MultiVeStA: Statistical Model Checking for Discrete Event Simulators *

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

Andrea Vandin
ECS, University of Southampton, UK,
IMT Institute for Advanced Studies Lucca, Italy
a.vandin@soton.ac.uk

ABSTRACT

The modeling, analysis and performance evaluation of large-scale systems are difficult tasks. 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 (e.g., logics) are used to express properties, which are automatically estimated again simulating the model at hand. These property specification languages provide a formal, compact and elegant way to express properties without hard-coding them in the model definition. This paper presents *MultiVeStA*, a statistical analysis tool which can be easily integrated with 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 instead a viable solution. A possible way to model the systems dynamics is via *discrete event simulations* (DESs), where systems evolve according to discrete *events*, and the time-flow is discretized in jumps among their occurrence. The popularity of DES is witnessed by the many existing DES-based tools. We cite among others ns-3 [19], OMNeT++ [26], DEUS [3] and Alchemist [16].

Exact quantitative analysis can be done via Probabilistic Model Checking [7], where, roughly, systems are abstracted in formal models, while properties are expressed in rigorous formalisms (e.g., logics), and are evaluated by exhaustively exploring models state-spaces. Although powerful, this approach suffers from the state-space explosion problem, i.e., it does not scale well when systems complexity grows. A possible solution is the use of statistical analysis techniques like *Statistical Model Checking* (SMC) [23], where properties are statistically estimated by resorting to simulations. Conversely, the main drawback is that it does not provide exact results, but only statistical guarantees.

Contribution. This paper presents MultiVeStA[1] [15], a Java tool extending PVeStA [2] (which in turn extends VeStA [1, 24]), allowing us to enrich discrete event simulators with distributed statistical analysis capabilities. MultiVeStA offers: (1) a clean way to integrate discrete event simulators; (2) a language (MultiQuaTEx) to compactly express 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.

We validate MultiVeStA by discussing its integration with DEUS, and prove the usefulness of the integration by analyzing a cloud system. The scenario, which will be used throughout the paper to ease presentation, regards the distribution of tasks requests among volunteer autonomous entities [21] based on Ant Colony Optimization (ACO) where a *colored pheromone field* is used to create a distributed guide for the search of a node willing to execute a task. Two classes of tasks exist, 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 satisfiable requests.

Synopsis. §2 discusses VeStA and PVeStA. Then §3 and §4 present the main improvements introduced in MultiVeStA:

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

ValueTools'13, December 10 – 12 2013, Turin, Italy

[1]License of Department of Computer Science at the University of Illinois at Urbana-Champaign, ©2013 The Board of Trustees of the University of Illinois

VALUETOOLS 2013, December 10-12, Torino, Italy

DOI 10.4108/icst.valuetools.2013.254377

§3 focuses on the extended property specification language, and on the improved performance, while §4 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. BACKGROUND: (P)VeStA AND QuaTEx

VeStA [24] is a statistical model checker and quantitative analyzer for probabilistic systems. It statistically evaluates quantitative temporal expressions (QuaTEx) [1], allowing to query expected values of observations on simulations of stochastic models. The tool also supports the transient fragments of PCTL [13] and CSL [6, 8], for which SMC algorithms based on statistical hypothesis testing exist [23]. We focus here on QuaTEx, as it generalizes both [1].

VeStA supports a language to express CTMCs, and PMaude, an algebraic specification language for probabilistic rewrite theories [1]. Actually, the analysis algorithms of VeStA are independent of the model specification language: it is only assumed that DES can be performed on the models. Thus, in principle, as discussed in [24], VeStA could be plugged to any discrete event simulator offering methods to initialize a simulation, to compute a simulation step, and to access the current state. By doing this, the simulator should be enriched with a property specification language (QuaTEx), and with automated 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 infrastructure to easily integrate new simulators.

VeStA estimates the expected value of QuaTEx expressions wrt two user-defined parameters: α and δ. Parameter δ defines a stopping criteria to the sample generation, i.e., when the *Confidence Interval* (CI), computed using the *Student's t-test*, radius is less or equal to $\delta/2$. 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 an expression, QuaTEx has to be "connected" with the simulated model: the state characteristics of interest have to be specified by implementing the `rval(`i`)` predicate for each observation i. E.g., for our cloud scenario `s.rval(0)` reduces to 1.0 if the simulation is completed in the current simulation state (denoted by the keyword `s`), and 0.0 otherwise, `s.rval(1)` returns the current simulated time, `s.rval(2)` counts the steps of simulation, `s.rval(5)` is the average time spent by tasks in queues, `s.rval(6)` returns the number of 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.

