Simultaneous Simulation of Alternative Configurations of Markovian System Models

Shravan Gaonkar and William H. Sanders
gaonkar@ieee.org, whs@illinois.edu

Version date: March 5, 2009

Abstract

Simulation is a useful tool for engineering systems, and it is used in numerous and diverse fields. Simulation of large systems can result in better designs and make the design process more efficient. One important use of simulation lies in comparing and choosing the best among different simulation models that represent competing or alternative designs before the actual deployment and implementation. We have developed the Simultaneous Simulation of Alternative System Configurations (SSASC) approach, which provides a methodology that exploits the structural similarity among the alternative configurations and results in an efficient simulation algorithm that evaluates alternative configurations of a system simultaneously. In addition, we designed and implemented an efficient data structure for simulation state management to speed up the SSASC algorithm. Our approach to simulation is orthogonal to parallel or distributed simulation. The resulting improved algorithm could form the basis of an efficient parallel simulation framework.

I. Introduction

Deployment of large-scale systems is often expensive and sometimes catastrophic, as these systems generally have large numbers of interacting components. Failure of those components adds uncertainty to the normal operation of the system. To manage that uncertainty, to understand the systems in depth before deployment, and to protect them from unexpected repairs and costs, engineers rely heavily on detailed discrete-event simulations.

Simulation is applied in numerous and diverse fields, such as manufacturing systems, communications and protocol design, financial and economic engineering, operations research, and design of transportation networks and systems, among others. While simulation is a powerful engineering tool for analyzing systems, several hurdles must be crossed before it can be accepted widely as a standard design approach. First, the correctness of the simulation studies depends heavily upon the accuracy of the system representation. Second, simulations of large systems take a significant amount of computational resources and time. Even though the clock speed of sequential processors improves every year, the complexity of the systems that must be modeled also increases every year. Furthermore, the real utility of simulation lies in comparing alternatives before actual implementation [17], suggesting that the system model has to be simulated and evaluated multiple times for a large number of design configurations and parameter values to allow determination of a good design configuration choice. To perform a thorough analysis of a large number of configurations with varying system design parameter values, it is important to develop efficient simulation methods that can evaluate a large number of system configurations quickly and accurately.

Researchers have focused on evaluating large discrete-event systems using parallel and distributed simulation methods [12], [23], [20], [26]. The novelty and appeal of parallel and distributed simulation methods have not resulted in the widespread use of these techniques in the real world. This can be attributed solely to the economic viability of the solution for companies, as it is difficult for them to justify the purchase of large clusters of computer nodes needed to run parallel simulation [20]. To encourage the widespread use of simulation, it is necessary to make simulation more efficient in evaluating large numbers of alternative design choices and configurations. If the system under evaluation is scrutinized carefully, one will notice that changing the system configuration or parameter values does not dramatically alter the structure or behavior. Rather, much of the system behavior is similar for most of the possible alternative configurations. That fact suggests the possibility of an efficient way to simulate multiple alternative system configurations simultaneously on a uniprocessor system.

In our work on Simultaneous Simulation of Alternative System Configurations (SSASC) described in this paper [9], our first contribution has been to extend the ideas of Vakili [30] about single-clock simulation and Chen et al. [4] with a methodology that exploits the structural similarity among the alternative configurations, while eliminating pseudo transitions. The result is an efficient simulation algorithm that evaluates all the alternative configurations of a system simultaneously, eliminating the limitations of the single-clock technique for models with event rates that vary greatly among alternative configurations, as is often the case in dependability evaluations. The second contribution of this paper is a datastructure to represent and manipulate the state of the simulated system to further improve the simulation efficiency. The alternative configurations often have very similar states, due to the structural similarity among alternative models during the execution of the simulation. Furthermore, state updates follow a unique pattern that allows us to build a data structure that enables us to represent the states more efficiently, thereby further improving the speed-up of the SSASC algorithm. In this paper, we develop of an efficient data structure that exploits the unique state update pattern and structural similarity among alternative configurations to enhance the speed-up of the simultaneous simulation algorithm. The speed-up we obtain is significant, and we show that simulation using our approach is 30 to 55 times faster than separate simulation of all of the alternative configurations. Our third and final contribution is integration of a variance reduction technique into the SSASC algorithm that exploits the speed-up obtained because of the reduction of variance due to the use of common random numbers. This improvement provides an additional speed-up that may be up to double the existing speed-up.

The rest of the paper is organized as follows. Section II provides insight into the current state of the art in distributed and parallel simulation, distinguishing it from the SSASC algorithm. Section III describes the approach, development, and correctness of SSASC. Section IV describes the algorithmic implementation of SSASC along with the data structure used to efficiently represent and update the state of the simulation model. Section V discusses how SSASC was implemented in Möbius. Section VI presents the analysis and experiment that explore the effectiveness of the SSASC. We conclude in Section VIII.

II. Background and Related Work

In this section, we explore the background and other related work relevant to simultaneous simulation. In each subsection, we review general concepts with additional pointers to references that provide greater detail. Furthermore, we provide insight into the novelty of our approach, which has been built to address the shortcomings of other existing research and techniques.

When the number of alternative configurations is in the thousands, discrete event simulation is often the best way to evaluate the system. Therefore, we first provide, in Section II-A, a brief overview of discrete event simulation of Markovian stochastic models. Finally, in Section II-B, we cover the topic of simultaneous simulation of a large number of alternative configurations and we compare the existing techniques to our approach and discuss approaches that could be used to extend the uniformization in simulation to non-Markovian system models.

A. Discrete-event Simulation

Here, we provide a literature review on adaptive uniformization as applied to simulation, using an example M/M/2/B queuing system as shown in Figure 1. This example is used throughout this paper to build the reader's understanding of SSASC. Readers are referred to [11] and [24] for more detailed descriptions of uniformization and adaptive uniformization in simulation.

In the example model, a Poisson stream of jobs arrives at rate α and is routed to two exponential servers with service rates μ_{slow} and μ_{fast} with probabilities p and $(1-p)$, respectively. Both servers have a finite buffers, whose size are denoted by B_{slow} and B_{fast}. Both servers provide service using the First Come First Serve ($FCFS$) policy. When a job completes, it departs from the system. The state of the system is represented by the queue length of jobs waiting to be served at the slow and fast servers.

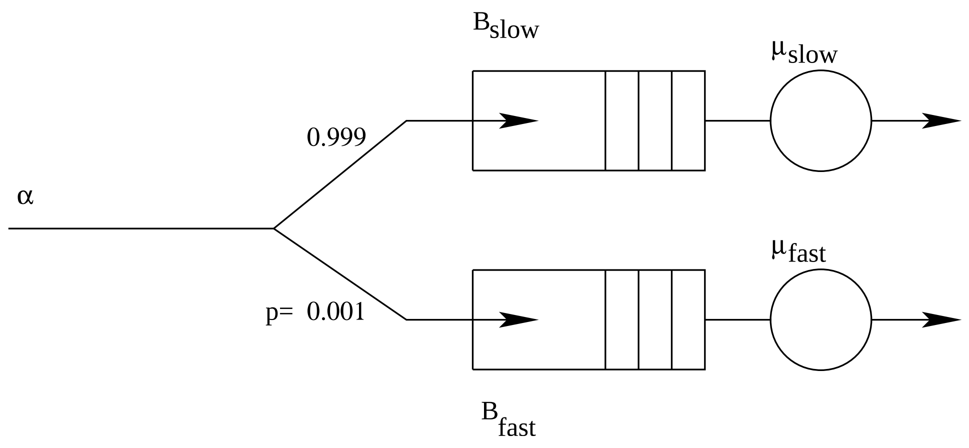

Fig. 1: M/M/2/B queuing network.

1) Uniformization: In order to simulate the queuing system using uniformization, it is necessary to generate events as Poisson processes with parameter λ, where $\lambda = \alpha + \mu_{slow} + \mu_{fast}$. Each transition event is designated as

(a) an arrival event, AE, with probability $\frac{\alpha}{\lambda}$,
(b) a potential departure from the slow server, PDSS, with probability $\frac{\mu_{slow}}{\lambda}$, or
(c) a potential departure from the fast server, PDFS, with probability $\frac{\mu_{fast}}{\lambda}$.

Execution or firing of the events changes the state of the system. The efficiency of simulation depends upon the fact that firing of an event results in a change to the state of the system. For example, whenever an event is designated as PDFS while the fast server has no pending jobs, the firing of the event does not update the state of the system. In such a situation, the firing of the event is called a *pseudo transition*. Since the state of the system does not change, the *pseudo transition* adds to simulation inefficiency.

2) Adaptive Uniformization: Since it is possible to have models whose simulations might lead to a large number of pseudo transitions, adaptive uniformization provides a methodology to alleviate these pseudo transitions to improve simulation efficiency [32]. In particular, for the given M/M/2/B queuing system, if the routing probability p is set very close to 1, i.e., $p \simeq 1$, and $\mu_{slow} \ll \mu_{fast}$, most of the incoming jobs will be routed to the slow server, but most of the events generated will be designated as departures from the fast server (PDFS). However, the fast server is idle most of the time, causing the simulator to label most of those events as pseudo transitions. Such conditions in the model add inefficiency in the discrete-event simulator. To alleviate the problem, it is possible to adaptively uniformize the firing rate based on the state of the system, as described in [24]. The adaptive uniformization rate for the M/M/2/B model, depending on the state of the system, is as shown below:

$\lambda \equiv \alpha$	When both servers are idle
$\lambda \equiv \alpha + \mu_{fast}$	When the slow server is idle
$\lambda \equiv \alpha + \mu_{slow}$	When the fast server is idle
$\lambda \equiv \alpha + \mu_{fast} + \mu_{slow}$	When both servers are busy

This technique of changing uniformization parameter values depending on the state of the system guarantees that the simulator never fires a pseudo transition. Thus, it enables continuous computational progress during the execution of the simulation model. However, one must note that this efficiency is obtained only while simulating individual simulation configurations. In this paper, we show how to extend it to simultaneous simulation of alternative configurations.

B. Simultaneous Simulation of Alternative System Configurations

In this section, we describe the existing approaches taken to simulate alternative design configurations simultaneously, and provide insight into their potential drawback, which is overcome by our SSASC algorithm [9], [8]. We conclude this section with a literature review on related work on the generalization of uniformization of simultaneous simulation to general distributions.

