Automating the Quantitative Analysis of Distributed Object Systems

Anonymous author
Anonymous affiliation

Abstract

Both *correctness* and *performance* properties are critical requirements of today's sophisticated distributed systems. Formal methods that can support analysis of both such properties from the early design stages are highly desirable. Since different tools and models are needed for different properties, semantic inconsistencies between such models can undermine the trustworthiness of the analyses. Most distributed systems can be naturally modeled as *generalized actor systems*, a generalization of Agha's actors, where actors, besides communicating through messages, can also perform internal "active object" actions; and can be formally specified as generalized actor rewrite theories (*GARwThs*) in Maude. Several research teams have demonstrated that both correctness and performance properties of distributed systems can be analyzed this way. In this paper we first introduce generalized actor systems and their rewrite theories. We then define an automatable transformation where a (non-probabilistic and untimed) *GARwTh*—ideal for correctness analysis—can be *enriched* into a purely probabilistic timed rewrite theory for performance estimation analysis through statistical model checking (SMC), based on user-specified *probability distributions* about the delays of message passing communication. We prove that the untimed and probabilistic system models agree semantically with each other; and that the enriched probabilistic model enjoys the *absence of non-determinism* (AND) property needed for SMC analysis. We present the *Actors2PMaude* tool, which automates the model transformations and provides easy access to the PVeStA parallel SMC tool. We also present a suite of case studies showing the usefulness of *Actors2PMaude* in analyzing the qualitative properties and comparing design alternatives for various distributed system designs, including an industrial data store and a state-of-the-art partitioned transaction system.

2012 ACM Subject Classification

Keywords and phrases ...

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.1

Acknowledgements Anonymous acknowledgements

1 Introduction

This paper presents several theoretical and practical contributions that can help overcome some practical challenges to the early adoption of formal methods supporting analysis of both logical and quantitative properties in the design of distributed systems. Virtually all such systems—from network protocols to distributed algorithms, and from cloud-based transaction systems to distributed cyber-physical systems—can naturally be modeled as systems of *concurrent objects* that communicate through message passing. This work adopts such a concurrent object view of distributed systems.

Problem Description. Logical correctness is a *necessary* but not sufficient condition to reach a high-quality distributed system, since *quantitative properties*, and in particular *performance* properties, are equally important: many applications, even if logically correct, become unusable if their performance is poor. Having a *unified way* of incorporating formal specification and analysis of *both* correctness and performance properties from the early stages of distributed system design, *and* supporting *high levels of automation* for system analysis are the twin problems we are addressing in this work. By a "unified way" we mean avoiding a kind of *modeling schizophrenia*, where separate models are used for logical and performance properties. Besides raising serious concerns about model consistency and model evolution, such modeling schizophrenia can also miss the opportunity of asking important *mixed property questions*, such as the following: "We know that Cassandra logically only satisfies eventual consistency. But *how often* doest it achieve strong consistency in practice?"

Prior Experience. We are building on a long experience by different research teams showing that *formal executable specifications* of distributed object systems in Maude [15] can indeed be incorporated from the early stages of system design; and that they support useful, tool-based analysis of the logical and performance properties of a distributed system design. This is supported in the following way: Maude specifications of distributed systems are rewrite theories [42, 15], which can be analyzed with respect to their logical properties using various model checking and theorem proving tools [44, 15]. But any such rewrite theory, say, $\mathcal{R}$, can be *enriched* with additional probabilistic information Π relevant for performance analysis purposes, yielding a *probabilisitic rewrite theory* [5], say $\mathcal{R}_\Pi$. The point is that performance measures 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. Furthermore, the time "delays" in a distributed system are mostly related to communication, i.e., to message delays, which for analysis purposes can be modeled as following certain *probability distributions* (see, e.g., [12]). Such distributions provide the probabilistic information Π in the enriched model $\mathcal{R}_\Pi$. Using this approach, various research teams have used Maude specifications to analyze both logical and performance properties for a wide range of distributed systems such as, e.g., sensor networks [25], protocols to defend client-sever systems against distributed denial of service (DDoS) attacks [4, 6, 18], and cloud-based data storage systems [13, 33, 35, 37].

Missing Links. Although the above work demonstrates that a unified way of supporting formal analysis of both logical and performance properties of distributed system designs is both possible and useful, this work addresses several important *missing links* at both the *foundational* and the *automation* levels that have up to now remained unresolved and place significant obstacles for the use and wider adoption of these methods in practice:

1. Up to now, the model enrichment process $\mathcal{R} \mapsto \mathcal{R}_\Pi$ has been performed *by hand*. This is awkward, time consuming, and error prone.
2. The probabilistic rewrite theory $\mathcal{R}_\Pi$ is a *non-executable* mathematical model. To analyze system performance, a further theory transformation $\mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ that makes $\mathcal{R}_\Pi$ executable is required. So far, this has also been performed by hand, raising similar concerns about the time and care needed to avoid model inconsistencies.
3. Even assuming that the hand transformations $\mathcal{R} \mapsto \mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ have been correctly performed, the correctness of the performance analysis of $\mathit{Sim}(\mathcal{R}_\Pi)$ crucially depends on $\mathcal{R}_\Pi$ being *purely probabilisitic*, i.e., on its enjoying the *absence of non-determinism* (AND) property. But, so far, no *theory-generic* meta-theorem has yet been proved ensuring that the AND property holds for any general input theory $\mathcal{R}$ (and initial state) that satisfies some general semantic requirements. In fact, the relationship between $\mathcal{R}$ and its probabilistic enrichments has not been studied before: this is one of our key contributions.

Main Contributions. We address these problems, and the problem of providing *high levels of automation* for system specification and analysis, in the following way: (1) In Section 3 we precisely define the input theories $\mathcal{R}$ as belonging to a class of rewrite theories, called *generalized actor rewrite theories* (*GARwTh*s), which are as expressive as possible to specify distributed object systems: they are even more general than the already very general class of

Actor Systems [3]. (2) In Sections 4 and 6 we formally define the theory transformations $\mathcal{R} \mapsto \mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$. (3) In Section 5 we prove that: (3.1) for any $\mathcal{R} \in \mathit{GARwTh}$ and initial states satisfying natural requirements, all probabilistic behaviors in $\mathcal{R}_\Pi$ satisfy the AND property with probability 1, and (3.2) $\mathcal{R}_\Pi$ is semantically related to $\mathcal{R}$ by means of a stuttering simulation. (3) In Section 6 we show that the executable theory $\mathit{Sim}(\mathcal{R}_\Pi)$ used for simulation and SMC analysis purposes is itself semantically consistent with $\mathcal{R}_\Pi$.

All this takes care of foundational issues; but by itself does not provide *high levels of automation*. This has been achieved by: (i) automating the transformations $\mathcal{R} \mapsto \mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ in Maude; and (ii) automating within the same environment the quantitative analysis of $\mathit{Sim}(\mathcal{R}_\Pi)$ in the PVeStA SMC tool [7], which parallelizes SMC analysis for further efficiency and supports specification of properties in the QuaTEx quantitative probabilistic temporal logic [5]. All this automation of theory transformations and SMC analysis is supported by the *Actors2PMaude* tool, which we present in Section 7.

To demonstrate the usefulness of this approach in supporting formal specification and analysis of both logical and performance properties of distributed system design with high levels of automation, we present in Section 8 a collection case studies that apply the *Actors2PMaude* tool to different kinds of distributed system designs. These case studies focus on SMC-based *quantitative* analysis of those designs in *Actors2PMaude*. For lack of space we do not explicitly discuss verification of *logical* properties or analysis of *mixed properties*. But Maude directly supports verification of logical properties by its own LTL model checker [15] and its theorem proving tools [44]; and ample evidence for these two other categories of properties in the case of cloud-based data storage systems can be found in [13, 33, 35, 37].

Without anticipating the more detailed discussion of related work in Section 9, two main points are that: (1) a unified approach to, and automated support for, formal specification and analysis of both logical and performance properties of system designs is both theoretically and pragmatically preferable to managing, and keeping consistency between, different models and tools for analyzing properties in these two different categories; and (2) the faithful quantitative analysis of object-based distributed systems by SMC and probabilistic model checking tools is challenging for some tools, particularly for automata-based ones, because there is a significant mismatch between their input languages and distributed object system, which often have challenging features such as: (i) object attributes with unbounded data strutures; (ii) dynamic object creation and therefore unbounded number of objects; and (iii) the need, not just for continuous probability distributions, but (as we explain in Section 4 and illustrate in Section 8) for *parametric* such distributions, so that a probabilistic transition specified by a probabilistic rewrite rule may need to instantiate the parameters of its distribution on-the-fly, depending on, e.g, the size of the message payload being sent, or the distance between sender and addressee.

2 Preliminaries

2.1 Rewriting Logic and Maude

Maude [15] is a rewriting-logic-based executable formal specification language and analysis tool for distributed systems.

A Maude module specifies a *rewrite theory* [42] $\mathcal{R} = (\Sigma, E, L, R)$, where:

- Σ is an algebraic *signature*, i.e., a set of *sorts*, *subsorts*, and *function symbols*.
- (Σ, E) is a *membership equational logic theory* specifying the system's data types, with E a set of (conditional) equations and membership 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 the syntax of Maude and refer to [15] for details. Operators are introduced with the `op` keyword: `op` f `:` $s_1 \ldots s_n$ `->` s and can have user-definable syntax. An operator with the `ctor` attribute is a constructor. Declaring an operator f with the `frozen` attribute forbids rewriting with rules in all proper subterms of a term having f as its top operator. Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl`, or `crl` for conditional rewrite rules. An equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with lefthand side $f(u_1, \ldots, u_n)$ can be applied. Equations and rules with the `nonexec` attribute are ignored by the Maude rewrite engine, which can for example be used at the metalevel for controlled execution. The mathematical variables are declared with the keywords `var` and `vars`, or can be declared on-the-fly, having the form *var*:*sort*. A module in Maude can be imported as a submodule of another using keywords such as `inc`. A comment is preceded by '`***`' and lasts till the end of the line.

A *class* declaration `class` C `|` att_1 `:` s_1`, ...,` att_n `:` s_n declares a class C of objects with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C is represented as a term `<` $o : C$ `|` $att_1 : val_1, \ldots, att_n : val_n$ `>`, where o (of sort `Oid`) is the object's *identifier*, and val_1 to val_n are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`. A system state is modeled as a term of sort `Configuration`, and consists of a *multiset* of objects and messages. The dynamic behavior of a system is axiomatized by specifying its transition patterns as rewrite rules. For example, the `l`-labeled rule

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

defines a family of transitions in which a message `m(O,w)` is read and consumed by an object `O` of class `C`, whose attribute `a1` is changed to `x + w`, and a new message `m'(O',x)` is generated. Attributes whose values do not change and do not affect the next state, such as `a3`, need not be mentioned in a rule.

In Maude there are two kinds of modules: *functional modules* and *system modules*. The top-level syntax is `fmod` *module* `is ... endfm`, resp., `mod` *module* `is ... endm`, where *module* is the module's name; '`...`' corresponds to all the declarations of submodule importations, sorts, subsorts, operators, variables, equations, rules (for system modules only), and so on.

Random Number Generation. Maude has a built-in function `random(`k`)` that returns the k-th pseudo-random number as a number between 0 and $2^{32} - 1$, and a built-in constant `counter` with an (implicit) rewrite rule `counter => N:Nat.` which rewrites to a different natural number each time it is rewritten. The rule [15]

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

rewrites the constant `rand` (used in Section 7) to a real number between 0 and 1 pseudo-randomly chosen according to the uniform distribution.

2.2 Probabilistic Rewrite Theories and Statistical Model Checking

Probabilistic rewrite theories [5] can express a wide range of models of probabilistic systems, including discrete/continuous-time Markov chains, Markov decision processes, probabilistic timed automata, and object-oriented stochastic real-time systems [15], and have rules of the form

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

where the term t' has new variables $\vec{y}$ disjoint from the variables $\vec{x}$ in term t. The concrete values of the new variables $\vec{y}$ in $t'(\vec{x}, \vec{y})$ are chosen probabilistically according to the probability distribution $\pi(\vec{x})$.

Statistical model checking (SMC) [50, 59] is a formal approach to analyzing probabilistic systems against quantitative temporal logic properties. A probabilistic system with *no un-quantified nondeterminism*, called a *purely probabilistic* system, is a necessary for SMC analysis. SMC uses Monte-Carlo simulations to verify a purely probabilistic system with respect to a path expression up to a certain statistical confidence. The expected value of the path expression (e.g., a performance measure such as system performance or latency) 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 belongs in the interval $[\bar{v} - \frac{\delta}{2}, \bar{v} + \frac{\delta}{2}]$.

