A Connection Formalism for the Solution of Large and Stiff Models [*]

David Daly and William H. Sanders
Department of Electrical and Computer Engineering
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St., Urbana, IL, U.S.A.
`http://www.crhc.uiuc.edu/PERFORM`

Abstract

Realistic computer systems are hard to model using state-based methods because of the large state spaces they require and the likely stiffness of the resulting models (because activities occur at many time scales). One way to address this problem is to decompose a model into submodels, which are solved seperately but exchange results. We call modeling formalisms that support such techniques "connection formalisms." In this paper, we describe a new set of connection formalisms that reduce state-space size and solution time by identifying submodels that are not affected by the rest of a model, and solving them separately. A result from each solved submodel is then used in the solution of the rest of the model. We demonstrate the use of two of these connection formalisms by modeling a real-world file server in the Möbius modeling framework. The connected models were solved one to two orders of magnitude faster than the original model, with one of these decomposition techniques introducing an error of less than 11%.

1 Introduction

In the attempt to model computer systems, tools are often overwhelmed by the complexity of the systems being modeled and by the fact that these systems have significant events at many time scales. The first problem leads to large state spaces when state-based analytical modeling is used, which can cause unacceptably large memory requirements. The second problem can cause the models to become stiff, making the time to solve the model by analytical methods or simulation very long.

A general way of combating these two problems is the use of connection formalisms [1, 7]. Connection is a form of modeling that decouples parts of a model. Using connection formalisms, it is possible to solve parts of a model separately to obtain intermediate results, which are used in the solution of other parts. Results can be passed in an arbitrary fashion, including iterative fashions. By separating a model into submodels, the size of the largest state space needed in the solution of the model can be reduced, and the range of time scales in each model can also be reduced, making each submodel less stiff. In this way, the connected models can be solved in less time than if the complete model were solved monolithically.

In this paper, we develop connection approaches that can be applied to a model that has submodels that affect, but are not affected by, the rest of the model. We develop four abstractions that can be used to make connection models from models exhibiting this property; all four involve solving, by itself, a part of the model that isn't affected by the rest of the model, and passing a result from that solution to the rest of the model. The abstractions that we consider involve passing a continuous-time random process, a discrete-time random process, a random variable, and an average value between the models. Of the abstractions developed, the average-passing and random-variable-passing abstractions are the two we are able to implement at this time. When these abstractions are applied, each submodel should have a smaller state space and fewer time scales than the complete model.

After developing these abstractions, we demonstrate their use by modeling the Network Appliance line of file servers [8]. In doing so, we develop a model of the system and measures on the model, and then apply the average-passing and random-variable-passing abstractions to the model. We solve this model using discrete event simulation both with and without the abstractions, and compare the solution times and results that each produces. The simulation times are one to two orders of magnitude faster for both abstractions, but while the average-passing technique gets good results for some of the measures, the random-variable-passing gets good results for all of the measures defined, with an error below 5% for most of the measures defined, and no higher than 11% for any of the measures in the example.

The rest of this paper is organized as follows. In Sec-

[*] This material is based upon work supported in part by the National Science Foundation under Grant No. 9975019 and by the Motorola Center for High-Availability System Validation at the University of Illinois (under the umbrella of the Motorola Communications Center). Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or of Motorola.

tion 2 we introduce the relevant notation and definitions we will use in the paper. In Section 3 we describe the connection model and the theory behind it. We then describe in Section 4 a case study to demonstrate this connection model, and analyze the results in Section 5. We close with some conclusions of the work in Section 6.

2 Concepts and Notation

We use the concepts of a *model* and *submodel* in this paper. Without loss of generality to other formalisms and frameworks, we base these definitions on the definition of an atomic model given by Deavours [4] for models in the Möbius modeling framework. We present the parts of interest of this definition in this paper. In particular, an *atomic model* $\mathcal{M}$ includes the definitions of S, A, and AF, where S is the set of "state variables," A is the set of "actions," and AF is a function called the "action function." A *state variable* is a variable that represents part of the state of the system, and the state of the system is determined by the values of all of the state variables. An *action* changes the value of state variables, and the *action function* determines exactly how each action behaves (e.g., a delay or a state change function). An atomic model is augmented with a set of "measures" M to form a *solvable model*. A *measure* is a metric defined on the model (concerning its performance or dependability, for example). If it is not clear which model something refers to, we will use a subscript of the model name (e.g., $S_{\mathcal{M}}$ denotes the state variables in $\mathcal{M}$).

2.1 Submodels

Following Deavours's definition of a model, we provide definitions for a "submodel," an "isolated submodel," and a "quotient submodel." A *submodel* $\mathcal{N} \subseteq \mathcal{M}$ is a model with state variables $S_{\mathcal{N}} \subseteq S_{\mathcal{M}}$, actions $A_{\mathcal{N}} \subseteq A_{\mathcal{M}}$, and AF restricted to the actions $A_{\mathcal{N}}$.

