement **6** messages created are only addressed to existing (including newly created) objects. Requirement **7** means that no actor can perform an uninterrupted infinite sequence of internal actions.

Note that a *GARwTh* abstracts from time. This reflects common design practice, where models usually are untimed, since timers constitute implementation choices and an algorithm's correctness is generally independent of the choices. For more on this, see the applications in Sections 3.3 and 9.

[5]The same object could be simultaneously enabled by requirement **2** *and* by requirement **1** with different messages; i.e., "local determinism" does not rule out nondeterministic choice between the local transitions currently possible for an object.

3.3 GARwThs in Practice

A wide range of distributed systems that have been formally modeled in Maude are directly, or with minor adaptation, generalized actor rewrite theories. These include: textbook distributed algorithms [Ölveczky 2017]; internetworking protocols [Agha et al. 2005; Wang et al. 2011, 2000]; mobile ad-hoc network (MANET) protocols [Liu et al. 2015, 2016a]; industrial distributed computing services such as Apache Zookeeper [Skeirik et al. 2013]; and cloud data storage systems, including the industrial data stores such as Apache Cassandra [Liu et al. 2014] and Google's Megastore [Grov and Ölveczky 2014], and distributed transaction systems [Liu 2022; Liu et al. 2016b, 2019a, 2018].

The following GARwTh, specified in Maude, defines a simple query protocol for handling "read" requests in a database system where multiple distributed servers may store the same data item.

The Query Protocol. For each user read request, a client issues read requests to the servers replicating the data item, and stores the read operation's result, which should be the latest written value among all returned values. This protocol can be seen as a simplified version of the protocol processing reads in the Cassandra key-value store [Cassandra 2022] (see also Section 9).

A client buffers the user requests, each of which is an operation `read(`id`,`k`)` on some data item or key k, in the `queries` queue (rule `req`). In rule `issue`, the client starts processing the first query in this queue: it finds the servers `R[K]` replicating the key `K`, and propagates the message `read(ID,K)` to them. Each server (called a replica) replies with the locally stored < *value*, *timestamp* > pair for the requested key (rule `reply`), with *timestamp* denoting when *value* was written. Upon receiving the reply, the client updates the corresponding record with the most recent value (with the `latest` timestamp) it has seen so far, removes the sender from its waiting list, and keeps waiting for the others (rule `update`). When all responses have been collected (the waiting list is `empty`), the client prepares to issue the next query in the queue by removing the current one (rule `finish`).

The following shows the Maude module `QUERY`, with some definitions omitted for brevity:

```
mod QUERY is
  ...
  vars O O' : Oid . var OS : Oids . var K : Key . vars V V'  : Value .  var ID : Id .
  vars TS TS' : Timestamp . vars DAT DAT' : Data . var Q : Query . vars QS QS' : Queries .
  var R : Map{Key,Oids} . var DB : Map{Key,Data} . var RS : Map{Id,Data} .

  class Client | queries : Queries, waiting : Oids, replicas : Map{Key,Oids}, results : Map{Id,Data}.
  class Server | database : Map{Key,Data} .  *** key-value stores map keys to data

  op propagate_to_from_ : Query Oids Oid  -> Msgs .
  eq propagate Q to (O' ; OS) from O = (propagate Q to OS from O) (to O' from O : Q) .
  eq propagate Q to empty from O = null .

  rl [req] :  (to O : QS')  < O : Client | queries: QS >  =>  < O : Client | queries: QS :: QS' >.

  crl [issue] :  < O : Client | queries : read(ID,K) :: QS, waiting : empty,
                                replicas : R, results : RS >
            =>      < O : Client | waiting : R[K], results : insert(ID,null,RS) >
                    (propagate read(ID,K) to R[K] from O)  if not $hasMapping(RS,ID) .

  rl [reply] :  (to O from O' : read(ID,K))    < O : Server | database : DB >
           =>      < O : Server | >    (to O' from O : reply(ID,DB[K])) .

  rl [update] :  (to O from O' : reply(ID,DAT'))
                 < O : Client | waiting : (O' ; OS), results : (RS, ID |-> DAT) >
            =>      < O : Client | waiting : OS, results : (RS, ID |-> latest(DAT,DAT')) > .
```

```
  rl [finish] :  < O : Client | queries : read(ID,K) :: QS, waiting : empty, results : (RS, ID |-> DAT) >
           =>      < O : Client | queries : QS > .
endm
```

The term `propagate read(ID,K) to R[K] from O` in rule `issue` reduces to a set of messages sent to K's replicas by the corresponding equations above. The module `INIT-QUERY` specifies an initial state with two clients, with incoming requests, and three servers, each storing two keys:

```
mod INIT-QUERY is including QUERY .
  ops c1 c2 s1 s2 s3 : -> Oid [ctor] .    ops k1 k2 k3 : -> Key [ctor] .

  op initconf : -> Configuration .
  eq initconf = (to c1 : (read(1,k1) :: read(2,k3)))    (to c2 : read(3,k2))
                < c1 : Client | queries : nil, waiting : empty, results : empty,
                                replicas : k1 |-> s1 s2, k2 |-> s2 s3, k3 |-> s1 s3 >
                < c2 : Client | queries : nil, waiting : empty, results : empty,
                                replicas : k1 |-> s1 s2, k2 |-> s2 s3, k3 |-> s1 s3 >
                < s1 : Server | database : k1 |-> < 23, 1 >, k3 |-> < 8, 4 > >
                < s2 : Server | database : k1 |-> < 10, 5 >, k2 |-> < 7, 3 > >
                < s3 : Server | database : k2 |-> < 14, 2 >, k3 |-> < 3, 6 > > .
endm
```

4 THE P TRANSFORMATION

Quantitative properties of distributed systems often involve *time*. For example, important performance metrics in transaction systems are *throughput* (completed transactions per second) and the average *latency* of each transaction. The time "delays" in a distributed system can often be attributed to communication, i.e., to message delays. For analysis purposes, message delays can be seen as following certain *probability distributions* (see, e.g., [Benson et al. 2010]), possibly also accounting for parameters such as payload size and the distance between sender and recipient.[6]

We *automatically transform* a generalized actor system into a timed probabilistic one by enriching it with user-defined probability distributions Π on communication delays: each message-triggered or internal action that generates new messages has a continuous distribution governing the delays of the generated messages. Also, GAS's internal actions are applied *eagerly*, i.e., as soon as enabled.

Section 4.2 formalizes this *P transformation*, mapping a nondeterministic untimed *GARwTh* $\mathcal{R}$ with an initial state *initconf*, which together satisfy the requirements 1–7 in Section 3, and a family Π of parametric probability distributions specifying the distributions of the delays of the generated messages, into a corresponding probabilistic rewrite theory $\mathcal{R}_\Pi$ and initial state $initconf_\Pi$.

4.1 Defining Probabilistic Message Delay Distributions

We need to define the message delays for (a) messages created by the application of a rewrite rule, and (b) the messages in the initial state. For (a), in a *probabilistic* rewrite rule $[l] : t(\vec{x}) \longrightarrow t'(\vec{x},\vec{y})$ **if** $cond(\vec{x})$ *with probability* $\vec{y} := \pi_l(\vec{x})$, the probability distribution π_l, from which the values of the new variables $\vec{y}$ are sampled, is *parametric* on the instantiation $\vec{x}\theta$, where θ is the matching substitution when applying the rule. For each rewrite rule l (which may generate new messages), we must therefore define a parametric probability distribution $\pi_l(\vec{x})$ from which message delays can be sampled. $\pi_l(\vec{x})$ and $\pi_{l'}(\vec{x'})$ may be *completely different* for $[l] \neq [l']$. For example, $\pi_l(\vec{x})$ may model message delays from clients to servers, while $\pi_{l'}(\vec{x'})$ may model the frequency of process failures, which can naturally be modeled by "failure messages" [Ölveczky 2017].

[6] The realistic modeling of message delay times, taking into account factors such as the above, as well as of, e.g., local processing times, is supported in two ways: (i) by using *parametric* probability distributions whose parameter values may vary depending on some of those factors; and (ii) by allowing the possibility of "modulating" the actual value of a delay as a value d sampled from a distribution by applying to d a *modulating function* δ, accounting for additional factors.

As explained in Footnote 6, in some applications we may need to *modulate* the sampled values to define more complex message delays that account for extra factors such as a transition's internal computation time. For this purpose, we allow the user to define a *modulation function* $\delta_l(\vec{x}, O, O', MC)$ for each rule l, which is parametric on $\vec{x}_l$ and on the message's receiver O, sender O', and message content MC, such that for each instantiation of its parameters, $\delta_l(\vec{x}_l, O, O', MC)$ becomes a function on the reals, so that the *actual delay* of a message `(to` o' `from` o `:` *msgContent*`)` is $\delta_l(\vec{x}\theta, o, o', msgContent)(d)$, where d is the "basic sampled delay" obtained by sampling π_l.

Likewise, for (b), the messages in the initial state, we must also define the "basic" delay distribution π_{init} and the modulation function $\delta_{init}(O, O', MC)$.

Assumptions on π and δ and Notation. We explain next: (1) the density function defining (2) a rule's continuous distribution; (3) a safe "envelope" for sampling values; (4) how δ modulates sampled delays; and (5) how Π gathers all π's and δ's. (1) For each rule label $l \in L$ and ground substitution $\theta = \{\vec{x}_l \mapsto \vec{a}\}$, instantiating the parameters $\vec{x}_l$ of rule l's left-hand side (resp. for $l = init$) there is a piecewise continuous *probability density function* $f_l(\vec{a}) : \mathbb{R} \to [0, +\infty)$ (resp. f_{init}) such that—since message delays are always non-negative—for $x < 0$, $f_l(\vec{a})(x) = 0$ (resp. $f_{init}(x) = 0$). (2) Each such density function defines a *probability distribution function* $\pi_l(\vec{a}) : \mathbb{R} \to [0, 1]$ (resp. $\pi_{init} : \mathbb{R} \to [0, 1]$) as a continuous and almost everywhere differentiable function of the form $\pi_l(\vec{a}) = \lambda x \in \mathbb{R}.\ \int_{-\infty}^{x} f_l(\vec{a})(t)\ dt$ (likewise for π_{init}). By the assumptions on $f_l(\vec{a})$ and f_{init}, $\pi_l(\vec{a})(x) = 0$ and $\pi_{init}(x) = 0$, for $x \leq 0$. More generally, yet equivalently ([Klenke 2006], Theorem 1.88), $f_l(\vec{a})$ defines a *probability measure* $\mu_l(\vec{a}) : \mathcal{B}(\mathbb{R}) \to [0, 1]$ on the Borel σ-algebra of $\mathbb{R}$, defined by $\mu_l(\vec{a}) = \lambda B \in \mathcal{B}(\mathbb{R}).\ \int_{t \in B} f_l(\vec{a})(t)\ dt$. (3) In what follows, $X_{f_l(\vec{a})}$, which we call the *support* of $f_l(\vec{a})$, will denote the set $X_{f_l(\vec{a})} = \{x \in \mathbb{R} \mid f_l(\vec{a})(x) > 0\}$, and $\overline{X}_{f_l(\vec{a})}$ will denote its topological closure, obtained by adding to $X_{f_l(\vec{a})}$ its limit points. The set $\overline{X}_{f_l(\vec{a})}$ provides a useful "envelope" for the values obtained by sampling $\pi_l(\vec{a})$. Indeed, by openness, for any $x \in \mathbb{R} \setminus \overline{X}_{f_l(\vec{a})}$ there is an open interval $(a, b) \subseteq \mathbb{R} \setminus \overline{X}_{f_l(\vec{a})}$ with $x \in (a, b)$, so that $\mu_l(\vec{a})((a, b)) = \pi_l(\vec{a})(b) - \pi_l(\vec{a})(a) = 0$. Therefore, any real number r obtained by sampling the distribution $\pi_l(\vec{a})$ must be inside the envelope $\overline{X}_{f_l(\vec{a})}$. Notice, furthermore, that the assumption on $f_l(\vec{a})$ and f_{init} forces the inclusions $\overline{X}_{f_l(\vec{a})} \subseteq [0, +\infty)$ and $\overline{X}_{f_{init}} \subseteq [0, +\infty)$. This is because, in all modules $\mathcal{R}_\Pi$, the intended meaning of a number r obtained by sampling $\pi_l(\vec{a})$ (resp. π_{init}) is that of a *message delay*. (4) We also allow users to specify that a time delay r is *modulated* by applying to r the function $\delta_l(\vec{a}, o, o', mc)$ (for the initial configuration, $\delta_{init}(o, o', mc)$). The main requirement about $\delta_l(\vec{a}, o, o', mc)$ (and likewise for $\delta_{init}(o, o', mc)$) is that it defines a function $\lambda r.\ \delta_l(\vec{a}, o, o', mc)(r) :\ [0, +\infty) \to [0, +\infty)$ that is *strictly monotonic*, i.e., $r < r' \Rightarrow \delta_l(\vec{a}, o, o', mc)(r) < \delta_l(\vec{a}, o, o', mc)(r')$. (5) All this information about the π_l and δ_l functions is denoted by Π and is *specified by the user* as an input to the P transformation.