Quantative Temporal Expressions (QuaTEx) [5] is a quantitative temporal logic where real-valued state and path functions are used to specify quantitative properties of probabilistic models. QuaTEx supports parameterized recursive function declarations, a standard conditional construct, and a *next* temporal operator for specifying temporal properties.

Given a quantitative property in QuaTEx and a Maude module of a *purely probabilistic* system, SMC analysis can be performed by using *discrete-event-based statistical model checkers* such as the VeStA-family of tools [51, 7, 49]. In particular, our tool (Section 7) integrates PVeStA [7], a parallelization of the VeStA tool [51].

The following shows an example QuaTEx formula and its interaction with the Maude model and PVeStA (see [5] for more details about QuaTEx and the interactions):

```
valueForThisSimul() = { s.rval(1) } ; eval E[ # valueForThisSimul() ] ;
```

where `valueForThisSimul()` returns the value of `rval(1)` (see below) on the final state `s` of the current simulation. The formula then asks for the value on the final state of next (denoted by `#`) simulation. Under the hood, for each simulation, PVeStA invokes Maude via `s.rval(`n`)` to execute the model from the initial state before evaluating the n-th function (in this case $n = 1$) defined in the Maude model on state `s` and returns a real.

2.3 Transition Systems and Stuttering Simulations

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., $\mathit{Reach}(a) = \{a' \in A \mid a \rightarrow^*_{\mathcal{A}} a'\}$, 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 function $\pi : \mathbb{N} \rightarrow A$ such that $\pi(0) = a_0$ and $\forall n \in \mathbb{N}$, $\pi(n) \rightarrow_{\mathcal{A}} \pi(n+1)$.

▶ **Definition 1.** *[41] Given total transition systems $\mathcal{A} = (A, \to_{\mathcal{A}}, a_0)$ and $\mathcal{B} = (B, \to_{\mathcal{B}}, b_0)$, a* stuttering simulation map*, denoted $h : \mathcal{A} \to \mathcal{B}$, is a function $h : Reach(a_0) \to Reach(b_0)$ 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 $\kappa : \mathbb{N} \to \mathbb{N}$ such that, for each $n \in \mathbb{N}$ and each i with $\kappa(n) \leq i < \kappa(n+1)$, $h(\pi(\kappa(n))) = h(\pi(\kappa(i))) = \rho(n)$.*

The states $\pi(i)$, with $\kappa(n) \leq i < \kappa(n+1)$, can be called the "stuttering states" of $\mathcal{A}$ simulated by $\rho(n)$ in $\mathcal{B}$. Definition 1 is conceptually appealing but hard to check directly. As explained in [41], not only for transition systems but for Kripke structures,[1] a more easily checkable characterization by Manolios [39] can be adapted to our setting as the following theorem:

▶ **Theorem 2.** *(Adapted from [39, 41]). Given total transition systems $\mathcal{A}$ and $\mathcal{B}$ with respective initial states a_0 and b_0, a function $h : Reach(a_0) \to B$ with $h(a_0) = b_0$ is a stuttering simulation map $h : \mathcal{A} \to \mathcal{B}$ iff there is a well-founded order $(W, >)$ and a function $\mu : Reach(a_0) \times Reach(b_0) \to W$ such that whenever $h(a) = b$ and $a \to_{\mathcal{A}} a'$, then either (i) there is a state $b' \in B$ such that $b \to_{\mathcal{A}} b'$ and $h(a') = b'$, or (ii) $h(a') = b$ and $\mu(a, b) > \mu(a', b)$.*

We can associate to a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ and an initial state $init \in T_{\Sigma/E,k}$[2] a corresponding total transition system, denoted $(\mathcal{R}, init)^\bullet$ and defined by $(\mathcal{R}, init)^\bullet = (T_{\Sigma/E,k}, \to_{\mathcal{R}}^\bullet, init)$, where $\to_{\mathcal{R}}^\bullet$ is the totalization of the one-step rewrite relation on states $\to_{\mathcal{R}}$ defined by $\mathcal{R}$.

3 Generalized Actor System Rewrite Theories

Generalized Actor Systems. Although distributed systems are highly nondeterministic—due to race conditions and the nondeterminism in the time it takes for messages to travel from sender to receiver—in practice quite often the behavior of each single node is deterministic. For example, *actors* [3] are a natural and popular formalism for modeling distributed systems, where an actor is a uniquely identified computational object that encapsulates a state. When an actor receives a message, it can: change its local state, send messages, and create new actors. According to Agha [3], these actions are *deterministic*; i.e., the new state of the actor, as well as the new generated messages and newly created actors, are uniquely determined by the received message and the actor's internal state.

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.

We 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 naturally be seen as GASs, since we can always trigger an actor's initial action by adding to the initial state an "initialization message" to that actor.

Generalized Actor Rewrite Theories. In the remainder of this section we formalize such (locally-deterministic) generalized actor systems in rewriting logic as *generalized actor rewrite theories* (*GARwThs*), which are standard object-oriented rewrite theories that (together with the initial state) satisfy natural requirements.

---

[1] All the concepts and results in this section extend to Kripke structures. However, for our purposes in this paper total transition systems will suffice.

[2] $T_{\Sigma/E,k}$ denotes the E-equivalence classes of ground Σ-terms of kind k [15].

As usual in object-oriented rewrite theories, the states (terms of sort `Configuration`) are multisets of *objects* (terms of sort `Object`) and *messages* (terms of sort `Msg`), where multiset union is modeled by an associative and commutative operator `__`, with `null` (denoting the empty multiset) its identity element:

```
sorts Object Msg Configuration .    subsorts Object Msg < Configuration .
op __ : Configuration Configuration -> Configuration [ctor assoc comm id: null] .
op null : -> Configuration [ctor] .
```

The rewrite rules in a *GARwTh* must have the form[3] (where parts in '[...]' are optional)

```
[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:

- *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:

  `(to` o_1 `from` $o\theta$ `:` mp_1`)` ... `(to` o_n `from` $o\theta$ `:` mp_n`)`

  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*, of appropriate classes, which are thus added to the existing configuration. The *names* of such new objects must all be distinct and different from those of objects in the current configuration. This can always be ensured by techniques such as those described in [3, 43].

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

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

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

with $1 \leq i_1 < ... < i_k \leq n$, 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 containing an object and a message addressed to it, if the object is *enabled* by a message-triggered rule to receive that message, then it is enabled by a *unique* rule, with a *unique* substitution.
2. In any concrete configuration containing an object, if the object is *enabled* to be rewritten by an object-triggered rule, then it is enabled by a *unique* rule, with a *unique* substitution.
3. *At most one* object in *initconf* is enabled to be rewritten by an object-triggered rule.

[3] Following Maude convention, logical *variables* appearing in rules are written in capital letters, while *terms* which need not be logical variables are written in italics.

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 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.
8. Hierarchical configurations (where an object can contain an "inner" configuration of objects and messages stored in one of its attributes) are allowed, but no rewrite rule can be applied to the inner configurations.

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

A wide range of distributed systems that have been formally modeled and analyzed in Maude are —or can be slightly modified to be— generalized actor rewrite theories, including:

- textbook distributed mutual exclusion algorithms, e.g., the token ring algorithm and Maekawa's voting algorithm [46];
- textbook cryptographic protocols such as the NSPK protocol [46].
- internetworking protocols, e.g., the IETF-standardized Border Gateway Protocol (BGP) [56] and TCP/IP [4] for the Internet, and the academic routing algorithm FBAR [57] for the active internetworks;
- mobile ad-hoc network (MANET) protocols, e.g., the IETF-standardized protocol AODV [31] and the well-known leader election protocol in [30];
- industrial distributed computing services such as Apache Zookeeper [53];
- cloud data storage systems, including the industrial datastores such as Apache Cassandra [36] and Google's Megastore [21], and a collection of state-of-the-art distributed transaction systems, such as RAMP [32], Walter [34], Jessy [33], ROLA [33], and the two-phase-locking-based transaction protocol [37].

4 The P Transformation

Generalized actor rewrite theories ($GARwThs$) specifying distributed systems as generalized actor systems are nondeterministic, non-probabilistic and (typically) untimed, which means that they are not well-suited for quantitative analysis, e.g., by statistical model checking.

Performance measures 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. For the most part, the time "delays" in a general actor system can be attributed to communication; i.e., to message delays.

For analysis purposes, these message delays can conveniently and, reasonably precisely, be seen as following certain *probability distributions* (see, e.g., [12]), possibly also taking into account parameters such as message payload size and distance between sender and recipient.[4]

Our idea to *automatically transform* an untimed nondeterministic generalized actor system into a timed probabilistic system by allowing users to specify the probability distributions Π of the message delays, and then perform an automatic transformation P that enriches the model with communication delays specified by Π.

In Section 4.2 we define this P *transformation* that transforms a nondeterministic untimed *GARwTh* $\mathcal{R}$ and its initial state *initconf*, which together satisfy the requirements 1–8 in Section 3, as well as a family Π of parametric probability distributions that specify the distributions of the *delays* of the generated messages, into a corresponding probabilistic rewrite theory (*PRwTh*) and associated initial state $P(\mathcal{R}, \mathit{initconf}, \Pi)$.

Since statistical model checking with PVeStA assumes that the systems are *purely probabilistic*, i.e., free of unquantified nondeterminism, we prove in Section 5 that the *PRwTh* generated by P is a *purely probabilistic rewrite theory almost-surely*, and also make precise the relationship between the original *GARwTh* $\mathcal{R}$ and the transformed theory $P(\mathcal{R}, \mathit{initconf}, \Pi)$.

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 the former, remember that in a *probabilistic* rewrite rule $[l] : t(\vec{x}) \longrightarrow t'(\vec{x}, \vec{y})$ **if** $\mathit{cond}(\vec{x})$ *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 therefore need to define such a parametric probability distribution $\pi_l(\vec{x})$ from which a message delay value can be sampled.

As already explained in Footnote 4, in some applications we may need to *modulate* this sampled value to define a more complex message delay that takes into account extra factors. 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 `:` $\mathit{msgContent}$`)` is $\delta_l(\vec{x}\theta, o, o', \mathit{msgContent})(d)$, where d is the "basic sampled delayed" obtained from π_l.

Likewise, for messages in the initial state, we also need to define the "basic" delay distribution π_{init} and the modulation function $\delta_{init}(O, O', MC)$. In simpler applications, the functions $\delta_l(\vec{x}_l, O, O', MC))$ and $\delta_{init}(O, O', MC)$ will typically be the identity function on the reals, and the distributions $\pi_l(\vec{x}_l)$, resp. π_{init}, will be used to choose message delays.

In more detail, we assume that: (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 lefthand side (resp. for $l = \mathit{init}$) 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. $f_{init}(x) = 0$). Each such density function then defines a *probability distribution function* $\pi_l(\vec{a}) : \mathbb{R} \to [0, 1]$

[4] Faithful modeling of message delay times taking account of factors such as the above, as well as of, e.g., local processing times, can and will be supported in two ways: (i) by using *parametric* probability distributions whose different 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 some extra factors.

(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}). Note that, 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 ([26], 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 $\mu_l(\vec{a}) = \lambda B \in \mathcal{B}(\mathbb{R}).\ \int_{t\in B} f_l(\vec{a})(t)\ dt$. 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 numbers 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} force the inclusions $\overline{X}_{f_l(\vec{a})} \subseteq [0,+\infty)$, $\overline{X}_{f_{init}} \subseteq [0,+\infty)$. 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*. However, we also assume that (2) such 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., $r < r' \Rightarrow \delta_l(\vec{a}, o, o', mc)(r) < \delta_l(\vec{a}, o, o', mc)(r')$. All this information about the π and δ functions is denoted by Π. It is *specified by the user* to automatically transform a generalized actor rewrite theory $\mathcal{R}$ into a probabilisitic version of it using the P transformation defined below.

4.2 Defining the P Transformation

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

- $\mathcal{R} = (\Sigma, E, L, R)$ is a generalized actor rewrite theory; and
- *initconf* is an initial state consisting of a set of objects and messages, so that $\mathcal{R}$, together with *initconf*, is an *GARwTh* satisfying requirements 1–8 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 Section 4.1. That is, for $l \in L$, $\pi_l(\vec{x}_l)$ is a *continuous probability distribution* for rule l, parametric on $\vec{x}_l$, and $\delta_l(\vec{x}_l, O, O', MC)$ is the *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 P-transformed probabilistic rewrite theory defined in detail below.
- $\mathit{initconf}_\Pi$ is the corresponding P-transformed initial state.

Intuitively, the resulting probabilistic rewrite theory $\mathcal{R}_\Pi$ behaves as follows. To model time, the state has the form `{` *config* `|` *clock* `}`, where *config* is the current configuration (consisting of objects and messages with their delivery times) and *clock* denotes the current time. Object-triggered rewrite rules are applied *eagerly*: time does not advance when an object-triggered rule is enabled. When there is a "ripe" message in the state, a message-triggered rule is applied to read the message. The messages created when applying either an object-triggered or a message-triggered rule are then "prepared to get their delays assigned" by placing them in a "delayed task object." If there is such a "delayed task" in the state, then its messages are assigned delays, one by one, by sampling the corresponding distribution function (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). Finally, when none of these rewrite rules can be applied, a "tick" rewrite rule (labeled `tick`) advances the global time of the system 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 $\mathit{initconf}_\Pi$ obtained by applying P to $(\mathcal{R}, \mathit{initconf}, \Pi)$.

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 make the terms $\pi_l(\vec{x}_l)$, π_{init}, $\delta_l(\vec{x}_l, O, O', MC)$, and $\delta_{init}(O, O', MC)$ definable by equations in E_Π, where $E \subseteq E_\Pi$.

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

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

In addition to objects and messages from $\mathcal{R}$, the multiset *config* may also contain: (i) a *delay-task term* (of sort `DTask`) which holds a list of outgoing messages *yet to be assigned a delay*, and (ii) a set of already *delayed messages* (of sort `DMsgs`, with each of sort `DMsg`), which are not yet ready to be received. We also define a subsort `Objects` for configurations consisting only of actor objects. The following summarizes the sorts and subsort relations added to Σ by P (the multiset union operator `__` for sort `Configuration` is overloaded on the subsorts `DMsgs`, `Msgs` and `Objects`):

```
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 entire state can be decomposed into a term `{`*objects msgs dmsgs dtask* `|` *clock*`}` consisting of (with their cardinality at any time): 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 (≤ 1) denoted by *dtask*.

