# MultiVeStA: Statistical Model Checking for Discrete Event Simulators [*]

Stefano Sebastio
IMT Institute for Advanced Studies Lucca
stefano.sebastio@imtlucca.it

Andrea Vandin
IMT Institute for Advanced Studies Lucca
andrea.vandin@imtlucca.it

## ABSTRACT

The modeling, analysis and performance evaluation of large-scale systems are difficult tasks. Due to the size and complexity of the considered systems, an approach typically followed by engineers consists in performing simulations of systems models to obtain statistical estimations of quantitative properties. Similarly, a technique used by computer scientists working on quantitative analysis is Statistical Model Checking (SMC), where rigorous mathematical languages (typically logics) are used to express systems properties of interest. Such properties can then be automatically estimated by tools performing simulations of the model at hand. These property specifications languages, often not popular among engineers, provide a formal, compact and elegant way to express systems properties without needing to hard-code them in the model definition. This paper presents *MultiVeStA*, a statistical analysis tool which can be easily integrated with existing discrete event simulators, enriching them with efficient distributed statistical analysis and SMC capabilities.

## Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking, Statistical methods; I.6.8 [**Types of Simulation**]: Discrete event; D.3.2 [**Language Classifications**]: Constraint and logic languages

## Keywords

Discrete event simulation, quantitative analysis, statistical analysis, statistical model checking.

## 1. INTRODUCTION

Many complex systems can be modeled as sets of interacting components. When dealing with large numbers of components, an analytical approach to their study may be too complex to handle. In these cases, a statistical approach based on simulations can be a viable solution to cope with the large-scale nature of the considered systems.

A possible way to model the systems dynamics is via *discrete event simulations* (DESs), where the system evolves in discrete steps according to the generation (and execution) of events, which model internal actions of components as well as interactions among them. In a DES, events occur at discrete points in time, and the time-flow is discretized in jumps among their occurrence, simplifying the modeling of the evolution of the systems. Intuitively, this abstraction is justified by the fact that not all the time instants are interesting, i.e. if there are neither events nor changes in the state. The popularity of DES is witnessed by the wide set of existing DES-based tools. We cite among others: ns-3 [27], OMNeT++ [35], PeerSim [22], DEUS [4], Alchemist [24].

Exact quantitative systems analysis can be done via Probabilistic Model Checking [9]. With such a technique, roughly, systems are abstracted in mathematically rigorous models, while systems properties are expressed in rigorous formalisms (typically logics), and are evaluated by exhaustively exploring models state-spaces. Although being powerful, this approach suffers the well known state-space explosion problem, i.e. it does not scale well when the system complexity grows.

A possible solution is the use of statistical analysis technique like *Statistical Model Checking* (SMC) [31], where systems properties are statistically estimated by resorting to simulations. The main advantage of SMC is that it only requires to simulate models, rather than generating theirs state-spaces. This in turn implies that SMC algorithms can be easily distributed, by distributing the simulations. Conversely, the main drawback is that it does not provide exact results, but only statistical guarantees on the evaluated properties.

**Contribution.** In this paper we present MultiVeStA, a lightweight Java tool which allows to enrich existing discrete event simulators with distributed statistical analysis capabilities. The tool extends VeStA [1,32] and PVeStA [2]. To sum up, MultiVeStA offers: (1) a clean way to integrate existing discrete event simulators; (2) a language (MultiQuaTEx) to compactly express systems properties; (3) the estimation of the expected values of MultiQuaTEx expressions wrt $n$ independent simulations, with $n$ large enough to respect a user-specified confidence interval; (4) the plot of the results in a minimal GUI, and the generation of gnuplot input files; (5) a client-server architecture to distribute simulations.

[*]Research partly supported by the European IP 257414 ASCENS, the European STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.

We validate our proposal by discussing the integration of MultiVeStA with DEUS, and prove the usefulness of the integration by analyzing a model of a cloud computing system. The scenario, which will be used throughout the paper to ease presentation, regards the sharing of computing resources among groups of volunteers autonomous entities. One of the challenges raised in the design of such kind of systems is the distribution of the tasks requests among the participant nodes. In a recent paper [30] we proposed an approach based on Ant Colony Optimization (ACO) [16] to solve the task distribution problem, where a *colored pheromone field* is used to create a distributed guide for the search of a node willing to execute a task. We distinguish among two classes of tasks, namely *large* and *small*, characterizing tasks with high resources requirements and low Quality of Service (QoS) restrictions, and vice-versa, respectively. Interesting observations for such systems are the number of tasks that the system is able to satisfy respecting the associated QoS.

**Synopsis.** §2 outlines the main characteristics of VeStA and PVeStA. Then §3 and §4 discuss the main improvements introduced in MultiVeStA: §3 focuses on the extended property specification language, while §4 focuses on the integration of new simulators and on the improved presentation of results. §5 validates our tool wrt DEUS, §6 discusses some related works, while §7 reports concluding remarks and future works.

## 2. VeStA, PVeStA and QuaTEx

VeStA [32] is a statistical model checker and quantitative analyzer for probabilistic systems. The tool performs a statistical evaluation (Monte Carlo based) of properties expressed as quantitative temporal expressions (QuaTEx) [1], allowing to query about expected values of observations performed on simulations of a probabilistic model. Some example queries that have been encoded are: "What is the expected probability that at least $n$ robots reach their goal?" [14], or "What is the expected fraction of clients that successfully connect to a server under denial of service attack?" [17].

The tool also supports the transient fragment of probabilistic computation tree logic (PCTL) [19] and continuous stochastic logic (CSL) [8, 10], for which SMC algorithms based on the invocation of a series of inter-dependent statistical hypothesis testing are implemented [31]. However, in this work we focus on QuaTEx as it generalizes the two logics [1].

Coming to the model specification languages (and their underlying simulation engines), the tool supports a language to express discrete- and continuous-time Markov chains, and PMaude, an executable algebraic specification language to describe models as probabilistic rewrite theories [1].