4.2 Defining the P Transformation

In this section we define the transformation $P : (\mathcal{R}, initconf, \Pi) \mapsto (\mathcal{R}_\Pi,\ initconf_\Pi)$, where:

- $\mathcal{R} = (\Sigma, E, L, R)$ is a generalized actor rewrite theory.
- *initconf* is an initial state consisting of a set of objects and messages, so that $\mathcal{R}$, together with *initconf*, is a *GARwTh* satisfying requirements 1–7 in Section 3.
- $\Pi = \{(\pi_l(\vec{x}_l), \delta_l(\vec{x}_l, O, O', MC))\}_{l \in L} \cup \{(\pi_{init}, \delta_{init}(O, O', MC))\}$ is a family of *pairs* of parametric functions, with $init \notin L$, defining the message delay distributions as explained in Sect. 4.1: $\pi_l(\vec{x}_l)$ is a *continuous probability distribution* for rule l, parametric on $\vec{x}_l$, and $\delta_l(\vec{x}_l, O, O', MC)$ is a *message delay modulation function*, parametric on $\vec{x}_l$ and on the message's receiver O, sender O', and message content MC. Likewise, π_{init} is a continuous probability distribution and $\delta_{init}(O, O', MC)$ is a real-valued function parametric on O, O', and MC.

- $\mathcal{R}_\Pi = (\Sigma_\Pi, E_\Pi, L_\Pi, R_\Pi)$ is the resulting probabilistic rewrite theory defined below.
- $initconf_\Pi$ is the corresponding P-transformed initial state, also defined below.

The resulting probabilistic rewrite theory $\mathcal{R}_\Pi$ is as follows. The state has the form { *config* | *clock* }, with *config* the current configuration (consisting of objects and messages with their delivery times) and *clock* the current time. Object-triggered rules are applied *eagerly*: time does not advance when they are enabled. When there is a message ready to be received in the state, a message-triggered rule is applied. The messages created when applying an object-triggered or a message-triggered rule are then "prepared to get their delays assigned" in a "delayed task object." If there is such a "delayed task" in the state, then its messages are individually assigned delays by sampling the corresponding distribution (rules $\texttt{delay}_{l.1}$ and $\texttt{delay}_{l.2}$, and $\texttt{delay}_{init.1}$ and $\texttt{delay}_{init.1}$ for messages in the initial state). When none of these rules can be applied, a "tick" rewrite rule advances the system's global time by increasing the clock to the value of the delivery time of the "next" message.

In what follows, we define in detail the probabilistic rewrite theory $\mathcal{R}_\Pi$ and the initial state $initconf_\Pi$ obtained by applying P to $(\mathcal{R}, initconf, \Pi)$.

4.2.1 The Equational Logic Theory (Σ_Π, E_Π). The equational theory (Σ_Π, E_Π) extends (Σ, E). It contains a sort `Real` for the real numbers, as well as the real number functions required to define the terms $\pi_l(\vec{x}_l)$, π_{init}, $\delta_l(\vec{x}_l, O, O', MC)$, and $\delta_{init}(O, O', MC)$ by equations in E_Π.

To represent *timed* probabilistic systems, the states in $\mathcal{R}_\Pi$ have the form {*config* | *clock* }:

```
op {_|_} : Configuration Real -> ClockedState [ctor] .
```

Besides objects and messages from $\mathcal{R}$, *config* may also contain: a *delay-task term* (of sort `DTask`) with a list of outgoing messages *yet to be assigned a delay*, and a set of already *delayed messages* (of sort `DMsgs`, with each message of sort `DMsg`) not yet ready to be received. A subsort `Objects` defines configurations consisting only of actor objects. A sort `MsgList` models a *list* of messages with concatenation operator `_;_` and identity `nil`. P adds to Σ the following sorts and subsorts:

```
sorts Real Objects DMsg DMsgs DTask Msgs MsgList ClockedState .
subsort Object < Objects .   subsorts Msg < Msgs MsgList .   subsort DMsg < DMsgs .
subsorts DMsgs DTask Msgs Objects < Configuration .
```

The state is a term {*objects msgs dmsgs dtask* | *clock*} consisting of (with cardinality): actor objects (≥ 1) denoted by *objects*, messages ready to be consumed (≤ 1) denoted by *msgs*, delayed messages not ready for consumption (≥ 0) denoted by *dmsgs*, and a delay-task term *dtask* (≤ 1).

The following variables are used in the definition of $\mathcal{R}_\Pi$:

```
vars O O' : Oid .     var MP : Payload .      vars T T' D : Real .
var OBJ : Object .    var OBJS : Objects .    var CF : Configuration .
var MSG : Msg .       var MSGS : Msgs .       var DMS : DMsgs .    var ML : MsgList .
```

P also defines the following operators used in $\mathcal{R}_\Pi$ and $initconf_\Pi$:

- A function `sort : Msgs -> MsgList` that turns a set of messages into a *sorted* list.[7]
- For each rule (labeled) l, an operator $\texttt{delay}_l : s_1 \ \ldots \ s_n$ `MsgList -> DTask` is used to generate the "delay task" which will assign the delays to the messages generated by the application of rule l. This operator takes as input a vector $\vec{a}_l$ of ground terms of sorts $\vec{s}_l = s_1, \ldots, s_n$ instantiating the corresponding variables $\vec{x}_l = x_1, \ldots, x_n$ appearing in the left-hand side of rule l, and a list of messages, and outputs a term of sort `DTask`.
- A delayed message of sort `DMsg` has the form `[`*time*`,`*msg*`]` indicating that *msg* (of sort `Msg`) will be delivered at global time *time*.

[7]The reason for sorting messages is explained in Footnote 8. Sorting can be implemented by using Maude's total order on terms available through Maude's `META-LEVEL` module.

- The predicate `objectEnabled` checks whether an object-triggered rule (type (‡)) is enabled on a set of objects. For each rule of the form (‡) a (possibly conditional) equation

  ```
  ceq objectEnabled(< o : C | atts >) = true [if cond] .
  ```

  is generated; two additional equations make `objectEnabled(`*objects*`)` `true` whenever an object in *objects* can be rewritten by an object-triggered rule.
- The function `init` converts *initconf* into the corresponding initial configuration $initconf_{\Pi}$:

  ```
  op init : Configuration -> ClockedState .
  eq init(OBJS MSGS) = { delay_init(sort(MSGS)) OBJS | 0.0 } .
  ```

  where the operator $\texttt{delay}_{init}$ is declared as `op` $\texttt{delay}_{init}$ `: MsgList -> DTask [ctor] .`

4.2.2 The Rewrite Rules R_{Π}. Each rule l of the form (†), resp. (‡), in $\mathcal{R}$ is transformed into a rule

```
[l.p] : { (to o from o' : mc)  < o : C | atts >  OBJS  DMS | T }
    => { < o : C | atts' >  delay_l(x_l, sort(msgs))  newobjs  OBJS  DMS | T }  [if cond]
```

respectively,

```
[l.p] : { < o : C | atts >  OBJS  DMS | T }
    => { < o : C | atts' >  delay_l(x_l, sort(msgs))  newobjs  OBJS  DMS | T }  [if cond]
```

When the configuration contains a term of sort `DTask`, the following *lifting rules* assign delays to the list of new messages one by one:

```
[delay_l.1] : { delay_l(x_l, (to O from O' : MP) ; ML)  CF | T }
        => { delay_l(x_l, ML)  [T + δ_l(x_l,O,O',MP)(D), (to O from O' : MP)]  CF | T }
         with probability D := π_l(x_l) .
```

where `CF` matches the rest of the configuration. The first message in the list is assigned a delay $\delta_l(\vec{x}_l, \texttt{O}, \texttt{O'}, \texttt{MP})(D)$, where the new variable `D` is assigned a value D sampled from $\pi_l(\vec{x}_l)$.

When each message in the delay task list has been assigned a delay, the $\texttt{delay}_l$ operator is removed from the configuration:

```
[delay_l.2] : delay_l(x_l, nil) => null .
```

Likewise, similar rewrite rules assign delays to the messages in the initial state:[8]

```
[delay_init.1] : { delay_init((to O from O' : MP) ; ML)  CF | T }
          => { delay_init(ML)  [T + δ_init(O,O',MP)(D), (to O from O' : MP)]  CF | T }
           with probability D := π_init .

[delay_init.2] : delay_init(nil) => null .
```

When none of the above rules can be applied (see below), the following "tick" rewrite rule advances global time in the system to the delivery time `T'` of the next message to be delivered:

```
[tick] : { OBJS  DMS  [T', MSG] | T }  =>  { OBJS  DMS  MSG | T' }
      if (not objectEnabled(OBJS)) /\ (T' <= times(DMS)) .
```

`times` gives the delivery times of the delayed messages `DMS`. The delayed message `[T', MSG]` becomes ready, as `MSG`, to be consumed. This rule can only be applied when no other rule is enabled:

- No object-triggered rule is enabled, because of the condition `not objectEnabled(OBJS)`.
- No message-triggered rule is enabled, since there cannot be any message without the '`[_,_]`' operator (i.e., of sort `Msg`) in the configuration in left-hand side of the tick rule.
- No lifting rule is enabled, since there is no term of sort `DTask` in the configuration.

[8] Note that since the samplings for the delays of messages in the sorted list of messages are independent of each other, the order of messages in the sorted list of messages is statistically immaterial, i.e., it does not affect the modeling. The only purpose of using a sorted list instead of a multiset of messages is to *eliminate the nondeterminism* implicit in an arbitrary choice of a message to be delayed in a message multiset. Similar rules are defined for initial messages without a sender.

4.3 Applying P: An Example

We now illustrate how we can specify in Maude the input Π to the P transformation, and the results of performing the P transformation, using the running example introduced in Section 3.3.

Specifying Probability Distributions. To automate the P transformation, the π and δ functions in Π must be specified in a machine-processable way. This can be achieved by using the following Maude module DISTR-LIB, which defines a *user-extensible* probability distribution library:

```
fmod DISTR-LIB is
  sort RFun .                                              *** functions on reals
  op _[_] : RFun Real -> Real  .                           *** function application

  op uniform : Real Real -> RFun [ctor] .                  *** min, max
  op exponential : Real -> RFun [ctor] .                   *** rate: lambda
  ops normal lognormal : Real Real -> RFun [ctor] .        *** mean: mu, sd: sigma
  op weibull : Real Real -> RFun [ctor] .                  *** shape: k, scale: lambda
  op zipfian : Real Real -> RFun [ctor] .                  *** skew: s, cardinality: n

  vars X MIN MAX RATE : Real .
  eq uniform(MIN,MAX)[X] = if MIN <= X and X <= MAX then 1.0 / (MAX - MIN) else 0.0 fi .
  eq exponential(RATE)[X] = e^(-RATE * X) .
  ...
endfm
```

This module includes equational definitions for six commonly used parametric probability distributions: the uniform, exponential, normal, lognormal, Weibull, and Zipfian distributions. These distributions are defined as elements of a sort RFun, for real-valued functions, with a function application operator _[_]. Users can use this data type to define additional distributions.