1) Single-Clock Multiple Simulations: Single-Clock Multiple Simulations (SCMS) [30] are a class of simulation techniques that exploit the commonality that exists during evaluation of the alternative configurations of a discrete-event system. In the SCMS approach, clock ticks are generated based upon a dominant process in the system, and the events are chosen by appropriate thinning of the process for each alternative configuration of the system. In that way, the clock update mechanism and the state update mechanism are decoupled. The state update mechanism provides no feedback to the clock update mechanism regarding generation of the next events. If a single configuration of the discrete-event model is being evaluated (as in traditional discrete-event simulation), SCMS would be equivalent to uniformization-based simulation as described in the subsection II-A1. Note that the single-clock multiple simulation approach does not maintain an enabled event set, which means that it cannot take advantage of the adaptive uniformization.

To illustrate the simultaneous evaluation of multiple configurations using the SCMS technique, reconsider the example of the M/M/2/B queuing system from Figure 1. Let the service rate of the slow server be the design parameter that needs to be determined. For simplicity, suppose that we have n possible choices for the service rate of the slow server. The SCMS would define the dominant Poisson process as $\lambda \equiv \alpha + \mu_{fast} + \widehat{\mu_{slow}}$, which would be used to generate the main clock tick where $\widehat{\mu_{slow}} = max(\mu^i_{slow}); 1 \leq i \leq n$ and i represents the i^{th} configuration. As in the uniformization approach described earlier, each clock tick is designated as an arrival, as a potential departure from the slow server, or as a potential departure from the fast server, and this information is sent to each alternative configuration of the model. Consider a scenario in which the event is designated as a potential departure of a customer from the slow server. Each alternative configuration i with a nonempty buffer would update its state with the probability $\frac{\mu^i_{slow}}{\widehat{\mu_{slow}}}$. This probability effectively thins the Poisson process as needed for each configuration for the departure event from the slow server. Using this technique, we can evaluate all the alternative configurations of the M/M/2/B queuing system together with a single dominant clock.

The salient feature of the technique is that it uses a single clock to update all the alternative configurations and eliminates the need to maintain any event list. However, as mentioned by Vakili [30], this technique gives rise to the possibility that an excessive number of pseudo transitions will be generated, creating the potential for inefficiency. In this paper, we propose the SSASC technique for simulating a family of alternative configurations of a system by using certain aspects from the single-clock technique and adaptive uniformization to achieve better efficiency in simulation.

2) Simultaneous Simulation of Non-Markovian Systems: The SSASC technique is quite efficient for simulating a large number of alternative configurations. However, its applicability is limited to a class of system models with exponential distributions. Here, we focus on other related work that attempts to extend uniformization from exponential to non-exponential distribution.

Several approximation techniques exist that try to match the first, second, and higher-order moments of the general distribution using phase-type distributions, such as hyper-exponential and hypo-exponential distributions [4], [2], [1], [29]. Quasi birth-death processes with exponential distributions have also been used to approximate general distributions to evaluate systems, particularly queuing systems with general distributions [19], [14]. In addition, research groups have developed hybrid simulation approaches, in which the simulation technique of uniformization for exponential distributions is combined with traditional event list management for non-exponential distributions [3], [25], [28].

III. SSASC: Markovian Models

In this section, we develop the theoretical representation of alternative simulation configurations of discrete-event systems in a unified framework to evaluate the models simultaneously. We describe an approach based on adaptive uniformization for simulation of a discrete-event system with multiple parameter value settings. We first describe the general formal model, generalized semi-Markov processes

TABLE I: GSMP representation of M/M/2/B queue

S	=	$\{n = [n_{slow}, n_{fast}]$: n_i is the number of jobs on server $i, i = \{slow, fast\}\}$,
E	=	$\{a, d_{slow}, d_{fast}\}$ (a = arrival, d_i = departure from server $i, i = \{slow, fast\}$),
P	=	$p([n_{slow}{+}1, n_{fast}], [n_{slow}, n_{fast}], a) = p$ if $n_{slow} \leq B_{slow}$, $p([n_{slow}, n_{fast}{+}1], [n_{slow}, n_{fast}], a){=}1 - p$ if $n_{fast} \leq B_{fast}$, $p([n_{slow}{-}1, n_{fast}], [n_{slow}, n_{fast}], d_{slow}){=}1$ if $n_{slow} > 0$, $p([n_{slow}, n_{fast}{-}1], [n_{slow}, n_{fast}], d_{fast}){=}1$ if $n_{fast} > 0$, and
ψ	=	$\{1\text{-}e^{(-\alpha x)}, 1\text{-}e^{(-\mu_{slow} x)}, 1\text{-}e^{(-\mu_{fast} x)}\}$.

($GSMPs$), that we use to represent the discrete-event model. We adapt this formal representation from [30] and [31] for consistency and clarity to show how our approach is a significant improvement over the SCMS technique. We then describe how the general GSMP, when restricted to exponentially distributed clock times, can be modified to represent configurations with different parameter values such that all the configurations of the system can be simulated simultaneously with adaptive uniformization. Finally, we provide necessary intuition into the workings of the SSASC algorithm by describing how the technique can be used to simulate alternative configurations simultaneously, and more efficiently than SCMS.

A. Generalized Semi-Markov Processes

A discrete-event simulation can be represented using a generalized semi-Markov process (GSMP) [10]. A GSMP is characterized by a triple, $GSMP = (S, E(s), p(., s, e))$, with a set of input distributions, $\psi = \bigcup F(.; s, e)$, $e \in E$, defined as follows:

S	=	a set of physical states of a system,
E	=	$\{e_1, e_2, ..., e_k\}$ is a set of finite events for the system,
$E(s)$	=	the set of possible events when the system is in state s,
$p(s', s, e)$	=	the probability of transition from s to s' when event e occurs,
ψ	=	the set of all clock distributions $F(.; s, e)$ for the model, where $F(.; s, e)$ is the probability distribution of the "clock time" of event e, when the system is in state s.

The M/M/2/B discrete-event system from Figure 1 can be represented using a GSMP as shown in Table I. In the above example, the parameters of interest that one could vary include the number of jobs that the servers can queue (i.e., B_{slow} and B_{fast}), the service rate of the servers (i.e., μ_{slow} and μ_{fast}), the inter-arrival rates of jobs (i.e., α), and the routing probability, p. All these parameter value variations result in alternative design configurations of the M/M/2/B model, and can be evaluated for the reward measures of interest using simulations.

B. Creating Alternative Configurations of the Simulation Model

In general, the behavior of a discrete-event system is governed by two components: (a) the state space of the system, and (b) the events and rate that cause the state changes. From the example in Figure 1, we see that alternative configurations of a discrete-event system can be created by varying parameter values, such as B_{slow} or B_{fast}, that change the system's state space, or by varying parameter values, such as p, α, μ_{slow}, or μ_{fast}, that change the rate of state transitions governed by the event rate.

TABLE II: Parameter values of the alternative design configurations for the M/M/2/B queuing model

Config #	0	1	2	3	4	5	6	7	8	9	10	11
B_{slow}	5	5	5	5	5	5	5	5	5	5	5	5
B_{fast}	7	9	7	9	7	9	7	9	7	9	7	9
μ_{slow}	1	1	2	2	1	1	2	2	1	1	2	2
μ_{fast}	3	3	3	3	4	4	4	4	5	5	5	5

In particular, for a system represented as a GSMP, it is possible to generate alternative configurations by varying the following parameter values:

- The probability transition function,$p(s', s, e)$, that captures the behavior of the parameters that controls the state space of the system. By varying either the values of the probability that governs the probability transition function or the conditions that control the probability transition function, one can create discrete-event models that have different state spaces.
- The input distribution function, ψ, and the probability transition function capture the behavior of the parameters that control the event rate of the system. Changing parameter values of the rate parameters of events or values of the probability that governs the probability transition function varies the rate of change of system behavior of a discrete-event model.

Using a combination of both types of parameters, it is possible to generate a large family of different configurations of the system that can be studied simultaneously. Table II describes a set of alternative configurations for the M/M/2/B queuing model.

C. Construction of Equivalent GSMP′s for Each Alternative Configuration

To construct alternative configurations of a model that can be simulated simultaneously and correctly, we construct a GSMP′ that augments the GSMP of each independent alternative configuration so that the behavior of the GSMP′ is statistically identical to that of the GSMP. The goal is to have a common input distribution function ψ for all the models. That would allow us to maintain a common enabled event set (EES) while simulating all the alternative configurations, thus amortizing the cost of event selection and firing. That necessitates modifications to the probability transition functions for each of the alternative configurations, to compensate for the existence of a common input distribution function. In this section, we describe the construction of GSMP′ necessary to modify each alternative configuration so as to enable the simultaneous evaluation of all the alternative configurations.

Consider a discrete-event model represented by a $GSMP = (S, E(s), p(., s, e))$. Let k be the number of parameters we wish to vary. Each parameter k_i is evaluated for n_j combinations, which results in $n = \prod n_j$ alternative configurations of the discrete-event model. It is fairly simple to generate the GSMP for each of the alternative configurations, as seen in the previous section. Let each alternative configuration be denoted by $GSMP^v = (S^v, E^v(s), p^v(., s, e))$. Let i denote the i^{th} alternative configuration, where $0 \leq i < n$.

Each new augmented GSMP$'^v$, which supports simultaneous simulation, is constructed from GSMP^v as follows.

1) The states S for the equivalent GSMP$'^v$ are to be the same as the states S^v of the particular configuration, i.e., $S'^v = S^v$.
2) The new set of actions for GSMP$'^v$ is the union of the actions of all the configurations ($E'^v = \bigcup_{i=1}^{n} E^i$). Note that an action $e^i \in E^i$ is said to be *equivalent* to $e^j \in E^j$, i.e., $e^i \equiv e^j$, if it has the same label, ignoring the timing aspect of the action.
3) The set of possible events that are enabled is the union of the possible events of all the configurations, i.e., $E'^v(s) = E^v(s)$.
4) The probability distribution $F(.; s, e)$ of each event $e \in E$ and state $s \in S$ is modified to correspond to the shortest holding time of the individual configurations. When the event is fired, the process is thinned to reflect its true behavior. The thinning of the Poisson process is done using the state

transition probability p described later. In terms of the rate parameter for the input distribution function, λ'_e is now defined as $\lambda'_e = MAX_{i=1}^{n} \lambda^i_e$. The holding time in a state s^v, given that the event e is enabled, is given by $F(.; s^v, e) = F^i(.; s^v, e'')$, provided that $e'' \in E^i(s^v)$, $e \equiv e''$, and $\lambda_{e''} = \lambda'_e$, where i is the index of variant that has the event e''.