A sort `MsgList` models a *list* of messages with concatenation operator `;` and identity `nil`:

```
op nil : -> MsgList [ctor] .
op _;_ : MsgList MsgList -> MsgList [ctor assoc id: nil] .
```

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 $\mathit{initconf}_\Pi$:

- A function `sort` that turns a set of messages into a *sorted* list:[5]

  ```
  op sort : Msgs -> MsgList .
  ```

[5] The purpose of sorting messages is explained in Footnote 7. Sorting can be implemented by using Maude's total order on terms in the meta-level.

- For each rule (labeled) l, an operator $\texttt{delay}_l$ 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 $\vec{x}_l = x_1, \ldots, x_n$[6] appearing in the lefthand side of rule l, and a list of messages, and outputs a term of sort `DTask`:

```
op delay_l : s_1 ... s_n MsgList -> DTask [ctor] .
```

- A delayed message of sort `DMsg` has the form `[`*time*`,`*msg*`]` indicating that *msg* (of sort `Msg`) will be delivered at global time *time*:

```
op [_,_] : Real Msg -> DMsg [ctor] .
```

- The predicate `objectEnabled` checks whether an object-triggered rule (type (‡)) is enabled on a set of objects:

```
op objectEnabled : Objects -> Bool [frozen] .
```

  For each object-triggered rule of the form (‡) a (possibly conditional) equation

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

  is generated. The following two equations are also introduced by P:

```
eq objectEnabled(OBJ OBJS) = objectEnabled(OBJ) or objectEnabled(OBJS) .
eq objectEnabled(OBJ) = false [owise] .
```

  The first equation makes `objectEnabled(`*objects*`) true` whenever an object in *objects* can be rewritten by an object-triggered rule; the second equation handles the remaining cases.
- 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 function $\texttt{delay}_{init}$ is declared:

```
op delay_init : MsgList -> DTask [ctor] .
```

4.2.2 The Rewrite Rules R_Π

Each rewrite rule l of the form (†), resp. (‡), in $\mathcal{R}$ is transformed into a rewrite 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]
```

resp.,

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

[6] $\vec{x}_l$ is a *list* of the different variables appearing in the lefthand side of rule l in the order of their first appearance.

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$_l$($\vec{x}_l$, (to O from O' : MP) ; ML)  CF | T }
          =>
             { delay$_l$($\vec{x}_l$,ML)  [T + $\delta_l$($\vec{x}_l$,O,O',MP)(D) , (to O from O' : MP)]  CF | T }
          with probability D := $\pi_l$($\vec{x}_l$)  .
```

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

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

```
[delay$_{l.2}$] : delay$_l$($\vec{x}_l$,nil) => null .
```

Likewise, the following rewrite rules assign delays to the messages in the initial state:[7]

```
[delay$_{init.1}$] : { delay$_{init}$((to O from O' : MP) ; ML)  CF | T }
            =>
               { delay$_{init}$(ML)  [T + $\delta_{init}$(O,O',MP)(D) , (to O from O' : MP)]  CF | T }
            with probability D := $\pi_{init}$  .

[delay$_{init.2}$] : delay$_{init}$(nil) => null .
```

Finally, 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:

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

The function times computes the set of delivery times in the delayed messages DMS. The delayed message [T',MSG] then becomes ready, as MSG, to be consumed. The tick 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.

4.2.3 The Rule Labels L_Π

The set L_Π of rule labels consists of l.p, $\mathtt{delay}_{l.1}$, and $\mathtt{delay}_{l.2}$ for each $l \in L$, and of the labels $\mathtt{delay}_{init.1}$, $\mathtt{delay}_{init.2}$, and tick.

4.3 Applying P: An Example

This section shows how the P transformation can be applied to a generalized actor rewrite theory, given using Maude, specifying a simple protocol for handling "read" requests in a database system where multiple distributed servers may store the same data item.

[7] 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.

4.3.1 The Query Protocol

For each user read request, a client issues read requests to the servers replicating the data item, and stores the result of the read operation, 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 [1] (see also Section 8).

A client buffers the user requests in a queue (attribute `queries`), each of which is a read operation with its identifier on some data item or key (rule `req`). Upon processing the first query `ID` on key `K` in the queue, the client initializes a placeholder `null` for the returned value, looks up the servers replicating the key (from the `replicas` attribute), and propagates the message `read(ID,K)` to them (rule `issue`). Each server (called a replica) replies to the client with the locally stored `<` *value*, *timestamp* `>` pair of the requested key (rule `reply`) with *timestamp* denoting when *value* was written. Upon receiving the reply, the client updates the corresponding record with the "freshest" value (having the `latest` timestamp) it has seen so far, removes that server from its waiting list, and keeps waiting for the rest (rule `update`). Once all responses have been collected (indicated by the `empty` waiting list), 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`:

```
mod QUERY is
inc PMAUDE .

*** sorts and subsorts
sorts Query Queries Key Value Timestamp Data Oid Oids Id Reply .
subsort Reply < Payload .  subsort Query < Queries < Payload .  subsort Nat < Id .
subsort Nat < Timestamp .  subsort Nat < Value .  subsort Oid < Oids .
subsort Msg < Msgs < Configuration .

*** variables declarations
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} .

*** basic data types and operators
op read : Id Key -> Query [ctor] .
op reply : Id Data -> Reply [ctor] .

op nil : -> Queries [ctor] .
op _::_ : Queries Queries -> Queries [assoc id: nil] .

op empty : -> Oids .
op _;_ : Oids Oids -> Oids [assoc comm id: empty] .

op <_,_> : Value Timestamp -> Data [ctor] .
op null : -> Data [ctor] .

op latest : Data Data -> Data .
eq latest(< V,TS >, < V',TS' >) = if TS >= TS' then < V,TS > else < V',TS' > fi .
eq latest(< V,TS >, null) = < V,TS > .
eq latest(null, < V,TS >) = < V,TS > .

*** objects
class Client | queries : Queries, waiting : Oids, read : Data,
               replicas : Map{Key,Oids}, results : Map{Id,Data}.
class Server | database : Map{Key,Data} .

*** msgs
op to_from_:_         : Oid Oid Payload -> Msg .
op to_:_              : Oid Payload     -> Msg .

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 .

*** rewrite rules specifying the protocol dynamics
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 (built-in) map lookup operator is represented with the notation `_[_]`. 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 above corresponding equations.

The following shows an example initial state, specified in the module `INIT-QUERY`, with two clients, having respective incoming requests, and three servers, each storing two keys:

```
mod INIT-QUERY is
  inc QUERY .

ops c1 c2 s1 s2 s3 : -> Oid . ops k1 k2 k3 : -> Key .

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
```

4.3.2 Applying the P Transformation

This section illustrates how we can specify in Maude the input Π to the P transformation, and the results of performing the P transformation, using our database example, that contains three (†) rules (`req`, `reply`, and `update`) and two (‡) rules (`issue` and `finish`).

4.3.2.1 Specifying Π

Specifying Probability Distributions. To automate the P transformation, the π and δ functions in Π must be available in a machine-processable way. This is achieved by a *user-extensible* probability distribution library as a Maude functional module DISTR-LIB:

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

  *** some probability distributions
  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
```

The module DISTR-LIB includes equational definitions for six commonly used parametric probability distributions, namely, 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 beyond the six mentioned above.

How is Π Specified. 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. This is done in the following module PI-QUERY:

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

  ...  *** delta functions and their applications for the other rules

  sorts Tuple Tuples .    subsort Tuple < Tuples .  *** interface for Pi

  op [_,_,_] : Qid RFun RFun -> Tuple [ctor] .
  op [_,_] : Qid RFun -> Tuple [ctor] .
  op empty : -> Tuples [ctor] .
  op _;;_ : Tuples Tuples -> Tuples [ctor comm assoc id: empty] .

  op tpls : -> Tuples .

  *** Rule-specific tuples. In this case delay is logarithmic to the 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
```

We exemplify the specification of the δ_l functions with $\delta_{\texttt{reply}}$, written `delta-reply` in Maude. In this case the sampled delay, whose `lognormal` probability distribution is parametric on the message's payload (determined by function `size`), is further modulated according to the distance (determined by 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 forms:

- `[`l`,`$\pi_l(\vec{x})$`,`$\delta_l(\vec{x}, o,$ `O':Oid,MC:Content`$)$`]`, indicating that the message delays in rule l follow the probability distribution $\pi_l(\vec{x})$ with the modulation δ_l; or
- `[`l`,`$\pi_l(\vec{x})$`]`, when δ_l is the identity function.

In δ_l the term o for the message sender is taken directly from the lefthand side of rule l, while the variables `O'` and `MC`, for the message receiver and content, respectively, are introduced on the fly as they may not be present in the rule (see, e.g., rule `issue` in module `QUERY`).

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

4.3.2.2 The Resulting Rewrite Rules

We exemplify the P transformation on the rules 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) .
```

4.3.2.3 Defining `objectEnabled`

The P transformation defines the predicate `objectEnabled` to hold when one of the *object-triggered* rules (`issue` and `finish`) is enabled (and also adds the two equations distributing this predicate over the objects):

```
ceq objectEnabled(< O : Client | queries : read(ID,K) :: QS, waiting : empty,
                                 replicas : R, results : RS >) = true
    if not $hasMapping(RS,ID) .

eq objectEnabled(< O : Client | queries : read(ID,K) :: QS, waiting : empty,
                                results : (RS, ID |-> DAT)  >) = true .
```

4.3.2.4 Transforming the Initial State

The initial state `initconf` in module `INIT-QUERY` 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 list of the initial messages is to be processed with the sampled delays by the following lifting rules defining the operator `delay-init`:

```
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 .
```

In this case, the clients are triggered to start at different times following the exponential distribution with the rate `0.1` (given by `['init,exponential(0.1)]` in `tpls` (Π)).

5 Correctness of the P Transformation

In this section we establish P's correctness by proving two fundamental properties: the *absence of non-determinism* in the P-transformed module, which is necessary for statistical model checking analysis, and its *faithful behavioral correspondence* with the input module.

As $\mathcal{R}_\Pi$ is a *non-executable* mathematical model, when reasoning about these properties, we actually refer to its *simulations*, which provide a computational model for $\mathcal{R}_\Pi$ and make statistical model checking possible. Simulation of probabilistic systems is a well-known subject, see, e.g., [20, 48]. In Section 6 we define a theory transformation $\mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ making $\mathcal{R}_\Pi$ executable by Monte Carlo simulation. The two key points about simulations are that: (i) we can faithfully simulate the behaviors of probabilistic systems (i.e., $\mathcal{R}_\Pi$ in our case) by *sampling* its associated probability distributions; and (ii) we can often *reduce* the problem of sampling a distribution function to the much simpler problem of sampling a *uniform* distribution using a (pseudo) random number generator.