An *isolated submodel* is a submodel that does not directly affect any of the measures $M_{\mathcal{M}}$ (none of the measures are defined on any of the submodel's state variables or actions), and is not affected by any of the state variables or actions not in the submodel. This situation is illustrated in Figure 1. The isolated submodel $\mathcal{N}$ has state variables $S_{\mathcal{N}} \subset S_{\mathcal{M}}$ and actions $A_{\mathcal{N}} \subset A_{\mathcal{M}}$. All the rest of the state variables and actions fall in the submodel $\mathcal{L}$. A certain collection of the state variables $S'_{\mathcal{N}} \subseteq S_{\mathcal{N}}$ and actions $A'_{\mathcal{N}} \subseteq A_{\mathcal{N}}$ in $\mathcal{N}$ affect the rest of the model, which is the submodel $\mathcal{L}$. However, none of the state variables or actions in the rest of the model affect the behavior of $\mathcal{N}$. Also, the measures, $M_{\mathcal{M}}$, are only defined on state variables and actions in the submodel $\mathcal{L}$, and not on any of the actions or state variables in $\mathcal{N}$. In general, a model may have more than one isolated submodel, and an isolated submodel may itself contain an isolated submodel.

The submodel $\mathcal{L}$ which consists of everything in $\mathcal{M}$ not in the isolated submodel $\mathcal{N}$, is called the *quotient sub-*

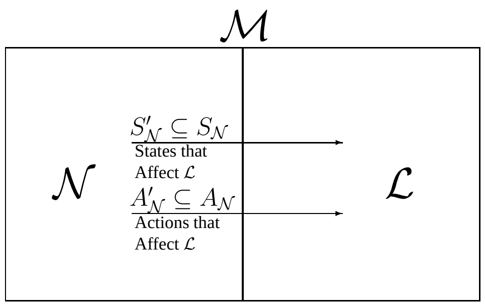
$\mathcal{M}$
$\mathcal{N}$
$S'_{\mathcal{N}} \subseteq S_{\mathcal{N}}$
States that Affect $\mathcal{L}$
$A'_{\mathcal{N}} \subseteq A_{\mathcal{N}}$
Actions that Affect $\mathcal{L}$
$\mathcal{L}$

Figure 1. Isolated Submodel and Quotient Submodel of a Model

model. In general, its set of measures $M_{\mathcal{L}}$ will be equal to $M_{\mathcal{M}}$, which is the set of measures of the complete model $\mathcal{M}$. $\mathcal{L}$ is not a complete model by itself, since its actions may depend on state variables or actions in $\mathcal{N}$. To be a complete model it must be augmented in some way. This augmentation, which consists of passing results to the model, will be provided later in the paper.

2.2 Basic Notation

Some notation is now introduced. We define $S(m)$ to be the solution of measure m as determined by some solution technique, and $ES(m)$ to be the exact solution of the measure. Note that the solutions will be random variables or random processes. We also define a function $\mathit{Diff}(m_1, m_2) \geq 0$. This is a measure of the difference between the solutions of the measures m_1 and m_2, and compares $S(m_1)$ and $S(m_2)$. The Diff function will be the measure of accuracy for any approximate model generated from a complex model. When possible, exact solutions of the measures on the original model and on the approximate model will be used in the evaluation of this function. However, when exact solutions are not available, the available solutions will be used. Possibilities for the Diff function include (but are not limited to) one minus the normalized correlation, a function of different moments, or the normalized difference in means. In general, this function will need to be specified by the modeler, to reflect differences in the important characteristics of the random variables with respect to the meanings of the particular performance or dependability variables used.

3 The Connection Formalism

Using the terminology just introduced, we now develop a set of abstractions that can be used to build a connection model from a model $\mathcal{M}$ that has an isolated submodel $\mathcal{N}$ and a quotient submodel $\mathcal{L}$. These abstractions will exploit the fact that the model has an isolated submodel, and are developed independently of any particular solution techniques and the constraints imposed by particular solution techniques. We focus on the trade-offs in accuracy and the information that needs to be passed between models, and analyze the feasibility of these abstractions with regard to known solution techniques. The derived connected model allows for the solution of the models $\mathcal{N}$ and $\mathcal{L}$, with results being passed between the two models, rather than the direct solution of $\mathcal{M}$. Actually, the models $\mathcal{N}^+$ and $\mathcal{L}^+$ are solved with results passed between them, where $\mathcal{N} \subseteq \mathcal{N}^+$ and $\mathcal{L} \subseteq \mathcal{L}^+$. Both models have some extra features for use in the connection. The model $\mathcal{N}^+$ can be solved by itself, since it does not depend on anything in $\mathcal{L}^+$. It is not clear, however, how to solve the model $\mathcal{L}^+$ by itself.

