Statistical Model Checking for Composite Actor Systems

Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, Martin Wirsing

Statistical Model Checking for Composite Actor Systems*

Jonas Eckhardt[1], Tobias Mühlbauer[1], José Meseguer[2], Martin Wirsing[3]

[1] Technische Universität München, [2] University of Illinois at Urbana-Champaign, [3] Ludwig-Maximilians-Universität München

Abstract. In this paper we propose the so-called composite actor model for specifying composed entities such as the Internet. This model extends the actor model of concurrent computation so that it follows the "Reflective Russian Dolls" pattern and supports an arbitrary hierarchical composition of entities. To enable statistical model checking we introduce a new scheduling approach for composite actor models which guarantees the absence of unquantified nondeterminism. The underlying executable specification formalism we use is the rewriting logic-based semantic framework Maude, its probabilistic extension PMaude, and the statistical model checker PVeStA. We formalize a model transformation which—given certain formal requirements—generates a scheduled specification. We prove the correctness of the scheduling approach and the soundness of the transformation by introducing the notions of strong zero-time rule confluence and time-passing bisimulation and by showing that the transformation is a time-passing bisimulation for strongly zero-time rule confluent composite actor specifications.

Key words: actor system, rewriting logic, Maude, composite actor, statistical model checking

1 Introduction

The actor model is a classical model for concurrent computation [16] which witnessed revived interest with the advent of multi-core programming and Cloud-scale computing. Several modern programming languages such as Erlang and Scala base their concurrency models on actors [7,14]. An actor is a concurrent object which operates asynchronously and interacts with other actors by sending asynchronous messages [3]. Temporal logic properties of actor-based models can be automatically verified either by exact model checking algorithms [13] or, in an approximate but more scalable way, by statistical model checking (see e.g., [4]).

These approaches are all based on the original "flat" actor model but many interesting applications such as the Internet and Cloud systems are not flat, as they are composed of various participants and systems and are hierarchically structured into different layers and networks. Such composed entities are often

* This work has been partially sponsored by the EU-funded projects FP7-257414 ASCENS and FP7-256980 NESSoS, and AFOSR Grant FA8750-11-2-0084. Tobias Mühlbauer is also partially supported by the Google Europe Fellowship in Structured Data Analysis. We thank Mirco Tribastone for his helpful comments on this paper. We further thank all reviewers for their valuable feedback.

safety- and security-critical, and have strong qualitative and quantitative formal requirements. The above mentioned analysis approaches, however, rely on flat actor models and cannot handle and model check composite models in a direct way.

In this paper, we extend the actor model to a so-called *composite actor model* that directly addresses hierarchical concurrent systems and present a model transformation which makes statistical model checking usable for composite actor model specifications. The composite actor model follows the so-called "Reflective Russian Dolls" model [19] and supports an arbitrary hierarchical composition of entities. As underlying executable specification formalism we use the rewriting logic language Maude and its real-time and probabilistic extensions.

Current statistical model checking methods require that the system is purely probabilistic, i.e., that there is no unquantified nondeterminism in the choice of transitions. This is nontrivial to achieve for distributed systems where many different components may perform local transitions concurrently. There are two complementary ways for guaranteeing the absence of nondeterminism: either by associating continuous probability distributions with message delays and computation time and by relying on the fact, that for continuous distributions the probability of sampling the same real number twice is zero (see e.g., [13]), or by introducing a scheduler that provides a deterministic ordering of messages (see e.g., [6]). We follow the latter approach and propose a new scheduling method for well-formed composite actor models that guarantees the absence of nondeterminism. We formalize the approach in Maude and study its soundness by proving the correctness of the scheduling approach, termination and confluence of the underlying equational specification, and by showing the absence of unquantified nondeterminism from any scheduled well-formed composite actor specification. We further formalize a model transformation which—given a composite actor specification that adheres to certain formal requirements—generates a scheduled specification. To prove the soundness of the transformation we introduce the notions of strong zero-time rule confluence and time-passing bisimulation and then show that the transformation, which is only a simulation by itself, is indeed a time-passing bisimulation for strongly zero-time rule confluent composite actor specifications. To the best of our knowledge, our solution is the first one making it possible to analyze such systems in a faithful way by statistical model checking. We have applied our method to several complex case studies (see [12,21,11,20]); for reasons of space we only illustrate a simple example.

Outline. The paper is structured as follows: In Sect. 2 we explicate shortly the (flat) actor model and the "Reflective Russian Dolls" model, and show how to build the composite actor model in Maude by applying the Russian Dolls approach to the flat actor model. In Sect. 3 we present our model transformation together with the scheduling approach, their formalization in Maude and prove the soundness properties. Sect. 4 then gives a short presentation of our methodology for the statistical model checking analysis of actor model-based specifications. We conclude by discussing related work, summarizing our results, and sketching further work.

2 The Composite Actor Model

2.1 The Actor Model of Computation

Our specifications are based on the *actor model of computation* [16,15,2], a mathematical model of concurrent computation in distributed systems. Similar to the *object-oriented programming paradigm*, in which the philosophy that *everything is an object* is adopted, the *actor model* follows the paradigm that *everything is an actor*. An *actor* is a concurrent object that encapsulates a state and can be addressed using a unique name. Actors communicate with each other using asynchronous messages. Upon receiving a message, an actor can change its state and can send messages to other actors. Actors can be used to model and reason about distributed and concurrent systems in a natural way [2].

2.2 The *Reflective Russian Dolls* Model

In rare situations, the state of a distributed system can be thought of as a *flat configuration* which contains objects and messages. Such a *flat configuration* can be modelled as a *flat soup* (i.e., a multiset) that consists of actors and messages. However, as a distributed system becomes more complex, hierarchies are introduced to better represent the structure of the system and its communication patterns. A flat model does not reflect boundaries in a hierarchical system which impose conditional communication barriers. In a flat model, every participant can communicate with everybody else. However, some concepts, like a firewall, rely on the existence of physical boundaries that messages from the outside have to cross in order to reach destinations within that boundary.

In [19], Meseguer and Talcott present the *Reflective Russian Dolls* (RRD) model which extends and formalizes previous work on actor reflection and provides a generic formal model of distributed object reflection. The rewriting logic-based model combines logical reflection and hierarchical structuring. In their model, the state of a distributed system is not represented by a *flat soup*, but rather as a *soup of soups*, each enclosed within specific boundaries. As with traditional Russian dolls, soups can be nested up to an arbitrary depth.