The analysis algorithms of VeStA are independent of the model specification languages: it is only assumed that DES can be performed on the model. Thus, in principle, as discussed in [32] VeStA could be plugged to any discrete event simulator offering: a method to initialize a simulation, a method to compute one step of simulation, and a method that duplicates and returns the current state. By doing this, VeStA would enrich the underlying simulator with a property specification language (QuaTEx), and would automate the analysis tasks by means of statistical analysis and SMC capabilities. Unfortunately, it is not possible to integrate new simulators in VeStA, as it does not offer this feature, and thus it would be necessary to study and modify its non publicly available source code. One of the extensions proposed in this paper is a clean way to integrate new simulators.

```
tOp() = if { s.rval(0) == 1.0 } then s.rval(6)
                                  else #tOp() fi;
eval E[ tOp() ] ;
```

**Listing 1: The QuaTEx expression $Q_1$**

VeStA estimates the expected value of QuaTEx expressions wrt two user-defined parameters: $\alpha$ and $\delta$. Specifically, estimations are computed as the mean value of $n$ samples (obtained from $n$ simulations), with $n$ large enough to grant that the size of the $(1-\alpha)*100\%$ *Confidence Interval* (CI) is bounded by $\delta$. The CI, computed using the *Student's t-test*, has center in $\overline{x}$ and radius $t_{n-1,\alpha/2}\sqrt{\hat{\sigma}/n}$, where $t_{n-1,\alpha/2}$ is computed with the Student's t-distribution, and $\hat{\sigma}$ is the sample variance [33]. The $\delta$ defines a stopping criteria to the samples generation, i.e. when the CI radius is less or equal to $\delta/2$. Thus, with probability $(1-\alpha)$, the sample values are in the provided CI, or, in other words, if a QuaTEx expression is estimated as $\overline{x}$, then, with probability $(1-\alpha)$, its actual expected value belongs to the interval $[\overline{x}-\delta/2, \overline{x}+\delta/2]$.

Before defining a QuaTEx expression, it is necessary to specify the state characteristics of interest to be observed. This model-specific step is necessary to "connect" QuaTEx with the simulated model. The state observations have to be offered via the `rval(`*i*`)` predicate which returns a number in the real domain (i.e. it is a *real-typed predicate*) for each observation $i$. For example, for the considered cloud scenario, `s.rval(0)` is defined such that it reduces to 1.0 if the simulation is completed in the current simulation state (denoted by the keyword `s`), and 0.0 otherwise. While `s.rval(1)` returns the current simulated time, `s.rval(2)` counts the number of performed steps of simulation, `s.rval(5)` corresponds to the average time spent by tasks in queues, `s.rval(6)` returns the number of running or executed tasks, and `s.rval(11)`, `s.rval(12)`, `s.rval(13)` correspond, respectively, to the ratio of successfully executed small, large and all tasks.

A detailed description of QuaTEx and of the procedure to estimate its expressions is given in [1]. This section discusses the syntax of the language with the help of the expression $Q_1$ of Listing 1, which, intuitively, reads as: "*compute the expected number of tasks executed in a simulation*".

Listing 2 presents QuaTEx's syntax. A QuaTEx expression (line `1`) consists of a set of *parameterized recursive temporal operators* "*DS*" ($Q_1$ has just one named `tOp()`), followed by

```
   Q   ::= DS eval E[PExp];                                  1
   DS  ::= set of Defn                                       2
  Defn ::= N(x_1,...,x_m) = PExp;                            3
  SExp ::= c | rval(i) | F(SExp_1,...,SExp_k) | x_j          4
  PExp ::= SExp | #N(SExp_1,...,SExp_n) |                    5
           if SExp then PExp_1 else PExp_2 fi                6
```

**Listing 2: QuaTEx syntax**

an *eval clause* "`eval E[`*PExp*`]`". A *definition of a temporal operator* "*Defn*" (line 3) consists of the name of the operator (e.g. `tOp()`) and of a path expression representing its body.

Line 4 specifies *state expressions* "*SExp*", i.e. real-typed expressions evaluated on a simulation state. Where an *SExp* is either a real number, an invocation of `rval`, an arithmetic or boolean expression involving state expressions, or a variable.

Finally, "*PExp*" (lines 5-6) is a *path expression*, i.e. a real-typed predicate possibly evaluated after performing steps of simulation. A *PExp* is either a state expression, a temporal operator preceded by the symbol `#`, or an `if_then_else` statement. The `if_then_else` statement behaves as expected, instead `#`$N(SExp_1, \ldots, SExp_n)$ means: "perform a step of simulation, and evaluate $N(SExp_1, \ldots, SExp_n)$ in the obtained state". In fact, the operator `#` (named "next"), triggers the execution of a step of simulation (in standard terminology [9], it is a primitive one-step temporal operator). Noteworthy, if it is used in recursive temporal operator definitions (like in $Q_1$), `#` allows to query properties of states obtained after an unspecified number of steps of simulation.

It is assumed that expressions are properly typed: the guards of `if_then_else` statements must be booleans, while the *PExp* appearing in `eval E[`*PExp*`]` must be real. Moreover, the semantics is given for a subset of the QuaTEx expressions named *bounded expressions*, where the value of any *PExp* can be determined from a finite number of steps of simulation.

$Q_1$ is now discussed in detail. VeStA associates `s` to the initial state of the simulation, and then evaluates the guard of the `if_then_else` statement (`s.rval(0) == 1.0`): "*is* `rval(0)` *equal to* 1.0 *in* `s`*?*", i.e. "*is* `s` *a final state of the simulation?*". If the guard is evaluated to true, then `s.rval(6)` is returned, i.e. the number of executed tasks. Otherwise the expression is evaluated as `#tOp()`: VeStA orders the simulator to advance of one step, updates `s`, and then recursively evaluates `tOp`. This is done until a state where `rval(0)` is evaluated to 1.0 is reached. The evaluation is repeated for several simulations, until the mean $\overline{x}$ of the obtained results satisfies the user-specified confidence interval, and $\overline{x}$ is returned as result. Noteworthy, suppose to have defined another `rval(`$j$`)`, such that it evaluates to 1.0 in case a certain event $e$ happens in a simulation, and to 0.0 otherwise. Then, by replacing `rval(6)` with `rval(`$j$`)`, $Q_1$ would estimate the probability of $e$ in a simulation. Indeed, `rval(`$j$`)` can be thought of as a random variable following a Bernoulli distribution, and thus its expected value represents the probability of $e$.

Another interesting expression is $Q_2$ of Listing 3, which reads: *compute the expected value of the mean time spent in queues by tasks, imposing* 30 *as maximum simulated time.* $Q_2$ shows that temporal operators can have parameters (variables). Variables have to be bounded, i.e. if they are in the right-

```
tOpB(x) = if { s.rval(1) >= x } then s.rval(5)
                                else #tOpB({x}) fi;
eval E[ tOp(30.0) ] ;
```

**Listing 3: The QuaTEx expression $Q_2$**

```
U(φ1,φ2) = if { φ2 } then 1
                     else if { φ1 } then #U(φ1,φ2)
                                    else 0 fi fi;
```

**Listing 4: The Until temporal operator**

hand-side of a *Defn* (i.e. after `=`), then they also have to be in its left-hand-side, so that a value can be assigned to them.