5.1 Absence of Non-Determinism (AND)

If 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 [15]), and (ii) execute each probabilistic rewrite rule labeled in $\mathcal{R}_\Pi$ with matching substitution θ by choosing the value for the extra variable of message delay in its righthand 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*) defines a *purely probabilistic* system, in the sense that the following AND property holds.

▶ **Definition 3** (AND). *For any reachable state there is at most one rewrite rule applicable, with a unique matching substitution θ, that can be applied to reach a next state, i.e., two rules, or the same rule but with different matching substitutions, can never be applied to a reachable state.*

More precisely, AND is an *inductive property*:
- It holds for `init`(*initconf*).
- For applications of *non-probabilistic* rules, if it holds for the given reachable state, then it always holds for the next state.
- For applications of *probabilistic* rules, if it holds for the given reachable state, then it also holds for the next state $\pi_l(\vec{x}_l)\theta$-*almost surely* [20] (resp. $\pi_{init}(\vec{x}_{init})\theta$-*almost surely*), i.e., with probability 1.

▶ **Theorem 4.** *The P-transformed module $\mathcal{R}_\Pi$ satisfies the AND property.*

Proof. Let us introduce some notation. We call the state of a concrete object o, *quiescent* if no rule of type (‡) can apply to it. Let *qobjs* denote a set of quiescent (ground) objects. Let *dtsk* denote a ground term of sort `DTask`. Let *dmsgs* denote a set of (ground) delayed messages. Let *odo* denote a concrete object enabled for the application to it of a rule of type (‡). Let *msg* denote a (ground) message.

The proof is by induction on the length of the sequence of rewrite steps starting from `init`(*initconf*) and reaching a given state, showing that (with probability 1) all states reachable from `init`(*initconf*) in a simulation are of one of the following four types:

1. { *qobjs dtsk dmsgs* | t} or { *qobjs odo dtsk dmsgs* | t}, where the set *dmsgs* could be `null`, and where different delayed messages in *dmsgs* have different delivery times.
2. { *qobjs dmsgs* | t}, where the set *dmsgs* could be `null`, and where different delayed messages in *dmsgs* have different delivery times.
3. { *qobjs msg dmsgs* | t}, where the set *dmsgs* could be `null`, and where different delayed messages in *dmsgs* have different delivery times.
4. { *qobjs odo dmsgs* | t}, where either *qobjs* or *dmsgs* (or both) could be `null`, and where different delayed messages in *dmsgs* have different delivery times.

We just need to prove, by induction on the number n of rewrite steps from `init`(*initconf*) that: (for $n = 0$) `init`(*initconf*) (i) belongs to one of these types of configurations and (ii) satisfies the AND property, and (assuming that (i) and (ii) holds for n) then (i) and (ii) also hold for a configuration obtained by $n + 1$ rewrite steps from `init`(*initconf*) almost surely, i.e., with probability 1. The *base case* $n = 0$ is trivial: `init`(*initconf*) is a configuration of type (1) with empty set of delayed messages. Furthermore, only one of the two lifting rules using the distribution π_{init} (the second rule would apply in the degenerate case when there are no messages in *initconf*). Furthermore, if *initconf* contains at least one message, the resulting configuration is also of type (1), since there is at most one delayed message in it; otherwise, it is of type either (2) or (4) with $dmsgs =$ `null`.

Let us now prove the *induction step* by type cases:

Type (1). Only a unique lifting rule involving an operator either $\mathtt{delay}_l$ or $\mathtt{delay}_{init}$ can apply. If it is the second lifting rule, corresponding to an empty list of to-be-delayed messages inside the `delay` operator, we obtain a configuration of type either (2) or (4) with same set of delayed messages. Otherwise, (non-empty list of to-be-delayed messages inside the `delay` operator) the first lifting rule is applied with ground matching substitution $\theta : \{\vec{x}_l \mapsto \vec{a}\}$ by sampling a delay d from the distribution $\pi_l(\vec{a})$ (resp. π_{init}). The result is a new configuration (as we shall show of type (1) almost surely), where the list of to-be-delayed messages inside the `delay` operator is the rest of the previous list, and where a new delayed message of the form $[t + \delta(\vec{a}, o, o', mc)(d), (\mathtt{to}\ o\ \mathtt{from}\ o' : mc)\theta]$ has been added to the previous set *dmsgs* of delayed messages. The only pending issue is whether the delivery time $t + \delta(\vec{a}, o, o', mc)(d)$ is different from the (by assumption all different) delivery times in *dmsgs*. But how could $t + \delta(\vec{a}, o, o', mc)(d)$ be equal to one of those delivery times? Remember that $f =_{def} \lambda\mathtt{D}.\ \delta(\vec{a}, o, o', mc)(\mathtt{D}) : [0, +\infty) \to [0, +\infty)$ is strictly monotonic, and therefore *injective*, and therefore *bijective* on $f([0, +\infty))$, with inverse bijection $f^{-1} : f([0, +\infty)) \to [0, +\infty)$. Let $t + d_1, \ldots, t + d_k$ be the delayed times in *dmsgs* such that $d_1, \ldots, d_k \in f([0, +\infty))$. Then, this duplication of delivery times can only happen iff $\delta(\vec{a}, o, o', mc)(d) \in \{d_1, \ldots, d_k\} \cap f([0, +\infty))$. Let $\{d_{j_1}, \ldots, d_{j_q}\} = \{d_1, \ldots, d_k\} \cap f([0, +\infty))$, with $1 \leq j_1 < \ldots < j_q \leq k$. Then, $\delta(\vec{a}, o, o', mc)(d) \in \{d_{j_1}, \ldots, d_{j_q}\}$ iff $d \in \{f^{-1}(d_{j1}), \ldots, f^{-1}(d_{j_q})\}$. But since the distribution $\pi_l(\vec{a})$ (resp. π_{init}) is *continuous* and obtained by *integration* of a corresponding density function, we of course must have

$\mu_l(\vec{a})(\{f^{-1}(d_{j1}),\ldots,f^{-1}(d_{j_q})\}) = 0$, and therefore $\mu_l(\vec{a})(\mathbb{R}\setminus\{f^{-1}(d_{j1}),\ldots,f^{-1}(d_{j_q})\}) = 1$, so that the delivery time $t + \delta(\vec{a}, o, o', mc)(d)$ is different from all the delivery times in *dmsgs* iff $d \in \mathbb{R}\setminus\{f^{-1}(d_{j1}),\ldots,f^{-1}(d_{j_q})\}$, i.e., with probability 1. Therefore, the resulting configuration is indeed of type (1) with probability 1.

Type (2). The only rewrite rule that can be applied to a state of the form $\{$ *qobjs dmsgs* $\mid t\}$ is the `tick` rule, and for this to happen we must have *dmsgs* $\neq$ `null`. Since, by assumption, the objects in *qobjs* are all quiescent, the negation of the `objectEnabled` predicate is `true`. Furthermore, there is a *unique matching substitution* θ with which the `tick` rule can be applied. This is because, since all delivery times in *dmsgs* are different, there is only one delayed message with *smallest possible* delivery time, and therefore a unique matching substitution satisfying the condition in the `tick` rule. Therefore, the resulting state is a state of type (3).

Type (3). We have a state of the form $\{$ *qobjs msg dmsgs* $\mid t\}$ with the delivery times in *dmsgs* all different. To such a state, only a rule of the form $l.p$ with l of type (†) in $\mathcal{R}$ can be applied. Note that all delayed messages are generated by delaying messages in either the list `sort`(*init.msgs*), or the list `sort`(*msgs*) of messages generated by the application of a rule of the form $l'.p$, associated to a rule l' in $\mathcal{R}$ of type (†) or (‡), by the assumptions about *init* and assumption (6) (Section 3). The addressee of *msg* must be exactly one of the quiescent objects in *qobjs*. But then, assumption (5) (Section 3) forces the rule $l.p$ and the matching substitution θ to be unique. Furthermore, the resulting state is of Type (1).[8]

Type (4). We have a state of the form $\{$ *qobjs odo dmsgs* $\mid t\}$, to which only a rule of the form $l.p$, with l of type (‡) in $\mathcal{R}$ can be applied, specifically to rewrite *odo*. But by assumption (b), $l.p$ and its matching substitution θ must be unique. Furthermore, the resulting state is of Type (1).

◀

5.2 Faithful Behavioral Correspondence (FBC)

We now give a precise statement and proof of how the simulations of $\mathcal{R}_\Pi$ from `init`(*initconf*) faithfully model corresponding behaviors of $\mathcal{R}$ from *initconf*.

The first thing to note is that if, by a "behavior of $\mathcal{R}$" we understand a sequence of state transitions of the given actor system, then it is unreasonable to expect that the simulations of $\mathcal{R}_\Pi$ will model *all* behaviors of $\mathcal{R}$. This is because $\mathcal{R}$ describes a fully asynchronous system that may easily have behaviors impossible in $\mathcal{R}_\Pi$. For example, in $\mathcal{R}_\Pi$ certain kinds of messages may always arrive to a given object *before* other kinds of messages, due to their different communication delays, but such restrictions do not generally exist in the asynchronous model $\mathcal{R}$. So, the kind of "faithfulness" we are looking for need not be expressed as a bisimulation. It may however be expressed as a *simulation*.

Second, regarding which system simulates which, since $\mathcal{R}$ has more behaviors, it should be able to *simulate* any behavior exhibited by a simulation of $\mathcal{R}_\Pi$. However, the simulation is not in lockstep: we should look for a *stuttering* simulation (see Section 2 and [39, 41]). Since $\mathcal{R}$ is executable while $\mathcal{R}_\Pi$ is not, for a simpler[9] apple-to-apple comparison we define a theory tranformation $\mathcal{R}_\Pi \mapsto NdEnv(\mathcal{R}_\Pi)$, where we call $NdEnv(\mathcal{R}_\Pi)$ the *non-deterministic envelope*

---

[8] This is so even if the set *msgs* of messages to be delayed is `null`, since the $\mathtt{delay}_l$ operator can be applied even to an empty list of to-be-delayed messages.

[9] In Section 6, an executable rewrite theory $Sim(\mathcal{R}_\Pi)$ that simulates $\mathcal{R}_\Pi$ is defined. But $Sim(\mathcal{R}_\Pi)$ involves unnecessary details —such as the choice of a random number generator and a seed— that are irrelevant for a semantic comparison with $\mathcal{R}$, and too specific to be usable in a semantic correctness proof.

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 $NdEnv(\mathcal{R}_\Pi)$, and (ii) we define a function h from states in $NdEnv(\mathcal{R}_\Pi)$ to configurations in $\mathcal{R}$ that is a stuttering simulation of $NdEnv(\mathcal{R}_\Pi)$ by $\mathcal{R}$.

Finally, there is a remaining point that all this would still miss: any simulation behavior of $\mathcal{R}_\Pi$ could be viewed as a behavior in $NdEnv(\mathcal{R}_\Pi)$ and shown to faithfully model a behavior in $\mathcal{R}$ via h. But what about *termination*? In particular, could there be a final state in $NdEnv(\mathcal{R}_\Pi)$ mapped by h to a non-final state in $\mathcal{R}$? This could be quite problematic, because it might mean that, as far as $\mathcal{R}$ is concerned, a simulation of $\mathcal{R}_\Pi$ has stopped *prematurely* for no good reasons. We shall show that this cannot happen.

The $\mathcal{R}_\Pi \mapsto$ *NdEnv*$(\mathcal{R}_\Pi)$ Tranformation. The signature of $NdEnv(\mathcal{R}_\Pi)$ is the same as that of $\mathcal{R}_\Pi$, except for the additions described in Footnote 10 below. The rewrite rules are all the same, except for the only probabilistics rules in $\mathcal{R}_\Pi$, namely, rules of the form $\texttt{delay}_{l.1}$, $l \in L \cup \{init\}$, which are replaced by rules of the form:[10]

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

where the parametric set $\overline{X}_{f_l(\vec{x}_l)}$ was defined in part (4) of the input requirements for P describing Π. Recall that, for each $\theta = \{\vec{x}_l \mapsto \vec{a}\}$, $\overline{X}_{f_l(\vec{a})}$ is the closure under limits of what we called the *support* of the density function $f_l(\vec{a})$ defining the probability distribution $\pi_l(\vec{a})$. The theory $NdEnv(\mathcal{R}_\Pi)$ is indeed a *non-deterministic envelope* of $\mathcal{R}_\Pi$ in the following straightforward sense: any rewrite $u \to v$ obtained by simulating $\mathcal{R}_\Pi$, so that u and v are reachable from $init_P$, is also a rewrite $u \to v$ in $NdEnv(\mathcal{R}_\Pi)$. If the rewrite $u \to v$ uses a non-probabilistic rule, this follows from the definition of $NdEnv(\mathcal{R}_\Pi)$. Instead, if a probabilistic rule of the form $\texttt{delay}_{l.1}$, $l \in L \cup \{init\}$, is used, this follows from the definition of $\overline{X}_{f_l(\vec{a})}$, which, as explained in the specification of Π, 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 $NdEnv(\mathcal{R}_\Pi)$.