Specifying Π. To specify Π, we must specify the modulation functions δ_l and the map associating rewrite rules (and the initial state) to their corresponding probability distributions and modulation functions. For our QUERY running example, this is done in the following module PI-QUERY:

```
mod PI-QUERY is including DISTR-LIB + QUERY .
  op delta-reply : Oid Oid Id Key Map{Key,Data} -> RFun .    *** 'delta' for rule 'reply'
  eq delta-reply(O,O',ID,K,DB)[D] = distance(O,O') * D .
  ...  *** delta functions for the other rules

  op tpls : -> Tuples .          *** Rule-specific tuples
  *** Here delay follows the lognormal distribution parametric on message payload:
  eq tpls = ['reply,lognormal(size(DB[K]),0.1),delta-reply(O,O',ID,K,DB)] ;;
            ['init,exponential(0.1)] ;;
            ...  *** tuples for the other rules
            [nonexec] .
endm
```

where **nonexec** declares a non-executable equation.

For example, in $\delta_{\texttt{reply}}$, written delta-reply in Maude, the sampled delay, whose lognormal probability distribution is parametric on the message's payload (given by the function size), is further modulated according to the distance (function distance) between the sender and receiver.

The mapping from rule labels to their distributions and modulation functions is given as a set of ;;-separated tuples of the form $[l, \pi_l(\vec{x}), \delta_l(\vec{x}, o, \texttt{O':Oid}, \texttt{MC:Content})]$, indicating that the message delays in rule l follow the probability distribution $\pi_l(\vec{x})$ with the modulation δ_l, or of the form $[l, \pi_l(\vec{x})]$ when δ_l is the identity function. In δ_l, the term o for the sender is taken directly

from the left-hand side of rule l, while the variables `O'` and `MC`, for the receiver and message content, respectively, are introduced on the fly as they may not be present in the rule (see, e.g., rule `issue`).

A rule label is represented as a quoted identifier (of sort `Qid`), e.g., `'reply`, where `'init` is the special label for the initial state. The constant `tpls` then defines the mapping Π.

The Resulting System. We exemplify the P transformation by showing how the rule `reply` has been transformed into a rewrite rule `reply.p` and a probabilistic rewrite rule $\texttt{delay}_{\texttt{reply.1}}$.

```
rl [reply.p] :
   {(to O from O' : read(ID,K))  < O : Server | database : DB > OBJS DMS | T}
=> {< O : Server | > delay-reply(O,O',ID,K,DB,sort(to O' from O : reply(ID,DB[K]))
    OBJS DMS | T} .
```

```
rl [delay_reply.1] :
   {delay-reply(O,O',ID,K,DB,(MSG ; ML)) CF | T}
=> {delay-reply(O,O',ID,K,DB,ML) [T + delta-reply(O,O',ID,K,DB)[D],MSG] CF | T}
    with probability D := lognormal(size(DB[K]),0.1) .
```

The initial state `initconf` is transformed into:

```
{ objs delay-init(sort((to c1 : (read(1,k1) :: read(2,k3))) (to c2 : read(3,k2)))) | 0.0 }
```

where *objs* denotes the client and server objects in `initconf`. The global clock is initialized to `0.0`. The initial messages are assigned different delays sampled from the exponential distribution with the rate `0.1` (given by `['init,exponential(0.1)]` in `tpls` (Π)):

```
rl [delay_init.1] : {delay-init(MSG ; ML) CF | T}
            =>   {delay-init(ML) [T+D,MSG] CF | T} with probability D := exponential(0.1) .

rl [delay_init.1] : delay-init(nil) => null .
```

5 CORRECTNESS OF THE P TRANSFORMATION

Does $\mathcal{R}_\Pi$ support *correct* quantitative analysis of $\mathcal{R}$? Most statistical model checkers require that the system is *purely probabilistic*. In Section 5.1 we show that $\mathcal{R}_\Pi$ is purely probabilistic by proving that it satisfies the *absence of nondeterminism* property. Intuitively, this holds for three reasons: (i) since the distributions in Π are *continuous*, the probability that any two messages are delivered at the same time is 0; (ii) rules in $\mathcal{R}$ are *locally deterministic* (see Section 3.2); (iii) internal actions are applied *eagerly*. In Section 5.2 we show that $\mathcal{R}$ and $\mathcal{R}_\Pi$ are *semantically consistent*: $\mathcal{R}_\Pi$'s behaviors *faithfully model* those of $\mathcal{R}$ *without any loss of information*. Section 5.2 formalizes this by proving that $\mathcal{R}_\Pi$ and $\mathcal{R}$ are related by a *stuttering simulation map*, mapping states in $\mathcal{R}_\Pi$ to states in $\mathcal{R}$.

5.1 Absence of Nondeterminism (AND)

Suppose that in the theory $\mathcal{R}_\Pi$ we: (i) execute each non-probabilistic rewrite rule in $\mathcal{R}_\Pi$ by the standard rewriting logic methods (as supported by Maude), and (ii) execute each probabilistic rewrite rule l in $\mathcal{R}_\Pi$ with matching substitution θ by choosing the value for the extra variable of message delay in its right-hand side by sampling the probability distribution $\pi_l(\vec{x}_l\theta)$ (π_{init} for the rule associated to $\texttt{delay}_{init}$). Then, the set of states reachable from `init(`*initconf*`)` and their transitions define a *purely probabilistic* system, in the sense that the following AND property holds.

Definition 5.1 (AND). With probability 1, for any reachable state there is at most one rewrite rule applicable, with a unique matching substitution θ; i.e., two different rules, or the same rule but with different matching substitutions, can never be applied to a reachable state.

AND is a property enjoyed by the states reachable from the initial state `init(`*initconf*`)` that can be proved by induction on the length of a path from `init(`*initconf*`)` by proving that:

- It holds for $\texttt{init}(\mathit{initconf})$.
- For applications of *non-probabilistic* rules, if it holds for a given reachable state, then it always holds for the next state.
- For applications of *probabilistic* rules, if it holds for a given reachable state, then it also holds for the next state $\pi_l(\vec{x}_l)\theta$-*almost surely* [Grimmett and Stirzaker 2001] (resp. $\pi_{init}(\vec{x}_{init})\theta$-*almost surely*), i.e., with probability 1.

Theorem 5.2. *The P-transformed module $\mathcal{R}_\Pi$ satisfies the AND property.*

The proof of Theorem 5.2 is given in the technical report [Liu et al. 2022a].

5.2 Faithful Behavioral Correspondence (FBC)

The simulations of $\mathcal{R}_\Pi$ from $\texttt{init}(\mathit{initconf})$ faithfully model corresponding behaviors of $\mathcal{R}$ from $\mathit{initconf}$, i.e., sequences of state transitions. The simulations of $\mathcal{R}_\Pi$ do not usually model *all* behaviors of $\mathcal{R}$, since $\mathcal{R}$ can have behaviors impossible in $\mathcal{R}_\Pi$. For example, in $\mathcal{R}_\Pi$ some messages may always arrive to a given object *before* other messages, due to their different communication delays, but in the asynchronous model $\mathcal{R}$ they might arrive in any order. By "faithful behavioral correspondence" we do *not* mean a bisimulation between $\mathcal{R}_\Pi$ and $\mathcal{R}$. We mean that a *simulation* between them exists. Moreover, such a simulation should not be expected to be in lockstep: we should look for a *stuttering* simulation (see Section 2). Since $\mathcal{R}$ is executable while $\mathcal{R}_\Pi$ is not, for a simpler apples-to-apples comparison we define a theory transformation $\mathcal{R}_\Pi \mapsto \mathit{NdEnv}(\mathcal{R}_\Pi)$, where we call $\mathit{NdEnv}(\mathcal{R}_\Pi)$ the *nondeterministic envelope* of $\mathcal{R}_\Pi$, such that: (i) any state transition corresponding to a simulation step for $\mathcal{R}_\Pi$ has a corresponding state transition in $\mathit{NdEnv}(\mathcal{R}_\Pi)$, and (ii) we define a function h from states in $\mathit{NdEnv}(\mathcal{R}_\Pi)$ to configurations in $\mathcal{R}$ that is a stuttering simulation of $\mathit{NdEnv}(\mathcal{R}_\Pi)$ by $\mathcal{R}$.

The $\mathcal{R}_\Pi \mapsto \mathit{NdEnv}(\mathcal{R}_\Pi)$ Transformation. The theory $\mathit{NdEnv}(\mathcal{R}_\Pi)$ transforms $\mathcal{R}_\Pi$ as follows. The rewrite rules are all the same, except for the probabilistic rules in $\mathcal{R}_\Pi$, namely, rules of the form $\texttt{delay}_{l.1}$, $l \in L \cup \{\mathit{init}\}$, which are replaced by rules of the form:

$$\begin{aligned}&\texttt{[delay}_{l.1}\texttt{]} : \{\,\texttt{delay}_l(\vec{x}_l\texttt{, (to O from O' : MC) ; ML) CF | T}\,\}\\ &\qquad \texttt{=>}\ \{\,\texttt{delay}_l(\vec{x}_l\texttt{, ML) [T +}\ \delta_l(\vec{x}_l\texttt{,O,O',MC)(D), (to O from O' : MC)] CF | T}\,\}\ \textbf{if}\ \texttt{D} \in \overline{X}_{f_l(\vec{x}_l)}\ .\end{aligned}$$

where the parametric set $\overline{X}_{f_l(\vec{x}_l)}$ is instantiated for each $\theta = \{\vec{x}_l \mapsto \vec{a}\}$ to the set $\overline{X}_{f_l(\vec{a})}$, i.e., to the closure under limits of the *support set* $X_{f_l(\vec{a})}$ of the density function $f_l(\vec{a})$ specifying the probability distribution $\pi_l(\vec{a})$, as defined in Section 4.1. $\mathit{NdEnv}(\mathcal{R}_\Pi)$ is a *nondeterministic envelope* of $\mathcal{R}_\Pi$ in the following sense: any rewrite $u \longrightarrow v$ obtained by simulating $\mathcal{R}_\Pi$ (see Section 6), so that u and v are reachable from init_P, is also a rewrite $u \longrightarrow v$ in $\mathit{NdEnv}(\mathcal{R}_\Pi)$. If the rewrite $u \longrightarrow v$ uses a non-probabilistic rule, this follows from the definition of $\mathit{NdEnv}(\mathcal{R}_\Pi)$. If a probabilistic rule of the form $\texttt{delay}_{l.1}$, $l \in L \cup \{\mathit{init}\}$, is used, this follows from the definition of $\overline{X}_{f_l(\vec{a})}$, which, as explained in the assumptions on π and δ, forces any $\texttt{D}$ sampled from $\pi_l(\vec{a})$ to belong to $\overline{X}_{f_l(\vec{a})}$. In summary, all simulation-based behaviors of $\mathcal{R}_\Pi$ are therefore *contained* in the behaviors of $\mathit{NdEnv}(\mathcal{R}_\Pi)$.

To prove the FBC property, we show that the behaviors of $\mathit{NdEnv}(\mathcal{R}_\Pi)$ "faithfully model" those of $\mathcal{R}$ because there is a stuttering simulation $h : (\mathit{NdEnv}(\mathcal{R}_\Pi), \texttt{init}(\mathit{initconf}))^\bullet \longrightarrow (\mathcal{R}, \mathit{initconf})^\bullet$. We need to understand what states reachable for init_P look like in $\mathit{NdEnv}(\mathcal{R}_\Pi)$. They look just like states of types (1)–(4) for states reachable from init_P by simulating $\mathcal{R}_\Pi$, in the proof of Theorem 5.2 (see [Liu et al. 2022a]), except that in all types (1)–(4) we drop the requirement that "different delayed messages in *dmsgs* have different delivery times," since this need not longer hold in $\mathit{NdEnv}(\mathcal{R}_\Pi)$.