To ease presentation, only "simple" expressions like $Q_1$ or $Q_2$ are considered in this paper. However, it is worth to remark that VeStA allows to express also more interesting properties, not easily definable without resorting to a property specification language. Some interesting examples are discussed in [1]. It is possible e.g. to encode well-known temporal operators [9], like the *until* one $\phi_1 \mathcal{U} \phi_2$ defined in Listing 4, with $\phi_i$ being boolean *SExp*. In a simulation, such operator evaluates to 1 if $\phi_2$ evaluates to *true* in a state `s`, and $\phi_1$ evaluates to *true* in all states before `s`. Considering for example a scenario where a message has to be sent, and assuming that `s.rval(20)` evaluates to true if a second copy of the message is sent in `s`, while `s.rval(21)` evaluates to true if the message has been received. Then, with `eval E[`$\mathcal{U}$`(`$\neg$`s.rval(20),s.rval(21))]`, the probability that the message is received without resending it is queried. Other examples shown in [1] regard expressions counting the occurrences of determined events, or more involved ones like the "*probability that if a message is sent in a given state, then it is received within* 100 *time units.*".

In principle, it would be possible to distribute the simulations performed by VeStA, obtaining better performance. This feature is offered by the recently proposed PVeStA [2]. The two tools have been exploited to analyze scenarios ranging from self-assembling robotic scenarios to service stability protocols in cloud systems (e.g. [3, 14, 17]).

## 3. MultiQuaTEx

QuaTEx allows to query only a measure at a time (e.g. `rval(6)` in $Q_1$, or `rval(5)` in $Q_2$), while one may be interested in more. For example, it may be interesting to study both the expected number of tasks executed in a simulation ($Q_1$), and the average time spent by them in queues ($Q_2$). Even if in principle it would be possible to evaluate the two properties by performing different observations on the same simulations, it is necessary to define two expressions (i.e. $Q_1$ and $Q_2$), and to separately analyze them via distinct simulation sets. To overcome this limitation we propose MultiQuaTEx, which extends QuaTEx allowing to query more measures at a time via multiple observations on the same simulations. This improves both the usability of the language, and the performance when evaluating several expressions.

### 3.1 Defining parametric multi-expressions

The proposed extension is minimal: as depicted in Listing 5, the only difference wrt QuaTEx is that a *multi-expression* "*MQ*" (i.e. a MultiQuaTEx expression) is composed by a set of temporal operator definitions followed by a list of `eval` clauses "*EL*", rather than just one. Intuitively, a multi-expression with $n$ `eval` clauses corresponds to $n$ QuaTEx expressions sharing the same temporal operators and having

```
    MQ ::= DS EL
    EL  ::= list of eval E[PExp];
     DS, Defn, SExp and PExp as in Listing 2
```

**Listing 5: MultiQuaTEx syntax**

one of the `eval` clauses. However, the multi-expression is more compact and is evaluated performing less simulations: just the maximal number of simulations required individually by the corresponding QuaTEx expressions.

Listing 6 provides a simple multi-expression $MQ_1$, having two temporal operators (`tOp()` and `tOpB(`$x$`)`), and two `eval` clauses. It is not difficult to see that $MQ_1$ corresponds to the two previously presented $Q_1$ and $Q_2$. Noteworthy, $MQ_1$ evaluates the expected value of `rval(6)` in final states, and that of `rval(5)` in states obtained after 30 units of simulated time. Thus, the observations `rval(6)` and `rval(5)` are done in different steps of the simulations, however this does not create inconsistencies, because by evaluating `tOp()` and `tOpB(`$x$`)` they either return a real-typed value and terminate, or require to perform one step of simulation.

Listing 7 provides another interesting multi-expression ($MQ_2$). Recalling that `s.rval(2)` returns the number of steps of simulation performed to reach state `s`, `tOpC(`$x$`)` causes the execution of new steps of simulation until reaching the value specified by $x$, and then evaluates `rval(5)`. Thus, the `eval` clauses (lines 3-5) allow to obtain the expected values of `rval(5)` at the varying of the number of steps (i.e. at steps 5, 15, 25, 35, 45 and 55).

We believe that properties like $MQ_2$ are quite useful, and we have thus introduced some syntactic sugar to facilitate their writing. In particular, we introduced the concept of *parametric multi-expression* (or *parametric expression* in short), i.e. a macro that allows to concisely write multi-expressions evaluated at the varying of a parameter. Listing 8 depicts a parametric expression corresponding to $MQ_2$. In line 3 the new keyword "`parametric`" is used: provided a path expression (`tOpC(`$x$`)`), a variable ($x$) and a range of values specified as min (5.0), increment (10.0) and max (55.0), the keyword is unrolled in the corresponding list of `eval` clauses (in this case those of Listing 7). Following the spirit of MultiQuaTEx, one may be interested in analyzing more path expressions at the varying of a parameter. For this reason, the first argument of `parametric` is actually a list of path expressions (each enclosed in `E[·]` to ease parsing). Thus, it is possible to write expressions like the one sketched in Listing 9, where `parametric` is unrolled by instantiating each path expression: i.e. `eval E[tOp1(m)] ; eval E[tOp2(m)] ; eval E[tOp3(m)] ; ... eval E[tOp1(M)] ; eval E[tOp2(M)] ; eval E[tOp3(M)]`.

```
  tOp() = if { s.rval(0) == 1.0 } then s.rval(6)
                                   else #tOp()      fi;
tOpB(x) = if { s.rval(1) >= x   } then s.rval(5)
                                   else #tOpB({x})  fi;
eval E[ tOp() ] ; eval E[ tOpB(30.0) ] ;
```

**Listing 6: The MultiQuaTEx expression $MQ_1$**

```
tOpC(x) = if { s.rval(2) <= x } then s.rval(5)
                                 else #tOpC({x})  fi;
eval E[ tOpC( 5.0) ] ; eval E[ tOpC(15.0) ] ;
eval E[ tOpC(25.0) ] ; eval E[ tOpC(35.0) ] ;
eval E[ tOpC(45.0) ] ; eval E[ tOpC(55.0) ] ;
```

**Listing 7: The MultiQuaTEx expression $MQ_2$**

```
tOpC(x) = if { s.rval(2) <= x } then s.rval(5)
                                 else #tOpC({x})  fi;
eval parametric(E[ tOpC(x) ],x,5.0,10.0,55.0) ;
```

**Listing 8: $MQ_2$ as a parametric expression**

Noteworthy, the `eval` clauses of a multi-expression may regard values of different orders of magnitude. For this reason, as discussed in Section 4.2, the user can specify a list of $\delta$s other than just one. If the list is provided, then, respectively, one $\delta$ per `eval` clause for multi-expressions, and one per path expression appearing in `parametric` for parametric multi-expressions are required. If just one $\delta$ is provided, than it is considered for all the `eval` clauses.