Hence, to prove the FBC property we only need to show that the behaviors of $NdEnv(\mathcal{R}_\Pi)$ "faithfully model" those of $\mathcal{R}$. In particular, we need to prove the following two properties:

(a) the simulation-based behaviors of $\mathcal{R}_\Pi$ cannot stop *prematurely*;
(b) there is a stuttering simulation
$$h : (NdEnv(\mathcal{R}_\Pi), \texttt{init}(initconf))^\bullet \to (\mathcal{R}, initconf)^\bullet.$$

Since all behaviors we care about are those involving reachable states in either system, the first order of business is to better understand what states reachable for $init_P$ look like in $NdEnv(\mathcal{R}_\Pi)$. The answer is simple: they look just like states of types (1)–(4) for states

[10] Admittedly, $NdEnv(\mathcal{R}_\Pi)$ is not executable for two reasons: (1) the choice of $\texttt{D}$ is (uncountably) non-deterministic; and (2) for each $\theta = \{\vec{x}_l \mapsto \vec{a}\}$, $\overline{X}_{f_l(\vec{a})}$ is an infinite set of real numbers which is not even definable in the signature of $\mathcal{R}_\Pi$. However, abandoning of course any pretensions of a finitary (or even countable!) equational specification, we can view the parametric predicate $\texttt{D} \in \overline{X}_{f_l(\vec{x}_l)}$ as *syntactic sugar* for the parametric predicate $\in_{f_l(\vec{x}_l)}(\texttt{D}) = \texttt{true}$, which is equationally defined for each $\theta = \{\vec{x}_l \mapsto \vec{a}\}$ by the uncountable set of ground equations: $\{\in_{f_l(\vec{a})}(x) = \texttt{true} \mid x \in \overline{X}_{f_l(\vec{a})}\} \cup \{\in_{f_l(\vec{a})}(x) = \texttt{false} \mid x \in \mathbb{R} \setminus \overline{X}_{f_l(\vec{a})}\}$. For proving that a stuttering simulation $h : NdEnv(\mathcal{R}_\Pi) \to \mathcal{R}$ exists, the non-executability of $NdEnv(\mathcal{R}_\Pi)$ is of course irrelevant.

reachable from $init_P$ by simulating $\mathcal{R}_\Pi$ in the proof of Theorem 4, 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 $NdEnv(\mathcal{R}_\Pi)$. This has a direct relevance for property (a) about the absence of premature termination of simulations of $\mathcal{R}_\Pi$ because: (i) A reachable state obtained by simulation of $\mathcal{R}_\Pi$ is terminating (no further probabilistic transition is possible) iff it is terminating in $NdEnv(\mathcal{R}_\Pi)$ (no further non-deterministic transition is possible). This is because: (a.1) all transitions are the same, except for the more general form of rule $\mathtt{delay}_{l.1}$ in $NdEnv(\mathcal{R}_\Pi)$. But the extra generality of rule $\mathtt{delay}_{l.1}$ only regards the *choice* of the specific `D`, and does not involve any difference in the enabledness status of either version of $\mathtt{delay}_{l.1}$ when applied to a given state. And (a.2) only states of type (2) may fail to be enabled. But for such states, the `tick` rule will be enabled iff the set *dmsgs* is non-`null`. In summary, the only terminating states reachable from $\mathtt{init}(initconf)$ in $NdEnv(\mathcal{R}_\Pi)$ are states of the form: $\{\ qobjs \mid t\}$, where the objects *qobjs* are all *quiescent*. But, as explained below, our desired simulation map h, in this case will give us: $h(\{\ qobjs \mid t\}) = qobjs$, and *qobjs* is clearly a terminating state in $\mathcal{R}$. This settles property (a): no premature terminations are possible.

Regarding property (b), to find the desired stuttering simulation h we first need to define a function $h : Reach(\mathtt{init}(initconf)) \to T_{\Sigma/E\cup B, Config}$, where $(\Sigma/E \cup B)$ is the underlying equational theory of $\mathcal{R}$, and then prove that it is indeed a stuttering simulation. The function h is defined by cases according to the types of reachable states in $Reach(\mathtt{init}(initconf))$ as follows:

1. $h(\{\ qobjs\ dtsk\ dmsgs \mid t\}) = (qobjs\ msgs(dtsk)\ undel(dmsgs))$, as well as $h(\{\ qobjs\ odo\ dtsk\ dmsgs \mid t\}) = (qobjs\ odo\ msgs(dtsk)\ undel(dmsgs))$, where: (i) $msgs(\mathtt{delay}_l(\vec{x}_l, ml)) = list2mset(ml)$ with *list2mset* the function sending a list of messages to its associates 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. $h(\{\ qobjs\ odo\ dmsgs \mid t\}) = (qobjs\ odo\ undel(dmsgs))$.

We then only need to prove the following theorem:

▶ **Theorem 5.** *$h : (\mathrm{NdEnv}(\mathcal{R}_\Pi), \mathtt{init}(initconf))^\bullet \to (\mathcal{R}, initconf)^\bullet$ is a stuttering simulation.*

Proof. By Theorem 2 (Section 2), it is enough to define a well-founded order $(W, >)$ and a function $\mu : Reach(\mathtt{init}(initconf)) \times Reach(initconf) \to W$ such that whenever $h(u) = v$ and $u \to^\bullet_{NdEnv(\mathcal{R}_\Pi)} u'$, then either (1) there is a $v' \in M$ s.t. $v \to^\bullet_M v'$ and $h(u') = v'$, or (2) $h(v') = u$ and $\mu(u, v) > \mu(u', v)$. We define $(W, >)$ as $(\mathbb{N} \times \mathbb{N}, >)$, with $>$ the lexicographic order on pairs of numbers. $\mu(u, v) = \mu(u) = (n, m)$, where: (i) if u has a subterm of the form $\mathtt{delay}_l(\vec{x}_l, ml)$, then $n = length(ml) + 1$, and otherwise $n = 0$, and (ii) m is the cardinality of the (possibly empty) set *dmsgs* of delayed messages in u.

By property (a), if u is a terminating state, so is $h(u)$ and condition (1) holds trivially. So we only need to look at transitions $u \to_{NdEnv(\mathcal{R}_\Pi)} u'$ and consider which rule is applied:

- For an application of a rule of the form $\mathtt{l}_p$, condition (1) clearly holds with the corresponding (†) or (‡) rule in $\mathcal{R}$.
- For an application of a rule of the form $\mathtt{delay}_{l.1}$ or $\mathtt{delay}_{l.2}$, $l \in L \cup \{init\}$, the first component of (n, m) decreases by 1, and the second component either increases by 1 or remains the same, and $h(u) = h(u')$, so condition (2) holds.

- For an application of the tick rule, $n = 0$, m decreases by 1, and $h(u) = h(u')$, so condition (2) holds.

◀

6 Simulating $\mathcal{R}_\Pi$: The Sim Transformation

The probabilistic rewrite theory $\mathcal{R}_\Pi$ associated by the P transformation to a generalized actor theory $\mathcal{R}$ is a *non-executable* mathematical model, since each application of a probabilistic rewrite rule l of $\mathcal{R}_\Pi$ requires a real number d, a *time delay*, obtained as the result of an *experiment* governed by the rule's probability distribution $\pi_l(\vec{a})$, where $\vec{a}$ is the instantiation of the rule's lefthand side variables $\vec{x}$. Therefore, to perform any kind of formal analysis on $\mathcal{R}_\Pi$ —including any statistical model checking analysis— we need to transform $\mathcal{R}_\Pi$ into an *executable* rewrite theory $Sim(\mathcal{R}_\Pi)$ which can *simulate* in an executable manner the experiments that randomly produce delays d governed by probability distributions $\pi_l(\vec{a})$ for the various rules l in $\mathcal{R}_\Pi$. This transformed theory is obtained by applying well-known *sampling methods* that generate sequences of values $\{d_n\}$ enjoying the statistical properties of random sequences for the distribution $\pi_l(\vec{a})$. Then, we can use $Sim(\mathcal{R}_\Pi)$ to perform Monte Carlo simulations of $\mathcal{R}_\Pi$.

To motivate the definition of the theory transformation $\mathcal{R}_\Pi \mapsto Sim(\mathcal{R}_\Pi)$ and explain the Inverse Transform Method on which it is based (see, e.g., [48]), we first need four basic notions: (1) A *measurable space* is a pair $(U, \mathcal{F})$ where $\mathcal{F} \subseteq \mathcal{P}(U)$ is a σ-algebra. In all our applications U will be a topological space and $\mathcal{F}$ the Borel σ-algebra $\mathcal{B}(U)$ generated by U's topology (for definitions of $\mathcal{B}(U)$ and σ-algebra see, e.g., [26]). (2) A *measurable map* $f : (U, \mathcal{F}) \to (V, \mathcal{G})$ between measurable spaces is a function $f : U \to V$ such that for each $A \in \mathcal{G}$, $f^{-1}(A) \in \mathcal{F}$. A useful fact is that if U and V are topological spaces and $f : U \to V$ is a *continuous* function, then $f : (U, \mathcal{B}(U)) \to (V, \mathcal{B}(V))$ is a measurable map and the mapping

$$(f : U \to V) \mapsto (f : (U, \mathcal{B}(U)) \to (V, \mathcal{B}(V)))$$

defines a *functor* from the category of topological spaces to that of measurable spaces (see [26], Theorem 1.88). (3) A *probability space* is a triple $(U, \mathcal{F}, \mu)$, where $(U, \mathcal{F})$ is a measurable space, and μ is a probability measure. Since in all our applications U will be a subspace of the topological space $\mathbb{R}$ and $\mathcal{F}$ will be its Boreal σ-algebra $\mathcal{B}(U)$ —so that the probability measure μ will be uniquely determined by a distribution function π (see [26], Theorem 1.60)— the explicit definition of a probability measure (for which see, e.g., [48, 20, 26]) is not needed. (4) A *map of probability spaces* $f : (U, \mathcal{F}, \mu) \to (V, \mathcal{G}, \mu')$ is a function $f : U \to V$ such that: (i) $f : (U, \mathcal{F}) \to (V, \mathcal{G})$ is a measurable map, and (ii) f is probability-measure-preserving, i.e., for each $A \in \mathcal{G}$, $\mu'(A) = \mu(f^{-1}(A))$. Probability spaces and their maps form a category with the usual function composition.

We are now ready to explain the basic facts about the Inverse Transfer Method. We do so by focusing on the nicest relevant case for our applications, and then briefly summarize its generalization to all other relevant cases. Since, by construction, in 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 $x < 0$, so that all delays d sampled from $\pi_l(\vec{a})$ must satisfy $d \geq 0$, we can disregard negative numbers and just consider distribution functions of the form: $\pi : [0, +\infty) \mapsto [0, 1]$ in the topological subspace $[0, +\infty)$. By the construction of π as the integral of a density function f, π is always an *increasing* function, i.e., $0 \leq x \leq y$ implies $\pi(x) \leq \pi(y)$. The nicest, and often occurring, case is when π is *strictly increasing*, i.e., $0 \leq x < y$ implies $\pi(x) < \pi(y)$. In such a case, the continuous function $\pi : [0, +\infty) \mapsto [0, 1)$ is not only bijective, but is actually a *homeomorphism*, i.e., the inverse function $\pi^{-1} : [0, 1) \mapsto [0, +\infty)$ is also continuous. This follows easily from the fact that π is an open map, i.e., $\pi(A)$ is open for any open $A \subseteq [0, +\infty)$. Let us now consider the *uniform distribution* on $[0, 1)$. This distribution 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* $id_{[0,1)}$. Let μ (resp. μ_U) denote the probability measure uniquely defined by π (resp. by $id_{[0,1)}$). The essential idea about the Inverse Transfer Method is based on the following lemma:

▶ **Lemma 6.** *A continuous, strictly increasing distribution function $\pi : [0, +\infty) \mapsto [0, 1)$, associated to a probability density function f, defines an* isomorphism *of probability spaces:*

$$\pi : ([0, +\infty), \mathcal{B}([0, +\infty)), \mu) \to ([0, 1), \mathcal{B}([0, 1)), \mu_U)$$

Proof. Since π is a homeomorphism, by functoriality it is also an isomorphism of measurable spaces $\pi : ([0, +\infty), \mathcal{B}([0, +\infty))) \to ([0, 1), \mathcal{B}([0, 1)))$. This means that there is a function $\pi^{-1} : \mathcal{B}([0, 1)) \ni A \mapsto \pi^{-1}(A) \in \mathcal{B}([0, +\infty))$ with inverse function $\pi : \mathcal{B}([0, +\infty)) \ni B \mapsto \pi(B) \in \mathcal{B}([0, 1))$. All we have left to prove is that: (a) π is probability-measure-preserving, i.e., $\mu \circ \pi^{-1} = \mu_U$, and (b) π^{-1} is probability-measure-preserving, i.e., $\mu_U \circ \pi = \mu$. To prove (a), by Example 1.56[11] in [26], it is enough to show $\mu(\pi^{-1}(A)) = \mu_U(A)$ for A an open interval in $[0, 1)$ of the form:[12] (x, y), with $0 < x < y \le 1$. Defining, by convention, $\pi^{-1}(1) = -\infty$ and $\pi(+\infty) = 1$, we have, $\mu(\pi^{-1}((x, y))) = \mu((\pi^{-1}(x), \pi^{-1}(y)) = \pi(\pi^{-1}(y)) - \pi(\pi^{-1}(x)) = y - x = id_{[0,1]}(y) - id_{[0,1]}(x) = \mu_U(x, y)$, as desired. Property (b) then follows easily from (a), since we have $\mu = \mu \circ \pi^{-1} \circ \pi = \mu_U \circ \pi$, as desired. ◀

The Inverse Transfer Method applies Lemma 6 as follows. We use the inverse *isomorphism* of probability spaces $\pi^{-1} : ([0, 1), \mathcal{B}([0, 1)), \mu_U) \to ([0, +\infty), \mathcal{B}([0, +\infty)), \mu)$ to *reduce* the nontrivial problem of sampling π to the much easier problem of sampling the uniform distribution, which has an easy solution 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., [48]). In other words, 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 $\pi^{-1}(r_i)$.