The desired stuttering simulation map is provided by a function $h : Reach(\texttt{init}(initconf)) \to T_{\Sigma/E\cup B,Config}$, where $(\Sigma/E \cup B)$ is the underlying equational theory of $\mathcal{R}$, which is defined by cases according to the types of reachable states in $Reach(\texttt{init}(initconf))$ as follows:

(1) $h(\{qobjs\ dtsk\ dmsgs \mid t\}) = (qobjs\ msgs(dtsk)\ undel(dmsgs))$, and $h(\{qobjs\ odo\ dtsk\ dmsgs \mid t\}) = (qobjs\ odo\ msgs(dtsk)\ undel(dmsgs))$, where: (i) $msgs(\texttt{delay}_l(\vec{x}_l, ml)) = list2mset(ml)$ with *list2mset* the function sending a list of messages to its associated multiset of messages; and (ii) $undel(dmsgs)$ erases all delays and delay operators and keeps the messages.
(2) $h(\{qobjs\ dmsgs \mid t\}) = (qobjs\ undel(dmsgs))$.
(3) $h(\{qobjs\ msg\ dmsgs \mid t\}) = (qobjs\ msg\ undel(dmsgs))$.
(4) $h(\{qobjs\ odo\ dmsgs \mid t\}) = (qobjs\ odo\ undel(dmsgs))$.

Theorem 5.3. $h : (\text{NdEnv}(\mathcal{R}_\Pi), \texttt{init}(initconf))^\bullet \to (\mathcal{R}, initconf)^\bullet$ *is a stuttering simulation.*

In the technical report [Liu et al. 2022a] we furthermore prove that simulations cannot stop prematurely: any terminating state in $NdEnv(\mathcal{R}_\Pi)$ is mapped by h to a terminating state in $\mathcal{R}$.

6 SIMULATING $\mathcal{R}_\Pi$: THE *Sim* TRANSFORMATION

$\mathcal{R}_\Pi$ is a *non-executable* model. For SMC analysis, $\mathcal{R}_\Pi$ must be transformed into an *executable* rewrite theory $Sim(\mathcal{R}_\Pi)$ that *simulates* the experiments that randomly produce delays d governed by probability distributions $\pi_l(\vec{a})$ for the rules l in $\mathcal{R}_\Pi$. $Sim(\mathcal{R}_\Pi)$ is obtained by applying the *inverse transform method* (see, e.g., [Grimmett and Stirzaker 2001; Rubinstein and Kroese 2017]) that generates sequences of values $\{d_n\}$ enjoying the statistical properties of random sequences for the distribution $\pi_l(\vec{a})$. We show that a general definition of the inverse function π^{-1} together with Lemmas 6.1–6.2 in Section 6.1 ensure that $Sim(\mathcal{R}_\Pi)$, as defined in Section 6.2, correctly simulates $\mathcal{R}_\Pi$. The format in Section 6.3 supports the automation of the $\mathcal{R}_\Pi \mapsto Sim(\mathcal{R}_\Pi)$ transformation.

6.1 The Inverse Transform Method

We first explain the inverse transform method for the simpler case of a strictly increasing distribution function π; then treat the general case where π is just increasing. Since for all distribution functions of the form $\pi_l(\vec{a})$ for our probabilistic rewrite theory $\mathcal{R}_\Pi$ we have $\pi_l(\vec{a})(x) = 0$ for all $x < 0$, we can disregard negative numbers and consider distribution functions $\pi : [0, +\infty) \mapsto [0, 1]$. Since π is the integral of a density function f, π is always an *increasing* function, i.e., $0 \le x \le y$ implies $\pi(x) \le \pi(y)$. The simplest case is when π is *strictly increasing*, i.e., $0 \le x < y$ implies $\pi(x) < \pi(y)$. Then, $\pi : [0, +\infty) \mapsto [0, 1)$ is both bijective and a *homeomorphism*, because the inverse function $\pi^{-1} : [0, 1) \mapsto [0, +\infty)$ is continuous, since $\pi(A)$ is open for any open $A \subseteq [0, +\infty)$. Consider the *uniform distribution* on $[0, 1)$. It has density function $f(x) = 1$, $0 \le x < 1$. Therefore, its distribution function maps each x, $0 \le x < 1$, to $\int_0^x 1\ dt = x$. That is, it is just the *identity function* $id_{[0,1)}$.

The inverse transform method uses the inverse function[9] $\pi^{-1} : [0, 1) \to [0, +\infty)$ to *reduce* the problem of sampling π to the much easier problem of sampling the uniform distribution, which is solved by means of (pseudo-)random number generation algorithms, that is, algorithms that can produce sequences of numbers in $[0, 1]$ enjoying the good statistical properties of a true uniformly distributed random sequence (see, e.g., [Rubinstein and Kroese 2017]). Random sequences sampling π are obtained by: (i) using a random number generator to generate uniformly distributed random numbers $r_i \in [0, 1)$, $i = 1, 2, \ldots, n, \ldots$ and (ii) generating for those r_i the π-distributed random values $\pi^{-1}(r_i)$. When π is increasing but *not strictly so*, π^{-1} is *only a binary relation*. However, π^{-1} *contains* a function, also denoted π^{-1} by abuse of notation, namely, the function:

$$\pi^{-1}(0) = sup\{x|\pi(x) = 0\} \quad and \quad \pi^{-1}(y) = inf\{x|\pi(x) \ge y\} \quad if \quad y > 0.$$

[9]When π is strictly increasing, π^{-1} becomes an isomorphism of probability spaces [Liu et al. 2022a].

We then generate $\{\pi^{-1}(r_i)\}$ as a π-distributed random sequence from the uniformly distributed random sequence $\{r_i\}$ obtained using a random number generator following steps (i)–(ii) as before (see, e.g., [Rubinstein and Kroese 2017], §2.3.1, where the above definition of $\pi^{-1}(y)$ for $y > 1$ is also applied to 0, which can cause $\pi^{-1}(0) \notin \overline{X}_f$; the definition of π^{-1} given above avoids this problem). Depending on whether $\pi(x) = 1$ for some $x \in [0, +\infty)$ or not, in our definition of the $\pi^{-1}(y)$ function we have $0 \leq y \leq 1$, or $0 \leq y < 1$. π^{-1} enjoys the following key property:

Lemma 6.1. *For $\pi : [0, +\infty) \to [0, 1]$ a continuous distribution: (1) if $\exists x \in [0, +\infty)$ s.t. $\pi(x) = 1$, then $\forall y \in [0, 1]$, $\pi^{-1}(y) \in \overline{X}_f$; (2) otherwise, $\forall y \in [0, 1)$, $\pi^{-1}(y) \in \overline{X}_f$.*

Another key property enjoyed by π^{-1} is the following *section property*:

Lemma 6.2. *$\forall y \in dom(\pi^{-1})$, $\pi(\pi^{-1}(y)) = y$.*

Lemma 6.1 ensures that all behaviors of $Sim(\mathcal{R}_\Pi)$ (defined below) are a subset of those of $NdEnv(\mathcal{R}_\Pi)$ and Lemma 6.2 forces $\pi(\pi^{-1}(r_i)) = r_i$, $i \in \mathbb{N}$. They ensure the correct simulation of $\mathcal{R}_\Pi$ by $Sim(\mathcal{R}_\Pi)$. The proofs of both lemmas are given in the technical report [Liu et al. 2022a].

6.2 Defining the $\mathcal{R}_\Pi \mapsto Sim(\mathcal{R}_\Pi)$ Transformation

$Sim(\mathcal{R}_\Pi)$ must support sampling the uniform distribution by random number generation. The key point is that, as explained in Section 2 and in [Clavel et al. 2007], subsequent rewritings of `rand` will produce a sequence of floating point numbers in $[0, 1]$ enjoying the statistical properties of a uniformly distributed random sequence. Since the signature and equations are left unchanged, all that is left to define the $\mathcal{R}_\Pi \mapsto Sim(\mathcal{R}_\Pi)$ transformation is to explain how the rewrite rules of $\mathcal{R}_\Pi$ are transformed into corresponding rewrite rules in $Sim(\mathcal{R}_\Pi)$: All *executable* rules in $\mathcal{R}_\Pi$ are imported unchanged into $Sim(\mathcal{R}_\Pi)$. This only leaves the probabilistic non-executable, rules $\texttt{delay}_{l.1}$ for each l, and $\texttt{delay}_{init.1}$, which are respectively transformed into the executable conditional rules:

$$
\begin{aligned}
&[\texttt{delay}_{l.1}] : \{\texttt{delay}_l(\vec{x}_l, \texttt{(to O from O' : MP) ; ML)}\ \texttt{CF} \mid \texttt{T}\} \\
&\quad => \ \{\texttt{delay}_l(\vec{x}_l, \texttt{ML})\ \texttt{[T} + \delta_l(\vec{x}_l, \texttt{O}, \texttt{O'}, \texttt{MP})\texttt{(D), (to O from O' : MP)]}\ \texttt{CF} \mid \texttt{T}\} \\
&\quad\ \ \texttt{if}\ \ \texttt{D} := \pi_l(\vec{x}_l)^{-1}\texttt{(rand)}\ .
\end{aligned}
$$

$$
\begin{aligned}
&[\texttt{delay}_{init.1}] : \{\texttt{delay}_{init}\texttt{((to O from O' : MP) ; ML)}\ \texttt{CF} \mid \texttt{T}\} \\
&\quad => \ \{\texttt{delay}_{init}\texttt{(ML)}\ \texttt{[T} + \delta_{init}\texttt{(O, O', MP)(D), (to O from O' : MP)]}\ \texttt{CF} \mid \texttt{T}\} \\
&\quad\ \ \texttt{if}\ \ \texttt{D} := \pi_{init}^{-1}\texttt{(rand)}\ .
\end{aligned}
$$

That is, the probabilistic rules $\texttt{delay}_{l.1}$ and $\texttt{delay}_{init.1}$ become executable by sampling their corresponding distributions $\pi_l(\vec{x}_l)$ and π_{init} using the Inverse Transform Method.

6.3 How the $\mathcal{R}_\Pi \mapsto Sim(\mathcal{R}_\Pi)$ Transformation is Automated in Practice

To automate the $\mathcal{R}_\Pi \mapsto Sim(\mathcal{R}_\Pi)$ transformation, the inverse functions $\pi_l(\vec{x}_l)^{-1}$ and π_{init}^{-1} in the transformed rules must be explicitly specified in each case. This is achieved by means of a *user-extensible* module `SAMPLING-LIB` analogous to the `DIST-LIB` module used to specify Π. In `SAMPLING-LIB` we specify the inverse function $\texttt{foo}(\vec{p})^{-1} = \lambda y.\ v(y, \vec{p})$ for each distribution $\texttt{foo}(\vec{p})$, parametric on $\vec{p}$, similarly to how $\texttt{foo}(\vec{p})$ is specified in `DIST-LIB`. The inverse function $\texttt{foo}(\vec{p})^{-1}$ is defined using the operator `sample` as follows: (1) $\texttt{foo}(\vec{p})^{-1}$ is *syntactically* specified as the term $\texttt{sample(foo}(\vec{p})\texttt{)}$, and (2) it is then *semantically* specified by the equation: $\texttt{sample(foo}(\vec{p})\texttt{)}[y] = v(y, \vec{p})$. The following fragment of `SAMPLING-LIB` illustrates this specification format:

```
fmod SAMPLING-LIB is including DISTR-LIB .
  op sample : RFun -> RFun [ctor]  .       *** operator for defining sampling functions
  *** sampling function for the exponential distribution:
  eq sample(exponential(RATE))[RAND] = (- log(RAND)) / RATE .
  *** sampling lognormal, with 'pi' approximating the value of π:
```

```
  eq sample(lognormal(MEAN,SD))[RAND]
   = exp(MEAN + SD * sqrt(- 2.0 * log(RAND)) * cos(2.0 * pi * RAND)) .
  ...   *** sampling functions for the other distributions
endfm
```

We illustrate the automatic transformation $\mathcal{R}_\Pi \mapsto Sim(\mathcal{R}_\Pi)$ for our running example as follows:

Example 6.3. The `delay-reply.1` rule (in Section 4.3) is transformed into:

```
crl [delay-reply.1] :
    {delay-reply(O,O',ID,K,DB,(MSG ; ML)) CF | T }
 => {delay-reply(O,O',ID,K,DB,ML) [T + delta-reply(O,O',ID,K,DB)[D], MSG] CF | T }
   if D := sample(lognormal(size(DB[K]),0.1))[rand] .
```

7 OBSERVING EVENTS: THE M TRANSFORMATION

Many properties cannot be defined on the models $\mathcal{R}$ and $Sim(\mathcal{R}_\Pi)$. For example, estimating the average delay of all queries in the distributed transaction system in Section 3.3 cannot be defined using QuaTEx on $Sim(\mathcal{R}_\Pi)$. One can *manually* modify a given specification to log relevant events. But this would defeat the dual purpose of this work: avoiding developing multiple models manually, and ensuring the semantic consistency between different models. We therefore define a transformation M that logs events during execution. M transforms a model $Sim(\mathcal{R}_\Pi)$ and initial state $\texttt{init}(initconf)$, together with a partial function m from rewrite rules in $\mathcal{R}$ to "events" to be observed, into a theory $M(Sim(\mathcal{R}_\Pi), m)$ with initial state $M(\texttt{init}(initconf))$.