MultiQuaTEx provides a further extension to QuaTEx: the *`last`* operator. This extension comes from two observations: the first one is that, typically, discrete event simulators have built-in mechanisms to specify the maximal "length" of a simulation, i.e. by setting a maximal number of steps or a maximal simulated time. The second observation is related to the frequent interest in properties regarding final states of the simulations only. In these cases it should be possible to let MultiVeStA to ask to compute whole simulations rather than just single steps. In this way the overhead introduced by these extra computations would be avoided (although in our experiments we noticed quite small overheads). For this reason, MultiVeStA allows the user to specify how to interpret the `#` operator by means of a newly introduced flag, whose value can be either *ONESTEP* or *WHOLESIMULATION*. In the case the user specifies *WHOLESIMULATION*, then MultiVeStA will ask the simulator to compute a whole simulation rather than just a step. In our experience, this efficiency trick should be used with great care, as it contrasts with the spirit of multi-expressions, and furthermore may lead to unintended interpretations of MultiQuaTEx expressions.

## 3.2 Evaluating parametric multi-expressions

MultiQuaTEx's extensions have to be reflected in the procedure to evaluate (multi-)expressions. Listing 10 shows the pseudo-code of a schematic version of the new procedure, where, to ease presentation, the distribution of simulations, the `last` operator, and lists of $\delta$s are omitted.

The procedure `evalOnceME` (lines `16-33`) evaluates the results (one per `eval` clause) of a multi-expression in a simulation.

```
tOp1(x) = ... ; tOp2(x) = ... ; tOp3(x) = ... ;
eval parametric(E[tOp1(x)],E[tOp2(x)],E[tOp3(x)],x,m,
     increment,M) ;
```

**Listing 9: A sample parametric multi-expression**

```
 1  double[] evalME(mq,α,δ)
 2    //initialization phase
 3    evals := array of eval clauses of (unrolled) mq;
 4    nEvals := size of evals;
 5    CIsReached := array of nEvals false;
 6    results := array of nEvals empty arrays of double;
 7    //iterative invocation of evalOnceME
 8    while(at least a false in CIsReached)
 9      initializeSimulatorWithNewRandomSeed();
10      s := obtain initial state of simulation;
11      resIteration := evalOnceME(s,DS(mq),evals,nEvals,
            CIsReached);
12      results := updateResults(results,resIteration);
13      CIsReached := checkCIs(results,CIsReached);
14    return computeMeansOfEachEval(results);
15
16  double[] evalOnceME(s,tOps,evals,nEvals,CIsReached)
17    moreStepsRequired := array of nEvals true;
18    resIteration := array of nEvals 0.0;
19    //perform the simulation step-by-step
20    while(at least a true in moreStepsRequired)
21      foreach(eval in evals)
22        //evals to be considered in this simulation
23        if(CIsReached[eval] == false)
24           //evaluation not completed yet
25           if(moreStepsRequired[eval] == true)
26            newEval := compute(s,eval,tOps);
27            //evaluation completed
28            if(newEval is a real number)
29              moreStepsRequired[eval] == false;
30              resIteration[eval] = newEval;
31            else
32              eval := newEval;
33    return resIteration;
```

**Listing 10: Evaluation of a multi-expression**

These values are exploited by `evalME` (lines `1-14`), the main procedure, which iteratively invokes `evalOnceME` to perform new simulations. At every iteration, `evalME` updates the means of the obtained results (one mean per `eval` clause), and terminates returning them to the user if the CI has been reached for every `eval` clause, or performs another iteration. Note that each `eval` clause may require a different number of simulations to reach the CI. Once the CI of an `eval` clause has been reached, it is ignored by `evalOnceME`.

The main procedure takes as parameters the multi-expression `mq`, and the specification of the required CI (line `1`), and is composed by an initialization phase (lines `3-6`) followed by the iterative invocation of `evalOnceME` (lines `8-13`). In the initialization phase, the `eval` clauses of (the unrolled version of) `mq` are stored in the array `evals`, and then the arrays `CIsReached` and `results`, having the same size of `evals`, are created. The former is an array of booleans initialized to false (line `5`) used to keep track of the `eval` clauses for which the required CI has been reached, and it is thus also used in the halting condition of the iterative phase (i.e. when it does not contain any *false*). `results` (line `6`) is instead an array of arrays of double: each position of `results` regards one of the `eval` clauses, and in particular stores all the results obtained in the performed simulations for that `eval` clause.

In the iterative phase, the simulator is initialized with a new random seed to perform a new simulation (lines `9-10`). Then, by invoking `evalOnceME`, the simulation is triggered, obtaining a real-typed evaluation for each `eval` clause. Those results are stored in the array of double `resIteration` (line `11`). In line `12`, `results` is updated with the obtained values, and finally in line `13` it is checked if any of the CIs has been satisfied (updating accordingly `checkCIs`). The iterations terminate once the CIs of every `eval` clause are satisfied (line `8`), and an array containing the means of the results of each `eval` clause (i.e. their expected value) is returned.

The procedure `evalOnceME` takes as parameters the initial state of the simulation (`s`), the temporal operator definitions of `mq`, the `eval` clauses (and their number) of `mq`, and `CIs-Reached`. The latter is used to ignore the `eval` clauses for which the required CI has been already reached. Line `17` defines an array of booleans with size `nEvals` initialized to true: as discussed, each `eval` clause may require a different number of steps of simulations to be evaluated, due to recursive temporal operators. This array is used to keep trace of those clauses whose evaluation has not terminated yet. Line `18` defines the array `resIteration` which will store the evaluations of the `evals`. Then, in lines `20-32` the step-by-step simulation is iteratively performed until all the `evals` have been evaluated (see line `20`), when the results are returned (line `33`). At each step of the simulation, only the `eval` clauses that have not reached the desired CI in a previous simulation (line `23`), and whose evaluation has not been completed yet in this simulation (line `25`) are considered. In particular, in line `26` each of the selected clauses is evaluated in the current state: if by evaluating one of them a real number is obtained, then its evaluation has been completed for this simulation (lines `28-30`), otherwise the `eval` clause is updated with the newly obtained path expression (line `32`), which will be in turn evaluated after another step of simulation.

## 4. MULTIVESTA

MultiVeStA comes as an extension of VeStA (and PVeStA), from which it inherits the license of Department of Computer Science at the University of Illinois at Urbana-Champaign, ©2013 The Board of Trustees of the University of Illinois. Section 3 discussed how MultiQuaTEx extends QuaTEx by providing better usability and performance. MultiVeStA further extends the two tools by allowing to integrate existing discrete event simulators in addition to the originally supported ones, and by improving the presentation of results.

### 4.1 Integrating a simulator and its models

MultiVeStA can coordinate and analyze the simulations performed by discrete event simulators. Essentially, the requirements are: (1) the support for the step-by-step simulations, and (2) the ability to inspect the states of the simulations to obtain real-typed observations. The downloadable version of MultiVeStA [23] supports some simulators, namely Alchemist [24], DEUS [4], MISSCEL [11], and the two originally supported ones [32] (i.e. PMaude and a CTMC engine).