Figure 1 illustrates the basic idea using a system that is guarded by a firewall. Each of the boxes represents a system. The firewall consists of a subsystem which itself is composed of several components $C_1 \dots C_n$. Message M is addressed to the innermost component C_n and as such has to pass the boundary of the firewall. The firewall possibly transforms the message to M' (e.g., tags a message with a security clearance). After that, the boundary of the sub-system has to be crossed which, respectively, can also alter the message to M''.

Mathematically, this can be modelled by boundary operators of the form

$$b : s_1, \dots, s_n, Configuration \rightarrow Configuration$$

where $s_1, \dots, s_n$ are additional sorts. These sorts are called the *parameters* of the boundary operator. Boundary operators encapsulate a configuration together with several parameters, and as with Russian dolls, they can be nested arbitrarily. Using the Russian Dolls model, sophisticated distributed systems, that rely on system boundaries, can be modeled [19].

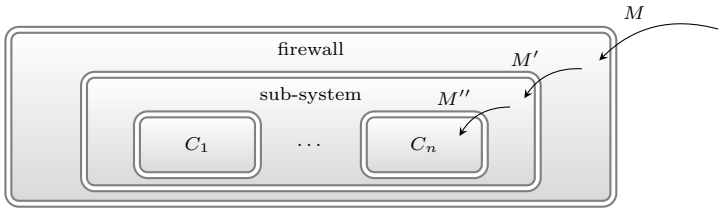

Fig. 1. Example of a Russian Dolls model of a system with boundaries.

2.3 The Composite Actor Model

We extend the original *actor model of computation* for flat configurations with the *RRD model* to allow the specification of hierarchically structured composite actor systems. Actors are now allowed to contain a soup, that is a multiset, of actors and messages in their state where each of the actors in the sub-configuration can again contain another soup of actors and messages, and so on. Referring to the aforementioned firewall example, the firewall actor can be represented by the term

`<0 :Firewall | config: <0.0 :Subsystem | config: <0.0.0 :C1> ... <0.0.N-1 :CN>>>`

Thereby the firewall actor has address `0` and contains a sub-configuration in its state denoted by the `config` attribute. In this sub-configuration there is a subsystem actor with address `0.0` which again contains a sub-configuration which consists of the components ($\texttt{C}_i$) with adresses `0.0.0`,...,`0.0.N-1`.

The naming of actors follows an hierarchical naming scheme. This naming scheme is comparable to Internet domain names, which are structured in top-level (e.g., "de"), second- (e.g., "google") and third-level (e.g., "www") domain names. The hierarchical naming scheme for the actor model basically builds a hierarchical name tree, in which children addresses are composed of their parents address and an appended number which uniquely identifies them among their siblings (e.g., a name could be 4.0.8.2). Further information on the naming scheme can be found in [11,20].

The analysis of real world systems often requires the system model to include a notion of time (e.g., as for quality of service properties in our case studies [6,21]). Thus, we introduce activation times for messages, i.e., times at which messages are intended to be processed.

We model composite actor models as hierarchically structured soups of messages and actors. In Maude, we represent a soup as a term of sort `Config`. A configuration is associative and commutative and is constructed by the operators `op null : -> Config` and `op __ : Config Config -> Config [assoc comm id: null]`. The sorts for actors and messages are both subsorts of `Config`. Additionally, we introduce the sort `Address`, which represents an actor's address, and which is a subsort of `Nat`. Addresses can be concatenated using the constructor `op _._ : Address Address -> Address [assoc]`. Messages are terms of sort `Msg` and are created by the constructor `(_,_<-_): Float Address Content -> Msg`, which takes the message's "activation" time, the receiver address and the actual contents of the message as arguments. Finally, actors can be created by

the constructor `op <_:_|_> : Address ActorType AttributeSet -> Actor`, which takes the address of the actor, the type of the actor, and an additional set of attributes (a term of sort `AttributeSet`, which represents an associative, commutative set of terms of sort `Attribute`) as arguments.

The hierarchical nature of the composite actor model is made explicit by the dedicated attribute `config`, which contains the inner soup of an actor. The constructor `op config:_ : Config -> Attribute [gather(&)]` creates this inner soup for an arbitrary term of sort `Config`. [1]

Following the idea of the actor model, actors in the composite actor model communicate only via asynchronous message passing. This convention allows the local specification of actors, i.e., the specification of an actor's semantics does not need to contain knowledge about the structure of the composed system and rewrite rules that capture the semantics do not have to include the receiving actor. As such, a specification of an actor only requires local transition rules (message handling) and boundary crossing rules to be specified. Thereby, boundary crossing rules are rules that insert a message into an actor's sub-configuration and rules that move a message from an actor's sub-configuration to the configuration the actor is in. More precisely, composite actor model specifications only contain (possibly probabilistic) rewrite rules of the following type:

*C*onsume: one actor consumes a message at time t and may emit timed ($\mathtt{t}_1,\ldots,\mathtt{t}_n$) messages or spawn new actors in its sub-configuration.

```
< A : T | config: C, AS> (t, A<-M) =>
  < A : T | config: C', AS'> (t1, A1<-M1) ... (tn, An<-Mn)
```

*B*oundary-Down: one actor consumes a message at time t and inserts it into its subconfiguration.

```
< A : T | config: C, AS> (t, X<-M) => < A : T | config: C, (t', X'<-M'), AS'>
```

*B*oundary-Up: one actor consumes a message from its subconfiguration at time t and emits it at its level in the composite actor hierarchy.

```
< A : T | config: (t, X<-M) C, AS> => < A : T | config: C, AS'> (t', X'<-M')
```

`A, A1,...,An, X, X'` are thereby terms of sort `Address`, `T` of sort `ActorType`, `C, C'` of sort `Config`, `t, t1,...,tn, t'` of sort `Float`, `M, M1,...,Mn, M'` of sort `Content` and `AS` and `AS'` of sort `AttributeSet`. Any of these rules may have added probabilistic information [4] or be subjected to a condition.

Local specifications of actors provide modularity which is a key technique to tackle the complexity of large distributed systems. In previous work [12] we describe how meta objects, i.e., distributed objects that mediate and control the behavior of one or more distributed objects, can be made highly reusable as formal patterns and how a distributed system can be modelled as a composition of such formal patterns.