To capture all the behavior of $\mathcal{L}$ in $\mathcal{L}^+$ as it would appear as part of $\mathcal{M}$, the complete effect of the isolated submodel $\mathcal{N}$ on $\mathcal{L}$ needs to be included in $\mathcal{L}^+$. Conceptually, this could be done by defining measures M' on all state variables $S'_{\mathcal{N}}$ and actions $A'_{\mathcal{N}}$ that affect $\mathcal{L}$. Informally speaking, for each state variable there would be a measure defined on it at each point in time, with value equal to the value of the state variable at that particular time. For each action there would be a measure defined on it for the time of each completion. There would be a measure of the times of the first, second, . . . , n^{th} completions. The exact solutions of these measures would form a random process that captures the complete details of the effect of $\mathcal{N}$ on $\mathcal{L}$. Therefore, if this random process $ES(M')$ was passed from $\mathcal{N}^+$ to $\mathcal{L}^+$ and used in $\mathcal{L}^+$ to replace the effect of $\mathcal{N}$, we would get exactly the same behavior in $\mathcal{L}$ whether it is solved in $\mathcal{L}^+$, or as part of $\mathcal{M}$. This first abstraction is called the *continuous-time random process abstraction*.

Another possibility would be to pass a discrete-time version of this process. This could be done by replacing the measures M' in the previous abstractions with measures M'_T, defined as the time-average value of the state variables and the time-average number of action completions over particular intervals of time, with each interval length being T. If the model $\mathcal{L}$ was insensitive to variations in $ES(M')$ that last less than T time units, the model $\mathcal{L}^+$ would get the same answer for all the measures M defined on the original model $\mathcal{M}$. For example, a model may be affected by an impulse of value a, but only by the increase of $\frac{a}{T}$ in the average value for that interval, and not by the sudden increase and decrease in value. We define the *time sensitivity* T_0 *of* L to be the largest value T such that $\forall m \in M$, $Diff(m_{\mathcal{M}}, m^+_{\mathcal{L}}) = 0$ when $\mathcal{L}$ is solved with the discrete time random process $ES(M'_T)$. The model $\mathcal{L}^+$ will get the same results as the model $\mathcal{M}$ when solved with the random process $ES(M'_T)$ if $T \leq T_0$.

It may be the case that the quotient submodel $\mathcal{L}^+$ will be mostly, but not completely, insensitive to variations of $ES(M')$ over periods of time T. While using this value of T would introduce some error, it might be small enough to warrant solving $\mathcal{L}^+$ with $ES(M'_T)$. We define the ϵ-time sensitivity T_ϵ to be the smallest T such that $Diff(m_{\mathcal{M}}, m_{\mathcal{L}^+}) \leq \epsilon, \forall m \in M$, when $\mathcal{L}^+$ was solved with $ES(M'_T)$. In that case the model $\mathcal{L}^+$ will get results acceptably close to the original model $\mathcal{M}$, when solved with the random process $ES(M'_T)$ when $T \leq T_\epsilon$. This second abstraction (whether using T_0 or T_ϵ) is called the *discrete time abstraction*.

If $T_0 = \infty$, there would be only one interval to solve for, and therefore only one random variable to pass using the discrete time abstraction. If the process were ergodic, then the variance of the one random variable would be zero. In that case, passing the average value would still yield a model that gets the same result as the original model $\mathcal{M}$. Likewise, if $T_\epsilon = \infty$, we could pass the average values and still have a model that gets results that are acceptably close to those of the original model $\mathcal{M}$. We call this third abstraction the *average-passing abstraction*. If the model were not ergodic, then a random variable would need to be passed.

If $T_\epsilon < \infty$ then passing just the average values would lead to a significant loss in accuracy. However, if $ES(M'_T)$ was independent and identically distributed (IID), a simpler form of the random process $ES(M_T)$ could be passed. The behavior of $ES(M'_T)$ could be completely captured by one random variable m' defined on any of the time intervals. This random variable m' could be passed and used to reconstruct the random process in $\mathcal{L}^+$. We would specify the reconstructed process as $P[i] = ES(m'),\ \forall i$, which would be equivalent to $ES(M'_T)$. This fourth abstraction is called the *random-variable-passing abstraction*.

Even if the process was not completely IID, this could be applied if the correlation between intervals was low, and the process was stationary. We define $\bar{T}_0$ to be the smallest value T such that values in different time periods would be uncorrelated, and $\bar{T}_\epsilon$ to be the smallest value T such that the error due to the correlation between distinct time periods would be less than ϵ. The model $\mathcal{L}^+$ with the random variable $ES(m')$ passed to it would get the same result as the model $\mathcal{M}$, if $\bar{T}_0 \leq T \leq T_0$. The model $\mathcal{L}^+$ with the random variable $ES(m')$ passed to it would get results acceptably close to those of the model $\mathcal{M}$, if $\bar{T}_{\epsilon_1} < T < T_{\epsilon_2}$. The relationship between $\epsilon, \epsilon_1, \epsilon_2$ would be dependent upon the definition of the $Diff$ function.