The integration of Java-based simulators is straightforward, as it only requires to extend one Java class. The procedure for non Java-based simulators is the same, but in addition it is necessary to apply wrapping-like approaches to overcome the barriers introduced by the use of different programming languages. Noteworthy, the non Java-based PMaude has been integrated exploiting ExpectJ [18], a Java library allowing to interact with external processes. Moreover, we are currently integrating the swarm robotic simulator ARGoS [6] (coming as a `C++` library), exploiting the Java Native Interface [21], which enriches Java with support for native code (i.e. `C/C++`).

Few steps are required to allow a new user to exploit MultiVeStA. Basically, it is necessary to extend the tool to support the new simulator engine, and then, for each model, it is needed to specify the relevant states observations. The steps that must be performed are sketched in the flow-chart of Figure 1. The dark ones are simulation-specific steps, and thus have to be tackled only once for each newly supported simulator. The bright ones instead are model-specific steps, i.e. they have to be tackled for every newly defined model. The blocks "Extend NewState" and "Define the model-specific state observations" are represented with thicker borders to stress the fact that their implementations are influenced by the programming language used to develop the simulator.

#### *4.1.1 Extending support for a new simulator*

In order to support a new simulator it is necessary to follow the dark blocks of Figure 1. First of all, MultiVeStA.jar must be downloaded from MultiVeStA's website [23], then a Java project referencing it must be created. As next step, the class *NewState*, the only point of interaction among MultiVeStA and the simulators, must be extended. Section 5 exemplifies the class *DeusState* integrating the Java-based simulator DEUS [4]. Essentially, three methods invoked by MultiVeStA have to be overridden (plus an optional one):

- `setSimulatorForNewSimulation(int randomSeed)`, invoked to (re)set the underlying simulator before performing a simulation run. The parameter is the seed that has to be used to initialize pseudo-random number generator of the simulator;
- `performOneStepOfSimulation()`, invoked to order the simulator to perform one step of simulation;
- `rval(int observation)`, invoked to obtain observations on the current state of the simulation. This method can be refined depending on the model at hand (see Section 4.1.2), however observations common to any model can be defined here once (e.g. the current simulation time);
- `performWholeSimualtion()`, an optional method invoked instead of `performOneStepOfSimulation()` if the user specifies the option *WHOLESIMULATION*, as discussed in Section 3.1. This method is invoked to order the simulator to perform a whole simulation.

Clearly, in order to implement these methods, the created project needs to have visibility of the simulator. As a general guideline, if the simulator is Java-based then the project should refer to a jar containing the simulator kernel classes, while the class extending *NewState* should have an instance of the simulator engine as a private field. Such a case is exemplified in Section 5. If the simulator comes as a blackbox external process, then it is possible to exploit the ExpectJ library, which allows to easily interact with external processes. As mentioned above, PMaude has been integrated following this approach: the *PMaudeState* class has as private field the class provided by ExpectJ to abstract external processes, while appropriate methods of the library are used to interact with the stdin and stdout of the PMaude process. A third option is that the simulator comes as a `C/C++` library. In this case it is possible to exploit the Java Native Interface to create a Java class wrapping the main class of the simulator, thus reducing the issue to the case of a Java-based simulator.

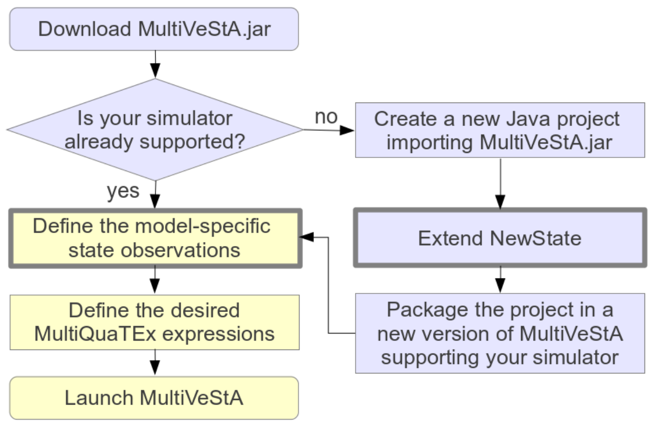
Download MultiVeStA.jar
Is your simulator already supported?
no
yes
Create a new Java project importing MultiVeStA.jar
Define the model-specific state observations
Extend NewState
Define the desired MultiQuaTEx expressions
Package the project in a new version of MultiVeStA supporting your simulator
Launch MultiVeStA

**Figure 1: Steps necessary to exploit MultiVeStA.**

Finally, the project can be packaged in a new version of MultiVeStA supporting the new simulator.

#### *4.1.2 Defining model-specific state observations*

Once a simulator is supported, before performing the analysis tasks it is necessary to refine the `rval` method defining the state observations of interest for the model at hand.

As a general guideline, if the simulator is Java-based, then the *IStateEvaluator* interface should be implemented. This simple interface consists of just the method `double getVal(int observation, NewState state)`, whose implementations compute model-specific observations accessing to the simulator state. When evaluating a MultiQuaTEx property, it is possible to provide the name of the class of a state evaluator, which is dynamically loaded and offered by the `getStateEvaluator()` method of *NewState*. Listing 11 schematizes a typical implementation of `rval` for a Java-based simulator.

If the simulator is not Java-based, then the interface *IStateEvaluator* may still be similarly instantiated. However it may happen that the internal state of the simulator cannot be accessed from Java. In these cases the computation of the observations should be dealt by the simulator (e.g. specified in the model definition), and thus `rval` should just forward requests to the simulator without requiring any state evaluator. This is the case of PMaude.

```
public double rval(int observation) {
  switch (observation) {
    case 0: return getTime();
    case 1: return getNumberOfSteps();
    // other common observations for this simulator
    case ...
    //model-specific observations
    default: return getStateEvaluator().getVal(
        observation, this);
  }
}
```

**Listing 11: Sample `rval` for Java-based simulators**

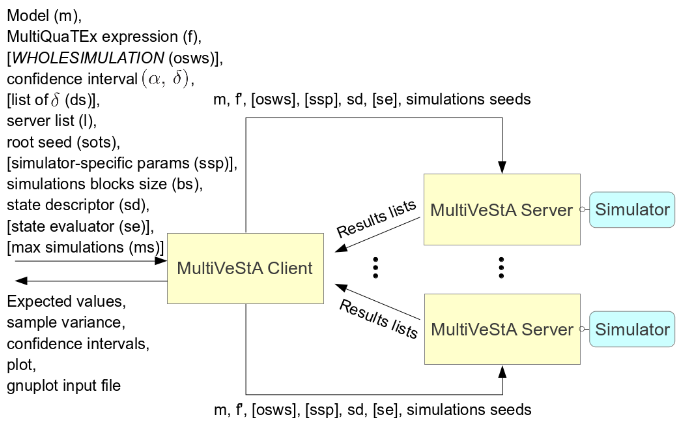

**Figure 2: MultiVeStA architecture.**

## 4.2 Architecture

Figure 2, adapted from [2], sketches the client-server architecture that MultiVeStA inherited from PVeStA. A concrete instance of MultiVeStA consists of a client interacting with the user and with a set of servers, each in turn interacting with an instance of the same simulator. The client coordinates the servers by sending them the parameters, collecting the results, computing statistical measures, and terminating the evaluation when the required accuracy is met. Different outputs are provided to the user by the client, depending on the MultiQuaTEx expression: the required expected values for multi-expressions, or plots visualized in a minimal GUI (realized using the JMathPlot library [20]) and gnuplot input files for parametric multi-expressions.