5) The transition probability for the new GSMP$'^v$ is the modified transition probability of the individual configuration. For each configuration that does not have the event e defined, i.e., $e \in (E'^v(s) - E^v(s))$, a pseudo transition with probability 1 is added. Additionally, the transition probability is appropriately thinned to account for the timing aspect associated with event e for the variant v for all $e \in E^v(s)$, i.e., $p'(s', s, e) = \frac{\lambda_e}{\lambda'_e}(p(s', s, e))$.

Note that the main difference between the construction of Vakili's GSMP, denoted by GSMP$''$, and our construction of GSMP$'$ is that every event is active in every state in GSMP$''$ while GSMP$'$ maintains the original active events $E(s)$ from the original GSMP. Furthermore, in our construction of GSMP$'$, we modify the probability distribution, $F(.; s, e)$, and probability transition function, $p(s', s, e)$, to accomplish the equivalent alternative GSMP$'$. That particular modification allows us to use adaptive uniformization.

We now show that the behavior of an alternative configuration of GSMP and the behavior of its augmented version that enables simultaneous simulation, GSMP$'$, are statistically equivalent for certain class of stochastic models. Here, the GSMPs are assumed to have exponential distributed clock times. Therefore, it suffices to show that the Markov chains of the processes represented by the original GSMP and the augmented GSMP$'$ are equivalent. We do so by showing that the generator matrices Q for both GSMP models are equal. Formally,

Proposition 3.1: Let $S = \{S(t); t \geq 0\}$ and $S' = \{S'(t); t \geq 0\}$ be the processes representing the original GSMP and the augmented GSMP$'$, respectively. Then S is stochastically equivalent to S'.

Proof: Since we consider GSMPs for which all the clock times are exponentially distributed, their behavior can be represented as continuous time Markov chains (CTMCs). By the definition of GSMP, the rate of going from state s_i to state s_j is the product of the probability transition function $p(s_j, s_i, e)$ and λ_e for each event e enabled when the model is in state s_i. Therefore, the generator matrix Q' of the CTMC that represents GSMP$'$ is given by $q'_{ij} = \sum\limits_{e \in E'(s_i)} p'(s_j, s_i, e)\lambda'_e$.

Recall that $E'(s_i) = E(s_i)$ from step 5 in the construction of the augmented GSMP$'$. Therefore, the generator matrix entry for Q' is modified as follows: $q'_{ij} = \sum\limits_{e \in E(s_i)} p'(s_j, s_i, e)\lambda'_e$.

Note that $E(s_i)$ are the events enabled in the original GSMP model. Furthermore, from step 6 of the construction of the GSMP$'$, we know that $p'(s_j, s_i, e) = \frac{\lambda_e}{\lambda'_e} p(s_j, s_i, e)$.

Replacing $p'(s_j, s_i, e)$ in the above equation and canceling common terms, we obtain $q'_{ij} = \sum\limits_{e \in E(s_i)} p(s_j, s_i, e)\lambda_e = q_{ij}$, where q_{ij} is the generator matrix of the original GSMP.

Therefore $Q = Q'$, which implies that S is stochastically equivalent to S'. $\diamondsuit$

Now that we have shown that the original GSMP and the modified GSMP$'$ are stochastically equivalent such that all the augmented GSMP$'$s have the same distribution functions ψ, we can simulate all alternative configurations of the model correctly using adaptive uniformization.

In the case of SCMS using uniformization [30], use of a Poisson process Λ that drives the process S', where $\Lambda = \sum \lambda_e$ for $e \in E'(s)$, leads to the possibility of a large number of pseudo transitions due to events $e' \in (E'(s) - E(s))$ for a given state of the system. We alleviate this problem by considering a non-homogeneous Poisson process ($NHPP$) Λ'_n that drives the process S', where n is the n^{th} transition epoch. The constant uniformization rate for every epoch is determined by the events that are enabled in each of the configurations of the model being evaluated. As argued in [18], that uniformization approach is valid even if one thins a nonhomogeneous Poisson process. In effect, in our technique, an NHPP with a piecewise constant epoch is used to uniformize after the firing of each event. To be conservative and prevent incorrect uniformization, care is taken to ensure that the adaptive rate is always greater than or equal to the actual possible transition rates of all the enabled events. In that way, the updating of the Λ_n is done as follows after each epoch: $\Lambda_i = \sum \lambda_e$, where $e \in \cup_{i=1}^{n} E(s^i)$.

Note that $E(s^i)$ is the set of events from the original representation of the discrete-event model. It is always true that $\cup_{i=1}^{n}(E(s^i)) \subset E'(s^v)$ for any variant v. There is always the possibility that $\cup_{i=1}^{n}(E(s^i)) = E'(s)$, i.e., the system could have all the events of all the variants enabled at all times. That could cause the adaptive uniformization to behave just like uniformization, except that the construction of the adaptive uniformization parameter will always ensure that there is useful computation from at least one of the configurations, i.e., the firing of an event in adaptive uniformization will change the state of at least one of the configurations, which might not be the case with the traditional uniformization technique. Hence, our technique will always guarantee progress in simulation for at least one of the configurations. The next section describes a practical implementation of the simulation algorithm that uses our new approach.

IV. Adaptive Uniformization Algorithm of SSASC

Continuing with the notations from the previous section, consider a scenario in which we want to simulate N alternative configurations of a system parameterized by its design parameter values. As we discussed earlier, we obtain the new configurations from the original model by modifying their probability transition functions $p(s', s, e)$, input distribution functions ψ, and the conditions that enable the state transition.

The simulation algorithm (See Algorithm 1) has three basic components: (1) a Common Adaptive Clock to generate the next event and to update the enabled event set, EES, based on the new state of the alternative configurations; (2) an Efficient State Management System, ESMS, to efficiently update the state of all the configurations simultaneously for the event that occurred; and (3) a reward redefinition process and an evaluation criterion that enables selection of the best alternative configuration using a statistical procedure based on a common random number generator. The algorithm is executed until a desired confidence interval is achieved through execution of multiple batches (in the case of steady-state simulation) or replication (in the case of terminating or transient simulation).

A. Common Adaptive Clock for SSASC

The SSASC algorithm begins with an empty EES (Line 1 in Algorithm 1). The simulation algorithm initially iterates through all the events in the model, adding them to the EES if they are enabled by any of the configurations (Line 2). Once the initial EES has been built, the simulator executes the basic components in a loop (Lines 5–11) until a terminating condition is satisfied.

In each iteration of the loop, the algorithm generates the next event epoch using an exponential random variable using the adaptive uniformization rate, Δ_n. An event, e, is picked randomly from the EES (Line 6(d)) and is weighted by the events firing rate, λ_e, that is in the EES. SSASC updates the state of the alternative configurations based on the firing of this event e (Line 7) provided that their probability transition function p is greater than u, where u is a uniform random variable between 0 and 1. Only those alternative configurations that are enabled for the particular fired event, e, have their state updated. Note that p has been modified to accommodate simultaneous simulation (See Section III-C).

SSASC updates EES to remove disabled events and add new enabled events (Line 8). Events that are disabled in all of the alternative configurations are removed from the EES. Events that are enabled in any of the alternative configurations are added to the EES. The algorithm computes the new adaptive uniformization rate for the updated EES (Line 8(c)). To improve the efficiency of the state update procedure, SSASC implements an ESMS that is described in the next subsection (See Section IV-B).

The reward measures, R, are computed for each alternative configuration based on the current state of the configuration. This loop is iterated until a defined terminating condition occurs. Often the terminating condition is either a fixed number of iterations or a certain confidence level obtained for the reward

Algorithm 1 SSASC using adaptive uniformization: Exponential distributions

```
1: Let
     $EES$            = $\emptyset$, enabled event set initialized to empty set,
     $N$              = number of alternative configurations,
     $E$              = number of exponential events in the system model,
     $v$              = index of the $v^{th}$ alternative configuration,
     $n$              = index to the $n^{th}$ event epoch,
     $\tau_n$         = $n^{th}$ event epoch,
     $n_e$            = event fired in the $n^{th}$ event epoch,
     $e_j$            = exponential event $j$ in discrete-event system model,
     $\lambda^v_{e_j}$ = exponential rate of event $j$ in configuration $v$,
     $\lambda_{e_j}$  = $max(\lambda^v_{e_j})$,
     $s^v_0$          = initial state of each configuration,
     $D(e)$           = dependency list that maintains the set of enabled events enabled due to firing of event $e$,
     $u$              = $U(0,1)$, uniform random variable,
     $R^v_k$          = $k^{th}$ reward measure defined on variant $v$,
     $erv$            = exponential random variable with rate 1,
     $\Lambda_n$      = adaptive uniformization rate.
2: $\forall e \in \bigcup_{v=0}^{N} E(s^v_0)$, $EES = EES + \{e\}$.
3: $\Lambda_0 = \sum \lambda_{e_j}$ where $e_j \in EES$.
4: $n = 0$, $\tau_0 = 0$.
5: repeat
6:     Generate next event.
       (a)  $\tau_{n+1} = \tau_n + \frac{erv}{\Lambda_n}$.
       (b)  $P[0] = 0$.
       (c)  $for(j = 1$; j$\leq$ |EES|; $j++)$
                  $P[j]$ = $P[j-1]$ + $\frac{\lambda_{e_j}}{\Lambda_n}$.
       (d)  $n_e = e_j$ where $e_j \in EES$
            iff $(P[j-1] \leq u < P[j])$.
7:     Update state (refer to Section IV-B).
       (a)  $\forall v$ with $n_e \in E(s^v_n)$ enabled, set $s^v_n$ to the next state $s'^v_{n+1}$ if $u > p(s'^v_{n+1}, s^v_n, n_e)$.
8:     Update EES
       (a)  $\forall e \in EES$, $EES = EES - \{e\}$,
            if $e \notin \bigcup E(s^v_{n+1})$.
       (b)  $\forall e' \in D(n_e)$, $e' \in \bigcup E(s^v_{n+1})$,
            $EES = EES + \{e\}$.
       (c)  $\Lambda_{n+1} = \sum \lambda_{e_j}$ where $e_j \in EES$.
9:     $\forall v, \forall k$, compute $R^v_k$.
10:    $n = n + 1$.
11: until a defined terminating condition. {Refer to Section IV-C for terminating condition.}
```

measures. To further improve simulation efficiency, SSASC redefines the reward measures to incorporate a variance reduction technique, as described in Section IV-C.