3.1 Practical Implications

Using the set of abstractions developed, we discuss the practical implications and implementations issues for each of these abstractions. The first thing we note is that an arbitrary model can not be solved for the continuous-time random process defined in the previous section. If the type of the random process is known, the model can be solved for the random process's parameters, but the type of the porcess is not normally known. Without knowing the type of the process, one could naively attempt to solve for the complete process by solving for a finite subset of the uncountably infinite collection of measures, but solving for all the joint distributions of the measures would be impractical. Therefore, while useful in developing other abstractions, the continuous-time random process passing abstraction can not be used to solve actual models.

We are also unable to solve an arbitrary model for the discrete-time random process also described in the previous section. While the number of measures is countable, it is still infinite, and a naive attempt to solve for these measures will still fail. Solving the model for a finite subset of the measures (e.g., if we are only interested in solving $\mathcal{M}$ for its measures at some time t) would still be impractical since the model would still need to be solved for all the joint probability distributions of the measures. Therefore, while the discrete-time random process passing abstraction is useful in developing the remaining abstractions, it also can not be used to solve actual models.

A model can be solved for a random variable. This is done by solving the model for the probability distribution of the random variable, which completely describes the random variable. Therefore, the random-variable-passing abstraction can be used. Similarly, a model can solved for the mean of a random variable. Therefore, we also can use the average-passing abstraction. In order for those two abstractions to be applicable, both of them require special properties. However, whenever the average-passing abstraction is applicable, so is the random-variable-passing abstraction, making the random-variable-passing abstraction more general.

4 Case Study

We now develop an example to demonstrate the use of these abstractions and their applicability, accuracy, and performance. Since the last two abstractions (passing a random variable and passing an average) are feasible, we apply both of them to the model developed. The example model needs to exhibit the properties required by the two abstractions (e.g., isolated submodel), and after we describe it, we show how it does indeed exhibit the necessary properties. All the models we develop are implemented in the Möbius modeling tool [3, 2] and solved with the Möbius discrete event simulator [9].

We model a file server from the Network Appliance line of filers. These filers provide NFS and Windows file services [8], and have demonstrated availabilities of up to 99.99% in production use [6]. In addition, they provide such services as online access to multiple backups called “snapshots” and a way to make consistent off-line backups of the system.

The NetApp filers provide this functionality using a novel architecture [5] that is specifically designed to be a file server. When a filer receives a write request, it goes through three steps: it updates its in-memory cache, it logs the request in non-volatile RAM (NVRAM), and it replies. The request isn’t written to disk until up to 10 seconds have passed, at which time all outstanding accesses are written to disk in one consistent step, generally in under one second. The architecture does this by writing all the changed file blocks and updated i-nodes to unused blocks. When this is done, it writes a new root node, which points to this new directory structure.

In that way, the on-disk file image is always consistent, and older consistent images are also accessible on disk, since they aren’t overwritten. This mechanism allows for the creation of consistent backups by copying snapshot images from the disk.

For this architecture, we want to know what fraction of time the system spends updating the disk image of the file system, and the fraction of time spent in the particular stages that compose these updates. We also desire to know how much time each of these updates, and the stages that compose these updates, take to execute, each time they execute. We will develop a model specifically to answer questions of this type.

4.1 Model

The first model developed to answer these questions is the *full model*, and does not initially employ any of the abstractions just described. This model will be used to test the quality of the connected models we develop. The full model contains an arrival process called the *request submodel*, which represents requests arriving and filling up the NVRAM, and a process called the *filer submodel*, which models the update of the disk. Using Möbius, we defined these two submodels and composed them to make the full model.

The request submodel represents the arrival of requests to the file server, and is shown in Figure 2. It has two main states, bursty and slow, represented by a token in either the place Bursty or Slow. The rate of the activity Arrival is exponentially distributed, and its rate is dependent on whether it is in the bursty or slow state, making this submodel a Markov-modulated Poisson process. This model was chosen to represent the bursty nature of requests to a file server. However, the model is just an estimate of the arrival process. Actual request traffic or behavior of the file server users was not used to develop this model; such study would lead to a more accurate model of the arrival process.