The rest of this section discusses QuaTEx's syntax resorting to the expression Q_1 of Listing 1, intuitively reading as: *compute the expected number of tasks executed in a simulation.* QuaTEx's syntax is presented in Listing 2: a QuaTEx expression (line 1) consists of a set of *parameterized recursive temporal operator definitions* "*DS*" (Q_1 has just one named `tOp()`), followed by an *eval clause* "`eval E[`*PExp*`]`". A *temporal operator definition* "*Defn*" (line 3) consists of the name of the operator (e.g., `tOp()`) and of a path expression representing its body. *State expressions*, or state observations, "*SExp*" (line 4) are either a real number (`c`), a predicate (`rval`), an arithmetic or boolean expression over *SExps*, or a variable (x_j). Finally, a *path expression* "*PExp*" (lines 5-6) is a real-typed predicate whose evaluation possibly requires to perform steps of simulation. A *PExp* is either a state expression *SExp*, a temporal operator preceded by the symbol `#`, or an `if_then_else` statement. The `if_then_else` statement behaves as expected, instead `#` requires to perform a step of simulation, and then to evaluate the associated temporal operator in the obtained state. Noteworthy, if `#` is used in recursive temporal operator definitions (like in Q_1), it allows to query properties of states obtained after an arbitrary number of simulation steps.

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`: "*is* `s` *a final state of the simulation?*". If true, then `s.rval(6)` is returned, i.e., the number of executed tasks. Otherwise Q_1 is evaluated as `#tOp()`: VeStA invokes the simulator to advance of one step, updates `s`, and then recursively evaluates `tOp()`. This is done until a state satisfying the guard is reached. Several simulations are evaluated, until the mean $\overline{x}$ of the results satisfies the user-specified CI, and $\overline{x}$ is returned as value of Q_1. Noteworthy, suppose a predicate `rval(`j`)` evaluating 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 the event 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 minimum simulated time.* Q_2 shows that temporal operators can have parameters (variables). Variables have to be bounded: if they are in the right-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" properties are considered in this paper, but QuaTEx allows us to express also more interesting ones, not easily definable without using a property specification language. Some interesting examples are discussed in [1]. It is possible e.g., to encode well-known temporal operators [7], like the *until* one $\phi_1 \mathcal{U} \phi_2$, with ϕ_i be-

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

Listing 2: QuaTEx syntax

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

Listing 1: The QuaTEx expression Q_1

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

Listing 3: The QuaTEx expression Q_2

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

Listing 4: MultiQuaTEx syntax

ing a boolean *SExp*: in a simulation, such operator evaluates to 1 if ϕ_2 evaluates to *true* in a state `s`, and ϕ_1 evaluates to *true* in all states before `s`. Other examples shown in [1] regard the counting of the occurrences of certain events, or more involved ones like the *probability that if a message is sent in a given state, then it is received within x time units.*

In principle, it would be possible to distribute simulations, obtaining better performance. This feature has been added in PVeStA [2], a recent extension of VeStA.

The two tools have been used to analyze scenarios ranging from self-assembling robots to service stability protocols in cloud systems (e.g. [10–12]).

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. E.g., one may study both the expected number of executed tasks (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 distinct expressions, 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 language's usability, and the performance when evaluating several expressions.

The language extension is minimal: as depicted in Listing 4, a *multi-expression* "*MQ*" (i.e. a MultiQuaTEx expression) may contain 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, but having one of the `eval` clauses each. However, the multi-expression is more compact and is evaluated performing less simulations: just the maximal number of simulations required by each of the corresponding QuaTEx expressions.

Listing 5 provides a simple multi-expression MQ_1, having two temporal operators (`tOp()` and `tOpB(`x`)`), and two `eval` clauses. It is easy to see that MQ_1 corresponds to 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 time. However this does not create inconsistencies.