In the general case where the continuous distribution π is increasing but not necessarily strictly so, π^{-1} need not be a function, but only a binary relation. However, the relation π^{-1} *contains* a function, also denoted π^{-1} by abuse of language, namely, the function:

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

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 the same steps (i)–(ii) as before (see, e.g., [48], §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 above avoids this problem). Depending on whether $\pi(x) = 1$ for some $x \in [0, +\infty)$ or not, then we respectively have $0 \le y \le 1$, or $0 \le y < 1$, in the above definition of the $\pi^{-1}(y)$ function.

[11] Applied to a subspace $V \subseteq \mathbb{R}$ with probability space $(V, \mathcal{B}(V), \mu)$ by using the inclusion map of probability spaces $j : (V, \mathcal{B}(V), \mu) \hookrightarrow (\mathbb{R}, \mathcal{B}(\mathbb{R}), \mu \circ j^{-1})$, with $j : V \ni x \mapsto x \in \mathbb{R}$. Also, since π is continuous and defined by integration, the choice of intervals (a, b) instead of $(a, b]$ is immaterial.

[12] $[0, x)$ is open in $[0, 1)$; but this case is covered by $(0, x)$, since $\mu_U([0, x)) = \mu_U((0, x)) = x$.

We of course would like to have $\pi^{-1}(y) \in \overline{X}_f$, as it should be the case for a correct simulation. This will ensure that all behaviors (rewrite sequences) of $\mathit{Sim}(\mathcal{R}_\Pi)$ (defined below) are a subset of those of $\mathit{NdEnv}(\mathcal{R}_\Pi)$. This is indeed the case:

▶ **Lemma 7.** *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, $\forall y \in [0, 1)$, $\pi^{-1}(y) \in \overline{X}_f$.*

Proof. By contradiction. Suppose in either case (1) or (2) that $\pi^{-1}(y) \notin \overline{X}_f$. Then there is an interval I, open in the subspace $[0, +\infty)$, with $I \subseteq [0, +\infty) \setminus \overline{X}_f$ and $\pi^{-1}(y) \in I$. There are two cases: $I = [0, b)$, with $\pi(0) = \pi(\pi^{-1}(y)) = \pi(b) = 0$, or $I = (a, b)$, with $\pi(a) = \pi(\pi^{-1}(y)) = \pi(b)$. In either case, the definition of $\pi^{-1}(y)$ is violated. ◀

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

The definition of the $\mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ transformation can now be easily given. First of all, $\mathit{Sim}(\mathcal{R}_\Pi)$ must support sampling the uniform distribution by random number generation. This is achieved by importing the Maude module (rewrite theory) `SAMPLE-UNIFORM` as a subtheory of $\mathit{Sim}(\mathcal{R}_\Pi)$, where `SAMPLE-UNIFORM` itself imports the Maude built-in modules `RANDOM` (the random number generator), `COUNTER` (a counter with constant `counter` incremented each time `counter` is rewritten), and `CONVERSION`, where `float`(a/b) converts the rational a/b into a floating point number. We refer to [15], Section 9.3, for the details about the `RANDOM` and `COUNTER` modules, and for an example module extending the functionality of `SAMPLE-UNIFORM` to sample the Bernoulli distribution. The key point for our present purposes is that `SAMPLE-UNIFORM` has a constant `rand` denoting a random value in $[0, 1]$, whose semantics is given by the rewrite rule:

```
rl rand => float(random(counter) / 4294967295) .
```

where, when `counter` holds n, then `random(counter)` is the nth random number in the natural number interval $[0, 2^{32} - 1]$ provided by the Mersenne twister random number generator. In other words, 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 $\mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ transformation is to explain how the rewrite rules of $\mathcal{R}_\Pi$ are transformed into corresponding rewrite rules in $\mathit{Sim}(\mathcal{R}_\Pi)$. But this is easy: all *executable* rules in $\mathcal{R}_\Pi$ are imported unchanged into $\mathit{Sim}(\mathcal{R}_\Pi)$. This only leaves the probabilistic, and therefore non-executable, rules $\texttt{delay}_{l.1}$ for each l, and $\texttt{delay}_{init.1}$, which are respectively transformed into the executable conditional rules:

```
[delay_{l.1}] : {delay_l(x⃗_l, (to O from O' : MP) ; ML) CF | T}
          =>
             {delay_l(x⃗_l, ML) [T + δ_l(x⃗_l, O, O', MP)(D), (to O from O' : MP)] CF | T}
            if  D := π_l(x⃗_l)^-1(rand) .

[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}
              if  D := π_init^-1(rand) .
```

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 Transfer Method.

6.2 How the $\mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ transformation is automated in practice

The above is a theoretical definition of the $\mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ transformation. But to automate such a 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 in our tool by means of a *user-extensible* module `SAMPLING-LIB` entirely analogous to the `DIST-LIB` module used to specify Π. Recall that in `DIST-LIB`, a distribution function $\texttt{foo}(\vec{p})$ parametric on $\vec{p}$ and having a parametric mathematical definition $\texttt{foo}(\vec{p}) = \lambda x.\ \mathit{exp}(x, \vec{p})$ is specified by: (1) declaring an operator `foo` with n arguments of sort `Real` corresponding to its n parameters $\vec{p}$ and with result sort `RFun`, and (2) actually specifying $\texttt{foo}(\vec{p})$ by an equation of the form: $\texttt{foo}(\vec{p})[x] = \mathit{exp}(x, \vec{p})$.

The `SAMPLING-LIB` module plays an entirely similar role for specifying the inverse function $\texttt{foo}(\vec{p})^{-1}$ of each $\texttt{foo}(\vec{p})$, parametric on $\vec{p}$, and having a parametric mathematical definition $\texttt{foo}(\vec{p})^{-1} = \lambda y.\ \mathit{exp}'(y, \vec{p})$. The inverse function $\texttt{foo}(\vec{p})^{-1}$ is specified using the operator

```
op sample : RFun -> RFun [ctor] .
```

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: $\texttt{sample(foo}(\vec{p})\texttt{)}[y] = \mathit{exp}'(y, \vec{p})$. Here is a fragment of `SAMPLING-LIB` illustrating these ideas:

```
fmod SAMPLING-LIB is including DISTR-LIB .
  *** unified operator for defining sampling functions
  op sample : RFun -> RFun [ctor]  .

  *** sampling function for the exponential distribution
  eq sample(exponential(RATE))[RAND] = (- log(RAND)) / RATE .

  *** sampling function for the lognormal distribution
  *** the built-in constant 'pi' approximates 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
```

An optimization in the automation of the $\mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ transformation is the observation that the matching conditions `D :=` $\pi_l(\vec{x}_l)^{-1}$`(rand)` (resp. `D :=` π_{init}^{-1}`(rand)`) in rules $\texttt{delay}_{l.1}$ (resp. $\texttt{delay}_{init.1}$) are superfluous, since we can just replace each appearance of D in the transformed rule's righthand side by $\pi_l(\vec{x}_l)^{-1}$`(rand)` (resp. π_{init}^{-1}`(rand)`). We can illustrate the automatic transformation $\mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$ for our running example as follows:

▶ **Example 8.** The `delay-reply.1` rule (in Section 4.3.2.2) is transformed into:

```
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)[sample(lognormal(size(DB[K]),0.1))[rand]],
                  MSG] CF | T } .
```

where the added red and blue parts refer to the lognormal sampling function and `delta-reply`'s application function, respectively; the entire part in italics will reduce to the actual delay (of sort `Real`) accordingly when this rule is applied. Likewise, the non-executable rule `delay-init.1` rule in Section 4.3.2.4 is transformed into an executable one by replacing the probability distribution with the corresponding sampling function:

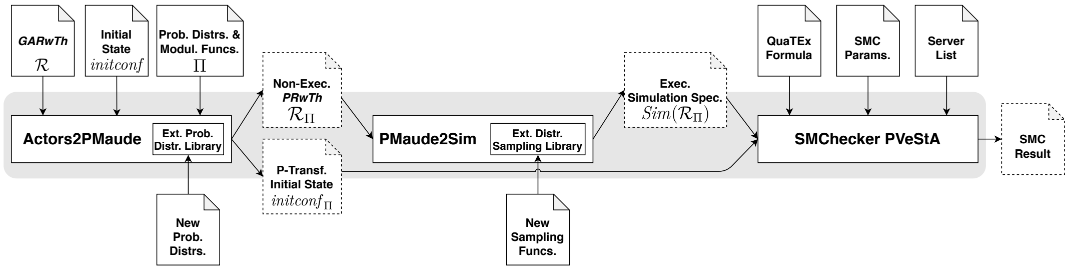

Figure 1 The architecture of the Actors2PMaude tool.

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

7 Quantitative Analysis in the Actors2PMaude Tool

We have implemented (in Maude, using Maude's meta-level facilities) the P and the *Sim* transformations and have also integrated parallelized PVeStA statistical model checking of the resulting $Sim(\mathcal{R}_\Pi)$ models into a single tool, called *Actors2PMaude*, which is available at `https://github.com/anonymousecoop/actors2pmaude`.

As shown in Fig. 1, the tool consists of three components:

1. The Actors2PMaude component implements the P transformation and includes the probability distribution library `DISTR-LIB`. The inputs are defined as in Example 4.3.2.1.[13]
2. The PMaude2Sim component applies the *Sim* transformation to the resulting module, and also includes an extensible sampling library.
3. The SMC component performs the SMC analysis with simulations obtained by executing the simulation model together with the associated initial state.

We use the statistical model checker PVeStA [7], which allows us to parallelize the Monte Carlo simulations to improve the performance of the analysis. PVeStA takes as input:

- the simulation model by the PMaude2Sim component 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.

The result of the statistical model checking is the expected value of the QuaTEx formula.

Using the Tool. Our Actors2PMaude tool is a client-server-based parallelization of SMC analysis. The user must first run the following script *at the client side* to launch each server:

```
./launch.sh serverlist
```

where *serverlist* is the file containing a list of available servers for running simulations. The user then starts the client by running the following script:

```
./run.sh -smc mode -pt gasys init distr -pv quatex confidence size serverlist
```

[13] Currently, the tool only accepts *core Maude* [15] modules as input; however, Full Maude can automatically transform any Full Maude module to its core Maude equivalent.

Turning *mode* `on` leads to a "one-button" SMC analysis where the QuaTEx formula can be defined without investigating the intermediate modules generated by the P or *Sim* transformation; otherwise, i.e., to see the P- and *Sim*-generated modules, *mode* should be `off`. *gasys*, *init*, and *distr* refer to the files containing the *GARwTh* (module `QUERY` in our example), the initial state (e.g., module `INIT-QUERY`), and Π (e.g., module `PI-QUERY`), respectively. The files *quatex* and *serverlist* contain the QuaTEx formula and the server list, and *confidence* and *size* denote the confidence level and size parameters, respectively.