Note that even though messages in the composite actor model are timed, the execution of composite actor model specifications in pure Maude would neither

[1] In general, it is possible to allow an actor to have its inner soup(s) in arbitrarily named attributes. The convention of having a single sub-configuration in a predefined attribute, however, simplifies scheduling approaches for statistical model checking.

take activation times nor the concept of a global time into account. Rather all possible executions regardless of activation times would be executed.

Definition (Well-Timedness). In the following we are only interested in "well-timed" executions where messages are processed according to their activation time and global time is advancing monotonously. More precisely, we call a run $u_0 \xrightarrow{r_0} u_1 \xrightarrow{r_1} \ldots$ of a composite actor specification well-timed if for any i, the transition $u_i \xrightarrow{r_i} u_{i+1}$ is triggered by the application of rule r_i at time t_i, and $t_i \leq t_{i+1}$ where t_i is the smallest activation time occurring in u_i.

3 Scheduling Approach for Composite Actor Models

The absence of unquantified nondeterminism is a requirement for statistical model checking using Maude/PMaude and PVeStA [4]. Currently, there are two approaches for assuring this requirement: (i) by associating continuous probability distributions with message delays and computation time [4] and (ii) by introducing a scheduler which guarantees a deterministic execution order of messages in the actor system [6]. Both approaches however rely on a flat soup of actors and cannot, in their current state, handle composite actor models. In this work we adapt the scheduling approach (ii) for composite actor models. In order to promote modularity and to make the scheduling approach transparent, we propose a model transformation approach that does not require the specification of an actor to have knowledge about the composition of the system and about the scheduling approach as such. Given a composite actor specification M—that adheres to certain formal requirements—we generate a scheduled composite actor specification SM by a model transformation $M \rightarrow SM$ such that SM guarantees the absence of unquantified nondeterminism.

3.1 Well-Formedness Requirements

To enable statistical model checking we require the following well-formedness conditions on the original composite actor specification. We call a composite actor specification M well-formed if an initial state is defined and the following two formal requirements are fulfilled:

(1.) The specification must adhere to the composite actor model, i.e., entities must be specified as actors which communicate only via asynchronous message passing and there are only rewrite rules of type *consume*, *boundary-down*, or *boundary-up* (see Sect. 2.3).
(2.) The specification must have no unquantified nondeterminism in the choice of rewrite rules, i.e., for each message there is at most one matching rewrite rule.

Moreover, we assume w.l.o.g. that (3.) any message `m = (t, A<-m')` is executed at its activation time `t` (i.e. there exists a matching rule which at time `t` is applied to `m` and the actor `<A : T | C>`). This is not a restriction since e.g. the loss of `m` can easily be modeled by the rule `(t, A<-M)<A : T | C> => <A : T| C>`.

If a specification M fulfills these requirements, it is still nondeterministic (e.g. if several messages occur in a configuration there may be a nondeterministic choice to which message a rewrite rule will be applied), but our scheduling

approach eliminates all unquantified nondeterminism in the generated scheduled composite actor model specification SM. Note that we introduce requirement (2.) as the scheduling approach introduced by the model transformation only eliminates unquantified nondeterminism in consumption of messages, but does not in the choice of rewrite rules. Together with the scheduling approach, requirement (2.) ensures no unquantified nondeterminism in the whole specification, which is a requirement for statistical model checking with PVeStA.

3.2 Model Transformation $M \rightarrow SM$

Given a module M that specifies a well-formed composite actor model specification, the transformation $M \rightarrow SM$ creates a module SM, for which the transformation preserves all sort declarations, all operators, and all equations. Moreover, we introduce new sorts that define an explicit scheduler and new message types to represent scheduled messages as well as two auxiliary sorts of messages where active and scheduled messages are annotated by the address of the sending actor:

```
sort Scheduler ScheduleList ScheduleMsg LocActiveMsg LocScheduleMsg .
subsorts Scheduler ScheduleMsg LocScheduleMsg LocActiveMsg < Config .
subsort LocScheduleMsg < ScheduleList .
```

where terms of sort `ScheduleMsg` are created by the constructor `[_] : Msg -> ScheduleMsg` and represent messages that are emitted by rewrite rules and that are to be inserted in the scheduler. The constructors `{_,_} : Address Msg -> LocActiveMsg` and `[_,_] : Address ScheduleMsg -> LocScheduleMsg`, generate terms of sort `LocActiveMsg` and `LocScheduleMsg` respectively. Both contain either a term of sort `Msg` (in case of a `LocActiveMsg`) or `ScheduleMsg` (in case of a `LocScheduleMsg`) and a term of sort `Address`[2].Terms of sort `ScheduleList`represent a list of `ScheduleMsg` using the constructor `op _;_ : ScheduleList ScheduleList -> ScheduleList [assoc id: nil]`. A term of sort `Scheduler` is created by the constructor `op {_|_} : Float ScheduleList -> Scheduler`, which contains the current time and a list of messages that are to be scheduled.

For each rewrite rule r (which is either of type *consume, boundary-down, or boundary-up*) in M, a rewrite rule r' is added to SM, where each term `MSG` of sort `Msg` on the right side of r is transformed to a term `[MSG]` of sort `ScheduleMsg`. We also add the specification of our scheduling approach to the new specification (for details see Sect. 3.3).

Moreover, any initial state st of M is transformed to a state st' of SM as follows: (i) every message `MSG` of sort `Msg` is transformed to a term `[MSG]` of sort `ScheduleMsg` and (ii) an empty scheduler `{0.0 | nil}` is added. Then the result of the model transformation is the canonical form $[st']$ of st', which is of the

[2] Terms of sort `LocScheduleMsg` are used to store the address of the actor in whose configuration a scheduled message was emitted (or the `topmost` address if the message was emitted at the top-most level) and the scheduled message itself. Similarly, terms of sort `LocActiveMsg` are used to store the same address but of a scheduled message that has been made active by the scheduler. These auxiliary messages are used in our scheduling approach.

form `AC {0.0 | SL}`, where `AC` represents the message-free actor hierarchy of *st* and `SL` is the message list in the scheduler that now contains all messages of *st* as scheduled messages.