7.1 Defining Events and the Event Map m

Events are terms of a *user-defined* sort `Event`. An *event map* m maps rule labels l in $\mathcal{R}$ to terms $t_l(\vec{x})$ of sort `Event`, where $\vec{x}$ is a subset of the variables occurring in the left-hand side of the rule l.

Example 7.1. To measure the duration of each query in the distributed transaction protocol in Section 3.3, we record the beginning and end of the processing of each query, which happen in the rules `issue` and `finish`, respectively. We can define the corresponding events as follows:

```
  ops startQuery endQuery : Id -> Event [ctor] .
```

The desired event map m is then $\{\texttt{issue} \mapsto \texttt{startQuery(ID)},\ \texttt{finish} \mapsto \texttt{endQuery(ID)}\}$.

7.2 The M Transformation

The effect of the M transformation is to add to the state a new object

```
  < log : Monitor | events : e1 @ t1 ; ... ; en @ tn >
```

where $e_1 \,@\, t_1 \ ; \ \ldots \ ; \ e_n \,@\, t_n$ is a list of time-stamped events $e_i \,@\, t_i$, denoting that an event e_i occurred at time t_i. This can be achieved by adding to $Sim(\mathcal{R}_\Pi)$ the following declarations:

```
  sorts Event TimedEvent TimedEvents .    subsort TimedEvent < TimedEvents .
  op empty : -> TimedEvents [ctor] .
  op _;_ : TimedEvents TimedEvents -> TimedEvents [ctor assoc id: empty] .
  op _@_ : Event Real -> TimedEvent [ctor] .

  class Monitor | events : TimedEvents .
  op log : -> Oid [ctor] .    var TES : TimedEvents .
```

and transforming each rewrite rule l.p, for $l \in \text{dom}(m)$, in $Sim(\mathcal{R}_\Pi)$ to

```
[l.p.m] : {(to o from o' : mc) < o : C | atts > < log : Monitor | events : TES > OBJS DMS | T }
      => {< o : C | atts' > delay_l(x⃗_l, sort(msgs)) newobjs
         < log : Monitor | events : TES ; m(l) @ T > OBJS DMS | T }  [if cond]
```

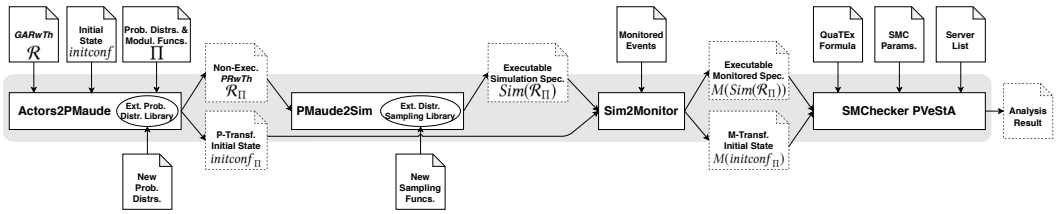

Fig. 1. Architecture of the Actors2PMaude tool (components in rectangular boxes).

(And similarly for object-triggered l.p rules.) The M transformation also adds an object `< log : Monitor | events : empty >` to the initial configuration.

Example 7.2. In our running example, M transforms the rule `finish.p` into the rewrite rule

```
rl [finish.p.m] :  {< log : Monitor | events : TES >
                    < O : Client | queries : read(ID,K) :: QS, waiting : empty,
                                   results : (RS, ID |-> DAT) > OBJS DMS | T}
                =>  {< log : Monitor | events : TES ; (endQuery(ID) @ T) >
                    < O : Client | queries : QS > OBJS DMS | T} .
```

Given an event map m, $(M(Sim(\mathcal{R}_\Pi), m)$ and $Sim(\mathcal{R}_\Pi)$ are bisimilar, with respective initial states $M(\texttt{init}(initconf))$ and $\texttt{init}(initconf)$ (see [Liu et al. 2022a] for the proof):

Theorem 7.3. *The function h that removes the monitoring object from the state is a bisimulation map $h : (M(Sim(\mathcal{R}_\Pi), m), M(\texttt{init}(initconf)) \rightarrow (Sim(\mathcal{R}_\Pi), \texttt{init}(initconf))$.*

7.3 Specifying M in Practice

The module `EVENTS` (see [Liu et al. 2022a]) defines the class `Monitor` and sorts for event maps. The user can then define the sort `Event` and the map m in a module `EVENTS-`*MOD*, where *MOD* is the name of the original module. The event map m is given by specifying the value of the constant `eventMap` as a `;;`-separated set of pairs $[l, t_l(\vec{x})]$, where $\vec{x}$ is a subset of the variables in the rule l.

Example 7.4. The user provides the following module to specify the event map in Example 7.1:

```
mod EVENTS-QUERY is including EVENTS + QUERY .
  ops startQuery endQuery : Id -> Event [ctor] .
  eq eventMap = ['issue, startQuery(ID)] ;; ['finish, endQuery(ID)] [nonexec] .
endm
```

8 THE ACTORS2PMAUDE TOOL

We have implemented in Maude the P, Sim, and M transformations. We have also incorporated the parallelized PVeStA statistical model checker into a single tool Actors2PMaude [Liu et al. 2022b] to analyze the resulting model $M(Sim(\mathcal{R}_\Pi), m)$. This tool consists of the four components in Figure 1:

(1) The Actors2PMaude component implements the P transformation and includes the probability distribution library `DISTR-LIB`. The inputs are defined as in Example 4.3.
(2) The PMaude2Sim component applies the Sim transformation to the resulting module, and includes the extensible sampling library `SAMPLE-LIB`.
(3) The Sim2Monitor component performs the M transformation with the additional input of "interesting" events defined as in Example 7.4.
(4) The SMChecker component carries out the SMC analysis on PVeStA, with simulations obtained by executing the M-transformed model on the initial state.

PVeStA takes as input: the $M(Sim(\mathcal{R}_\Pi), m)$ model and the initial state; the quantitative property defined as a QuaTEx formula; the SMC parameters, i.e., the confidence level and threshold (see Section 2); and a list of servers, each given as "address:port" on which to run the SMC simulations. Each analysis result consists of the expected value of the QuaTEx formula (returned by PVeStA).

The following exemplifies the use of Actors2PMaude on our running example (Section 3.3). We refer to [Liu et al. 2022b] for details about the tool and its usage.

Example 8.1. In our running example, the modules `QUERY`, `INIT-QUERY`, `PI-QUERY`, and `EVENTS-QUERY` specify the *GARwTh*, the initial state, Π, and the event map, respectively. In addition to the SMC parameters and the server list, we also provide the tool with the following QuaTEx formula for statistically measuring the average latency ("processing time") of a query.

```
avgLatForThisSimul() =
    if { s.sat(0) } then { s.rval(1) } else # avgLatForThisSimul() fi ;
eval E[avgLatForThisSimul()] ;
```

The first two lines define the temporal operator `avgLatForThisSimul()`: PVeStA returns the average latency (computed by the Maude function to which `rval(1)` refers) if all queries are finished (checked by the predicate `sat(0)`) in the current state `s`. Otherwise, it evaluates `avgLatForThisSimul()` on the *next* state, where ◯ is denoted by #. The last line computes the expected average latency.

The functions are defined over the *timed events* recorded in the `Monitor` object `log`:

```
*** QuaTEx interaction with Maude
op sat : Nat ClockedState -> Bool .        op val : Nat ClockedState -> Real .
op allFinished : TimedEvents -> Bool .     op avgLatency : TimedEvents -> Real .
eq sat(0,{ OBJS < log : Monitor | events : TES > | T }) = allFinished(TES) .
eq val(1,{ OBJS < log : Monitor | events : TES > | T }) = avgLatency(TES) .
```

The predicate `allFinished` checks whether all queries have been successfully completed:

```
eq allFinished(TES1 ; (startQuery(ID) @ T) ; TES2 ; (endQuery(ID) @ T') ; TES3)
 = allFinished(TES1 ; TES2 ; TES3) .
eq allFinished(empty) = true .          eq allFinished(TES) = false [owise] .
```

The function `avgLatency` computes the average latency of all recorded queries by dividing the total latency (`totalLatency`) by the number of queries (`numberOfQueries`, whose definition is not shown):

```
eq avgLatency(TES) = totalLatency(TES) / numbOfQueries(TES) .

ops totalLatency numberOfQueries : TimedEvents -> Real .
eq totalLatency(TES1 ; (startQuery(ID) @ T) ; TES2 ; (endQuery(ID) @ T') ; TES3)
 = totalLatency(TES1 ; TES2 ; TES3) + (T' - T) .
eq totalLatency(TES) = 0.0 [owise] .
```

Avoiding Model Schizophrenia and Supporting Tool Independence. *Model schizophrenia* is avoided by analyzing the *same* distributed system model $\mathcal{R}$ with respect to both *qualitative* properties in, e.g., temporal logic using $\mathcal{R}$, and *quantitative* ones in some probabilistic temporal logic using $\mathcal{R}_\Pi$. To illustrate the use of different tools for these purposes, we first analyze a distributed transaction system $\mathcal{R}$ (an extension of our running example[10]) with respect to eventual and strong consistency using three well-known LTL model checkers: Maude LTL [Clavel et al. 2007], LTSmin [Kant et al. 2015], and Spot [Duret-Lutz et al. 2016]. We provide the same (nondeterministic, untimed) Maude model to all three model checkers, with the connection between Maude and LTSmin (resp. Spot) supported by the `umaudemc` tool [Rubio et al. 2021]. The three model checkers all verify eventual consistency and find counterexamples to strong consistency.

[10]The main extension consists of adding write operations, since both properties are trivially satisfied with only reads. The new models, and the user-input modules (i.e., `PI-QUERY` and `EVENTS-QUERY`) can be found at [Liu et al. 2022b].

Table 1. Qualitative/quantitative analysis of the extended running example with six tools.

Property	Metric	LTL Model Checker			Statistical Model Checker		
		Maude LTL	LTSmin	Spot	PVeStA	MultiVeStA	MC2
Strong Consistency	Time	**2.366ms**	82.865ms	205.461ms	**8min 35s**	136min 13s	11min 14s
	Mem (Mb)	**32.44**	43.68	67.62	205.48	**136.76**	732.28
Eventual Consistency	Time	**140.057ms**	243.294ms	208.169ms	**1s**	11s	**1s**
	Mem (Mb)	**47.58**	75.38	67.30	148.42	134.25	**39.00**

We then ask *mixed property* questions: *How often* does strong (resp. weak) consistency hold? They can be answered for $\mathcal{R}_\Pi$ using three well-known SMC tools: PVeStA, MultiVeStA [Sebastio and Vandin 2013], and MC2 [Donaldson and Gilbert 2008], which are the only tools that, to the best of our knowledge, can analyze $\mathcal{R}_\Pi$. Other SMC tools require input models incompatible with models like $\mathcal{R}_\Pi$ based on user-definable continuous probability distributions [Agha and Palmskog 2018; Bakir et al. 2017]. The properties were formalized in QuaTEx for PVeStA and MultiVeStA, and in PTLc for MC2. The probabilities estimated by these tools for strong consistency to hold are in the (50%, 60%) interval; for eventual consistency all yield a 100% probability. Since PVeStA has been integrated into our tool, the PVeStA SMC analysis is performed by providing to Actors2PMaude the model used in the qualitative analysis, together with the probability distribution and monitoring information, and the QuaTEx formulas. MultiVeStA has a built-in connection to Maude. For MC2, the simulations were generated with the Maude Python library prior to running its SMC algorithm. The probabilistic model analyzed by MultiVeStA and MC2 is the one generated by Actors2PMaude.