▶ **Example 9.** In our running example, we give the following command (for appropriate file names) to statistically estimate the throughput (number of queries processed per time unit) with 95% confidence, i.e., $\alpha = 0.05$, and threshold $\delta = 0.01$:

```
./run.sh -smc on -pt query.maude init-query.maude pi-query.maude
         -pv throughput.quatex 0.05 0.01 serverlist
```

In this case, the `throughput.quatex` file contains the QuaTEx formula

```
thrForThisSimul() = { s.rval(1) } ; eval E[ # thrForThisSimul() ] ;
```

where `1` corresponds to the `throughput` function defined in the Maude model:

```
*** QuaTEx interaction with Maude
op val : Nat ClockedState -> Real .        eq val(1,CS) = throughput(CS) .

op throughput : ClockedState -> Real .     eq throughput({OBJS | T}) = 3.0 / T .
```

This QuaTEx formula defines the value of `3.0 / T` at the end of each run (when all three queries have been processed), where `T` is the value of the global clock.

The `serverlist` file contains a local server and a remote server:

```
localhost:49046
xxx.xxx.xxx.xxx:49046
```

The output file shows the result of the statistical model checking, where 60 simulations were performed to obtain the expected throughput 0.0968:

```
Confidence (alpha):  0.05            Threshold (delta):  0.01
Samples generated:  60               Result: 0.0968
Running time: 0.34 seconds
```

8 Applications

In this section we illustrate the usefulness of our tool by using it to study the performance of a range of generalized actor systems, specified as *GARwTh*s in Maude. In particular, some of the case studies illustrate the need to take into account the size of the message payload and the distance between sender and receiver when defining the message delays.

As shown in Table 1, we have used our tool on 11 case studies, as well as on our running example. The case studies include simple textbook distributed algorithms such as Maekawa's voting algorithm [38], internetworking protocols like FBAR [57], IETF-standardized mobile ad-hoc (MANET) protocols such as AODV [47], complex industrial storage systems like Apache Cassandra [1], and state-of-the-art academic transaction systems such as RAMP [9]. Note that the Maude models contain a significant proportion of object-triggered (‡) rules.

Table 1 Summary of our case studies. "Query Protocol" is our running example. •-marked protocols are first specified for this paper.

GA System	Maude LOC	(†) : (‡) (#rules)	GA System	Maude LOC	(†) : (‡) (#rules)
Mutex Alg.			**Database**		
Token Ring [46]	60	4 : 2	Query Protocol	90	3 : 2
Maekawa's [46]	150	6 : 3	RAMP-F [32]	330	10 : 4
Internetwork			RAMP-S [32]	250	10 : 4
FBAR [57]	260	11 : 1	Prepare-F HA•	750	14 : 6
MANET			Sticky HA•	810	15 : 6
AODV [31]	570	13 : 3	Cassandra [36]	940	13 : 7
			Cassandra-TA [28]	970	17 : 5

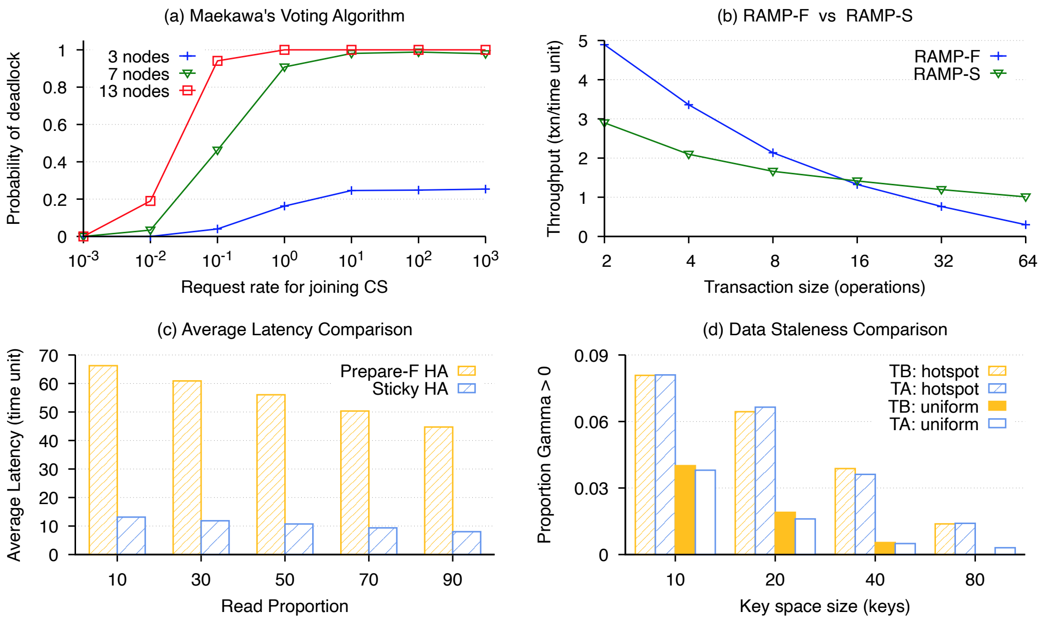

Figure 2 Statistical model checking analysis results obtained using the Actor2PMaude tool.

The Actors2PMaude tool allow us to use SMC to analyze *quantitative* and *probabilistic* properties of a range of distributed systems, modeled as *GARwThs*; this facilitates the comparison of different system designs. In the following we discuss four examples.

We employ 50 d710 Emulab machines [58] to parallelize the SMC analysis. For the experiments in Fig. 2 (b–d), we have also implemented in Maude a parametric[14] database workload generator to produce initial states.

Distributed Mutual Exclusion. Fig. 2 (a) shows the *probability of deadlock* in the Maekawa voting algorithm [38] for distributed mutual exclusion, against nodes' request rates for entering the critical section, with initial states (with 3, 7, and 13 nodes) taken from [38]. As expected, fewer nodes and/or infrequent entering lead to lower probability of deadlock.

[14] The parameters are: read/write proportion, number of clients, servers, keys, transactions, and operations per transaction, and key access distribution.

RAMP. The Read Atomic Multi-Partition (RAMP) transaction system [9] developed at UC Berkeley provides high-performance operations for large-scale partitioned data stores (in a non-replicated setting). RAMP has different versions, such as RAMP-F and RAMP-S, which offer different trade-offs between the *size* of the messages and *system performance*. Fig. 2 (b) shows the throughput (completed transactions per time unit) of RAMP-F and RAMP-S with varying transaction size. RAMP-F tends to perform worse than RAMP-S when the number of operations per transaction increases. The P-transformed models refine the manually transformed probabilistic versions [29] of the Maude models of RAMP-F and RAMP-S in [32], which did not take into account the effect of the size of the message payload on transmission delays. Therefore, we now obtain statistical results that are consistent with the Java implementation-based evaluations in [9] (see also Fig. 3 in Appendix A).

The RAMP developers also conjectured two designs [9], called Sticky HA and Prepare-F HA, in a *replicated* setting, with the former expected to incur much lower latency due to the client's *stickiness* (its *short distance* to its local data center). We have formally modeled these two algorithms as *GARwThs* in Maude. Fig. 2 (c) plots their comparison on average latency with varying read proportion in the workload (from write-heavy workloads with 10% read operations to read-heavy ones with 90% read operations). By taking into account the effect of client/server distance on the message delays, our experimental results are consistent with the conjecture of the RAMP developers.

Cassandra. Apache Cassandra [1] is a distributed, scalable, and highly available NoSQL database design used by Apple, eBay, Instagram, Netflix, and thousands of other companies.

In the original Cassandra design (called TB), a coordinator uses *timestamps* to decide which value to return to the client. In [28], Liu *et al.* proposed an alternative *time-agnostic* design (called TA) that uses the *values* themselves to determine what value to return. The authors of [28] only compared the two designs on the probability of satisfying certain consistency properties. In this paper we complement those results by comparing the two designs in terms of the *staleness* ("age") of the client-observed data.

Fig. 2 (d) shows the SMC results obtained by our tool. For staleness, we use the *Gamma* metric [19] to count the proportion of values involved in consistency anomalies: the higher the proportion is, the more frequently the anomalies occur. Our results show that the two Cassandra designs are incomparable, with varying key-space size under different key-access distributions (i.e., *hotspot* and *uniform* [16]). Our results for TB's staleness are consistent with the Java-implementation-based evaluations in [19] (see also Fig. 4 in Appendix B).

9 Related Work

The most closely related work is earlier work on using probabilistic rewrite theories and Maude to analyze performance properties of distributed object systems. Our work builds on the extensive experience of several research teams in this area [25, 4, 6, 13, 33, 35, 37, 18, 53, 28, 29, 55, 8, 37]. The similarities are many. The differences appear in resolving the "missing links" mentioned in the Introduction by: (1) Changing the passage from an actor model $\mathcal{R}$ of a distributed system to its probabilistic enrichments $\mathcal{R}_\Pi$ and $\mathit{Sim}(\mathcal{R}_\Pi)$ from an art requiring effort and expertise to two *correct-by-construction automatic transformations* $\mathcal{R} \mapsto \mathcal{R}_\Pi \mapsto \mathit{Sim}(\mathcal{R}_\Pi)$. (2) Having precisely defined a very general class of *generalized actor rewrite theories* (*GARwThs*) as the input domain for such transformations. The main difference with the Actor PMaude language in [5] is that actor rules are restricted to message-triggered rules. Our experience in specifying many distributed systems is that allowing also *object-triggered rules*, modeling what sometimes are called *active objects*, makes specifications more natural and simpler. (3) Proving *metatheorems* for any input theory $\mathcal{R}_\Pi \in \mathit{Sim}(\mathcal{R}_\Pi)$ (and initial states) ensuring the semantic correctness of SMC analysis relative to the distributions in Π. (4) Having provided high levels of automation for such SMC analysis through the *Actors2PMaude* tool. In relation to such previous work, it seems fair to say that the main thrust of the advances in this work are in developing the semantic foundations and in facilitating the use and wider adoption of these methods in practice.

Our main contributions is this paper is a unified framework in which a *single model* of a *distributed system* can be used to both verify the correctness of a (non-probabilistic, untimed) distributed systems, and (after being automatically enriched) also can be subjected to formally-based quantitative analysis, e.g., by statistical model checking.

There are a number of formal framework for quantitative analysis, typically based on statistical or probabilistic model checking. For example, the UPPAAL-SMC [17] statistical model checker belongs to the timed automaton-based Uppaal [11] family of formal tools for real-time systems. UPPAAL-SMC inputs are networks of stochastic timed automata, with communication modeled by instantaneous broadcast channels. These input models are assumed to be purely probabilistic. Unlike in our case, where the probability distributions (on message delays) can be any continuous probability distribution—possibly even augmented by modulation functions—in their models the distributions are either uniform or exponential, albeit with (like in our approach) parameters of these distributions that may depend on the states instead of being merely constants. Whereas we introduce time (only) on message delays, in UPPAAL-SMC communication is instantaneous and the delays instead apply to how long a node stays in a given location (which can be used to encode messaging delays).

PRISM [2, 27] supports the modeling and probabilistic and statistical model checking of MDPs and (nondeterministic) probabilistic timed automata, restricted to uniform and exponential distributions. For statistical model checking purposes, nondeterministic choices are resolved by simple random choice (i.e., sampling from a uniform distribution).

$\mathcal{S}$BIP [40, 45] is a sophisticated statistical model checker for stochastic real-time BIP models; that is, a collection of stochastic timed automata composed using multi-party interactions. Like us, $\mathcal{S}$BIP supports user-defined probability distributions. There is no unquantified nondeterminism to resolve, since the stochastic real-time BIP modelling formalism guarantees that all nondeterminism is resolved through stochastic choices over interactions.

The main difference between these frameworks and our contribution is that we are not aware of any semantic mappings from models of distributed systems, such as the actor model, to these frameworks. This also seems to be the case for $\mathcal{S}$BIP, even though (non-probabilistic and untimed) BIP [10] is a well-established formalism for distributed systems.

Researchers in the Modest research group have achieved an impressive unification of probabilistic and stochastic automata-based models and tools. This includes not only the Modest toolset and language [22], but also the JANI modeling and tool interaction language [14]. A key idea, exploited by both Modest and JANI, is to derive a rich hierarchy of probabilisitic and/or timed automata-based models as special cases of the model of (networked) stochastic hybrid automata (SHA), including, e.g., stochastic and probabilisitic timed automata, discrete- and continuous-time Markov chains, or even just labeled transition systems, as *special cases*. Stochastic hybrid automata is a model well suited for specification and analysis of cyber-physical systems. At the tool support level, the combined power of Modest and JANI extends far beyond the Modest toolset: Models on UPPAAL, Prism, Generalized Stochastic Petri Nets, I/O Stochastic Automata, and Probabilistic Guarded Command languages can be specified and various tools beyond Modest-based ones, including the already-discussed Prism and UPPAAL ones, can be invoked to perform probabilistic or statistical model checking analyses. Furthermore, the Jani Interaction Protocol also supports various model transformations. One similarity with our work is that, although it does not seem to be currently supported, the inclusion of labeled transition systems in the above-mentioned hierarchy of model classes could enable temporal logic model checking of untimed, nondeterministic systems. Another attractive similarity is support for model transformations. The main differences are semantic, of the type already-mentioned above when discussing Prism and Uppaal: to be able to support both correctness and performance analysis of Actor models of distributed systems, semantically faithful model transformations enriching such systems with probability distributions and presumably mapping them into networked SHAs should be available. However, to the best of our knowledge such model transformations have not yet been defined.