3.3 The Scheduling Approach

On a high level of abstraction, in order to remove all unquantified nondeterminism, our scheduling approach takes control of when a rewrite rule is executed. Our well-formedness requirements specify that only three types of rewrite rules (*consume*, *boundary-down*, and *boundary-up*) are allowed; and that there is no unquantified nondeterminism in the choice of these rules. Each of these rewrite rules requires an active message, i.e., non-scheduled message, to be present either at the hierarchy level of the actor or in its sub-configuration. Thus, a rewrite can occur only if an active message is present in the composite actor hierarchy. Furthermore, since we require that one single message is consumed by exactly one actor, this rewrite can be determined up to probabilistic choices. In order to ensure that only one message at a time is emitted we introduce an operation `step` and, similar to the special tick rule that is defined in [4], define a one-step computation of a model written in the scheduled composite actor model as a transition of the form

$$u \xrightarrow{step} v \to w$$

where

(i) u is a canonical term, which represents the global state of a scheduled composite actor system and in which all messages are contained in the scheduler.
(ii) v is a canonical term obtained by removing the next addressed scheduled message `[A,[(T, A'<-M)]]` from the scheduler, and by inserting it, as a non-scheduled addressed message `{A,(T, A'<-M)}`, into the configuration of the actor at address `A`. Additionally, the global time in the scheduler is advanced to T. This operation is called `step`.
(iii) w is a canonical term obtained after zero or one rewrites the actor performs to forward (*boundary-down* or *boundary-up*) or *consume* the message.

More precisely, the scheduling approach for the composite actor model consists of the following steps:

(1.) If no active message is present at any level of the hierarchy, the next scheduled message is marked as active and inserted in the top-most configuration by the scheduler.
(2.) Then, the message is pushed down the hierarchy by equational simplification, until it eventually reaches the configuration where it has been emitted. The resulting term is in canonical form.
(3.) Since the specification adheres to the composite actor model, there exists at most one rewrite rule of type *consume*, *boundary-down*, or *boundary-up* which consumes the message and possibly produces several scheduled messages.
(4.) Finally, the scheduled messages are pulled up the hierarchy by equational simplification, until it eventually reaches the top-most level where the scheduled messages are inserted into the scheduler which stores the messages in a strict

order, i.e., by the scheduled activation time and if equal by Maude's built-in term order. The resulting term is in canonical form.

In the following, we present the Maude specification of the scheduling approach which forms the final part of the model transformation $M \rightarrow SM$. To efficiently distinguish composite actor hierarchies that contain messages in any of its sub-configurations and those that do not, we use conditional sort memberships. The sort of term `ActorConfig`, a subsort of `Config`, thereby represents a configuration which contains no messages in any of its sub-configurations. In order to be able to make this distiction, we introduce the sort `InertActor`, which represents a term of sort `Actor`, that is either flat, i.e., that does not contain any subconfiguration, or whose subconfiguration is of sort `ActorConfig`. This is expressed by the following (conditional) sort membership axioms:

```
mb < A : T | config: AC, AS > : InertActor .
cmb ACT : InertActor if flatActor(ACT) .
op flatActor : Actor -> Bool .
eq flatActor(< A : T | config: C, AS >) = false .
eq flatActor(ACT) = true [owise] .
```

where `A` is a variable of sort `Address`, `T` of sort `ActorType`, `AC` of sort `ActorConfig`, `AS` of sort `AttributeSet`, `ACT` of sort `Actor`, and `C` of sort `Config`.

Having these (conditional) sort membership axioms, we can easily identify configurations that do not contain any active message: terms of sort `ActorConfig`. These are the configurations that can advance, i.e., the `step` operation can be called on these configurations. The `step` operation is defined as

```
op step : Config -> Config [iter] .
eq step(AC {gt | [A1, [(t1 , A <- M1)]]; SL}) = {A1, (t1 , A <- M1)} AC {t1 | SL} .
```

where `AC` is a term of sort `ActorConfig`, `gt` and `t1` of sort `Float`, `A1` and `A` of sort `Address`, `M1` of sort `Msg`, and `SL` of sort `ScheduleList`. The `step` equation can only be applied on a term of sort `Config` which consists of a term of sort `ActorConfig` and a term of sort `Scheduler`. It inserts the first message of the scheduler in the configuration and updates the time of the scheduler.

The remaining equations use the following variables: `A` and `A'` of sort `Address`, `T` of sort `ActorType`, `SM` of sort `ScheduleMsg`, `LSM` of sort `LocScheduleMsg`, `S` of sort `Scheduler`, `SL` of sort `ScheduleList` `C` of sort `Config`, `AS` of sort `AttributeSet`, `AM` of sort `Msg`, and finally `gt`, `t1`, and `t2` of sort `Float`.

To put scheduled messages as active messages back into the configuration where they have been emitted, the address of the actor containing the scheduled message needs to be stored together with the scheduled message. The equations `create-loc-msg1` and `create-loc-msg2` create a `LocScheduleMsg` from a scheduled message that is either in an actor's sub-configuration or at the top-most level of the hierarchy.

```
eq [create-loc-msg1] : <A : T | config: SM C, AS> = [A,SM] <A : T | config: C, AS> .
eq [create-loc-msg2] : SM C S = [ toplevel, SM ] C S .
```

Then, terms of sort `LocScheduleMsg` are pulled up the hierarchy by the `pull-up` equation and finally inserted in the scheduler by the `insert-in-scheduler` equation.

```
eq [pull-up] : < A : T | config: LSM C, AS > = LSM < A : T | config: C, AS > .
```

```
eq [insert-in-scheduler] : LSM S = insert(S, LSM) .
```

Terms of sort `LocActiveMsg` are pushed down the hierarchy by the equation `push-down` and are finally inserted in the correct subconfiguration by the equation `insert-in-configuration` or by the equation `insert-toplevel`, if the scheduled message has been emitted in the top-most level.

```
eq [push-down] : < A : T | config: C, AS > {A . A', AM}
  = < A : T | config: C {A . A', AM}, AS > .
eq [insert-in-configuration] : < A : T | config: C, AS > {A , AM}
  = < A : T | config: C AM, AS > .
eq [insert-toplevel] : {toplevel, AM} S = AM S .
```

Finally, the operator `insert` inserts a term of sort `LocScheduleMsg` into the scheduler.

```
op insert : Scheduler LocScheduleMsg -> Scheduler .
op insert : ScheduleList LocScheduleMsg -> ScheduleList .
eq insert({gt | SL}, LSM) = {gt | insert(SL,LSM)} .
eq [insert-list] : insert([A1,[(t1, A<-M1)]]; SL, [A2,[(t2, A'<-M2)]]) =
  if (t1 < t2) or ((t1 == t2)) and lt(M1,M2) then
    [A1,[(t1, A<-M1)]] ; insert(SL,[A2,[(t2, A'<-M2)]])
  else
    [A2,[(t2, A'<-M2)]] ; [A1,[(t1, A<-M1)]] ; SL
  fi .
eq insert(nil, LSM) = LSM .
```

3.4 Correctness of the Scheduling Approach

In this section we analyse the correctness of the scheduling approach by proving that the equational specification is terminating and confluent modulo associativity and commutativity and by showing that the introduction of the scheduler eliminates all unquantified nondeterminism.