B. Efficient State Management System (ESMS) for SSASC

SSASC has been shown to produce substantial execution speed-up because of a common adaptive clock (refer to experimental results in Section VII). However, there are significant overheads due to the state-saving/updating operation in the simultaneous simulation algorithm (Line 7 in Algorithm 1), which can be further optimized. The loss of the expected speed-up of the SSASC algorithm is caused by the large memory footprint used to represent the states of the alternative configurations and the operations used to update the state of the model. In particular, each time an event is fired, the simulation algorithm needs to check and update the state variables of all alternative configurations. If the number of alternative configurations is large, a substantial overhead is caused by the need to iterate through the state variables of the simulation configurations and update the states.

In order to understand the basic state management approach in SSASC, consider the M/M/2/B queuing system as shown in Figure 1. The state of each alternative configuration is represented by a tuple, $< n_{slow}, n_{fast} >$. Using the 12 alternative system configurations (refer to Table II for parameter values)

TABLE III: Simulation state of the alternative design configurations for the M/M/2/B queuing model

Config #	0	1	2	3	4	5	6	7	8	9	10	11
n_{slow}	0	0	0	0	0	0	0	0	0	0	0	0
n_{fast}	0	0	0	0	0	0	0	0	0	0	0	0

TABLE IV: Trace of SSASC simulation of the M/M/2/B queuing network

	Activity (Event) Fired	State Variable	
		n_{slow}	n_{fast}
1	initial state	000000000000	000000000000
2	λ, arrives at slow server	111111111111	000000000000
3	λ, arrives at fast server	111111111111	111111111111
4	μ_{fast}, only configurations with rates 4, 5	111111111111	111100000000
5	λ, arrives at fast server	111111111111	222211111111
6	μ_{slow}, only configurations with rate 2	001100110011	222211111111
7	μ_{fast}	001100110011	111100000000
8	μ_{fast}	001100110011	000000000000

of the queuing system, we can represent the initial state of the state variable of all configurations as an array of 12 integers, as shown in Table III.

For each state variable update after the firing of an event, the SSASC iterates all of the configurations' states and updates them individually. Table IV traces out the state of the state variables n_{slow} and n_{fast} for a particular simulation trajectory of the SSASC algorithm. That update process adds a large overhead when the number of configurations is very large (in the thousands). However, in Table IV, it's evident that the change of state of the different configurations follows structured patterns. The reason is that the configurations share similar simulation model structures and have very similar stochastic behavioral properties. For example, configurations 0 and 1 differ only in the buffer capacity of the fast server (refer to Table II). For the given parameter values, the simulation trajectories of both of the configurations will be almost identical for most of the simulation time. This creates the opportunity to design an efficient state representation that would reduce cost overhead, in terms of both execution time and memory, to update the state variable of the configurations.

Furthermore, for each event fired, only a regular subset of the alternative configurations' states are updated, due to the thinning of the poisson process. From the traces of simulation, we see that when μ_{fast} fired only for rates 4 and 5 (see line 4 in Table IV), only configurations 4 through 11 were updated. Similarly, for μ_{slow} (see line 6 in Table IV), configurations 1, 3, 5, 7, 9, and 11 were updated. All of these patterns can be predetermined before the running of the simulation algorithm to provide a regular structure to update the state of the affected alternative configurations. That would significantly reduce the overhead of updating states in the SSASC algorithm.

Finally, it is important to note that the most common operations on the state variable are always accesses and updates to all the alternative configurations. Therefore, the data structure can be designed in a manner that is most efficient for the group access. Using the properties discussed above, we present the data structure and operations supported by it to enable the speed-up in updating the state as the simulation progresses.

1) Data-structure for ESMS: The ESMS necessary to perform efficient state management has two components. The first component is a data structure that encodes all of the individual states of all the alternative configurations to make them compact to represent, and efficient to access and update. Thus, the complete state of all the alternative configurations with M individual states is encapsulated by M compact data structure representations. We call an individual state a *state variable* in the remainder of this paper. For the M/M/2/B queuing system, n_{slow} and n_{fast} are represented using the compact state representation or CSR.

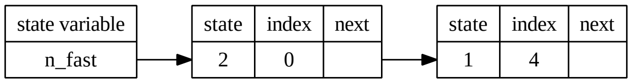

Fig. 2: Compact state representation of n_{fast} using linked lists

TABLE V: Trace of SSASC simulation with ESMS of a M/M/2/B queuing network

	Activity (event) fired	State Variable	
		n_{slow}	n_{fast}
1	initial state	[0,0,11]	[0,0,11]
2	λ, arrives at slow server	[1,0,11]	[0,0,11]
3	λ, arrives at fast server	[1,0,11]	[1,0,11]
4	μ_{fast}, only configurations with rates 4, 5	[1,0,11]	[1,0,3],[0,4,11]
5	λ, arrives at fast server	[1,0,11]	[2,0,3],[1,4,11]
6	μ_{slow}, only configurations with rate 2	[0,0,5],[1,6,11]	[2,0,3],[1,4,11]
7	μ_{fast}	[0,0,5],[1,6,11]	[1,0,3],[0,4,11]
8	μ_{fast}	[0,0,5],[1,6,11]	[0,0,11]

The second component is an additional support data structure, called the Indirect Reference List, or IRL, that provides a one-to-one mapping between state variable and alternative design configurations. IRL provides regularity to the access pattern to SSASC when the simulation updates the states of the model. That prevents fragmentation of CSR.

2) Compact State Representation (CSR): In SSASC, any access or update operation on the state variable includes exactly N operations, where N is the number of alternative configurations. Table IV shows that the state of the system is updated $N = 12$ times for each fired event. If we were to represent each state variable in a more compact form, we could reduce the average number of operations to be less than N.

CSR is achieved with a simple data structure that encodes the state of the alternative configuration. This encoding can be represented using linked lists. Each cell in the list has three elements: the *state* of the model, the *index* of the current cell, and a pointer to the *next* cell. The linked list of tuples, [*state*, *index*, *next*], represents the state variable of the simulation model. Figure 2 represents the CSR data structure for the simulation trace when a customer arrives to the fast server (Line 5) as depicted in Table IV. Table V traces the same simulation trajectory with the CSR data-structure enabled for the state variables.

Since the size of the CSR is bounded by the number of alternative configurations, N, an array implementation of the linked list is very efficient. Furthermore, accesses and updates are executed on a single contiguous block of memory, which makes them efficient on processors that provide pre-fetching and caching of blocks.

3) Indirect Reference List: From our example M/M/2/B queuing model, event rate parameter μ_{fast} is in the sorted order that matches the ordering of the alternative configurations. Thus, all state variables affected by μ_{fast}, such as n_{fast}, will benefit from the CSR data-structure. However, state variable n_{slow}, which is affected by μ_{slow}, will be fragmented if represented by CSR (as seen in line 6) as a series of "01" strings (as shown in Table IV). It is possible to mitigate this fragmentation by building an Indirect Referencing List (IRL) for each state that has a different order of event rates that is different from the actual configuration order. IRL is built only once, during the initialization of the simulation model. The SSASC algorithm uses IRL to access and update state variables based on indirect referencing of the state variables. Table VI presents the IRL for the M/M/2/B queuing model. The simulation trace shown in Table IV is modified by ESMS is now shown in Table V.

TABLE VI: Indirect reference list for the M/M/2/B queuing model

Config #	0	1	2	3	4	5	6	7	8	9	10	11
n_{slow}	0	2	4	6	8	10	1	3	5	7	9	11
n_{fast}	0	1	2	3	4	5	6	7	8	9	10	11

C. Reward Redefinition to Determine the Best Alternative Configuration

In Appendix A, we present a statistical procedure, two-stage selection (developed in [7]) to choose the best alternative configuration automatically. SSASC generates *common random numbers* (CRN) automatically to seed its simulation, which have been considered one of the best and most popular approaches to variance reduction [17]. We exploit this inherent advantage provided by SSASC to reduce the number of batches (in steady state simulation) or replications (in terminating simulation) to show the real advantage of SSASC over traditional simulation approaches. SSASC incorporates the two-stage selection as its terminating condition in Algorithm 1 (in line 11) by automating the process as follows.

Given that R_i and R_j are the reward measures defined on alternative configuration E^i and E^j respectively, we define D_{ij} as $R_i - R_j$, where $0 \leq i < j < N$. This enables the SSASC to incorporate the cross-correlation due to the use of a common random number generator in the simulation algorithm. Suppose that the choice of the best configuration is based on the largest reward R_i, i.e, E^i is the best configuration iff $R_i \geq R_j$ for all j, $j \neq i$. Then, the choice is equivalent to choosing the largest D_{ij} and picking the configuration E^j, where j is the column subscript. If the choice of the best configuration is based on the smallest R_j, then it is equivalent to choosing the smallest D_{ij} and picking the configuration E^j, where j is the column subscript. With the reward redefinition process, the SSASC algorithm terminates quicker based on two-stage algorithm when compared to traditional terminating conditions. That is due to the fewer number of replications or batches that are necessary to obtain the same confidence-level interval to choose the best design configuration.

V. Implementation of SSASC in Möbius

The Möbius tool was built on the observations that no formalism is best for building and solving models, that no single solution method is appropriate for solving all models, and that new formalisms and solution techniques are often hindered by the need to build a complete tool to handle them. Möbius addressed these issues by providing a broad framework in which new modeling formalisms and model solution methods can be easily integrated. In Möbius, a *model* is a collection of *state variables*, *events*, and *reward variables* expressed in some formalism. Briefly, *state variables* hold the state information of the model. *Events* change the state of the model over time. *Reward variables* are quantitative measures of interest defined by the Möbius user to evaluate his or her models. In the next section, we present the details on how SSASC implements the state variable representation, event management, and reward measure computation as the algorithms were integrated into Möbius framework.