Listing 6 provides the multi-expression (MQ_2). Recalling that `s.rval(2)` returns the current number of simulation steps, `tOpC(`x`)` causes the execution of new steps 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 different steps of simulation (i.e. 5, 15, 25, 35, 45 and 55). Parametric properties like MQ_2 are 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 7 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 6). Actually, the first argument of `parametric` may be a list of path expressions (each enclosed in `E[·]` to ease parsing), allowing to analyze more path expressions at the varying of a parameter.

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

Listing 5: The MultiQuaTEx expression MQ_1

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

Listing 6: The MultiQuaTEx expression MQ_2

Noteworthy, the `eval` clauses of a multi-expression may regard values of different orders of magnitude. For this reason, the user can specify a list of δs other than just one. If the list is provided, then, respectively, one δ per `eval` clause for multi-expressions, or one per path expression appearing in `parametric` for parametric multi-expressions are required.

A further extension is provided: 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 is possible to let MultiVeStA ask to compute entire simulations rather than just single steps, thus reducing the overhead introduced by these interactions (although in our experiments we noticed quite small overheads). In particular, the user can specify how to interpret the `#` operator by means of a newly introduced flag, whose value can be either *ONESTEP* or *WHOLESIMULATION*.

The presented extensions have been reflected in the procedure to evaluate (multi-)expressions. A particular care has been taken, as each `eval` clause may require a different number of steps in order to be evaluated in a simulation, and, moreover, a different number of simulations may be required by each `eval` clause to reach its CI. Roughly, once an `eval` clause has been evaluated in a simulation, it is ignored until all her clauses have been evaluated in that simulation. Similarly, once the CI of an `eval` clause has been reached, it is

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

Listing 7: MQ_2 as a parametric expression

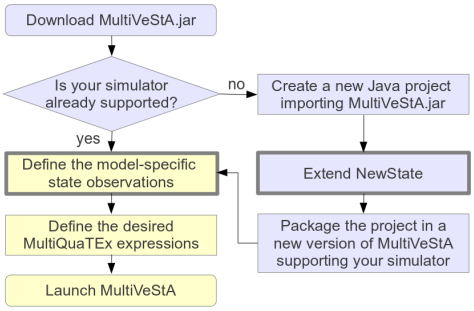

Figure 1: Steps necessary to exploit MultiVeStA.

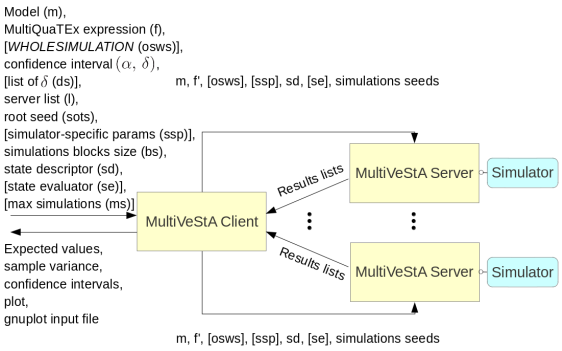

Figure 2: MultiVeStA architecture.

ignored until no more simulations are necessary for the other `eval` clauses, when the evaluation procedure terminates, and a real-value for each `eval` clause is returned. [22] provides an informal but thorough discussion of the new evaluation procedure.

4. MULTIVESTA

MultiVeStA extends PVeStA, which in turn extends VeStA, 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 currently supports some simulators, namely Alchemist [16], ARGoS [5], DEUS [3], MISSCEL [14], and the two originally supported ones [24], but just a few steps are required to integrate a new DES. Basically, it is necessary to extend the tool to support the new simulator, and then, for each model, the relevant states observations have to be specified. The required steps are sketched in 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. Integrating 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 wrappings are necessary to overcome the barriers introduced by the use of different programming languages. Noteworthy, the non Java-based PMaude has been integrated exploiting ExpectJ, a Java library allowing to interact with external processes, while the `C++` swarm robotic simulator ARGoS has been integrated using the Java Native Interface, which enriches Java with support for native `C/C++` code.