Proposition. The equational specification of the scheduling approach is terminating and confluent modulo associativity and commutativity (AC).

Proof sketch (termination). The equations that need to be discussed are the recursive ones: `insert-list`, `pull-up`, and `push-down`. `insert-list` terminates since the `ScheduleList` argument of the `insert-list` operator in the recursive call gets smaller. `pull-up`, for a specific scheduled message, terminates, since the distance of the message's location to the toplevel gets smaller with each level of recursion. `push-down`, for a specific addressed active message, terminates, since the distance between the message's address and the addresses of the actors in the sub-configuration the message is inserted into gets small with each level of recursion. □

Proof sketch (confluence modulo AC). As the equational specification of the scheduling approach is terminating, it is sufficient to prove local confluence. For most equations local confluence is achieved by applying the AC property of the configurations. The only exception are the equations that insert a message into the scheduler. For the insertion equations local confluence results from the fact that the message order in the scheduler list is a total ordering. The messages are ordered by activation time and if equal by Maude's built-in term order. □

Lemma 1. The scheduling approach emits at most one active message at a time (after a call to the `step` operation).

Proof sketch. The `step` operator takes an `ActorConfig` together with a `Scheduler` as its argument. By construction an `ActorConfig` does not contain any message and all messages are in the scheduler. `step` emits one addressed active message into the toplevel. If the message's address is the toplevel the message is then "unwrapped", i.e., converted into an active message; otherwise it is pushed down to the level it addresses and is "unwrapped" there. As only one addressed active message is emitted, the scheduling approach emits at most one active message after a call to the `step` operation. □

Theorem 1. Let SM be a scheduled composite actor specification. If SM satisfies the requirements for the scheduling approach for scheduled composite actor models (see Sect. 3.1), then for any one-step computation $u \xrightarrow{step} v \to w$ of SM, there is no unquantified nondeterminism possible; however, there may be probabilistic choices in the application of an actor rewrite rule in v.

Proof sketch. We prove the theorem by reductio ad absurdum. Assume there exists unquantified nondeterminism in a one-step computation of SM. As a `step` operation is possible for u, the configuration in a state u has to be of sort `ActorConfig`, which means that it contains no active messages. As all the rewrite rules require an active message, no unquantified nondeterminism is possible in u. w is a state after a rewrite triggered by an active message. However, in a scheduled composite actor model specification all rewrite rules may only produce scheduled messages and thus no active messages. Since no active messages are present in the configuration of a state w, no rewrites are possible and thus no unquantified nondeterminism is possible in w. Thus there has to be unquantified nondeterminism in a state v. Because of the well-formedness of the original specification an actor can only react to an active message with rewrite rules that fulfill the unquantified nondeterminism requirement, there have to be at least two active messages in the configuration of a state v to get unquantified nondeterminism. Since the `step` operation works on terms of sort `ActorConfig` in which there are no active messages, this means that the scheduling approach has to emit more than one active message. Lemma 2 however states that the scheduling approach emits at most one active message after a call to the `step` operation. Thus there cannot be two active messages in the configuration of a state u and as a consequence, no unquantified nondeterminism is possible in u. As there is no unquantified nondeterminism possible in u, v, and w, there is no unquantified nondeterminism possible for any one-step computation $u \xrightarrow{step} v \to w$ of SM. □

3.5 Soundness of the Model Transformation

In this section we analyse the soundness of the model transformation. In particular, we show that SM is a simulation of M for any model transformation $M \to SM$ where M is well-formed. Correctness needs the additional assumption of strong rule confluence: if M is well-formed and strongly rule confluent, then M and SM are time-passing bisimilar.

First, we introduce some definitions for (timed) probabilistic labelled transition systems and recall the notions of simulation and bisimilation [18].

Definition (Timed Probabilistic Labelled Transition System). A timed probabilistic (tp) labelled transition system $\mathcal{A}=(A, L, \mu, \xrightarrow{l,t})$ consists of a set of states A, a family of probabilistic distributions $\mu : A \times L \to [0,1]$ and a labelled transition relation $\xrightarrow{l,t} \subseteq A \times A$ where $l \in L$ and $t \in \mathbb{R}_{\geq 0}$. We write $u \xrightarrow{l,t} v$ for a transition from u to v where the label l indicates which rule r of the rewriting specification is applied to u, $\mu(u, l_r)$ is the probability, and t is the time delay, i.e., the difference of global time between u and v.

In the following, the specifications M and SM form the labelled transition systems $\mathcal{M} = (Can(M), L_{\mathcal{M}}, \mu_{\mathcal{M}}, \xrightarrow{L_{\mathcal{M}},t}_{\mathcal{M}})$ of M and $\mathcal{SM} = (Can(SM), L_{\mathcal{SM}}, \mu_{\mathcal{SM}}, \xrightarrow{L_{\mathcal{SM}},t}_{\mathcal{SM}})$ of SM, where $L_{\mathcal{SM}} = L_{\mathcal{M}}$ is the set of rules in M, $\mu_{\mathcal{M}}$ is a family of distribution functions, $\mu_{\mathcal{SM}}$ is a family of distribution functions which coincides with $\mu_{\mathcal{M}}$ on the image of the model transformation, and $Can(M)$ and $Can(SM)$ are the canonical terms of sort `config` of M and SM, respectively. The labelled transition systems $\mathcal{M}$ and $\mathcal{SM}$ describe the set of all well-timed one-step rewrites of the probabilistic rewrite theory of M and SM, respectively.