A. Integration of SSASC into Möbius

The SSASC algorithm, described in Section IV, is implemented as a C++ module extending the simulation features in the Möbius tool [5]. The implementation first separates the state of the model and the clock event generation mechanism in the Möbius simulator. The state of the model for each alternative configuration is replicated and represented using an ESMS data structure. The event generations and management module for all the alternative configurations are merged to obtain a single set of events, while maintaining the information on parameter values of distributions used for each event in the individual configurations. During the running of the simulation, when any event is fired from the event list management module, the event is checked to determine whether it is an actual state transition or a pseudo state transition. The state of the configuration is updated based on the outcome of the classification of the event. To further optimize the algorithm, the event list manager keeps track of only those configurations

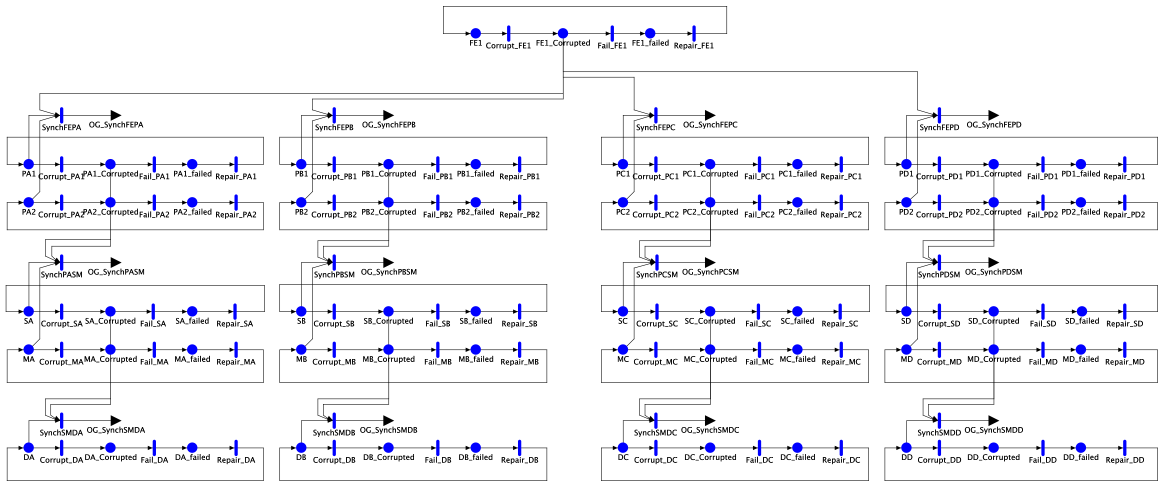

Fig. 3: SAN model of the DISS

that are active for a particular event, thus eliminating the need to test all of the configurations when an event is fired.

The computational savings from the amortization of the cost of event list management in a simultaneous simulation of all the configurations are quite substantial. These savings are illustrated in the next sections using a model of a distributed information service system for transient analysis, and a model of a fault-tolerant computer with repair for steady-state analysis. Moreover, the integration allows us to perform a fair comparison between SSASC and TDES, as both these algorithms were built on the same Möbius framework.

VI. Experimental setup

This section presents two Stochastic Activity Network (SAN) models of dependable systems as case studies to evaluate the SSASC algorithm. The transient availability of a distributed information service system (DISS) adapted from [16] and [22] is used to analyze SSASC's terminating simulation. The model represents different components of an information service and the interaction and propagation of faults across the components. The steady state availability of a fault-tolerant computer system that is adapted from [6] and [27] is evaluated to compare SSASC with the traditional discrete-event simulation (TDES) algorithm. We evaluate all the models by varying parameter values to generate alternative configurations. We show that the SSASC algorithm is efficient and scalable for evaluation of large numbers of alternative configurations.

A. *Evaluation Environment*

The SSASC and TDES simulators are run on an AMD Athlon XP 2700+ processor running at 2.2 GHz with 4Gb RAM in a Unix environment. The implementation was compiled using $g++$ 3.4 with optimization level -03. SSASC is integrated into the Möbius simulator. This integration gives us a fair way to compare the TDES built into Möbius against our simultaneous simulator.

B. *SAN Model of Distributed Information Service System (DISS)*

The distributed information service system has a single front-end module that interacts with four processing units [16], [22]. Each processing unit has two processor units, one unit of memory, a switch, and a back-end database. Each of the units can be in any of the following four states: *Working*, *Corrupted*, *Failed*, and *Repaired*. The units cycle through those states, as shown in a SAN model representation in Figure 3 [21].

TABLE VII: Failure, corruption, and repair rates of sub-models of DISS

Submodel		Corruption Rate	Failure Rate	Repair Rate
Front-end		1/3000 * FFactor	10/1000	10/10
Processor	A1	4/5000 * P1Factor	4/1000	9/10
	B1	3/5000 * P1Factor	4/1000	9/10
	C1	2/5000 * P1Factor	4/1000	9/10
	D1	1/5000 * P1Factor	4/1000	9/10
	A2	4/7000 * P1Factor	4/1000	9/10
	B2	3/7000 * P2Factor	4/1000	9/10
	C2	2/7000 * P2Factor	4/1000	9/10
	D2	1/7000 * P2Factor	4/1000	9/10
Switch	A	4/11000 * SFactor	3/1000	8/10
	B	3/11000 * SFactor	3/1000	8/10
	C	2/11000 * SFactor	3/1000	8/10
	D	1/11000 * SFactor	3/1000	8/10
Memory	A	4/13000 * MFactor	2/1000	7/10
	B	3/13000 * MFactor	2/1000	7/10
	C	2/13000 * MFactor	2/1000	7/10
	D	1/13000 * MFactor	2/1000	7/10
Database	A	4/17000 * DFactor	1/1000	6/10
	B	3/17000 * DFactor	1/1000	6/10
	C	2/17000 * DFactor	1/1000	6/10
	D	1/17000 * DFactor	1/1000	6/10
Note: FFactor, P1Factor, P2Factor, SFactor, MFactor, and DFactor are global variables whose values are varied from 1 to 1000 by a multiplicative factor of 10.				

TABLE VIII: Error propagation rates in the DISS model

Error Propagation Rate					
Error	Rate	Error	Rate	Error	Rate
SynchFEPA	$4.5/3000$	SynchPASM	$4.5/5000$	SynchSMDA	$4.5/7000$
SynchFEPB	$3.5/3000$	SynchPBSM	$3.5/5000$	SynchSMDB	$3.5/7000$
SynchFEPC	$2.5/3000$	SynchPCSM	$2.5/5000$	SynchSMDC	$2.5/7000$
SynchFEPD	$1.5/3000$	SynchPDSM	$1.5/5000$	SynchSMDD	$1.5/7000$

1) Fault Model: The SAN model describes how the fault propagates through the various components as each of them is corrupted. Each component can become corrupted internally. While corrupted, some of the components can corrupt other units. Errors are propagated from a corrupted component onto other working components based on the following rules.

The corrupted front end may propagate an error to either of the two working processors in any of the four processing units. The propagation occurs through the common event between the front end and processors. After the failure has been propagated, the front end might still remain in the corrupted state and could possibly corrupt the processors on the other *Working* processing units. When both of the processors of a processing unit are in the *Corrupted* state, they may corrupt the *Working* switch or the memory unit. Like the front end, the processor might remain in the *Corrupted* state until it fails. Both the memory unit

TABLE IX: Failure and repair rates of FTCSR

Sub-model	Failure or Repair rates		
Repair	10.0	*	RFactor
CPU	0.0052596	*	CFactor
RAM	0.0052596	*	RAMFactor
I/O port	0.0052596	*	IOFactor
Inter	0.0017532	*	IFactor
Error handler	0.0017532	*	ErrFactor

Note: RFactor, CFactor, RAMFactor, IOFactor, IFactor, and ErrFactor are global variables whose values are varied from 1 to 1000 by a multiplicative factor of 10.

and the switch unit in their *Corrupted* state can corrupt the back-end database unit by propagating their error to it. Both the memory unit and switch unit can remain in the *Corrupted* state independently before they move to the *Failed* state.

The distributed information server is said to be available if the front end is able to communicate with the back-end database. The model parameters used in the experiments are presented in Table VII and Table VIII. The availability of the system is measured at the transient time point of 0.1 given that all the components were in *Working* state at time point 0. We use the traditional Möbius simulator to evaluate the configurations to compare the accuracy and scalability with our technique. In order to have an accurate comparison between our technique and standard discrete-event simulation, we ran a simulation of each configuration for one million batches. We show that our approach evaluates the measures of interest accurately for all configurations of the model and is scalable to the number of configurations.

C. SAN Model of Fault-tolerant Computer System with Repair (FTCSR)

The fault-tolerant computer system is a multiprocessor computer with redundant modules that provide high availability as shown in Figure 4 and Figure 5 [6], [27]. The computer is composed of 3 memory modules, of which one is a spare unit; 3 CPU units, of which one is a spare unit; 2 I/O ports, of which one is a spare unit; and 2 non-redundant error-handling chips. In addition to those modules, the computer has a repair module that detects module failures and repairs them while the system is up and running.

Each of the memory modules consists of 41 RAM chips and 2 interface chips. Each CPU module has 3 processor chips, one of which is a spare. Each I/O port has 2 chips, one of which is a spare. The computer system is said to be available if at least 2 memory modules, at least 2 CPU units, at least 1 I/O port, and both the error-handling chips are functioning. A memory module is said to be available if at least 39 of its 41 RAM chips and 2 of its interface chips are functioning. In addition, the fault-tolerant computer system has a failure detection/repair module that detects failed chips and modules and automatically replaces them. We assume that this module is a black box and suffers no failures. Since our goal is to obtain the availability of the computer system, we are only interested in the relative failure rates and repair rates of the components. Table IX provides the relative failure rate for the computer system for each component.