For all of the experiments, the low rate was 50 arrivals per second, while the bursty rate was varied between 800 and 900 arrivals per second. The slow state had an average holding time of 4.5 seconds, while the bursty state had an average holding time of 1 second. These numbers were selected so that the average number of arrivals per ten seconds would be at about the level needed to cause a disk update

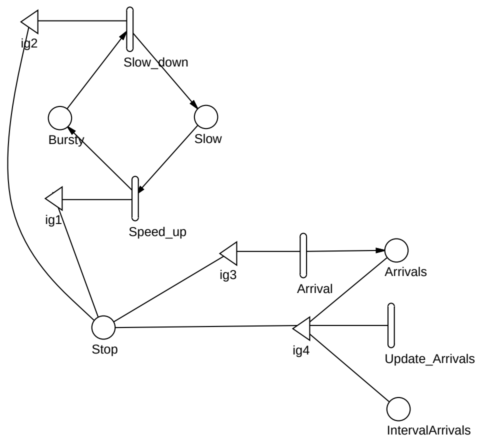

Figure 2. Model of Markov-modulated Arrival Process

to clear the NVRAM. The model should have interesting behavior in this operating region, with a significant number of updates being started by timeouts, and a significant number being started by the filling up of the NVRAM.

The place Stop is used to disable this model. If there is a token in Stop, none of the activities are enabled. In some cases, this provides an improvement in solution time. All submodels composed with the filer submodel will have this feature.

The filer submodel represents the disk update process and is shown in Figure 3. When either the arrivals fill up the NVRAM bank (represented by the place Arrivals), or the timeout occurs (deterministically timed activity Timeout), the arrival queue is processed. When either of these events happens, the contents of Arrivals is copied to Processed_Arrivals, and Arrivals is cleared. As mentioned in the file server description, the filer has five separate phases through which it progresses to update the disk. Activities phase1-5 represent these phases. Phase1, phase2, and phase4 are modeled as exponentially distributed activities to reflect service time variability due to file size, while phase3 and phase5 are modeled as deterministic distributed activities with a constant delay. Phase 4 is the write to disk and is the dominant phase with regard to delay.

The arrival process is an isolated submodel (see Section 2.2.1) of the full model, since it is not affected by any of the state variables or actions in the rest of the model. Therefore, the developed abstractions can be applied to this model. For each of the abstractions, a representation of the arrival process needs to be passed from the isolated submodel (request submodel) to the quotient submodel (filer submodel). The abstractions that represent the arrival process with a random variable and with a mean are used.

However, before applying these abstractions, we define $Diff$ in order to compare the different abstractions.

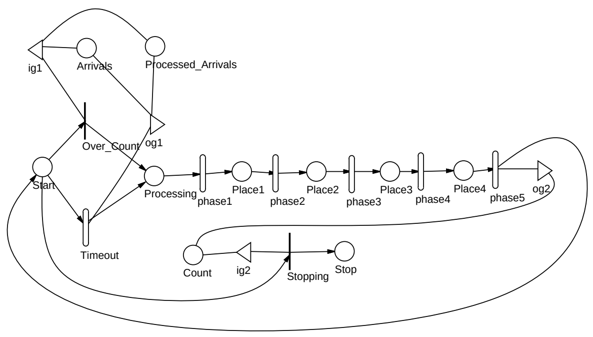

Figure 3. Model of NetApp Filer Creating a New Snapshot

The $Diff$ function is a measure of how accurate the approximate models are compared to the original. In particular, we let $Diff$ be the absolute value of the percent error on the means, as shown here:

$$Diff(m_{\mathcal{M}}, m_{\mathcal{L}}^{+}) = \left| \frac{E[S(m_{\mathcal{M}})] - E[S(m_{\mathcal{L}}^{+})]}{E[S(m_{\mathcal{M}})]} \right| \qquad (1)$$

It will always be expressed as a percent, rather than a decimal.

To apply the average-passing abstraction, it is necessary tp define a measure on the arrivals in the request submodel to determine the average rate of arrivals. For the abstraction to be applicable, the quotient submodel needs to have $T_\epsilon = \infty$ as its time sensitivity to the arrival process.

In order to apply the random-variable-passing abstraction, a period T is needed such that $T < T_\epsilon$, the time sensitivity of the quotient submodel. If $T \leq T_\epsilon < \infty$, and $T > \bar{T}_\epsilon$ (see Section 3), this abstraction should give better results than the average-passing technique. Reviewing the model, we note that the disk updates every 10 seconds, or when the queue fills up. We choose $T = 1$ second for the period. Compared to the 10 seconds of the timeout, and the time it takes the queue to fill up, this second should be less than T_ϵ for any of the fraction-of-time measures or holding-time measures described in the system description. We also believe that $\bar{T}_\epsilon \leq 1$ second, since the holding time of the bursty state is 1 second.