Definition (Simulation and Bisimulation). Given timed probabilistic (tp) labelled transition systems $\mathcal{A}=(A, L_{\mathcal{A}}, \mu_{\mathcal{A}}, \xrightarrow{l,t}_{\mathcal{A}})$ and $\mathcal{B}=(B, L_{\mathcal{B}}, \mu_{\mathcal{B}}, \xrightarrow{l,t}_{\mathcal{B}})$ and a bijection of the labels $L_{\mathcal{A}} \leftrightarrow L_{\mathcal{B}}$, where $\mu_{\mathcal{A}} : A \times L_{\mathcal{A}} \to [0,1]$ and $\mu_{\mathcal{B}} : B \times L_{\mathcal{B}} \to [0,1]$ are families of probability distributions. Then a simulation of tp-transition systems is a binary relation $H \subseteq A \times B$ such that if $a \xrightarrow{l_{\mathcal{A}},t}_{\mathcal{A}} a'$ and aHb then there is b' such that $b \xrightarrow{l_{\mathcal{B}},t}_{\mathcal{B}} b'$ and $a'Hb'$, $\mu_{\mathcal{A}}(a, l_{\mathcal{A}}) = \mu_{\mathcal{B}}(b, l_{\mathcal{B}})$, and $l_{\mathcal{A}} \leftrightarrow l_{\mathcal{B}}$. If both H and H^{-1} are simulations, then we call H a bisimulation.

Remark (Relation between *M* and *SM*). We relate the scheduled composite actor specification SM with the composite actor specification M as follows: For any canonical term u of SM in form `AC {t | SL}` one can construct a term $\hat{u}$ of M by inserting all messages of `SL` into the right subconfiguration of `AC`: for each term `[A , [(at1, A1 <- M)]]` of sort `LocScheduleMsg` of `SL`, the message `(at1, A1 <- M)` is inserted into the subconfiguration `AC` at adress `A` (or into the toplevel configuration of `AC` if `A = toplevel`).

Theorem 2. Let M be a well-formed composite actor model specification and SM the scheduled composite actor model specification that is the result of the model transformation $M \to SM$. Then $\mathcal{SM}$ is a simulation of $\mathcal{M}$.

Proof sketch. $H : \mathcal{SM} \to \mathcal{M}$ is a simulation where $H = \{(u, \hat{u}) | u =$ `AC {gt | SL}` for some `AC, gt, SL`$\}$. □

To show the existence of a time-passing bisimulation between $\mathcal{M}$ and $\mathcal{SM}$ we need some further definitions:

Definition (Delay and Zero-Time Transition). We distinguish between delay transitions $a \xrightarrow{l,t} a'$ with $t > 0$, which indicate the passing of time, and zero-time transitions $a \xrightarrow{l,0} a'$ which are executed without taking any time. By $A_{>0} = \{a \in A |$ there is no a', l such that $a \xrightarrow{l,0} a'\}$ we denote the set of all terms to which no zero-time transition is applicable.

Definition (Complete Zero-Time *-Transition). A zero-time *-transition $a \Rightarrow_L a'$ consists of a finite sequence of zero-time transitions; it is complete if it cannot be extended further:

For $n \in \mathbb{N}, L : \{1, \ldots, n\} \to L_{\mathcal{A}}$ we call $a \Rightarrow_L a'$ a zero-time transition if and only if $\exists a_1, \ldots, a_n$ such that $a \xrightarrow{L(1),0} a_1 \to \ldots \to a_{n-1} \xrightarrow{L(n),0} a', a_n = a'$. It is called complete if $a' \in A_{>0}$.

If $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a permutation, we write $a \Rightarrow_{\pi(L)} a'$ for the zero-time *-transition $a \Rightarrow_{L'} a'$, where $L'(i) = L(\pi(i))$ for $i = 1, \ldots, n$.

Definition (Time-passing Simulation). Given tp-labelled transition systems $\mathcal{A} = (A, L_{\mathcal{A}}, \mu_{\mathcal{A}}, \xrightarrow{l,t}_{\mathcal{A}})$ and $\mathcal{B} = (B, L_{\mathcal{B}}, \mu_{\mathcal{B}}, \xrightarrow{l,t}_{\mathcal{B}})$ and a bijection $L_{\mathcal{A}} \leftrightarrow L_{\mathcal{B}}$. Then $H \subseteq A \times B$ is called a time-passing simulation if $\forall a, b, l_{\mathcal{A}}, t > 0, a', L, a'' :$

$$
\begin{aligned}
& aHb \wedge a \xrightarrow{l_{\mathcal{A}},t}_{\mathcal{A}} a' \wedge a' \Rightarrow_L a'' \text{ complete} \\
& \Rightarrow \exists \text{ a permutation } \pi : \{1, \ldots, n\} \to \{1, \ldots, n\}, b', b'', l_{\mathcal{B}}, \text{ s.t.} \\
& b \xrightarrow{l_{\mathcal{B}},t}_{\mathcal{B}} b' \wedge b' \Rightarrow_{\pi(L)} b'' \wedge a'' H b'' \wedge l_{\mathcal{A}} \leftrightarrow l_{\mathcal{B}}
\end{aligned}
$$

Corollary. Every simulation (of tp transition systems) is a time-passing simulation.

Definition (Strong Zero-Time Rule Confluence). An actor specification is called strongly zero-time rule confluent if the order of applying any two zero-time actor rules to the same term does not matter:

A specification M is strongly zero-time rule confluent, if for any two zero-time rules r_1, r_2 with labels l_{r_1}, l_{r_2} and for all canonical terms u, v, w of sort `Config`: $u \xrightarrow{l_{r_1},0} v$ and $u \xrightarrow{l_{r_2},0} w$ implies that there exist v', w' such that $v \xrightarrow{l_{r_2},0} v'$ and $w \xrightarrow{l_{r_1},0} w'$ and $v' = w'$.

Many composite actor specifications satisfy the condition of strong zero-time rule confluence; in particular, except for one [12], all our published case studies studies [21,20,11] are strongly zero-time rule confluent.

Lemma 2. Let M be a well-formed and strongly zero-time rule confluent composite actor specification. For any zero-time *-transition $v_0 \Rightarrow_L v_n$ of M of form $v_0 \xrightarrow{L(1),0} v_1 \ldots v_{n-1} \xrightarrow{L(n),0} v_n$ and any rule r which is applicable with label l to a message m in v_0, the following holds:

(1.) If r is not applied in any of the v_i then m occurs in all v_i and r stays applicable with label l in all $v_i, i = 0 \ldots n$.