An user provides several parameters to the client: the file name of the model definition ($m$), the MultiQuaTEx expression to be evaluated ($f$), the optional flag *WHOLESIMULATION* ($osws$), the $\alpha$ and $\delta$ specifying the CI, the optional list of $\delta$s ($ds$), the IP network addresses of the servers ($l$), and the root seed ($sots$). The latter is used to initialize a pseudo-random number generator, which in turn creates the seeds for the simulations performed by the servers. A simulator may require additional parameters other than the model specification: if needed, those can be provided using the *ssp* option. Furthermore, the user provides the name of the Java class extending `NewState` ($sd$), and an optional state evaluator ($se$) to compute the model-specific state observations. As discussed, MultiQuaTEx expressions are iteratively estimated by performing blocks of simulations until the desired CI is reached. The number of simulations performed at each iteration can be specified via "simulations blocks size" ($bs$). Finally, it is possible to specify a stopping criteria through the maximum number of simulation runs to be performed ($ms$). This is indeed a further extension introduced in MultiVeStA. In this case, due to the interruption of the evaluation, properties may not have been computed with the required CI. Thus, other than the result of the query, also the sizes of the computed confidence intervals are returned to the user.

The client and servers interact as follows: each server receives $m$, the MultiQuaTEx property *f'* obtained by preprocessing *f*, *sd*, and the optional parameters *osws*, *ssp* and *se*. Furthermore, servers receive the number of simulations to be performed in an iteration ($\lceil bs/\text{number of servers}\rceil$), and the list of simulations' seeds. Then, for each simulation, a server interacts with the simulator to initialize it and evaluate *f'*.

```
1  private Engine deusInstance;
2  private AutomatorParser deusAP;
3  
4  public DeusState(ParametersForState params){
5    super(params);
6    deusAP = new AutomatorParser(getModelName());
7  }
8  
9  public void setSimulatorForNewSimulation(int seed) {
10   deusAP.getEngine().setSeedAndResetSimulator(seed);
11   deusInstance = deusAP.getEngine();
12 }
13 
14 public void performOneStepOfSimulation(){
15     deusInstance.runStep();
16 }
17 
18 public void performWholeSimulation(){
19     deusInstance.run();
20 }
21 
22 public double rval(int which) {
23   switch (which) {
24   case 0: if(simulationCompleted()) return 1.0;
25           else                      return 0.0;
26   case 1: return getTime();
27   case 2: return getNumberOfSteps();
28   case 3: return deusInstance.getNodes().size();
29   case 4: return deusInstance.getEvents().size();
30   default: return getStateEvaluator().getVal(
         observation, this);
31   }
32 }
33 
34 private boolean simulationCompleted(){
35   return getTime() >= deusInstance.getMaxTime() ||
36          deusInstance.getEvents().size() == 0;
37 }
38 
39 public double getTime() {
40   return deusInstance.getVirtualTime();
41 }
42 
43 public Engine getDeusInstance() {
44   return deusInstance;
45 }
```

**Listing 12: DeusState class extending NewState**

## 5. VALIDATION

In order to demonstrate the feasibility and benefits of the integration of existing discrete event simulators with MultiVeStA, this section discusses the integration of DEUS [4], and shows some examples of the analysis done for the already mentioned volunteer cloud computing scenario. DEUS [4] is a general-purpose, open-source, Java-based simulation environment for the analysis of complex and large scale systems, characterized by extreme ease of use and flexibility.

### 5.1 Integrating MultiVeStA and DEUS

The steps of Figure 1 have to be followed in order to integrate DEUS. Listing 12 reports the class *DeusState* (omitting unnecessary details), which extends *NewState*. As discussed in Section 4.1.1, `DeusState` has an instance of the simulator engine as private field (lines `1-2`). In the constructor (lines `4-7`), all the parameters are contained in `params`. Those common to any simulator (e.g. the file name of the model definition) are handled by the constructor of `NewState` (line

5). Eventual simulator-specific parameters can be parsed from the string `params.getOtherParameters()`. In line 6 the DEUS-specific initialization code is performed: i.e. the model definition is loaded (once for each MultiVeStA server). The actual (re)initilization of the simulator is done in `setSimulatorForNewSimulation` (lines 9-12), invoked before any new simulation to set a new seed, to obtain the initial state of the simulation, and to reset the internal state of the simulator. The methods `performOneStepOfSimulation` and `performWholeSimulation` order DEUS, respectively, to perform a step of simulation or to independently perform a simulation until an internal halting condition is met. Finally, `rval` evaluates state observations common to every DEUS model (e.g. the number of performed steps), and supports eventual model-specific state evaluators (line 30).

It may be worth to remark that such a step was done once and for all: since then, DEUS is supported by MultiVeStA.

## 5.2 Test: volunteer cloud simulation

It is now possible to define and evaluate properties of interest, like the ratio of successfully executed tasks (*Hit+Running ratio*) during the simulation progress, for small, large and all tasks. The corresponding parametric multi-expression is reported in Listing 13. Here, new steps of simulations are triggered upon reaching the simulation times specified by the variable $x$. Once such states are reached, the observations number 11, 12 and 13, corresponding to the ratios of tasks executed or still running for different types of tasks, are evaluated. Note that the expression is estimated for $x$ ranging from 100 to 360000 (i.e. 1 hour of simulated time), with step of 10000. In order to evaluate such properties, `s.rval(1)` must be associated to the simulated time, while `s.rval(11)`, `s.rval(12)`, and `s.rval(13)`, respectively, to the ratio of executed tasks for small, large and both. The first is not model-specific, and in fact it is handled in `rval` of Listing 12. While for the others a state evaluator must be defined, as sketched in Listing 14.