The activity $update\_load$ is used when the request submodel is solved by itself for the results needed for the two abstractions. Every time the interval of time defined by $interval$ has passed, update_load copies all the tokens from Arrivals to IntervalArrivals, making IntervalArrivals represent the number of arrivals in the past interval. Changing the length of the interval effectively changes the period T. A reward variable is defined on the IntervalArrivals place and solved for its distribution of values (for random variable passing) and its mean (for average passing), using the

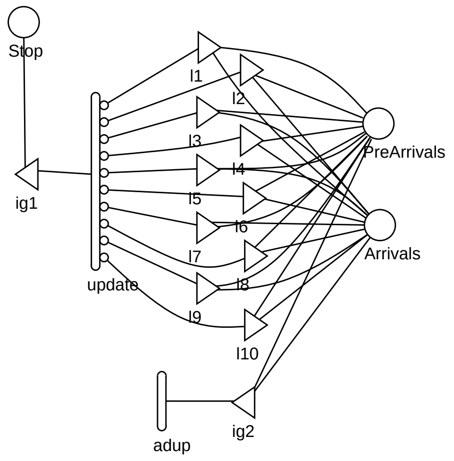

Figure 4. Driver Submodel for Simplified Model

same model solution for both abstractions.

A small model called the *driver submodel* (shown in Figure 4) is built to recreate the random arrival process, as required for the random-variable-passing abstraction. Every time $interval$ time units pass, it picks another value from the distribution of the random variable. Unfortunately, we do not currently have a general way to sample from a distribution in our simulator implementation. Therefore, an activity with cases was used to mimic the distribution (a case is chosen at random when the activity fires). For the experiments, 10 cases were used to represent the 0^{th} to 10^{th}, 10^{th} to 20^{th}, . . . , 90^{th} to 100^{th} percentiles. The cases use the values of the 5^{th}, 15^{th}, . . . , 95^{th} percentiles to represent these probability intervals, which are all equally probable.

The requests are first put in the place PreArrival, by the output gate selected by the action. Each output gate represents one of the probability intervals. After one second the requests are moved to the place Arrivals, which is shared with the filer submodel. A problem arises if the number of requests in this batch overflows the queue. In reality, the system update would occur before the next complete second completed. The action adup is used to represent this early update. If the queue can only accommodate 1/3 of the current batch (tokens in Arrival), then adup copies 1/3 of those requests to Arrivals2 after 1/3 of a second, triggering a system update.

The driver submodel composed in that way with the filer submodel is the *random-variable-passing connection model*. The driver submodel is also used to implement the average-passing connection model. This is done by composing it with the filer submodel and setting one case's probability to 1. When that case is selected, the average number of arrivals are placed in Arrival. The composed model configured in this way is called the *average-passing connection model*.

4.2 Model Measures

These models were developed to determine the fraction of time that the file server spends in certain sets of states in the update process, and the holding times in these states. The sets of states of particular interest are the states in which the system cannot process new requests, the update state, and any state that is possibly less fault-tolerant than the rest.

The first two phases of processing in the system are blocking, which means that many file requests can not be processed in those phases. Therefore, the model will be considered to be in the blocked state if it is in phase1 or phase2. The last phase is the most critical to correctness, as it is the time when the system goes from one consistent disk image to another. An error in this phase could lead to corrupt data; therefore, it is considered to be the "risky" phase. The model is in the update state if it is in any of the update phases. This is of interest because the whole update process is system-intensive, and therefore degrades the response time of any requests that arrive during that period.

The fractions of time spent in those states, and the holding times of those states (except for the risk state, since it has a known constant duration) need to be measured. The fraction-of-time measures are defined as steady-state performance variables in Möbius, and are shown in Table 1, while the duration measures are defined as transient performance variables in Möbius, and are shown in Table 2. The Möbius reward variable formalism uses C++ syntax to define performance variables as functions, which then return values. The Mark() method is called on a place to determine the number of tokens in the place.

Several other steady-state measures are also defined. Arrivals is defined to be the value of the Arrivals place, while Arrivals2 is defined to be the value of the Processed_Arrivals place, and Full is defined to be the value of the place Full. The place Full is set to 1 whenever the filling up of the queue causes a snapshot, and is cleared whenever the timeout causes a snapshot.

The duration variables show the effect that these states have on requests that occur during these states. The shorter the durations, the lower the performance impact on such requests. The steady-state variables, defined on the fraction of time in those states, represent the probability of being in those states. The lower the value of those variables, the lower the probability that a request will be impacted at all by those stages. The three models (full, average-passing, and random-variable-passing) are solved for these measures.

Table 1. Steady-State Performance Variables

Proc Time	if (filer−>Processing−>Mark() + filer−>Place1−>Mark() + filer−>Place2−>Mark() + filer−>Place3−>Mark() + filer−>Place4−>Mark() $>$ 0) return (1); else return (0);
Arrivals	return(filer−>Arrivals−>Mark());
Arrivals2	return(filer−>Processed_Arrivals−>Mark());
Full	return(filer−>full−>Mark());
Blocking	if (filer−>Place1−>Mark() + filer−>Processing−>Mark() $>$ 0) return (1); else return (0);
High Risk	return(filer−>Place4−>Mark());