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 [15], then a Java project referencing it must be created. As next step, the class `NewState`, the only point of interaction between MultiVeStA and the simulators, must be extended. Essentially, three methods invoked by MultiVeStA have to be overridden, plus an optional one: **1**) `setSimulatorForNewSimulation(int randomSeed)`, invoked to reset the simulator and change its random seed before performing a simulation run; **2**) `performOneStepOfSimulation()`, invoked to order the simulator to perform one step of simulation; **3**) `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, however predicates common to any model of the considered simulator can be defined here once; **4**) `performWholeSimulation()`, an optional method invoked instead of `performOneStepOfSimulation()` if the user specifies the option *WHOLESIMULATION* (see Section 3). More details and examples can be found in [15, 22].

Before analyzing a model of a supported simulator it may be necessary to refine the `rval(int observation)` method to provide model-specific state observations. For Java-based simulators, the *IStateEvaluator* interface, consisting only of the method `double getVal(int observation, NewState state)`, has to be implemented. During the analysis phase, the user can then provide the class-name of the evaluator, which is dynamically loaded. If the simulator is not Java-based, then *IStateEvaluator* may still be similarly instantiated. However it may happen that the simulator state 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). This is the case of PMaude.

4.2 Architecture

Figure 2, adapted from [2], sketches the client-server architecture that MultiVeStA inherited and enriched 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 and the requests of simulations, collecting the results, computing statistical measures, and terminating the evaluation when the required accuracy is met. Outputs are provided to the user in different formats: lists of expected values for multi-expressions, or plots visualized in a minimal GUI and gnuplot input files for parametric multi-expressions.

A user provides several parameters to the client, among which we mention: the file names of the model and of the expression to be evaluated, the α, δ parameters and the the optional list of δs, the IP addresses of the servers, and the root seed. The latter is used to initialize a pseudo-random number generator which creates the seeds for the simulations. Additional simulator-specific parameters can be provided using the *ssp* option. Furthermore, the user provides the name of the class extending `NewState`, and the optional state evaluator. Finally, it is possible to specify the maximum

number of simulations to be performed, in which case also the sizes of the computed confidence intervals are returned to the user. This is indeed a further extension introduced in MultiVeStA. More details can be found in [15, 22].

5. VALIDATION

To demonstrate the feasibility and benefits of the integration of simulators with MultiVeStA, this section discusses the case of DEUS [3], and sketches some of the analysis tasks done for the mentioned cloud scenario.

The steps of Figure 1 have been followed to integrate DEUS. In particular, the class *DeusState*, extending *NewState*, has been provided, a detailed discussion of which is given in [22]. After completing the integration, it has been then possible to evaluate properties of the cloud scenario, like the ratio of successfully executed tasks during the simulation progress from 100 to 360,000 seconds at intervals of 10,000 seconds, for small, large and all tasks. The corresponding expression is reported in Listing 8. Here the observations number 11, 12 and 13, corresponding to the ratios of tasks executed for the different types of tasks, are estimated for the required intervals (recall that `s.rval(1)` provides the current simulated time). Note that the observations regarding ratios of executed tasks are model-specific, and required the definition of a state evaluator, whose details can be found in [22].

The expression of Listing 8 has been evaluated on a laptop having a 2.0 Ghz Quad Core and 16 GB of RAM. α and δ have been setted respectively to 0.05 and 0.004, thus the expected values have been computed wrt a 95% CI with a radius of 0.002. Three servers were used, and 100 has been setted as maximum number of simulations. About 4 hours, and 18 simulations distributed in the 3 servers in groups of 6 simulations have been required to evaluate the expression. Figure 3, generated using the gnuplot input file returned by MultiVeStA, plots the results of the expression. Without detailing the obtained results, it is important to note that, it has been possible to study three properties varying the simulated time: each point of the three lines is actually an expected value of the corresponding property, when instantiating the parameter x.

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.

The following discusses how the described analysis would be much more complex by using either DEUS alone or with PVeStA. As discussed, a MultiQuaTEx expression corresponds to a set of QuaTEx expressions sharing the same temporal operators, each having one of the `eval` clauses. The MultiQuaTEx expression of Listing 8 has three `eval` clauses, each having a parameter (x) instantiated with the 36 values from 100 to 360,000, with step 10,000. 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 $108 * 18 = 1,944$ simulations rather than 18. MultiVeStA thus provide a drastic performance improvement. In [17], 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 launch the simulations, to compute the CIs, and to launch 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 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.