The expression of Listing 13 has been evaluated on a laptop having a 2.0 Ghz Quad Core and 16 GB of RAM. The client was launched with the parameters reported in Listing 15. Thus, the expected values have been computed wrt a 95% CI with a radius of 0.002, while 3 servers were used. Moreover, the batch block size was fixed to 6, and 100 was setted as maximum number of simulations. Finally, the state evaluator of Listing 14 has been specified. For easiness of presentation, Listing 14 just sketches the evaluator. In line 4, the simulator engine is extracted from the `DeusState`, and it is used in line 5 to build the object `simulationInfo`, which contains information about received and missed tasks. Such an object is then used to perform the model-specific state observations.

```
HitSmall(x) = if {s.rval(1) >= x} then {s.rval(11)}
  else #HitSmall({x}) fi ;
HitLarge(x) = if {s.rval(1) >= x} then {s.rval(12)}
  else #HitLarge({x}) fi ;
HitOverall(x) = if {s.rval(1) >= x} then {s.rval(13)}
  else #HitOverall({x}) fi ;
eval parametric(E[HitSmall(x)], E[HitLarge(x)], E[
    HitOverall(x)],x,100.0,10000.0,360000.0);
```

**Listing 13: Ratios of executed tasks on time**

```
1  public class VolunteerCloudStateEvaluator implements
       IStateEvaluator {
2
3    public double getVal(int which, NewState state) {
4      Engine engine = ((DeusState)state).
           getDeusInstance();
5      //Access engine to build the object "
           simulationInfo" which contains information
           about received and missed tasks
6
7      switch(which){
8        case 11:  return 1.0 - (simulationInfo.
             getSmallTasks().getMiss() / simulationInfo.
             getSmallTasks().getSize());
9        case 12: ...
10       case 13: ...
11 }}}
```

**Listing 14: IDeusStateEvaluator for the cloud**

```
-m examples/acoLoadBalancing_s1.xml
-l examples/serverlist3
-f examples/quatex/acoPerformanceIndicators_mp.quatex
-bs 6 -a 0.05 -d1 0.004
-se it.imtlucca.cloud.multivesta.
    VolunteerCloudStateEvaluator
-osws ONESTEP -sots 12345 -sd deus.DeusState -ms 100
```

**Listing 15: Parameters of the MultiVeStA client**

About 4 hours have been required to evaluate the expression. Noteworthy, the setted maximum number of simulations did not constitute a restriction, since the required $\delta$ has been reached after just 18 simulations, distributed in the 3 servers in groups of 6 simulations. The plot in Figure 3, depicting the evaluation of the expression, has been generated using the gnuplot input file returned by MultiVeStA.

The compactness and expressiveness of MultiQuaTEx has facilitated the evaluation of the ACO approach in distributing tasks in a volunteer cloud. Without detailing the obtained results, it is important to note that, by evaluating a single multi-expression, it has been possible to study three properties (ratios of executed small, large and all tasks) at the varying of the simulated time: each point of the three lines is actually an expected value of the corresponding property, when instantiating the parameter $x$.

## 5.3 Experience using DEUS with MultiVeStA

By integrating DEUS with MultiVeStA, the former has been enriched with capabilities far from those usually offered by simulation tools. Mainly, it is now possible to define in a clean and compact way the properties of interest, to decouple them from the model definition, and to automatize their evaluation. Furthermore, distribution of simulations is now supported. Another important benefit is the ability of evaluating more properties at once via parametric multi-expressions, thus reducing the number of required simulations.

In the following, we discuss how the analysis described in Section 5.2 would be much more complex by using either DEUS alone or with PVeStA. As discussed in Section 3.1, a MultiQuaTEx expression corresponds to a set of QuaTEx expressions sharing the same temporal operators, and each having one of the `eval` clauses of the MultiQuaTEx expression.

The MultiQuaTEx expression of Listing 13 has three `eval` clauses (`HitSmall(`$x$`)`, `HitLarge(`$x$`)`, and `HitOverall(`$x$`)`), each having parameter $x$, instantiated with the 36 values from 100 to 360000, with step 10000. The MultiQuaTEx expression thus corresponds to $36 * 3 = 108$ QuaTEx expressions. MultiVeStA required 18 simulations to evaluate the MultiQuaTEx expression, meaning that 18 is the maximal number of simulations required to reach the provided CI for each of the 108 `eval` clauses. Hence, VeStA and PVeStA would require in the order of $108*18 = 1944$ simulations rather than 18 to evaluate the 108 corresponding QuaTEx expressions. MultiVeStA thus provide a dramatic performance improvement. In [25], presenting the integration of MultiVeStA with Alchemist, we show how MultiVeStA's ability to reuse simulations allows to perform a 68 times faster analysis of an expression regarding 5 parametric properties instantiated with 50 values each, wrt the case of not reusing simulations.

An analysis similar to the one done with MultiVeStA, including the reuse of simulations, can be performed using DEUS alone, by logging the required measures during simulations. However, no CI evaluation facilities are provided, thus it would be necessary to do it by hand by performing the means of the obtained values, and launching new simulations if necessary. Moreover, each of the 18 simulations performed by MultiVeStA required about 40 minutes on our test machine, but given that 3 servers were used, 3 groups of 6 simulations were performed in parallel, requiring about 4 hours to evaluate the expression. Performing with DEUS simulations like the ones performed resorting to MultiVeStA still requires about 40 minutes, in fact we noticed an overhead in the order of few minutes. However, DEUS does not provide distribution of simulations, and thus the required time would be of the order of $18 * 40 = 720$ minutes (or 12 hours).

To sum up, the advantage in chaining PVeStA to DEUS (if this would be possible), would be the possibility to express complicated queries, and to automatize and distribute their evaluation. However, the implicit ability of reusing simulations by logging would be lost, leading to a sensible degradation of performance when evaluating several expressions. MultiVeStA shares the same benefits of PVeStA, but not its drawbacks, as in addition it allows to reuse simulations via MultiQuaTEx expressions, thus leading to sensible improvements in both usability and performance of DEUS.

## 6. RELATED WORKS

In the literature there exist many tools supporting statistical analysis. In particular, a partial list of those supporting SMC is: APMC [5], COSMOS [15], YMER [36], and SAM [28]. Moreover, statistical extensions of established tools like BIP [12], PRISM [26] and UPPAL [34] exist. However, differently from MultiVeStA, most of these tools have their own engines and model specification languages, as they do not aim at exploiting and enriching existing discrete event simulators.

An interesting tool sharing our aims and motivations is PLASMA-lab [13], which can be integrated to existing simulators via a plug in system. The tool allows to answer to two kinds of questions: (1) "*does a property is satisfied by a model with a probability greater than a given threshold?*", and (2) "*what is the probability that a model satisfies a property?*". In addition to probabilities, (Multi)VeStA further