(2.) If M is strongly zero-time rule confluent and $v_{n-1} \xrightarrow{L(n),0} v_n$ is the first rule application of r with label $L(n) = l$, then for appropriate v_i' , $v_0 \xrightarrow{L(n),0} v_1' \xrightarrow{L(1),0} v_2' \ldots v_{n-1}' \xrightarrow{L(n-1),0} v_n'$ and $v_n = v_n'$.

Proof. Proof by induction on the length n of the transition. □

Lemma 3. Let M be well-formed and strongly zero-time rule confluent and SM be the result of the model transformation $M \to SM$. For all canonical terms v_0 of sort `Config` of M , w_0 of SM with $\hat{w}_0 = v_0$, and any zero-time *-transition

$v_0 \Rightarrow_L v'$ with $v' \in M_{>0}$ there exist $w' \in SM_{>0}$ and a permutation π such that $w_0 \Rightarrow_{\pi(L)} w'$ and $\hat{w}' = v$.

Proof. Proof by induction on the length of the transition using Lemma 2. □

Theorem 3. Let M be a well-formed and strongly zero-time rule confluent composite actor model specification and SM be the result of the model transformation $M \to SM$. Then M and SM are time-passing bisimilar.

Proof. Consider the relation H as in Theorem 2 and let $H_{>0} \subseteq SM_{>0} \times M_{>0}$. □

Remark. One can also solve the general case where interdependent messages have the same activation time and the specification is not zero-time rule commutative by assigning random delays to the scheduled activation times of the dependent messages. As in [4], we assume that the probability that a random number is sampled twice is 0. If this process is recursively applied to the dependent messages in the scheduler, a fix point is reached in which there exist no two dependent messages with the same scheduled activation time.

3.6 Example: Scheduling Approach for the Composite Actor Model

As an example of how the scheduling approach works, we model a forest of binary trees, where the leaf nodes send messages to each other while the intermediate nodes only forward and delay these messages. The leaf nodes in the forest are of type `Leaf` and the intermediate nodes of type `Node`. Message contents are created by the operator `cnt : -> Contents`. The following listing shows the specification of the behavior of the intermediate and leaf nodes, where the original composite actor specification M is shown on the left side and the modified rules of the specification SM after the transformation on the right side:

```
crl [intermediate-boundary-up1] :
  <A : Node |config: (t, A'<-cnt) C> =>
  <A : Node |config: C> (t+0.1, A'<-cnt)
  if |A'| <= |A| .
crl [intermediate-boundary-up2] :
  <A : Node |config: (t, A'<-cnt) C> =>
  <A : Node |config: C> (t+0.1, A'<-cnt)
  if |A'| > |A| /\ pref(A', |A|) =/= A .
rl [intermediate-boundary-down] :
  <A : Node |config: C> (t, A.A'<-cnt) =>
  <A : Node |config: C (t+0.1, A.A'<-cnt)> .
rl [leaf-receive-and-send] :
  <A : Leaf |nil> (t, A<-cnt) =>
  <A : Leaf |nil> (t+0.1, rndA(|A|)<-cnt) .
```

```
crl [intermediate-boundary-up1] :
  <A : Node |config: (t, A'<-cnt) C> =>
  <A : Node |config: C> [(t+0.1, A'<-cnt)]
  if |A'| <= |A| .
crl [intermediate-boundary-up2] :
  <A : Node |config: (t, A'<-cnt) C> =>
  <A : Node |config: C> [(t+0.1, A'<-cnt)]
  if |A'| > |A| /\ pref(A', |A|) =/= A .
rl [intermediate-boundary-down] :
  <A : Node |config: C> (t, A.A'<-cnt) =>
  <A : Node |config:C [(t+0.1,A.A'<-cnt)]> .
rl [leaf-receive-and-send] :
  <A : Leaf |nil> (t, A<-cnt) =>
  <A : Leaf |nil> [(t+0.1, rndA(|A|)<-cnt)] .
```

In both specifications, `A` and `A'` are variables of sort `Address`, `t` of sort `Float`, `C` of sort `Config`, `pref: Address Nat -> Address` an operator returning a prefix of a given length, `|_| : Address -> Nat` an operator returning the length of a given address, and `rndA: Nat -> Address` an operator returning a random Address of a given length for a binary tree.

The initial configuration consists of two actor trees and the top-level scheduler which contains one message. The following listing shows this configuration, where the original one (M) is shown on the left, and SM on the right.

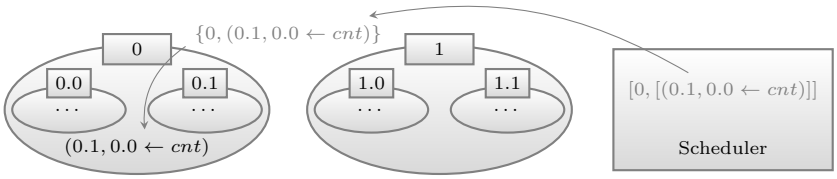

Fig. 2. The term in canonical form after one step.

```
<0: Node | config: (0.1, 0.0 <- cnt)
  <0.0 : Leaf | nil> <0.1 : Leaf | nil>>
<1: Node | config:
  <1.0 : Leaf | nil> <1.1 : Leaf | nil>>
```

```
<0: Node | config:
  <0.0 : Leaf | nil> <0.1 : Leaf | nil>>
<1: Node | config:
  <1.0 : Leaf | nil> <1.1 : Leaf | nil>>
{0.0 | [0, [(0.1, 0.0 <- cnt)]]}
```

Figure 2 illustrates the resulting term in canonical form after one step, where the term in canonical form is depicted in solid black and in addition, the intermediate messages in gray. The arrows show the corresponding equational simplifications.

4 Statistical Model Checking Methodology

We propose the following methodology to verify hierarchically structured distributed systems:

(1.) Specification of the real-world system as a composite actor system in Maude/ PMaude using the composite actor model as a foundation and fulfilling the formal requirements for the scheduling approach for composite actor models.
(2.) Definition of standard probabilistic temporal logic and quantitative temporal logic properties for the system.
(3.) (Automated) transformation of the model specified in (1.) to a scheduled composite actor model specification and specification of an initial state which consists of the initial state of the specification of the model defined in (1.) and an instance of the top-level scheduler of the scheduling approach for the composite actor model.
(4.) Formal analysis of the properties defined in (2.) over the initial state using the statistical model checker PVeStA.