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

Listing 8: Ratios of executed tasks on time

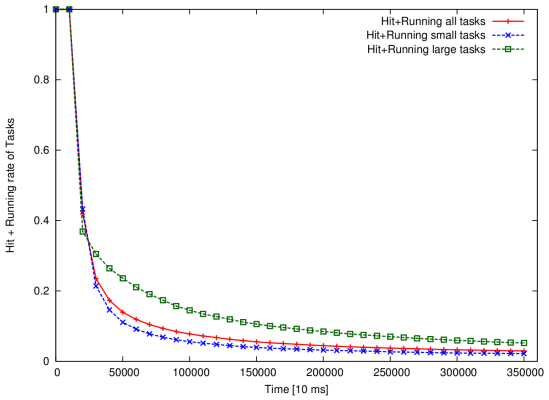

Figure 3: Analysis results.

6. RELATED WORKS

Many tools supporting statistical analysis exist. Examples of those supporting SMC are: APMC [4], YMER [27], and SAM [20]. Moreover, statistical extensions of established tools like PRISM [18] and UPPAAL [25] exist. However, differently from MultiVeStA, these tools have their own engines and model specification languages, as they do not aim at exploiting and enriching discrete event simulators.

An interesting tool sharing our aims is PLASMA-lab [9], which can be as well integrated to simulators. As typical SMC tools, it allows to answer questions like: *is a property satisfied by a model with a probability greater than a given threshold?*, and *what is the probability that a model satisfies a property?*. MultiVeStA further allows to obtain expected values of real-typed properties, like the mean time spent by tasks in queue. This is an important feature, as users of discrete event simulators are often interested on such real-typed measures. Moreover, it is not mentioned if PLASMA-lab reuses simulations, another important feature implicitly adopted by users of discrete event simulators when logging several measures. Nevertheless, the tool is interesting, as it has algorithms that use importance sampling to reduce the number and the length of simulations. It would be interesting to understand how these algorithms can be integrated in our approach.

7. CONCLUSIONS AND FUTURE WORK

We presented MultiVeStA, a statistical analysis tool allowing enrich discrete event simulators with distributed statistical analysis capabilities. Moreover, MultiQuaTEx enables to express system properties in a compact way, and to efficiently evaluate them. The tool has been used to reason on collision-avoidance robots [14], volunteer clouds [21] and crowd steering [17] scenarios, in the context of the European projects ASCENS, and SAPERE. As future work we plan to extend the set of supported simulators, e.g., considering ns-3 and OMNeT++. Moreover, we plan to improve our GUI, which currently consists in an interactive plot of the results.

8. ACKNOWLEDGMENT

We thank the proposers of (P)VeStA and QuaTEx: Gul Agha, Musab AlTurki, José Meseguer, Koushik Sen and Mahesh Viswanathan. We also thank Danilo Pianini, who provided many suggestions. Finally, we thank Michele Amoretti and Alberto Lluch Lafuente who supported us in this work.

9. REFERENCES

[1] G. A. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *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 *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 386–392. Springer, 2011.

[3] M. Amoretti, M. Agosti, and F. Zanichelli. DEUS: a discrete event universal simulator. In *Proceedings of Simutools '09*, pages 58:1–58:9, 2009.

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

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

[6] 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.

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

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

[9] 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.

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

[11] R. Bruni, A. Corradini, F. Gadducci, A. Lluch-Lafuente, and A. Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. *Submitted to Science of Computer Programming*, 2012.

[12] J. Eckhardt, T. Mühlbauer, M. AlTurki, J. Meseguer, and M. Wirsing. Stable availability under denial of service attacks through formal patterns. In *Proceedings of FASE'12*, pages 78–93. Springer-Verlag, 2012.

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

[14] MISSCEL: `sysma.lab.imtlucca.it/misscel`.

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

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

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

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

[19] G. Riley and T. Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, pages 15–34. Springer, 2010.

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

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

[22] S. Sebastio and A. Vandin. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. Technical Report `eprints.imtlucca.it/1798`.

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

[24] K. Sen, M. Viswanathan, and G. A. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *QEST*, pages 251–252, 2005.

[25] UPPAAL: `www.uppaal.org`.

[26] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of Simutools '08*, pages 60:1–60:10, 2008.

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