Rebeca [52] is an actor-based formal modeling language for distributed system. Timed Rebeca extends Rebeca to discrete-time real-time systems. In [23], the authors define a statistical model checker for such (non-probabilistic) Timed Rebeca models based on simulation traces generated by the McErlang tool. Nondeterminism is resolved by the scheduler of McErlang, which randomly chooses the next transition based on the uniform distribution. PTRebeca [24] extends Timed Rebeca to the probabilistic setting, where a variable may be assigned a value drawn from a discrete probability distribution, and subject such models to probabilistic model checking using Prism and IMCA.

Although Rebeca is suitable to model actor-based distributed systems, it does not support continuous probability distributions in its PTRebeca extension, which in our experience is important for performance analysis. Furthermore, we are not aware of any work automatically enriching Rebeca distributed systems models to the timed and probabilistic versions of Rebeca.

10 Conclusion

Distributed systems are hard to design and verify: their design flaws can be subtle and may impact both correctness and performance.Formal methods that can be integrated into the design process and support analysis of both correctness performance properties are highly desirable. This work has advanced these goals for Maude-based formal executable specification of distributed systems, focusing on semantic foundations on the one hand, and on achieving high levels of automation and model reuse on the other. Both tasks are intimately related: the P and *Sim* transformations that we have formally defined and proved to generate probabilistic models *semantically consistent* with the original distributed system Maude specification, are precisely those that have been *automated* in the *Actors2PMaude* tool, and have been integrated with SMC analysis using PVeStA. Furthermore, the case studies have demonstrated that much can be learned about a distributed system before it is built; and that what is learned reveals performance trends and comparisons between alternative system designs that have been independently validated by experimental evaluations.

Although, due to lack of space, we have focused on quantitative analyses, Maude supports verification of correctness properties by breadth first search reachability analysis and LTL model model checking, which can be used to verify generalized actor rewrite theories [15]. They can also be deductively verified to satisfy Hoare logic and reachability logic properties using the reachability logic prover [54]. In the case of distributed transaction systems further studied in e.g., [13, 33, 35, 37], model checking verification of a wide range of *consistency properties* has been automated in the tool described in [35]. In the near future the following further steps should be advance this research: (i) more case studies should be developed; (ii) an *Actors2PMaude* manual with a representative collection of tutorial examples should be written, and (iii) it would be highly desirable to achieve an integration of the *Actors2PMaude* tool with the tool described in [35], and with the tool generating correct-by-construction distributed implementations from already verified generalized actor models described in [37].

References

1. Apache cassandra, 2020. `https://cassandra.apache.org`.
2. PRISM-SMC, 2020. `https://www.prismmodelchecker.org/manual/RunningPRISM/StatisticalModelChecking`.
3. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.
4. Gul Agha, Carl Gunter, Michael Greenwald, Sanjeev Khanna, Jose Meseguer, Koushik Sen, and Prasanna Thati. Formal modeling and analysis of DoS using probabilistic rewrite theories. In *Workshop on Foundations of Computer Security (FCS)*, 2005.
5. Gul A. Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. *Electr. Notes Theor. Comput. Sci.*, 153(2), 2006.
6. M. AlTurki, J. Meseguer, and C. Gunter. Probabilistic modeling and analysis of DoS protection for the ASV protocol. *Electr. Notes Theor. Comput. Sci.*, 234:3–18, 2009.
7. Musab AlTurki and José Meseguer. PVeStA: A parallel statistical model checking and quantitative analysis tool. In *CALCO'11*, volume 6859 of *LNCS*, pages 386–392. Springer, 2011.
8. Musab A. Alturki and Grigore Rosu. Statistical model checking of randao's resilience to pre-computed reveal strategies. In *Formal Methods. FM 2019 International Workshops*, volume 12232 of *LNCS*, pages 337–349. Springer, 2019.
9. Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Scalable atomic visibility with RAMP transactions. *ACM Trans. Database Syst.*, 41(3):15:1–15:45, 2016.
10. Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Softw.*, 28(3):41–48, 2011.
11. Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems, SFM-RT 2004*, volume 3185 of *Lecture Notes in Computer Science*. Springer, 2004.
12. Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC'10*, pages 267–280. ACM, 2010.
13. Rakesh Bobba, Jon Grov, Indranil Gupta, Si Liu, José Meseguer, Peter Csaba Ölveczky, and Stephen Skeirik. Survivability: Design, formal modeling, and validation of cloud storage systems using Maude. In *Assured Cloud Computing*, chapter 2, pages 10–48. Wiley-IEEE Computer Society Press, 2018.
14. Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. JANI: quantitative model and tool interaction. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2017.*
15. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.
16. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SOCC'10*, pages 143–154. ACM, 2010.
17. Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *Int. J. Softw. Tools Technol. Transf.*, 17(4):397–415, 2015.
18. Jonas Eckhardt, Tobias Mühlbauer, Musab AlTurki, José Meseguer, and Martin Wirsing. Stable availability under denial of service attacks through formal patterns. In *FASE*, pages 78–93, 2012.

19 Wojciech M. Golab, Muntasir Raihan Rahman, Alvin AuYoung, Kimberly Keeton, and Indranil Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *ICDCS*, pages 493–502. IEEE Computer Society, 2014.

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

21 Jon Grov and Peter Csaba Ölveczky. Formal modeling and analysis of Google's Megastore in Real-Time Maude. In *Specification, Algebra, and Software*, volume 8373 of *LNCS*. Springer, 2014.

22 Arnd Hartmanns and Holger Hermanns. The Modest toolset: An integrated environment for quantitative modelling and verification. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 593–598. Springer, 2014.

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

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

25 Michael Katelman, José Meseguer, and Jennifer C. Hou. Redesign of the lmst wireless sensor protocol through formal modeling and statistical model checking. In *Proc. FMOODS 2008*, volume 5051 of *LNCS*, pages 150–169. Springer, 2008.

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

27 M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV'11*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

28 Si Liu, Jatin Ganhotra, Muntasir Rahman, Son Nguyen, Indranil Gupta, and José Meseguer. Quantitative analysis of consistency in NoSQL key-value stores. *Leibniz Transactions on Embedded Systems*, 4(1):03:1–03:26, 2017.

29 Si Liu, Peter Csaba Ölveczky, Jatin Ganhotra, Indranil Gupta, and José Meseguer. Exploring design alternatives for RAMP transactions through statistical model checking. In *ICFEM*, volume 10610 of *LNCS*, pages 298–314. Springer, 2017.

30 Si Liu, Peter Csaba Ölveczky, and José Meseguer. Formal analysis of leader election in MANETs using Real-Time Maude. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering*, volume 8950 of *LNCS*, pages 231–252. Springer, 2015.

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

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

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

34 Si Liu, Peter Csaba Ölveczky, Qi Wang, and José Meseguer. Formal modeling and analysis of the Walter transactional data store. In *WRLA*, volume 11152 of *LNCS*, pages 136–152. Springer, 2018.

35 Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. Automatic analysis of consistency properties of distributed transaction systems in Maude. In *TACAS'19*, volume 11428 of *LNCS*, pages 40–57. Springer, 2019.

36 Si Liu, Muntasir Raihan Rahman, Stephen Skeirik, Indranil Gupta, and José Meseguer. Formal modeling and analysis of Cassandra in Maude. In *ICFEM*, volume 8829 of *LNCS*. Springer, 2014.

37 Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. Generating correct-by-construction distributed implementations from formal Maude designs. In *NFM'20*, volume 12229 of *LNCS*. Springer, 2020.

38 Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

39 P. Manolios. A compositional theory of refinement for branching time. In *CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2003.

40 Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, Mahieddine Dellabani, Axel Legay, and Saddek Bensalem. $\mathcal{S}$BIP 2.0: Statistical model checking stochastic real-time systems. In *ATVA'18*, volume 11138 of *LNCS*, pages 536–542. Springer, 2018.

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

42 José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

43 José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

44 José Meseguer. Twenty years of rewriting logic. *J. Algebraic and Logic Programming*, 81:721–781, 2012.

45 Ayoub Nouri, Braham Lotfi Mediouni, Marius Bozga, Jacques Combaz, Saddek Bensalem, and Axel Legay. Performance evaluation of stochastic real-time systems with the SBIP framework. *Int. J. Crit. Comput. Based Syst.*, 8(3/4):340–370, 2018.

46 Peter Csaba Ölveczky. *Designing Reliable Distributed Systems - A Formal Methods Approach Based on Executable Modeling in Maude*. Undergraduate Topics in Computer Science. Springer, 2017.

47 Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir R. Das. Ad hoc on-demand distance vector (AODV) routing. *RFC*, 3561:1–37, 2003.

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

49 Stefano Sebastio and Andrea Vandin. MultiVeStA: Statistical model checking for discrete event simulators. In *ValueTools*, pages 310–315. ICST/ACM, 2013.

50 Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In *CAV'05*, volume 3576 of *LNCS*. Springer, 2005.

51 Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. VESTA: A statistical model-checker and analyzer for probabilistic systems. In *QEST'05*, pages 251–252. IEEE Computer Society, 2005.

52 Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae*, 63(4):385–410, 2004.

53 Stephen Skeirik, Rakesh B. Bobba, and José Meseguer. Formal analysis of fault-tolerant group key management using ZooKeeper. In *CCGRID*, pages 636–641, 2013.

54 Stephen Skeirik, Andrei Stefanescu, and José Meseguer. A constructor-based reachability logic for rewrite theories. *Fundam. Inform.*, 173(4):315–382, 2020.

55 Abraão Aires Urquiza, Musab A. AlTurki, Max I. Kanovich, Tajana Ban Kirigin, Vivek Nigam, Andre Scedrov, and Carolyn L. Talcott. Resource-bounded intruders in denial of service attacks. In *CSF*, pages 382–396. IEEE, 2019.

56 Anduo Wang, Carolyn L. Talcott, Limin Jia, Boon Thau Loo, and Andre Scedrov. Analyzing BGP instances in Maude. In *FMOODS'11*, volume 6722 of *LNCS*, pages 334–348. Springer, 2011.

57 Bow-Yaw Wang, José Meseguer, and Carl A. Gunter. Specification and formal analysis of a PLAN algorithm in Maude. In *ICDCS Workshop on Distributed System Validation and Verification 2000*, pages E49–E56, 2000.

58 Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*. USENIX Association, 2002.

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

A The Original RAMP Evaluations in [9]

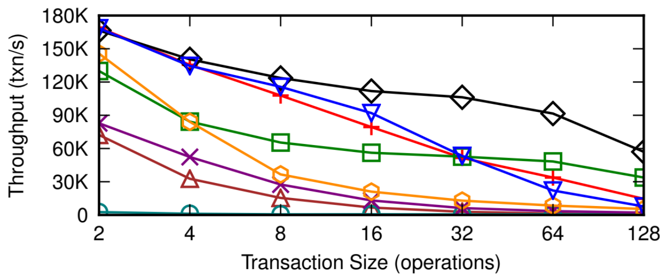

Figure 3 RAMP-F (blue) vs RAMP-S (green): throughput under varying transaction size by the Java implementation-based evaluations in [9].

B The Original Cassandra Evaluations in [19]

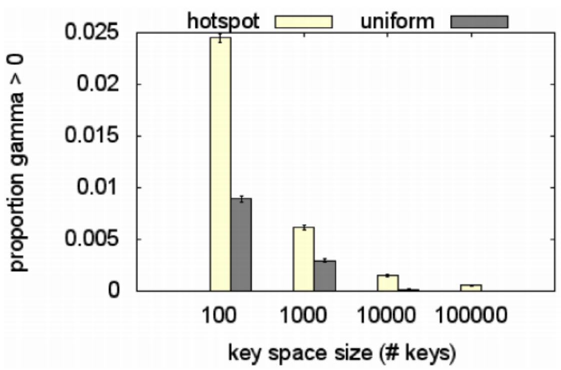

Figure 4 The original Cassandra design implemented in Java: data staleness measured by the *Gamma* metric under varying key space size for two different key-access distributions in [19].