Besides the formal requirements, we further require that the specification does not contain zero-time loops, i.e., a recurring series of messages is produced where each message has the same activation time.

More technically the statistical model checker PVeStA calls the operator `run` to start the execution of a sample of a composite actor model. The operator is repeatedly called by PVeStA until a specified amount (denoted by the variable `LIMIT`) of global time has passed. In Maude, the `run` operator is defined as follows:

```
op run : Config Float -> Config .
eq run(AC {gt | SL}, LIMIT) =
  if (gt <= LIMIT) then run(step(AC {gt | SL}), LIMIT) else AC {gt | SL} fi .
```

Several case studies have been conducted to validate this methodology. For example, the approach has been used to detect bugs in the design of a group key distribution service [21,11], to predict the performance of a distributed broker-based Publish/Subscribe service [21,20], and to improve the availability of service

of Internet-based service architectures such as Clouds using a denial of service protection mechanism together with dynamic resource provisioning [12].

From a more practical point of view, the statistical model checking methodology for hierarchical specifications based on the composite actor model has proven itself effective in our case studies. Compared to LTL model checking, we were able to model check meaningful qualitative as well as quantitative properties of rather large instances of the specifications. E.g., during the work on [12], we model checked quantitative properties with a high confidence of 99% on specification instances with up to 500 individual actors. Thereby, the statistical model checking process was running for several hours on a cluster of 32 machines. Model checking just simple qualitative properties with LTL model checking on the same specification instances would have exceeded a graspable timeframe.

5 Related Work & Concluding Remarks

This work is mainly related to the ideas of actors [16,15,2], PMaude [4], the statistical model checker PVeSTA [5], and the original scheduling approach for flat actor configurations mentioned in [6]. Only recently Ölveczky et al. [8] proposed a probabilistic strategy language for probabilistic rewrite theories that is implemented in Maude and offers the possibility of statistical model checking with a model checking algorithm implemented as a Maude meta-level functionality. Bruni et al. [9] have shown that a framework to describe adaptive behavior in multi-layered component hierarchies can naturally be realized in Maude based on the Reflective Russian Dolls Model and quantitatively analyzed using PVeSTA.

In a broader sense this work is related to process calculi such as the Mobile Ambients calculus and its probabilistic extension [10,17]. While process calculi mainly focus on dynamic behavior, our approach also emphasizes the representation of data.

In this paper we have presented the composite actor model and argued that it is well-suited for specifying concurrent Cloud and Internet-based systems which are composed of various participants and subsystems and are hierarchically structured in different layers and networks. Our model extends the actor model of concurrent computation and supports an arbitrary hierarchical composition of entities. As a second main result we have defined a model transformation which extends a composite actor specification by a new scheduling approach that guarantees the absence of nondeterminism and, as a consequence, enables statistical model checking. To show the soundness of our approach we have proven termination and confluence of the (equational part of the) scheduler specification and shown the absence of unquantified nondeterminism in scheduled composite actor specifications. To prove the soundness of the model transformation we introduced the notions of strong zero-time rule confluence and time-passing bisimulation and showed that the transformation is a time-passing bisimulation for strongly zero-time rule confluent composite actor specifications.

Until today we have successfully applied the composite actor approach to three non-trivial Cloud case studies [12,21,20,11] by formally specifying and analyzing them using Maude and PVeSTA.

In the future we plan to extend and complement our composite actor modeling approach by a correct-by-construction model-driven program synthesis. As a target system we choose the ØMQ (zeromq) socket library [1] to act as a concurrent framework for Cloud services. Currently, we have partially implemented one of our case studies [12] as a ØMQ application based on the Maude model and shown that large parts of the transformation process can be automated.

References

1. ØMQ: The Intelligent Transport Layer. `http://www.zeromq.org/` (2012/08/07).
2. G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.
3. G. Agha and C. Hewitt. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, 1988.
4. G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. *ENTCS*, 153(2):213–239, 2006.
5. M. AlTurki and J. Meseguer. PVeStA: A parallel statistical model checking and quantitative analysis tool. In *CALCO*, volume 6859 of *LNCS*, pages 386–392, 2011.
6. M. AlTurki, J. Meseguer, and C. A. Gunter. Probabilistic Modeling and Analysis of DoS Protection for the ASV Protocol. *ENTCS*, 234:3–18, 2009.
7. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
8. L. Bentea and P. C. Ölveczky. Probabilistic real-time rewrite theories and their expressive power. In *FORMATS*, volume 6919 of *LNCS*, pages 60–79, 2011.
9. R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. Modelling and Analyzing Adaptive Self-assembly Strategies with Maude. In *WRLA*, volume 7571 of *LNCS*, pages 118–138, 2012.
10. L. Cardelli and A. Gordon. Mobile ambients. In *FOSSACS*, pages 140–155, 1998.
11. J. Eckhardt. A Formal Analysis of Security Properties in Cloud Computing. Master's thesis, LMU Munich, TU Munich, 2011.
12. J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, and M. Wirsing. Stable Availability under Denial of Service Attacks through Formal Patterns. In *FASE*, volume 7212 of *LNCS*, pages 78–93, 2012.
13. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In *WRLA*, volume 71 of *ENTCS*, pages 162–187, 2002.
14. P. Haller and F. Sommers. *Actors in Scala*. Artima Developer, 2012.
15. C. Hewitt and H. G. Baker. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992, 1977.
16. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.
17. M. Kwiatkowska, G. Norman, D. Parker, and M. G. Vigliotti. Probabilistic Mobile Ambients. *TCS*, 410(12–13):1272–1303, 2009.
18. K. G. Larsen and A. Skou. Bisimulation through Probabilistic Testing. *Inf. Comput.*, 94(1):1–28, 1991.
19. J. Meseguer and C. L. Talcott. Semantic Models for Distributed Object Reflection. In *ECOOP*, pages 1–36, 2002.
20. T. Mühlbauer. Formal Specification and Analysis of Cloud Computing Management. Master's thesis, LMU Munich, TU Munich, 2011.
21. M. Wirsing, J. Eckhardt, T. Mühlbauer, and J. Meseguer. Design and Analysis of Cloud-Based Architectures with KLAIM and Maude. In *WRLA*, volume 7571 of *LNCS*, pages 54–82, 2012.