For a fair comparison to MC2, all analyses were performed on a single machine with a 2.1 GHz 32 cores Intel Xeon Silver 4216 processor and 16 GB RAM. For time and memory usage for all properties see Table 1. Further details are provided in [Liu et al. 2022b].

9 APPLICATIONS, AND PREDICTIVE POWER OF SMC-BASED ANALYSIS

9.1 Case Studies

We have applied our tool to 13 applications of different kinds:[11] our running example and its extension; a well-known token ring distributed mutual exclusion algorithm; the N-Tube inter-domain bandwidth reservation algorithm [Weghorn et al. 2022]; the FBAR [Wang et al. 2000] internetworking protocol; the AODV IETF-standardized mobile ad-hoc network protocol [Perkins et al. 2003]; the Apache Cassandra [Cassandra 2022] industrial data storage system and its variant Cassandra-TA [Liu et al. 2017a]; and five variations and extensions of the RAMP state-of-the-art academic transaction system [Bailis et al. 2016; Liu et al. 2016b].

We discuss eight of these applications below and showcase the analysis of:

- *quantitative properties*, such as transaction throughput and data staleness; and
- *mixed properties*, such as the probability of satisfying data consistency guarantees.

Table 2 summarizes the main system features modeled in each case study, with the runtime and memory usage of the analyses. System models were derived in different ways: the Cassandra model was developed by studying both high-level descriptions of Cassandra and the actual Cassandra implementation; the RAMP-F, RAMP-S, and N-Tube models were developed based on the high-level and pseudo-code descriptions published by the system designers; the alternative Cassandra design, RAMP-1PW, and the two replicated RAMP algorithms were formalized directly in Maude.

To parallelize the SMC analysis with PVeStA we employed 50 Emulab machines [White et al. 2002], each with a 2.4 GHz Quad Core Xeon processor and 12 GB RAM. We set the statistical confidence to 95% ($\alpha = 0.05$) and the error margin δ to 0.01. We used the lognormal or exponential

[11] Details about the executable models for the case studies are available at [Liu et al. 2022b].

Table 2. Summary of the case studies, with the size of the QuaTEx properties (including state expressions defined in Maude), and memory usage and execution times of their SMC analysis. Mixed properties are in *italics*. The experimental results are aligned with the properties, since, for the same property, execution time and memory usage are quite similar for different variants of a system.

System Model	Model LOC	Features Modeled	Property	QuaTEx LOC	Mem (Mb)	SMC Time
N-Tube	1088	path-aware architecture; attacker model; bandwidth reservation; reservation renewal	attack resistance	15	105.3	4.9s
RAMP Family		read/write transactions; two-phase commit;	throughput;	24	729.4	21min 2s
RAMP-F	503	multi-version database; database sharding;	*data consistency*;	28	301.8	27min 36s
RAMP-S	521	RA concurrency control mechanisms	*read your writes*	36	320.6	43.5s
RAMP-1PW	517					
Replicated RAMP		the above for RAMP family;	average latency	20	314.4	2min 48s
Sticky HA	1141	replication strategies; multi-datacenter				
Prepare-F HA	994					
Cassandra		key partitioning; key-value store;	data staleness	104	326.9	1hr 40min
Original Design TB	905	consistency levels; quorum replication;				
Alter. Design TA	1010	timestamp policies				

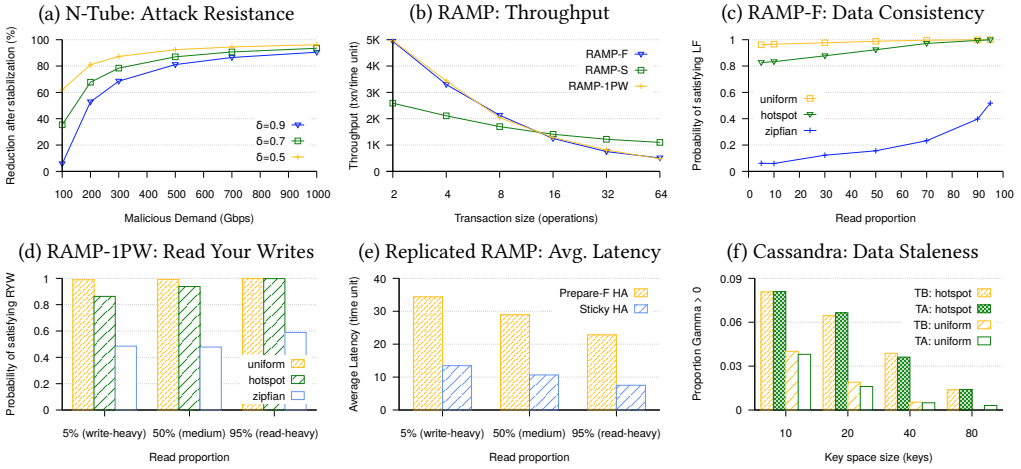

Fig. 2. Statistical model checking analysis results obtained using the Actors2PMaude tool.

distribution characterizing the network latency in realistic deployments [Benson et al. 2010; Ghosh and Ramchandran 2018]. We also considered a linear modulation for message payload size and distance [Günther and Hoene 2005]. The transformation of each model takes around 1 second.

N-Tube. The N-Tube distributed bandwidth reservation algorithm [Weghorn et al. 2022] reserves end-to-end bandwidth along network paths. It guarantees available bandwidth allocations for benign bandwidth demands even during adversarial demand bursts. N-Tube is currently being integrated into the SCION Internet architecture [Giuliari et al. 2021; SCION 2022]. We use Actors2PMaude to generate a probabilistic model and to statistically estimate N-Tube's resistance to attacks.

Figure 2(a) plots the malicious demand reduction after N-Tube stabilizes the bandwidth allocations for different proportions δ of free bandwidth that can be reserved. N-Tube effectively resists increasing malicious power: The reduction increases as the adversaries demand more bandwidth. Our analysis helped the developers validate their hand proofs of N-Tube's correctness properties and quantitatively explore N-Tube's resistance to adversaries prior to the algorithm's implementation.

RAMP. The *Read Atomic Multi-Partition* (RAMP) transaction system [Bailis et al. 2016] provides high-performance operations for large-scale partitioned data stores (in a non-replicated setting) and guarantees *read atomicity* data consistency. RAMP has different versions; RAMP-F and RAMP-S offer different trade-offs between the *size* of the messages and system performance. The developers also sketched alternative designs such as RAMP with one-phase writes (RAMP-1PW).

Figure 2(b) shows the throughput (completed transactions per time unit) of RAMP-F, RAMP-S, and RAMP-1PW, with varying transaction sizes. RAMP-F and RAMP-1PW perform worse than RAMP-S when the number of operations per transaction increases. The Actors2PMaude-generated models refine the manually developed probabilistic versions [Liu et al. 2017b] of the Maude models in [Liu et al. 2016b], which do not account for the effect of message payload size on transmission delays. We therefore now obtain statistical results that are consistent with the Java-implementation-based comparisons between RAMP-F and RAMP-S in Figures 3(c) and 3(f).

We also investigate the probability of satisfying stronger—but not guaranteed—consistency properties. Figure 2(c) shows the probability of RAMP-F satisfying the *latest freshness* (LF) property (reads always return the latest committed writes), and Figure 2(d) shows the probability of RAMP-1PW satisfying the *read your writes* (RYW) property (all writes performed by a client are visible to its subsequent reads); both for different key-access distributions and workloads.

The RAMP developers also sketched two designs, Sticky HA and Prepare-F HA, in a *replicated* setting. Sticky HA is expected to incur lower latency due to the client's *short distance* to its local data center. We used Actors2Maude to transform the Maude models in [Liang and Liu 2021] to performance estimation models. Figure 2(e) shows the transaction latency with varying workloads. By taking into account the effect of the distance between the client and the server on message delays, our performance estimates agree with the conjectures of the RAMP developers.

Cassandra. Apache Cassandra [Cassandra 2022] is a distributed NoSQL database design used by Apple, Netflix, and many other companies. In the Cassandra design (called TB), a coordinator uses *timestamps* to decide which value to return to the client. In [Liu et al. 2017a], an alternative design (called TA) was proposed using the *values* themselves to determine which value to return. The paper [Liu et al. 2017a] compared the designs on the probability of satisfying certain consistency properties. In this paper we compare the two designs in terms of the *staleness* ("age") of the client-observed data. Figure 2(f) shows the SMC results obtained by our tool. We use the *Gamma* metric [Golab et al. 2014] to count the proportion of values involved in consistency anomalies. Our results show that the two designs are incomparable, with varying key-space size under different key-access distributions (*hotspot* and *uniform* [Cooper et al. 2010]). Our results for TB are consistent with the Java-implementation-based evaluations in [Golab et al. 2014] (see also [Liu et al. 2022a]).

9.2 Relating Probabilistic Model Simulations to Actual System Evaluations

SMC-based analysis can be used to *predict* the performance of a system design prior to its implementation. But under what assumptions are model-based performance predictions likely to agree with the actual performance of a subsequent system deployment? We first discuss some caveats to avoid unrealistic expectations about the performance predictions produced by model-based analysis. We next discuss ways of choosing probabilistic models with good predictive power, and how such models can account for other phenomena besides message transmission. We then discuss how to increase the confidence in, and robustness of, SMC-based predictions about a design before it is implemented. Finally, since various factors are unknown before a deployed system is evaluated, we discuss how *a posteriori* information about a system's evaluation can be used to further instantiate a model to reach quantitative predictions reasonably close to actual system evaluation values.

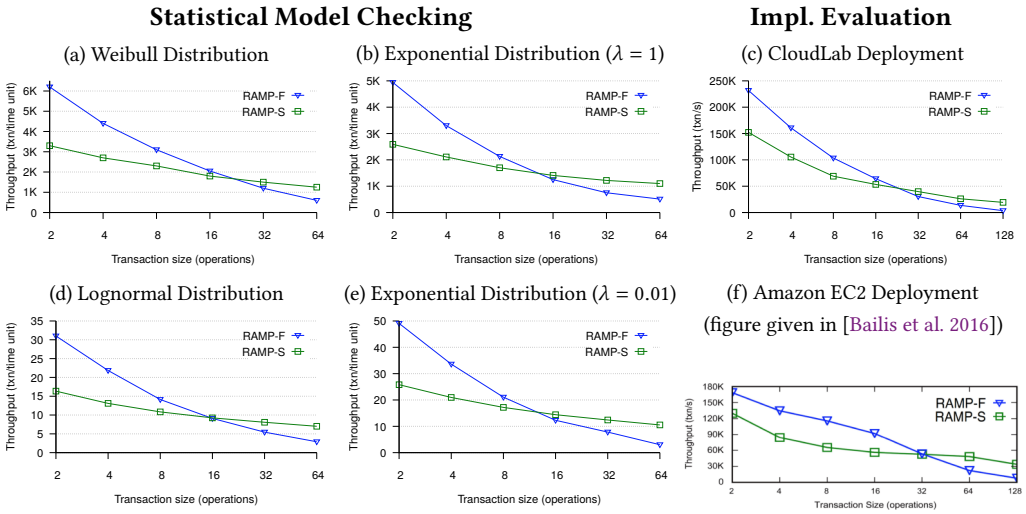

Fig. 3. Throughput comparison of two RAMP algorithms: SMC-based performance estimations and implementation-based performance evaluations.