Table 2. Duration Performance Variables

Wait Time	if (race−>Count−>Mark() == 1 && race−>Processing −>Mark() + race−>Place1−>Mark() + race−>Place2−>Mark() + race−>Place3−>Mark() + race−>Place4−>Mark() $>$ 0) return (1); else return (0);
Blocking	if (race−>Count−>Mark() ==1 && race−>Place1−>Mark() + race−>Processing−>Mark() $>$ 0) return (1); else return (0);

5 Results

The full model, random-variable-passing model, average-passing model, and request model are solved using discrete event simulation for a confidence interval of $\pm 1\%$ with a probability of 95%. Each of the three models was solved for the duration variables and the steady-state variables with three different bursty rates. The times required to solve these models are given in Table 3. Since the average-passing and random-variable-passing models require the solution of the request model, we also present the solution time for these models plus the solution time of the request model in the Average Total and R.V. Total columns. However, the same request model solution was used for results for the average model and random-variable-passing model, for both the duration and steady-state solutions. Each request model solution is thus used at least four times in this example.

Both abstractions were simulated much faster than the full model. The abstractions ran about one order of magnitude faster than the full model, with the request model solution time included, and almost two orders of magnitude faster with the request model solution time excluded. This result makes sense, since in the full model there are activities firing hundreds of times per simulated second, whereas in the average-passing and random-variable-passing models, activities fire no more than once per simulated second.

The results for the steady-state variables for the three models, and the $Diff$ of these results, are presented in Table 4. The random-variable-passing model did well on all the variables, with $Diff$ up to 11%, while the average-passing model only performed well on the Proc_Time and Blocking variables. However, both of those variables are

Table 3. Solution Times in Seconds

Bursty	*Request*	*Avg.*	*Avg.*	*R.V.*	*R.V.*	*Full*
Rate	*Model*	*Pass*	*Total*	*Pass*	*Total*	
800 Dur.	241.74	34.55	276.29	39.29	281.03	2468.08
850 Dur.	268.14	34.33	302.47	39.48	307.62	2536.67
900 Dur.	280.69	34.36	315.05	39.18	319.87	2477.51
800 Steady	241.74	78.98	320.72	80.52	322.26	6658.02
850 Steady	268.14	80.18	348.32	81.21	349.35	6865.78
900 Steady	280.69	80.95	361.64	82.17	362.86	7178.21

dependent on the number of arrivals being processed, and not on the frequency with which the system writes snapshots, making $T_\epsilon = \infty$ for those variables. Thus, it makes perfect sense for the average-passing model to perform well on those variables, and we note that random-variable-passing also works well in those cases, as it should whenever the average-passing works well. For all the other variables, the number of snapshots written is important, and for that reason, those variables show a much better accuracy with random-variable-passing than with average-passing.

The random-variable-passing model, with $Diff$ values from 1 to 3%, performed much better than the average-passing model, with $Diff$ values from 16 to 20% for the duration variables. The results are presented in Table 5. Not only does the random-variable-passing model provide accurate values for mean values, it also accurately measures the distribution of these variables, as seen in Figure 5. The random-variable-passing model gives a very tight fit to the full model distribution, while the average-passing model is much lower. If we change the exponentially distributed actions to deterministically distributed actions by accounting for the variability of request size in the arrival process, then the average-passing model does much worse, while random-variable-passing does about as well as it did here. The average model distribution begins to look much more like a step function. This is clear even if only the most dominant action (phase3) is changed to have a deterministic distribution, as shown in Figure 6.

6 Conclusion

We have described a technique for the solution of large models. By separating one large system model into two parts, we get a connection model that is simpler to solve. We developed the ideas of isolated and quotient submodels, and used those ideas to develop four abstractions that result in connection models. The four abstractions involve passing a continuous-time random process, a discrete-time random process, a random variable, and a mean value between models. Currently only the last two of these are feasible, and we demonstrated them with an extensive case study of a real file server. Both abstractions provided speedups approximately in the range of one to two orders of magnitude, when solved using discrete event simulation. The random-variable-passing abstraction provided accurate re-

Table 4. Steady-State Results and $Diff$ Values, Bursts of 850

Variable	*Full Model value* (m_1)	*RV Passing value* (m_2)	*Avg. Passing value* (m_3)	$Diff$ (m_1, m_2)	$Diff$ (m_1, m_3)
Proc Time	$0.1064 \pm 7\times10^{-4}$	$0.1041 \pm 6\times10^{-4}$	$0.1053 \pm 7\times10^{-4}$	2.1%	1.1%
Arrivals	776 ± 3	786 ± 2	909 ± 2	1.7%	17.2%
Arrivals2	1629 ± 5	1737 ± 4	1983 ± 4	6.7%	21.8%
Full	$0.510 \pm 4\times10^{-3}$	$0.568 \pm 3\times10^{-3}$	$0.865 \pm 3\times10^{-3}$	11.4%	69.8%
Blocking	$0.00731 \pm 5\times10^{-5}$	$0.00721 \pm 7\times10^{-5}$	$0.00726 \pm 4\times10^{-4}$	1.4%	0.7%
High Risk	$1.18\times10^{-4} \pm 4\times10^{-7}$	$1.11\times10^{-4} \pm 3\times10^{-7}$	$9.82\times10^{-5} \pm 2\times10^{-7}$	6.1%	17.0%