1) Fault/Repair Model: Each module (CPU, RAM, I/O port, Inter, and Error handler) can become corrupted internally. In this SAN model, failures do not propagate from failed modules to working modules.

VII. Evaluation and Analysis of SSASC

In this section, we present the evaluation and analysis of SSASC against TDES as a simulation framework. In Section VII-A, we first verify the correctness of SSASC and its implementation. In Sections VII-B and VII-C, we scale the number of simultaneous simulation configurations to show how we achieve more than 1 order of magnitude in speed-up compared to TDES. In the following sections, we analyze the impact of the host environment, (such as compiler optimization, processor's L1 cache, and

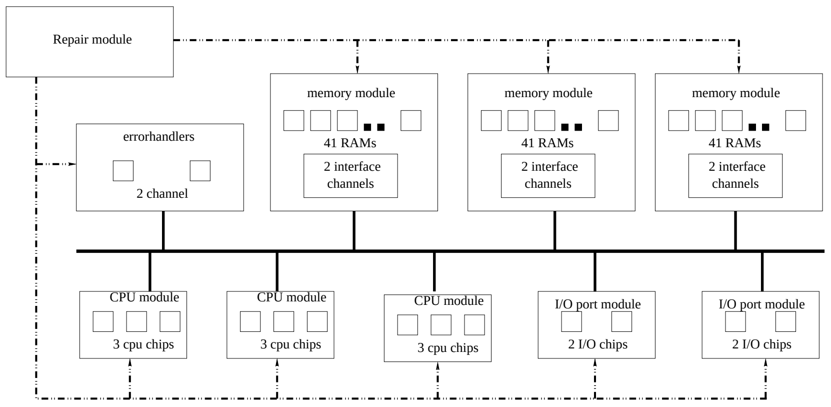

Fig. 4: Architecture of the FTCSR

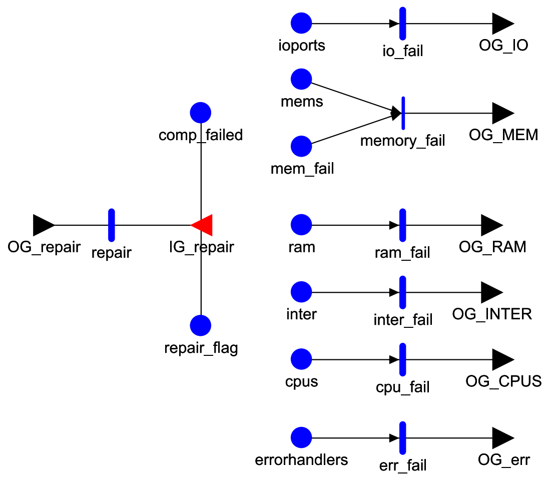

Fig. 5: SAN model of the FTCSR

TABLE X: Comparison of correctness/accuracy of SSASC and TDES using DISS

Component failure rate		TDES			SSASC		
P1Factor	P2Factor	Availability $\times 10^{-02}$	SD $\times 10^{-05}$	Time (seconds)	Availability $\times 10^{-02}$	SD $\times 10^{-05}$	Time (seconds)
5	7	8.464	5.600	10.40	8.461	5.126	
5	70	8.999	4.716	10.40	8.995	4.725	
5	700	9.055	4.601	10.40	9.050	4.327	
5	7000	9.056	4.557	10.40	9.060	4.589	
50	7	9.227	4.214	10.61	9.229	3.686	
50	70	9.829	2.068	10.21	9.831	2.438	
50	700	9.892	1.656	10.21	9.893	1.965	23.90
50	7000	9.899	1.597	10.21	9.899	1.702	
500	7	9.306	4.017	10.40	9.309	2.969	
500	70	9.917	1.447	10.21	9.919	1.328	
500	700	9.982	6.820	10.40	9.982	5.320	
500	7000	9.988	5.588	10.21	9.988	5.306	
5000	7	9.316	3.989	10.21	9.318	2.795	
5000	70	9.926	4.601	10.21	9.927	1.189	
5000	700	9.991	4.922	10.21	9.991	4.777	
5000	7000	9.997	2.823	10.21	9.997	2.115	
Total Time				164.90			23.90

memory overhead) and the impact of individual subcomponents (such as pseudo-transitions, two-stage selection, and ESMS) on the performance of SSASC.

A. *Correctness and Efficiency of SSASC Algorithm Using DISS*

To illustrate the correctness and efficiency of SSASC from an implementation and practical perspective, we compare the simulation results obtained using SSASC with TDES. We varied the individual corruption rate of processor 1 and processor 2 by a multiplicative factor of 10 to obtain 16 alternative configurations of the DISS model. The corruption rate of other components, such as the memory, the switch, the front end, and the database, were fixed. Table X shows the expected instant-of-time availability measures for these configurations. The table compares the traditional serial simulation to the SSASC algorithm. The results were obtained at a 95% confidence level. As one can see from the table, the availability measures obtained from our technique and the traditional method fall in each other's confidence intervals. However, the key finding is the difference in total simulation time. The TDES takes 164.9 seconds, whereas SSASC

TABLE XI: Scalability experiments of SSASC and TDES algorithm: Evaluating DISS using terminating simulation

Number of alternative configurations	Simulation time (seconds)			Speed-up	
	TDES	SSASC	SSASC with ESMS	SSASC	SSASC with ESMS
1	10.21	10.59	11.16	0.96	0.91
4	41.40	13.17	11.13	3.14	3.72
16	164.90	23.90	12.71	6.90	12.97
64	665.60	68.46	19.17	9.72	34.72
256	$2,648.00$	165.91	47.31	15.96	55.97
1024	$10,559.00$	$1,617.00$	183.10	6.53	57.67
4096	$41,978.00$	$8,088.00$	$1,688.90$	5.19	24.86

completes the same simulation in 23.9 seconds. In the next section, we will compare TDES and SSASC to illustrate the speed-up obtained due to SSASC.

B. *Scalability Evaluation of SSASC: Evaluating DISS using Terminating Simulation*

In this set of experiments, the corruption rates of the memory (MFactor), switch (SFactor), front-end (FFactor), proc1 (P1Factor), proc2 (P2Factor), and database (DFactor) were varied by a multiplicative factor of 10 from values 1 to 1000 (refer to Table VI-B for parameter values).

1) Time/Speed-up Characteristics: Table XI shows the speed-up obtained by running all the alternative configurations to obtain the DISS availability. SSASC achieved an average fivefold speed-up compared to TDES, as we simultaneously simulate 4096 configurations. The integration of ESMS into SSASC provided an additional fivefold speed-up compared to TDES for the same 4096 configurations.

Note that in most cases, the speed-up can be as much as an order of magnitude, but it begins to decrease once the number of alternative configurations hits 1024. The decrease in speed-up can be attributed to three factors. First, one should note that the relative length of the trajectory that is required to compute the instant-of-time availability is short (0–0.1 time units) in the system we have described. Thus, some of the computation time is spent in initializing the simulation of batches rather than in executing events. Since a million replications were run, the cost of initializing the simulation was the largest overhead for this particular example. We measured this initialization overhead to be around 10%–30%. Second, as the number of configurations is increased, the dissimilarity among configurations increases. That dissimilarity has a direct impact on speed-up. Third, we noted that when the number of configurations increases beyond a certain threshold, we lose the the advantage obtained by the locality of reference for memory access by the processor to update the state of the model or to compute the reward measures. Due to the use of large arrays to represent the state of the system for each configuration, the overhead of updating the state of the system and computing the reward measures decreased the speed-up of the simulation. Thus, the speed-up becomes comparatively moderate for the DISS model when the number of alternative configurations is greater than 1024.

C. *Scalability Evaluation of SSASC: Evaluating FTCSR using Steady-State Simulation*

To complete the comparative study of the speed-up obtained by using SSASC instead of TDES, we evaluated the FTCSR model using steady-state simulators using both approaches. The parameter values described in Table IX were varied by a multiplicative factor of 10 from values of 1 to 1000 to obtain 4096 alternative configurations. The simulation model was run for 10000 batches. Table XII presents the speed-up obtained from SSASC with ESMS against TDES. We documented a further speed-up of 1-2 times when a 2-stage selection process was used to evaluate all the alternative configurations. We omit the results, as they add no additional value to the results shown in Table XV.

TABLE XII: Scalability experiments of SSASC and TDES algorithm: Evaluating FTCSR using steady-state simulation

Number of alternative configurations	Simulation time (seconds)		Speed-up
	TDES	SSASC with ESMS	SSASC with ESMS
1	0.31	0.63	0.50
4	103.13	51.20	2.01
16	542.68	130.54	4.15
64	$2,771.00$	453.06	6.12
256	$16,327.00$	$1,776.00$	9.14
1024	$88,460.00$	$9,125.00$	9.69
4096	$403,310.00$	$90,320.00$	4.46

TABLE XIII: Comparison of computational overheads to evaluate reward measure defined on DISS: Row-access versus column-access of state variables representing sub-components in DISS

Number of alternative configurations	Row-access	Column-access
1	1.0899	1.0
4	0.7857	1.0
16	0.8571	1.0
64	0.6896	1.0
256	0.7272	1.0
1024	0.8088	1.0
4096	0.5622	1.0

Note: The overhead is normalized with respect to column-access.

D. Impact of Compiler Optimization and Processor Cache on Speed-up

We performed a controlled experiment to evaluate the impact of compiler optimization and processor cache on the speed-up results presented here. The reward measure of the DISS model states that DISS is only available if all of its subcomponents are in the *working* state. The DISS model has 21 subcomponents, as shown in Figure 3. Therefore, the function that computes the reward measure of DISS in TDES only has to check the status of 21 memory locations to determine the availability of DISS. This operation is very efficient in TDES simulation. However, when SSASC with N configurations represents the state variables using an array, the allocated memory block (M) uses $21 \times N$ memory locations. In order to compute the availability of DISS for all alternative configurations, the evaluation can proceed in either of two ways. In the first approach, the SSASC implementation evaluates the reward measure for each alternative configuration before proceeding to the next alternative configuration. This is equivalent to performing column-access on M, and we currently implement this approach in SSASC. In the second approach, the implementation evaluates the reward measures for all alternative configurations by accessing the state of subcomponents in a consecutive order. This is equivalent to performing row access on M.