9.2.1 Caveats. A model's SMC estimation and a deployment's experimental evaluation should exhibit *closely similar behaviors* up to some scaling factors. Taking the two alternative RAMP designs as examples, the conjecture is that RAMP-S outperforms RAMP-F for large transactions. There are at least five reasons why sound predictions will rarely numerically agree with experimental values:

- **Different Platforms.** Figures 3(c) and (f) confirm experimentally on *different platforms* that RAMP-S outperforms RAMP-F for large transactions. Figure 3(c) gives our experimental results on the CloudLab platform [CloudLab 2022], and Figure 3(f) reports those by the RAMP designers using Amazon EC2 [Amazon 2022]. They show that *identical Java implementations* have *different throughput values, crossover points, and curve shapes* on the two platforms.
- **Different Distribution Parameters.** SMC analysis requires specific parameter choices. Figures 3(b) and (e) plot the throughput predictions under two different choices for the λ parameter in the exponential distribution.
- **Different Distributions.** We present SMC-based analyses for the exponential, lognormal, and Weibull distributions. All of them have been shown to faithfully model network delay in different cloud storage systems and data centers [Benson et al. 2010; Ghosh and Ramchandran 2018]. Figures 3(a), (b), (d), and (e) all predict that RAMP-S outperforms RAMP-F for large transactions, *but with different throughput and crossover values.*
- **Different Initial States.** Since Monte Carlo simulation cost increases substantially for large states, the *initial states* used in SMC-based analysis and in experimental evaluations may differ significantly. Often we cannot compare predicted values and experimentally measured ones for the *same* initial states: we can only do so for initial states with similar features.
- **Simulation vs. Actual Time.** The probabilistic model $\mathcal{R}_\Pi$ is a *real-time* model in which quantitative values are relative to a *unit of time*. Thus, many quantitative values predicted by the model can only approximate the values observed in a system's deployment *up to a scaling factor* unknown at design time.

9.2.2 Defining Probabilistic Models with Good Predictive Power. Distributions should be validated empirically to closely model actual message communication delays. To account for additional phenomena—such as message processing time, message size, or actual physical distance between

the sender and the receiver—our framework allows $\mathcal{R}_\Pi$ to include user-defined *modulation functions* δ_l in probabilistic rules. For example, our probabilistic models of RAMP-F and RAMP-S in Table 2 account for the size of message payloads. This is not done in the probabilistic models in [Liu et al. 2017b], which therefore do not exhibit the crossover point of both algorithms.

9.2.3 Prediction Robustness. As explained above, more than one distribution can closely model communication delays. To increase one's confidence in a distributed system design before it is built, one can analyze it, not for a single choice of distribution (and parameters), but for several such sound choices. For instance, the confidence in the conjecture that RAMP-S outperforms RAMP-F for larger transactions would be increased if, as illustrated in our examples, it holds true not just for a single probabilistic model, but for several such models based on different choices of distributions.

9.2.4 A Posteriori Agreement between Model-Based Predictions and Experimental Evaluations. For a given system deployment and evaluation, one can asses *a posteriori* whether a probabilistic model agrees quantitatively with the experimental evaluation. Such an agreement should not be reached by "cooking" the model: it should follow from general principles. For example, among our RAMP models, the Weibull-based one has the best prediction of the *crossover point* compared with the CloudLab evaluation. As shown in Figures 3(a) and (c), the predicted throughput values are still quite different from the values as measured in CloudLab. Since throughput is the number of transactions processed per time unit, this is due to the abstract nature of a time unit (t.u.) in the model, whose scaling factor with respect to actual time in a concrete deployment can only be assessed a posteriori. Slowing time in Figure 3(a) by the factor $1\,t.u. = 34\,s.$, Figure 4 shows a significant *quantitative* agreement between the Weibull-based SMC predictions and the CloudLab evaluation.

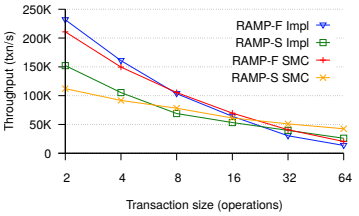

Fig. 4. Comparison between the SMC predictions with Weibull (1 t.u. = 34 s) and the CloudLab evaluation of RAMP-F and RAMP-S.

10 RELATED WORK

The most closely related work is earlier research on using probabilistic rewrite theories and Maude to analyze performance properties of distributed systems [Agha et al. 2005; AlTurki et al. 2009; Alturki and Rosu 2019; Bobba et al. 2018; Eckhardt et al. 2012; Katelman et al. 2008; Liu et al. 2017a,b, 2019a,b, 2020; Skeirik et al. 2013; Urquiza et al. 2019]. Compared to that work, we resolve the "missing links" mentioned in the introduction and support objects with internal actions.

Formal frameworks for quantitative analysis are based on statistical or probabilistic model checking. Uppaal-SMC [David et al. 2015] is an SMC tool for purely probabilistic networks of stochastic timed automata. Whereas our probability distributions can be any parametric continuous ones, in their models the distributions are uniform or exponential. We introduce time on message delays, while in Uppaal-SMC communication is instantaneous and the delays apply to how long a node stays in a location. PRISM [Kwiatkowska et al. 2011; PRISM 2022] supports probabilistic and statistical model checking of MDPs and probabilistic timed automata, restricted to uniform and exponential distributions. For SMC, nondeterministic choices are resolved by simple random choice. $\mathcal{S}$BIP [Mediouni et al. 2018] is a statistical model checker for stochastic real-time BIP models (i.e., stochastic timed automata composed using multi-party interactions). $\mathcal{S}$BIP also supports user-defined probability distributions. There is no unquantified nondeterminism to resolve since BIP's modeling formalism guarantees that all nondeterminism is resolved through stochastic choices over interactions. The main difference between these frameworks and our work is that, to the

best of our knowledge, no semantic mappings have been worked out from models of distributed systems, such as the actor model (or even untimed BIP [Basu et al. 2011]), to these frameworks.

The Modest research group has unified a variety of probabilistic and stochastic automata-based models and tools, including the Modest language and toolset [Hartmanns and Hermanns 2014], and the JANI modeling and tool interaction language [Budde et al. 2017]. A key idea is to derive a rich hierarchy of probabilistic and timed automata-based models as special cases of (networked) stochastic hybrid automata (SHA). Models in Uppaal, Prism, Generalized Stochastic Petri Nets, I/O Stochastic Automata, and Probabilistic Guarded Command languages can be specified and various tools can be invoked for probabilistic or statistical model checking. However, this work does not seem to provide a method of mapping qualitative distributed system models to quantitative ones.

Timed Rebeca [Aceto et al. 2011] extends the Rebeca actor language [Sirjani et al. 2004] to (non-probabilistic) discrete real-time systems, and supports SMC analysis by randomly choosing the next transition based on a uniform distribution [Jafari et al. 2016a]. The PTRebeca extension [Jafari et al. 2016b] allows random assignment to variables of values using discrete probability distributions. Continuous distributions, important for performance analysis, are not supported. We are unaware of any work automatically enriching Rebeca models to timed and probabilistic versions.

More recent work [Ashok et al. 2019; Bogdoll et al. 2011; Henriques et al. 2012; Lassaigne and Peyronnet 2012; Wang et al. 2020] on statistical model checking has relaxed the requirement that models must be purely probabilistic by using techniques like reinforcement learning to reduce an MDP to a purely probabilistic model. In contrast to those approaches, our $\mathcal{R} \mapsto \mathcal{R}_\Pi$ transformation ensures that $\mathcal{R}_\Pi$ is purely probabilistic, thereby making SMC verification simpler and more efficient.

11 CONCLUDING REMARKS

We have addressed the challenge of bridging the gap between qualitative and quantitative models of distributed systems in a *semantically consistent* manner. We have proposed the very general semantic framework of generalized actor rewrite theories for this purpose. By composing the *P*, *Sim*, and *M* theory transformations, we have shown how the gap can be bridged in a semantically consistent way and that the entire process can be automated all the way to the SMC analysis of quantitative properties. We have demonstrated this automation by applying the Actors2PMaude tool to a rich variety of state-of-the-art case studies. This automation makes it much easier for non-specialists to analyze the quantitative properties of complex distributed systems.

Our results and methods are widely applicable and *language-independent*. They have been demonstrated using Maude, PVeStA, and QuaTEx; but they can be applied to any actor-based concurrent programming language to soundly enrich actor-based distributed systems into quantitative models and then analyze them using any suitable statistical model checker and any chosen probabilistic logic. For example, developing language-specific versions of the *P*, *Sim*, and *M* transformations for actor languages like Erlang and Scala Akka would provide automated support for the quantitative analysis of actor systems in these languages and seems a promising topic for future research.

ACKNOWLEDGMENTS

We thank the reviewers for their insightful comments which have significantly improved the paper. The ETH team gratefully acknowledges support from the Werner Siemens-Stiftung. We thank Rubén Rubio for help with the `umaudemc` tool and Ziwei Zhou for help setting up the experiments.

REFERENCES

Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. 2011. Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. *Electronic Proceedings in Theoretical Computer Science* 58 (2011), 1–19.

Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

Gul Agha, Carl Gunter, Michael Greenwald, Sanjeev Khanna, Jose Meseguer, Koushik Sen, and Prasanna Thati. 2005. Formal Modeling and Analysis of DoS Using Probabilistic Rewrite Theories. In *Workshop on Foundations of Computer Security (FCS).*

Gul Agha and Karl Palmskog. 2018. A Survey of Statistical Model Checking. *ACM Trans. Model. Comput. Simul.* 28, 1 (2018), 6:1–6:39.

Gul A. Agha, José Meseguer, and Koushik Sen. 2006. PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. *Electr. Notes Theor. Comput. Sci.* 153, 2 (2006).

Musab AlTurki and José Meseguer. 2011. PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In *CALCO'11 (LNCS, Vol. 6859)*. Springer, 386–392.

M. AlTurki, J. Meseguer, and C. Gunter. 2009. Probabilistic Modeling and Analysis of DoS Protection for the ASV Protocol. *Electr. Notes Theor. Comput. Sci.* 234 (2009), 3–18.

Musab A. Alturki and Grigore Rosu. 2019. Statistical Model Checking of RANDAO's Resilience to Pre-computed Reveal Strategies. In *Formal Methods. FM 2019 International Workshops (LNCS, Vol. 12232)*. Springer, 337–349.

Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. 2015. Theory in Practice for System Design and Verification. *ACM SIGLOG News* 2, 1 (2015).

Amazon. Accessed April, 2022. Amazon EC2. https://aws.amazon.com/ec2/.

Pranav Ashok, Jan Kretínský, and Maximilian Weininger. 2019. PAC Statistical Model Checking for Markov Decision Processes and Stochastic Games. In *Proc. CAV 2019 (LNCS, Vol. 11561)*. Springer, 497–519.

Christel Baier, Luca de Alfaro, Vojtech Forejt, and Marta Kwiatkowska. 2018. Model Checking Probabilistic Systems. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 963–999.

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (2016), 15:1–15:45.

Mehmet Emin Bakir, Marian Gheorghe, Savas Konur, and Mike Stannett. 2017. Comparative Analysis of Statistical Model Checking Tools. In *Membrane Computing - 17th International Conference, CMC 2016 (Lecture Notes in Computer Science, Vol. 10105)*. Springer, 119–135.

Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. 2011. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Softw.* 28, 3 (2011), 41–48.

Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *IMC'10*. ACM, 267–280.

Lucian Bentea and Peter Csaba Ölveczky. 2011. Probabilistic Real-Time Rewrite Theories and Their Expressive Power. In *Proc. FORMATS 2011 (LNCS, Vol. 6919)*. Springer.

Rakesh Bobba, Jon Grov, Indranil Gupta, Si Liu, José Meseguer, Peter Csaba Ölveczky, and Stephen Skeirik. 2018. Survivability: Design, Formal Modeling, and Validation of Cloud Storage Systems Using Maude. In *Assured Cloud Computing*. Wiley-IEEE Computer Society Press, Chapter 2, 10–48.

Jonathan Bogdoll, Luis María Ferrer Fioriti, Arnd Hartmanns, and Holger Hermanns. 2011. Partial Order Methods for Statistical Model Checking and Simulation. In *Proc. FMOODS/FORTE 2011 (LNCS, Vol. 6722)*. Springer, 59–74.

Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. 2017. JANI: Quantitative Model and Tool Interaction. In *Proc. TACAS 2017 (LNCS, Vol. 10206)*. Springer, 151–168.

Apache Cassandra. Accessed April, 2022. Open Source NoSQL Database. https://cassandra.apache.org.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. 2007. *All About Maude*. LNCS, Vol. 4350. Springer.

CloudLab. Accessed April, 2022. CloudLab: Flexible, scientific infrastructure for research on the future of cloud computing. https://www.cloudlab.us/.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SOCC'10*. ACM, 143–154.

Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Danny Bøgsted Poulsen. 2015. Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transf.* 17, 4 (2015), 397–415.

Robin Donaldson and David R. Gilbert. 2008. A Model Checking Approach to the Parameter Estimation of Biochemical Pathways. In *CMSB 2008 (LNCS, Vol. 5307)*. Springer, 269–287.

Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. 2016. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *ATVA 2016 (LNCS, Vol. 9938)*. Springer, 122–129.

Jonas Eckhardt, Tobias Mühlbauer, Musab AlTurki, José Meseguer, and Martin Wirsing. 2012. Stable Availability under Denial of Service Attacks through Formal Patterns. In *Proc. FASE (LNCS, Vol. 7212)*. Springer, 78–93.

Avishek Ghosh and Kannan Ramchandran. 2018. Faster Data-access in Large-scale Systems: Network-scale Latency Analysis under General Service-time Distributions. In *56th Annual Allerton Conference on Communication, Control, and Computing,*

Allerton 2018, Monticello, IL, USA, October 2-5, 2018. IEEE, 757–764.

Giacomo Giuliari, Dominik Roos, Marc Wyss, Juan Ángel García-Pardo, Markus Legner, and Adrian Perrig. 2021. Colibri: a cooperative lightweight inter-domain bandwidth-reservation infrastructure. In *CoNEXT'21*. ACM, 104–118.

Joseph Goguen and José Meseguer. 1992. Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theoretical Computer Science* 105 (1992), 217–273.

Wojciech M. Golab, Muntasir Raihan Rahman, Alvin AuYoung, Kimberly Keeton, and Indranil Gupta. 2014. Client-Centric Benchmarking of Eventual Consistency for Cloud Storage Systems. In *ICDCS*. IEEE Computer Society, 493–502.

G. Grimmett and D. Stirzaker. 2001. *Probability and Random Processes (3rd, Ed.)*. Oxford University Press.

Jon Grov and Peter Csaba Ölveczky. 2014. Formal Modeling and Analysis of Google's Megastore in Real-Time Maude. In *Specification, Algebra, and Software (LNCS, Vol. 8373)*. Springer.

André Günther and Christian Hoene. 2005. Measuring Round Trip Times to Determine the Distance Between WLAN Nodes. In *NETWORKING 2005 (LNCS, Vol. 3462)*. Springer, 768–779.

Hans Hansson and Bengt Jonsson. 1994. A Logic for Reasoning about Time and Reliability. *Formal Asp. Comput.* 6, 5 (1994), 512–535.

Arnd Hartmanns and Holger Hermanns. 2014. The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In *Proc. TACAS 2014 (LNCS, Vol. 8413)*. Springer, 593–598.

David Henriques, João G. Martins, Paolo Zuliani, André Platzer, and Edmund M. Clarke. 2012. Statistical Model Checking for Markov Decision Processes. In *Proc. QEST 2012*. IEEE Computer Society, 84–93.

Ali Jafari, Ehsan Khamespanah, Haukur Kristinsson, Marjan Sirjani, and Brynjar Magnusson. 2016a. Statistical model checking of Timed Rebeca models. *Comput. Lang. Syst. Struct.* 45 (2016), 53–79.

Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, Holger Hermanns, and Matteo Cimini. 2016b. PTRebeca: Modeling and analysis of distributed and asynchronous systems. *Science of Computer Programming* 128 (2016), 22–50.

Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. 2015. LTSmin: High-Performance Language-Independent Model Checking. In *TACAS 2015 (LNCS, Vol. 9035)*. Springer, 692–707.

Michael Katelman, José Meseguer, and Jennifer C. Hou. 2008. Redesign of the LMST Wireless Sensor Protocol through Formal Modeling and Statistical Model Checking. In *Proc. FMOODS 2008 (LNCS, Vol. 5051)*. Springer, 150–169.

Achim Klenke. 2006. *Probability Theory*. Springer.

M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV'11 (LNCS, Vol. 6806)*. Springer, 585–591.

Richard Lassaigne and Sylvain Peyronnet. 2012. Approximate planning and verification for large Markov decision processes. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012*. ACM, 1314–1319.

Lei Liang and Si Liu. 2021. Exploring Design Alternatives for Replicated RAMP Transactions Using Maude. In *TASE*. IEEE, 111–118.

Si Liu. 2022. All in One: Design, Verification, and Implementation of SNOW-Optimal Read Atomic Transactions. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 43 (mar 2022), 44 pages. https://doi.org/10.1145/3494517

Si Liu, Jatin Ganhotra, Muntasir Rahman, Son Nguyen, Indranil Gupta, and José Meseguer. 2017a. Quantitative Analysis of Consistency in NoSQL Key-Value Stores. *Leibniz Transactions on Embedded Systems* 4, 1 (2017), 03:1–03:26.

Si Liu, José Meseguer, Peter Csaba Ölveczky, Min Zhang, and David Basin. 2022a. *Bridging the Semantic Gap between Qualitative and Quantitative Models of Distributed Systems*. Technical Report. http://hdl.handle.net/20.500.11850/563291.

Si Liu, José Meseguer, Peter Csaba Ölveczky, Min Zhang, and David Basin. 2022b. The Actors2PMaude Tool. https://doi.org/10.5281/zenodo.7071693.

Si Liu, Peter Csaba Ölveczky, Jatin Ganhotra, Indranil Gupta, and José Meseguer. 2017b. Exploring Design Alternatives for RAMP Transactions through Statistical Model Checking. In *ICFEM (LNCS, Vol. 10610)*. Springer, 298–314.

Si Liu, Peter Csaba Ölveczky, and José Meseguer. 2015. Formal Analysis of Leader Election in MANETs Using Real-Time Maude. In *Software, Services, and Systems (LNCS, Vol. 8950)*. Springer, 231–252.

Si Liu, Peter Csaba Ölveczky, and José Meseguer. 2016a. Modeling and analyzing mobile ad hoc networks in Real-Time Maude. *J. Log. Algebraic Methods Program.* 85, 1 (2016), 34–66.

Si Liu, Peter Csaba Ölveczky, Muntasir Raihan Rahman, Jatin Ganhotra, Indranil Gupta, and José Meseguer. 2016b. Formal modeling and analysis of RAMP transaction systems. In *SAC*. ACM.

Si Liu, Peter Csaba Ölveczky, Qi Wang, Indranil Gupta, and José Meseguer. 2019a. Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. *Formal Asp. Comput.* 31, 5 (2019), 503–540.

Si Liu, Peter Csaba Ölveczky, Qi Wang, and José Meseguer. 2018. Formal Modeling and Analysis of the Walter Transactional Data Store. In *WRLA (LNCS, Vol. 11152)*. Springer, 136–152.

Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. 2019b. Automatic Analysis of Consistency Properties of Distributed Transaction Systems in Maude. In *TACAS'19 (LNCS, Vol. 11428)*. Springer, 40–57.

Si Liu, Muntasir Raihan Rahman, Stephen Skeirik, Indranil Gupta, and José Meseguer. 2014. Formal Modeling and Analysis of Cassandra in Maude. In *ICFEM (LNCS, Vol. 8829)*. Springer.

Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. 2020. Generating Correct-by-Construction Distributed Implementations from Formal Maude Designs. In *NFM'20 (LNCS, Vol. 12229)*. Springer.

Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, Mahieddine Dellabani, Axel Legay, and Saddek Bensalem. 2018. *S*BIP 2.0: Statistical Model Checking Stochastic Real-Time Systems. In *ATVA'18 (LNCS, Vol. 11138)*. Springer, 536–542.

José Meseguer. 1992. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* 96, 1 (1992), 73–155.

José Meseguer. 1993. A Logical Theory of Concurrent Objects and its realization in the Maude Language. In *Research Directions in Concurrent Object-Oriented Programming*, Gul Agha, Peter Wegner, and Akinori Yonezawa (Eds.). MIT Press, 314–390.

J. Meseguer, M. Palomino, and N. Martí-Oliet. 2010. Algebraic simulations. *J. Log. Algebr. Program.* 79, 2 (2010), 103–143.

J. Meseguer and R. Sharykin. 2006. Specification and Analysis of Distributed Object-Based Stochastic Hybrid Systems. In *HSCC (LNCS, Vol. 3927)*. Springer, 460–475.

Microsoft. 2018. High-level TLA+ specifications for the five consistency levels offered by Azure Cosmos DB. https://github.com/Azure/azure-cosmos-tla.

C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (April 2015), 66–73.

Peter Csaba Ölveczky. 2017. *Designing Reliable Distributed Systems - A Formal Methods Approach Based on Executable Modeling in Maude*. Springer.

Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir R. Das. 2003. Ad hoc On-Demand Distance Vector (AODV) Routing. *RFC* 3561 (2003), 1–37.

PRISM. Accessed April, 2022. *PRISM-SMC*. https://www.prismmodelchecker.org/manual/RunningPRISM/StatisticalModelChecking.

R. Rubinstein and D.P. Kroese. 2017. *Simulation and the Monte Carlo Method (3rd, Ed.)*. J. Wiley & Sons.

Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo. 2021. Strategies, model checking and branching-time properties in Maude. *J. Log. Algebraic Methods Program.* 123 (2021), 100700. https://doi.org/10.1016/j.jlamp.2021.100700

SCION. Accessed April, 2022. *SCION: Scalability, Control, and Isolation on Next-Generation Networks*. https://scion-architecture.net/.

Stefano Sebastio and Andrea Vandin. 2013. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. In *ValueTools*. ICST/ACM, 310–315.

Koushik Sen, Mahesh Viswanathan, and Gul Agha. 2005a. On Statistical Model Checking of Stochastic Systems. In *CAV'05 (LNCS, Vol. 3576)*. Springer.

Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. 2005b. VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems. In *QEST'05*. IEEE Computer Society, 251–252.

Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. 2004. Modeling and Verification of Reactive Systems using Rebeca. *Fundamenta Informaticae* 63, 4 (2004), 385–410.

Stephen Skeirik, Rakesh B. Bobba, and José Meseguer. 2013. Formal Analysis of Fault-tolerant Group Key Management Using ZooKeeper. In *CCGRID*. 636–641.

Abraão Aires Urquiza, Musab A. AlTurki, Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn L. Talcott. 2019. Resource-Bounded Intruders in Denial of Service Attacks. In *CSF*. IEEE, 382–396.

Anduo Wang, Carolyn L. Talcott, Limin Jia, Boon Thau Loo, and Andre Scedrov. 2011. Analyzing BGP Instances in Maude. In *FMOODS'11 (LNCS, Vol. 6722)*. Springer, 334–348.

Bow-Yaw Wang, José Meseguer, and Carl A. Gunter. 2000. Specification and Formal Analysis of a PLAN Algorithm in Maude. In *ICDCS Workshop on Distributed System Validation and Verification 2000*. E49–E56.

Yu Wang, Nima Roohi, Matthew West, Mahesh Viswanathan, and Geir E. Dullerud. 2020. Statistically Model Checking PCTL Specifications on Markov Decision Processes via Reinforcement Learning. In *59th IEEE Conference on Decision and Control, CDC 2020*. IEEE, 1392–1397.

Thilo Weghorn, Si Liu, Christoph Sprenger, Adrian Perrig, and David Basin. 2022. N-Tube: Formally Verified Secure Bandwidth Reservation in Path-Aware Internet Architectures. In *CSF 2022*. IEEE. To appear.

Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*. USENIX Association.

Håkan L. S. Younes and Reid G. Simmons. 2006. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.* 204, 9 (2006), 1368–1409.