Table 5. Duration Results and $Diff$ Values, Bursts of 850

Variable	*Full Model value* (m_1)	*RV Passing value* (m_2)	*Avg. Passing value* (m_3)	$Diff$ (m_1, m_2)	$Diff$ (m_1, m_3)
Wait Time	$0.909 \pm 9\times10^{-3}$	$0.926 \pm 9\times10^{-3}$	$1.063 \pm 1\times10^{-3}$	1.0%	16.1%
Blocking	$0.06234 \pm 5\times10^{-4}$	$0.0646 \pm 5\times10^{-4}$	$0.0751 \pm 6\times10^{-4}$	3.6%	20.4%

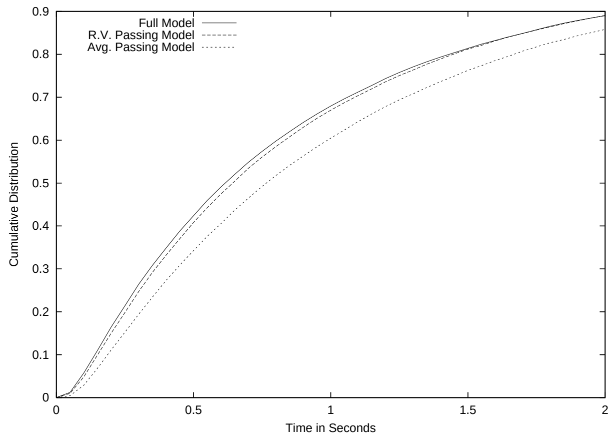

Figure 5. Distribution of Delays for Complete Snapshot for Bursts of 850

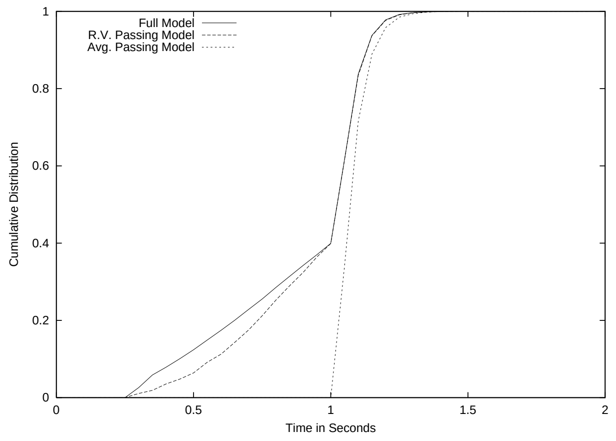

Figure 6. Distribution of Delays for Complete Snapshot for Bursts of 850, Deterministic phase3 Distribution

sults on all the variables, while the average-passing abstraction only provided accurate results on the two variables that exhibited a special property.

References

[1] A. Christensen. Result specification and model connection in the Möbius modeling framework. Master's thesis, University of Illinois, 2000.

[2] G. Clark, T. Courtney, D. Daly, D. D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster. The Möbius tool. In *Submitted to PNPM'01*, 2001.

[3] D. Daly, D. D. Deavours, J. M. Doyle, P. G. Webster, and W. H. Sanders. Möbius: An extensible tool for performance and dependability modeling. In B. R. Haverkort, H. C. Bohnenkamp, and C. U. Smith, editors, *Computer Performance Evaluation: Modelling Techniques and Tools: Proceedings of the 11th International Conference, TOOLS 2000, Schaumburg, IL*, number 1786 in Lecture Notes in Computer Science, pages 332–336. Springer, 2000.

[4] D. D. Deavours and W. H. Sanders. Atomic models in the Möbius framework. In *Submitted to PNPM'01*, 2001.

[5] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. Technical Report TR3002, Network Appliance, 1995.

[6] S. R. Kleiman, S. Schoenthal, A. Rowe, S. H. Rodrigues, and A. Benjamin. Using NUMA interconnects to implement highly available filer server appliances. Technical Report XP1004, Network Appliance.

[7] W. H. Sanders. Integrated frameworks for multi-level and multi-formalism modeling. In *Proceedings of PNPM'99: 8th International Workshop on Petri Nets and Performance Modeling, Zaragoza, Spain*, pages 2–9, September 8-10 1999.

[8] A. Watson and P. Benn. Multiprotocol data access: NFS, CIFS, and HTTP. Technical Report TR3014, Network Appliance, 1999.

[9] A. Williamson. Discrete event simulation in the Möbius modeling framework. Master's thesis, University of Illinois, 1998.