Our first hypothesis is that the current implementation of reward computation in SSASC (the column-access approach) has no impact on the overall speed-up of SSASC, as the number of configurations is increased because of one or more of the following implementation factors: (a) the memory layout of the program and/or compiler optimizations, and (b) the processor cache architecture. In order to test our hypothesis, we implemented both approaches. We instrumented the SSASC reward computation function to collect timing information. The results of our controlled experiments, illustrated in Table XIII, allow

TABLE XIV: Worst-case memory overhead of SSASC for the DISS model

SAN component	Factor	Total count	Worst-case memory footprint
Places	2	63	$126N$
Activities[1]	2	67	$134N$
Global variables[1]	4	6	$24N$
IRL	1	6	$6N$
Total			$290N$

us to reject the hypothesis. The implication of this controlled experiment is that it is possible to improve the speed-up of SSASC further using smart compiler optimization techniques that are tailored to improve the efficiency of simultaneous simulation.

E. Memory Overhead Characteristics

Table XIV tabulates the worst-case memory overhead of SSASC compared to TDES for the DISS model, where N is the number of alternative configurations. Suppose N = 4096; then the memory overhead would be about 4.5 megabytes. From experimental evaluation, we see that the memory footprint of TDES averages at 8MB. The memory footprint for SSASC averages at 8MB for 1 configuration and 16MB for 4096 configurations. We can conclude that the memory requirement grows linearly as the number of configurations is increased. However, the speed-up achieved is far more significant, which improves the overall utility of simultaneous simulation.

F. Pseudo Transitions

With regard to the issue of pseudo transitions, one should note that unlike the SCMS, SSASC insures that at any given point in time, at least one configuration of the discrete-event model will be performing useful computation that leads to progress in simulation. It is always possible to have a family of models in which one or more alternative configurations might have events that are fired at very different time scales (rates), which could potentially cause other configurations to have significant numbers of pseudo event transitions. However, SSASC guarantees useful simulation progress in at least one of those configurations.

G. Two-stage Selection of Best Alternative Configuration

In this subsection, we show how SSASC can be augmented with smart statistical techniques, such as R&S or the MCB procedures described in [13]. Here, we illustrate how a two-stage selection approach for choosing the best alternative configurations can further speed-up the SSASC algorithm.

1) Procedure for performing two-stage selection: Refer to Appendix A for details on how to set up two-stage selection process to choose the best alternative configuration. Here, for this experiment, we set $n_0 = 10000$. We estimate the variance for the traditional approach, i.e., $max_{i,j}^{N}\left[S_{ij}^{2}(n_{f_{trad}})\right]$. Using that, we compute the required $n_{f_{SSASC}}$ such that $max_{i,j}^{N}\left(S_{ij}^{2}(n_{f_{SSASC}})\right) < max_{i,j}^{N}\left(S_{ij}^{2}(n_{f_{trad}})\right)$.

2) Analysis: Table XV shows the speed-up for the simulation time of traditional simulation and SSASC. Note that the speed-up of SSASC over TDES for the DISS model is 1.5–1.7 times greater than the speed-up shown in Table XI. That is attributable to the use of common random numbers in SSASC due to its common adaptive clock. Common random numbers reduce the variance of computed reward measures [17]. Therefore, SSASC has to execute fewer batches or replications to achieve the same confidence interval as TDES.

Note that we distinguish SSASC and SSASC with ESMS only in the above experiments, to compare the individual contribution in speed-up provided by the common adaptive clock and the ESMS data structure.

[1]Activity firing rates and global variables are floating point numbers.

TABLE XV: Scalability experiments of SSASC and TDES algorithm: Evaluating DISS using terminating simulation with 2-stage selection of best alternative configuration

Number of alternative configurations	Simulation time (seconds)			Speed-up	
	TDES	SSASC	SSASC with ESMS	SSASC	SSASC with ESMS
1	10.21	10.59	11.16	0.96	0.91
4	41.40	12.17	11.10	3.40	3.72
16	164.90	18.90	12.71	8.72	12.98
64	665.60	46.40	18.10	14.34	36.74
256	$2,648.00$	90.10	47.31	27.8	95.27
1024	$10,559.00$	$1,010.60$	114.40	10.44	92.31
4096	$41,978.00$	$4,757.00$	993.00	8.82	42.27

We don't make that distinction in the remaining experiments. All the remaining experimental evaluations have both the common adaptive clock and the ESMS data structure enabled in the SSASC algorithm.

H. *Comparative Evaluation of Cost of Event-Generation and State Update: SSASC versus TDES*

As discussed earlier, SSASC achieves speed-up because of two components in its simulation algorithm: (a) the common adaptive clock, and (b) ESMS. Table XVI illustrates the comparison of SSASC and TDES with respect to each of these components for each of the case-study models. 1000000 replications of alternative configurations of DISS were executed for terminating simulation. 10000 batches of alternative configurations were executed for steady-state simulation. The table reiterates the fact that combination of alternative configurations into one large simulation model has significant advantages. Furthermore, use of an appropriate data structure to manipulate state update provides further improvement in speedup. In the next section, we will analyze the impact of ESMS data structure on simulation, and discuss how one can make optimal use of ESMS.

I. *Analysis of ESMS*

As with any data structure, the efficiency of the execution of ESMS depends upon the data access and update patterns from the simulation algorithm. In Section IV-B, we looked at some of the state update characteristics and built an ESMS data structure to be optimized for those operations. We will now look into some strategies that would make the best use of the ESMS data structure to improve the SSASC algorithm.

Any operation that causes two or more state updates to directly interact reduces the simulation efficiency. Consider the M/M/2/B queuing system from Figure 1. Suppose that there is an event, called *transfer*, that transfers customers from the slow server to the fast server. Whenever transfer is fired, in SSASC without ESMS, it would take exactly N operations to access and/or update the state of the simulation model, where N is the number of alternative configurations. Due to the use of IRL, SSASC with ESMS would add an additional $m * N$ operations for each state update operation, where m is the number of interacting state variables (here, $m = 2$).

In general, the order in which the IRL is built for each state variable in the model has some impact on the speed-up. Since faster-rate events are fired more often, we achieve better speed-up if the IRL list is built based on the order of the dominant event rate that affects each state variable. In situations in which the state variable of a model is affected by more than one event, it is better to build the IRL based on the fastest event affecting the state variable.

In the discussion about state update characteristics in Section IV-B, we noted that minimizing the size of the linear list improves efficiency. The reason is that one could pre-compute the sorted order of firing of a particular activity for all the alternative configurations. However, note that the state-dependent activity

TABLE XVI: Comparison of SSASC against TDES using total number of events generated and state updates for evaluating alternative configurations

Number of alternative configurations	DISS			
	Generated events		State updates	
	TDES	SSASC	TDES	SSASC
1	5.6109×10^{05}	5.7037×10^{05}	2.4936×10^{07}	3.1508×10^{07}
4	2.1049×10^{06}	5.9200×10^{05}	9.9590×10^{07}	8.2953×10^{07}
16	5.6845×10^{06}	6.6062×10^{05}	3.8059×10^{08}	9.9101×10^{07}
64	1.4833×10^{07}	9.8219×10^{05}	1.4607×10^{09}	5.0861×10^{08}
256	4.0003×10^{07}	1.3229×10^{06}	5.6717×10^{09}	2.3451×10^{09}
1024	1.3851×10^{08}	1.3311×10^{07}	2.2492×10^{10}	6.33981×10^{09}
4096	5.4912×10^{08}	6.5311×10^{07}	8.9870×10^{10}	3.33284×10^{10}

Number of alternative configurations	FTPCS			
	Generated events		State updates	
	TDES	SSASC	TDES	SSASC
1	2.4765×10^{05}	2.4657×10^{05}	1.4340×10^{06}	2.7320×10^{06}
4	6.7939×10^{07}	2.6983×10^{07}	5.5780×10^{08}	6.8780×10^{08}
16	3.5932×10^{08}	9.4765×10^{07}	2.8662×10^{09}	1.7386×10^{09}
64	1.8717×10^{09}	3.7657×10^{08}	1.3593×10^{10}	8.2765×10^{09}
256	1.1186×10^{10}	2.3456×10^{09}	8.1333×10^{10}	5.8345×10^{10}
1024	6.3356×10^{10}	4.4512×10^{09}	4.1348×10^{11}	1.5434×10^{11}
4096	2.8790×10^{11}	6.2545×10^{10}	1.8948×10^{12}	1.0645×10^{11}

does not guarantee this property of sorted order of firing. Therefore, it might be efficient to avoid using IRL lists and stick to the linear array representation when an event's rate depends on the state of the simulation model.

Finally, it is possible to significantly improve efficiency of the simulation by viewing the simulation programming paradigm as N independent alternative simulations that depend on each other for computational efficiency, rather than taking the traditional approach of viewing it as a simulation of N independent replications of the model with different parameter values. In simplistic terms, the simulation tool user should view the process of developing models in SSASC as similar to vector, set, or matrix manipulation.

VIII. Conclusions

In this paper, we discussed a new approach to simultaneous simulation of alternative configurations of dependability models through a combination of adaptive uniformization and the SCMS technique. We showed that a significant speed-up can be obtained by this new approach due to the amortization of the cost of event set management, and due to correlation among the trajectories for most of the alternative configurations. We expect that our technique will open up new research issues and ideas in optimization of simulations, sensitivity analysis, and parameter optimization of systems.

One of the goals of this work is to facilitate the use of simulation as an objective and/or constraint function in optimization of stochastic systems. Stochastic optimizations are often nonlinear, and it is not possible to quantify them analytically. That eliminates the possibility of exact calculations of local gradients, upon which traditional optimization solvers rely. Our approach provides a way to explore a large number of parameter values that could potentially allow us to use alternative approaches, such as compass search, direct search, or other unconstrained optimization more efficiently [15]. Furthermore, as the configurations are simulated, our approach provides a seamless framework that enables us to prune out designs that would not meet the required objectives. The pruning criteria can be defined to eliminate uninteresting designs to improve the efficiency of the simulations.