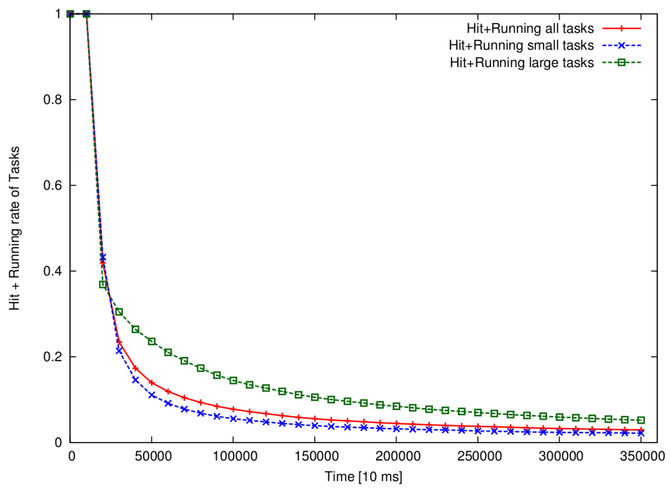

**Figure 3: Evaluation of three properties at the varying of the simulated time.**

allows to obtain expected values of real-typed properties, like the latency of a probabilistic communication protocol, the mean time spent by tasks in queue, or the number of entities reaching a certain goal. This is a very important feature, as users of discrete event simulators are often interested to such a kind of measures when evaluating the performance of a system. Moreover, it is not mentioned if PLASMA-lab allows to reuse simulations when evaluating several properties, another important feature implicitly adopted by users of discrete event simulators when logging measures of a simulation. Nevertheless, the tool seems quite interesting and promising to us, as it is equipped with algorithms that use importance sampling to reduce the number and the length of simulations. It would be interesting to understand how these algorithms could be integrated with the approach used by us in reducing the number of required simulations by evaluating more properties at once.

## 7. CONCLUSIONS AND FUTURE WORK

This paper presented MultiVeStA, a statistical analysis tool which allows for an easy integration with existing discrete event simulators, enriching them with distributed statistical analysis capabilities. Moreover, MultiQuaTEx enables users to express in a compact way system properties of interest, and to efficiently evaluate them. The tool has been used to reason on scenarios regarding collision-avoidance robots [11], volunteer clouds [30] and crowd steering [25], in the context of the European projects ASCENS [7], and SAPERE [29].

MultiVeStA currently supports the simulators Alchemist [24], DEUS [4], MISSCEL [11], and the two originally supported ones [32] (i.e. PMaude and a CTMC engine). As future work, we plan to extend the set of supported simulators, e.g. it may be interesting to consider ns-3 and OMNeT++. Moreover, we plan to improve our GUI (possibly simplifying the deployment and the management of the MultiVeStA servers in the network), which currently consists in an interactive plot of the results. Finally, it may be useful to provide the user with the plot of the distribution of all the obtained samples, and to analyze it by adding fitting function capabilities.

## 8. ACKNOWLEDGMENT

We are grateful to the proposers of (P)VeStA and QuaTEx, our starting point: Gul Agha, Musab AlTurki, José Meseguer, Koushik Sen and Mahesh Viswanathan. Furthermore, we thank the author of Alchemist (Danilo Pianini), which provided many suggestions to improve MultiVeStA. Finally, we thank Michele Amoretti and Alberto Lluch Lafuente which supported us in the preparation of the tool and of this paper.

## 9. REFERENCES

[1] G. A. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In A. Cerone and H. Wiklicky, editors, *QAPL 2005*, volume 153(2) of *ENTCS*, pages 213–239. Elsevier, 2006.

[2] M. AlTurki and J. Meseguer. Pvesta: A parallel statistical model checking and quantitative analysis tool. In A. Corradini, B. Klin, and C. Cîrstea, editors, *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 386–392. Springer, 2011.

[3] M. AlTurki, J. Meseguer, and C. A. Gunter. Probabilistic modeling and analysis of DoS protection for the ASV protocol. In D. J. Dougherty and S. Escobar, editors, *SecReT 2008*, volume 234 of *ENTCS*, pages 3–18. Elsevier, 2009.

[4] M. Amoretti, M. Agosti, and F. Zanichelli. Deus: a discrete event universal simulator. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 58:1–58:9, ICST, Brussels, Belgium, Belgium, 2009.

[5] APMC: `sylvain.berbiqui.org/apmc`.

[6] ARGoS: `iridia.ulb.ac.be/argos`.

[7] ASCENS Project: `www.ascens-ist.eu`.

[8] A. Aziz, V. Singhal, and F. Balarin. It usually works: The temporal logic of stochastic systems. In P. Wolper, editor, *CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 155–165. Springer, 1995.

[9] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[10] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time markov chains. In J. C. M. Baeten and S. Mauw, editors, *CONCUR*, volume 1664 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 1999.

[11] L. Belzner, R. De Nicola, A. Vandin, and M. Wirsing. Reasoning (on) Service Component Ensembles in Rewriting Logic. To appear in the proceedings of SAS 2014, Springer LNCS Festschrift.

[12] BIP: `www-verimag.imag.fr/BIP-Tools,93.html`.

[13] B. Boyer, K. Corre, and S. Sedwards. PLASMA-lab: a flexible distributable statistical model checking library. *To appear in the proceedings of QEST 2013*, 2013.

[14] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. In F. Durán, editor, *WRLA 2012*, volume 7571 of *LNCS*, pages 118–138. Springer, 2012.

[15] COSMOS: `www.lsv.ens-cachan.fr/~barbot/cosmos`.

[16] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Trans. Evolutionary Computation*, 1(1):53–66, 1997.

[17] J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, and M. Wirsing. Stable availability under denial of service attacks through formal patterns. In *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 78–93, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] ExpectJ: `expectj.sourceforge.net`.

[19] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.

[20] JMathPlot: `code.google.com/p/jmathplot`.

[21] JNI: `docs.oracle.com/javase/7/docs/technotes/guides/jni`.

[22] A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. In H. Schulzrinne, K. Aberer, and A. Datta, editors, *Peer-to-Peer Computing*, pages 99–100. IEEE, 2009.

[23] MultiVeStA: `code.google.com/p/multivesta/`.

[24] D. Pianini, S. Montagna, and M. Viroli. Chemical-oriented simulation of computational systems with Alchemist. *Journal of Simulation*, 2013.

[25] D. Pianini, S. Sebastio, and A. Vandin. Statistical analysis of chemical computational systems with MultiVeStA and Alchemist. Submitted `eprints.imtlucca.it/1697`.

[26] PRISM: `www.prismmodelchecker.org`.

[27] G. Riley and T. Henderson. The ns-3 network simulator. In K. Wehrle, M. GÃÂijneÅŸ, and J. Gross, editors, *Modeling and Tools for Network Simulation*, pages 15–34. Springer Berlin Heidelberg, 2010.

[28] SAM: `rap.dsi.unifi.it/SAM`.

[29] SAPERE Project: `www.sapere-project.eu`.

[30] S. Sebastio, M. Amoretti, and A. Lluch-Lafuente. A computational field framework for collaborative task execution in volunteer clouds. Manuscript `eprints.imtlucca.it/1651`.

[31] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2005.

[32] K. Sen, M. Viswanathan, and G. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 251–252, 2005.

[33] K. S. Trivedi. *Probability and statistics with reliability, queuing and computer science applications*. John Wiley, New York, 2nd edition edition, 2002.

[34] UPPAL: `www.uppaal.org`.

[35] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008.

[36] Ymer: `www.tempastic.org/ymer`.