SSASC also introduces an incentive to look at a new programming paradigm for describing simulation models. The current programmer's view of a simulation execution sees it as a single model and its behavior. The speed-up of this approach is greatly enhanced if the programmer understands programmatic dependencies and interactions between alternative configurations, even though they're stochastically independent. Development of a programming paradigm to maximize the utility of the speed-up achieved by the algorithm would be an interesting research area.

The utility of simulation of different models and statistical analysis of the outputs lies in comparing the alternatives before the actual implementation or deployment of the system. Even though there is a coupling between the state update and the clock-event generation mechanism, the structural and behaviorial similarity that configurations display provide an opportunity to reduce the computational cost of updating and maintaining the enabled event set. That allows dynamic comparison of the configurations at the run time of the simulations, which would provided a fast, efficient way to evaluate a large number of alternative design choices.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Number CNS-0406351 and a generous gift from HP Labs. We would also like to thank Jenny Applequist for her editorial comments.

REFERENCES

[1] T. Altiok, "Approximate analysis of queues in series with phase-type service times and blocking," *Operations Research*, vol. 37, no. 4, pp. 601–610, 1989. [Online]. Available: http://www.jstor.org/stable/171260

[2] ——, "On the phase-type approximations of general distributions," *IIE Transactions*, vol. 17, pp. 110–116, June, 1985.

[3] C.-H. Chen, "A hybrid approach of the standard clock method and event scheduling approach for general discrete event simulation," in *WSC '95: Proceedings of the 27th Winter Simulation Conference*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 786–790.

[4] C.-H. Chen and Y.-C. Ho, "An approximation approach of the standard-clock method for general discrete-event simulation," *Control Systems Technology*, vol. 3, no. 4, pp. 309–318, 1995.

[5] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster, "The Möbius modeling tool," in *Proceedings of the 9th International Workshop on Petri Nets and Performance Models*, September 11–14, 2001, pp. 241–250.

[6] J. D. Diener, "Empirical Comparison of Uniformization Methods for Continuous-time Markov Chains." Master's thesis, University of Arizona, 1994.

[7] E. J. Dudewicz and S. R. Dalal, "Allocation of observations in ranking and selection with unequal variances." *Sankhya: The Indian Journal of Statistics*, vol. 37, pp. 28–78, 1975.

[8] S. Gaonkar, T. Courtney, and W. H. Sanders, "Efficient state management to speed up simultaneous simulation of alternate system configurations," in *Proceedings of the International Mediterranean Modelling Multiconference*, Barcelona, Spain, 2006, pp. 31–36.

[9] S. Gaonkar and W. H. Sanders, "Simultaneous Simulation of Alternative System Configurations," in *Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing*, Changsha, Hunan, China, 2005, pp. 41–48.

[10] P. W. Glynn, "On the role of generalized semi-Markov processes in simulation output analysis," in *WSC '83: Proceedings of the 15th Winter Simulation Conference*, 1983, pp. 39–44.

[11] P. Heidelberger and D. M. Nicol, "Conservative parallel simulation of CTMC using uniformization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 8, pp. 906–921, 1993.

[12] M. Hybinette and R. Fujimoto, "Cloning: A novel method for interactive parallel simulation," in *WSC '97: Proceedings of the 29th Winter Conference on Simulation*. ACM Press, 1997, pp. 444–451.

[13] S. H. Jacobson and L. W. Schruben, "Techniques for simulation response optimization," *Operations Research Letters*, vol. 8, no. 1, pp. 1–9, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/B6V8M-48MPRV7-2/2/117ae1af164ea61dc96b78d9578915eb

[14] K. Kawanishi, "QBD approximations of a call center queueing model with general patience distribution," *Queues in Practice, Computers and Operations Research*, vol. 35, pp. 2463–2481, August, 2008.

[15] T. G. Kolda, R. M. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods," *Society for Industrial and Applied Mathematics (SIAM) Review*, vol. 45, no. 3, pp. 385–482, July 2003.

[16] V. V. Lam, P. Buchholz, and W. H. Sanders, "A structured path-based approach for computing transient rewards of large ctmcs," *In the Proceedings of First International Conference on the Quantitative Evaluation of Systems (QEST)*, pp. 136–145, Sept. 2004.

[17] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*. McGraw Hill, 2000.

[18] P. A. W. Lewis and G. S. Shedler, "Simulation of nonhomogeneous Poisson processes by thinning," *Naval Research Logistics Quarterly*, vol. 26, no. 3, pp. 403–413, 1979.

[19] S.-Q. Li and J. Mark, "Performance of voice/data integration on a TDM system," *IEEE Transactions on Communications*, vol. 33, no. 12, pp. 1265–1273, Dec 1985.

[20] Y. Low, C. Lim, W. Cai, S. Huang, W. Hsu, S. Jain, and S. Turner, "Survey of languages and runtime libraries for parallel discrete-event simulation," *Simulation*, vol. 72, pp. 170–186, Mar. 1999.
[21] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proceedings of the International Workshop on Timed Petri Nets*, July 1985, pp. 106–115.
[22] R. R. Muntz and J. Lui, "Computing bounds on steady-state availability of repairable computer systems," *Journal of the ACM*, vol. 41, no. 4, pp. 676–707, 1994.
[23] D. M. Nicol and P. Heidelberger, "Optimistic parallel simulation of continuous time Markov chains using uniformization," *J. Parallel Distrib. Comput.*, vol. 18, no. 4, pp. 395–410, 1993.
[24] ——, "Parallel simulation of Markovian queueing networks using adaptive uniformization," *SIGMETRICS*, vol. 21, no. 1, pp. 135–145, 1993.
[25] V. Nicola, P. Heidelberger, and P. Shahabuddin, "Uniformization and exponential transformation: Techniques for fast simulation of highly dependable non-markovian systems," *Proc. of 22nd International Symposium on Fault-Tolerant Computing*, pp. 130–139, Jul 1992.
[26] S. Prasad and Z. Cao, "Parallel distributed simulation and modeling methods: Syncsim: A synchronous simple optimistic simulation technique based on a global parallel heap event queue," in *Proceedings of the 35th Winter Conference on Simulation*, 2003, pp. 872–880.
[27] W. H. Sanders and L. M. Malhis, "Dependability evaluation using composed SAN-Based reward models," *Journal of Parallel and Distributed Computing*, vol. 15, no. 3, pp. 238–254, 1992.
[28] J. G. Shanthikumar and R. G. Sargent, "A unifying view of hybrid simulation/analytic models and modeling," *Operations Research*, vol. 31, no. 6, pp. 1030–1052, 1983. [Online]. Available: http://www.jstor.org/stable/170837
[29] E. Smeitink and R. Dekker, "A simple approximation to the renewal function [reliability theory]," *IEEE Transactions on Reliability*, vol. 39, no. 1, pp. 71–75, Apr 1990.
[30] P. Vakili, "Massively parallel and distributed simulation of a class of discrete event systems: A different perspective," *ACM Trans. Model. Comput. Simul.*, vol. 2, no. 3, pp. 214–238, 1992.
[31] ——, "Massively parallel and distributed simulation of a class of discrete event systems: a different perspective," *ACM Trans. Model. Comput. Simul.*, vol. 2, no. 3, pp. 214–238, 1992.
[32] A. P. A. Van Moorsel and W. H. Sanders, "Transient solution of markov models by combining adaptive and standard uniformization," *IEEE Transactions on Reliability*, vol. 46, no. 3, pp. 430–440, Sep 1997.

Appendix A
Two-stage selection of best alternative configuration

Detailed description of two-stage selection can be found in [17]. Let R_i be the reward of interest defined of alternative configuration E^i. Let μ_i be its mean. Let us define D_{ij} as the difference between reward measures for alternative configurations E^i and E^j ($0 \leq i < j < N$), where $D_{ij} = R_i - R_j$. Let μ_{ij} be the mean of the differences.

Due to inherent randomness of the observation, the user cannot expect to pick the best configuration with probability one. Instead, the user can pre-specify the probability of choosing the best alternative configuration (BAC) = p^*. Furthermore, to prevent the method from computing a large number of replications that differentiate two configurations, the user needs to specify *indifference* amount d^*. d^* allows the method to be indifferent to configurations that satisfy the criterion $\mu_i - \mu_j \leq d^*$.

In the first stage, the SSASC executes a fixed number of replications or batches of simulations, n^0, of all N alternative configurations. Let D^k represent the reward measure obtained from the k^{th} replication or batch. The means and variances of the first stage are computed as follows

$$\mu_{ij}(n^0) = \frac{\sum\limits_{k=1}^{k=n^0} D_{ij}^k}{n^0} \tag{1}$$

$$S_{ij}^2(n^0) = \frac{\sum\limits_{k=1}^{k=n^0} \left[D_{ij}^k - \mu_{ij}(n^0)\right]^2}{n^0 - 1} \tag{2}$$

In the second stage, we compute the total number of batches, n_i^f, that are necessary for each alternative configuration i to make sure that the measures computed on these system models are within the specified confidence level intervals. We use n^f which is the maximum of all n_i^f to perform the next set of simulations

$$n^f = max_m^{m=0..\frac{N(N-1)}{2}} \left\{ n^0 + 1, \left(\frac{h^2 * S_{ij}^2(n^0)}{(d^*)^2} \right) \right\} \tag{3}$$

Here, h can be numerically computed, since F and f are CDF and PDF of a *t*-distribution, which can be assumed to be normally distributed,

$$p^* = \int_{-\infty}^{\infty} [F(t+h)]^{\frac{(N-1)N}{2}} f(t)dt \tag{4}$$

Now, the means for reward measures for the remaining batches are defined as

$$\mu_{ij}(n^f - n^0) = \frac{\sum_{k=n^0+1}^{k=n_i^f} D_{ij}^k}{n^f - n^0} \tag{5}$$

where the weights are

$$W1_{ij} = \frac{n^0}{n^f}\left[1 + \sqrt{1 - \frac{n^f}{n^0}\left(1 - \frac{(n^f - n^0)(d^*)^2}{h^2 S_{ij}^2(n^0)}\right)}\right] \tag{6}$$

and $W2_{ij} = 1 - W1_{ij}$ for $0 \leq j \leq i \leq N$. The weighted sample means are

$$\mu_{ij}(n^f) = W1_{ij} * \mu_{ij}(n^f - n^0) + W2_{ij} * \mu_{ij}(n^0) \tag{7}$$