Integrating Reliability Analysis with a Performance Tool

David M. Nicol[*] — College of William and Mary
Daniel L. Palumbo — NASA LaRC
Michael Ulrey — Boeing Defense and Space

Abstract

A large number of commercial simulation tools support performance oriented studies of complex computer and communication systems. Reliability of these systems, when desired, must be obtained by remodeling the system in a different tool. This has obvious drawbacks: (i) substantial extra effort is required to create the reliability model, (ii) through modeling error the reliability model may not reflect precisely the same system as the performance model, (iii) as the performance model evolves one must continuously reevaluate the validity of assumptions made in that model. In this paper we describe an approach—and a tool that implements this approach—for integrating a reliability analysis engine into a production quality simulation based performance modeling tool, and for modeling within such an integrated tool. The integrated tool allows one to use the same modeling formalisms to conduct both performance and reliability studies. We describe how the reliability analysis engine is integrated into the performance tool, describe the extensions made to the performance tool to support the reliability analysis, and consider the tool's performance.

[*]This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480 while the first author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001

1 Introduction

Discrete-event simulation is emerging as the dominant means of designing and evaluating the performance of complex computer and communication systems. A thriving industry of simulation tool and model developers exists to support these developments (e.g., see the recent tools survey [9]). By performance we mean metrics like buffer usage, CPU utilization, and throughput. The simulators evaluate these quantities by directly emulating the behavior of the modeled system, e.g., pushing packets through network models. Many of these systems have stringent reliability requirements: consider the systems that control fly-by-wire aircraft, telephone switching systems, or the FAA flight management systems. However, it is difficult to use a performance-oriented simulator to assess reliability with any accuracy. Failures happen rarely, a direct simulation that mixes performance events and reliability events must run for a long long time (if no tricks are employed). In special cases there are exceptions. Importance sampling[2] is a way of accelerating the occurrence of failures and "fixing" the statistics. However, as powerful as it can sometimes be, importance sampling requires the modeler to be expert in its application; it is not yet appropriate for general use on general models.

Existing reliability modeling tools require a whole different view-point of the modeled system than do performance modeling tools. In practice, someone trained in reliability modeling is called upon to develop a system's reliability model. We believe that because of this, reliability modeling is not used as extensively as it might be if its methods were naturally integrated with performance modeling. Towards this end, since 1989 we have developed experimental reliability modeling tools with the goal of being able to analyze very large models, requiring a minimum degree of involvement by the modeler [4, 5, 3, 10]. Believing that goal to be achieved, we set out to develop a tool that incorporates our analysis techniques, but can be easily integrated into model-development front-ends. This paper describes the results of that effort.

The mathematical details of the approach we describe are for reliability analysis. Availability analysis is also of wide-spread interest, but imposes a much larger storage requirement to save generated states. However, all that is required to support availability analysis in our framework is a different algorithm for managing and generating the underlying state-space. We have accomplished this already with a separate interface solution engine that exploits the collective memory available in workstation networks [1]. That engine already uses the same interface between modeling tool and solution engine that we describe here, in the context of reliability modeling.

Section 2 introduces the concepts we use with a discussion of Markov modeling. Next, Section 3 describes the interface to the general purpose analysis. Section 4 describes the extensions we made to the BONeS (Block Oriented Network Simulator) Designer simulation environment to support integration of reliability analysis. Section 5 discusses types of integrated analysis that are supported by our approach. Section 6 then describes some run-time optimizations that, when applicable, substantially accelerate reliability model solution time. Section 7 details experiments we conducted with the integrated tool, and section 8 presents our conclusions.

2 Markov Modeling and Analysis

In Markov modeling, one describes the modeled system's temporal behavior as a mathematical construct called a continuous time Markov chain (CTMC). At every instant in time the CTMC is in some *state* that describes the essential characteristics of the modeled system. For instance, the state of a computer system that starts with n identical processors and 2 spares might be the vector (g, r), where g is the number of processors currently "good" (i.e., not yet failed), and r is the number of remaining spares. Alternatively one might describe the state as a vector of binary digits, each one indicating the state of a particular computer or spare. *Transitions* occur, moving the CTMC from one state into another. If while in state $(5,1)$ a processor fails and is replaced by the spare, the CTMC modeling the system makes a transition from $(5,1)$ to $(5,0)$. In a general CTMC there may be a number of different transitions possible from any given state. In reliability models, a transition usually corresponds to a component failure or completion of a repair activity. The CTMC model ascribes a *rate* $\lambda_{i,j}$ to the transition between state i and state j (i and j here are integer codes for state vectors). If this transition corresponds to the failure of some component, then $\lambda_{i,j}$ is the failure rate of that component (which is the inverse of its mean-time-to-failure). The random component lifetime defining the i to j transition is exponentially distributed, with mean $1/\lambda_{i,j}$. CTMCs assume that the different activities that define transitions are mathematically independent. As a consequence, the random time the system remains in state i before *any* transition occurs is exponentially distributed with rate $\lambda_i = \sum_{\text{all } j} \lambda_{i,j}$. The different transitions possible from state i can be thought of as competing with each other to achieve the next transition. The probability that a transition to state j succeeds in this competition (i.e., the random completion time of its activity is least among all competitors) is $\lambda_{i,j}/\lambda_i$.

The set of all states the system might possibly enter is called the *state-space*. State-spaces are frequently envisioned as directed graphs, with nodes representing states, and arcs representing transitions. Arc labels describe the corresponding transition rates. The behavior of the modeled system in time is viewed as tracing a path through the state-space graph. As we have already seen, the time the CTMC spends in a given state is a random variable, and the choice of transition out of a state is also random. If we then specify some specific amount of time T and an initial starting state (starting implicitly at time 0), the path a system chooses over time $[0, T]$ is random. For any given state i, there is some probability $p_i(T)$ that the system enters state i before time T. Each state in a reliability-focused CTMC can be classified as representing system failure, or not. We may assume that once the CTMC has entered a failure state, it does not leave. Letting $\mathcal{S}$ denote the set of all failure states, the system unreliability is the probability that the system enters some state in $\mathcal{S}$ within the mission time T. Figure 1 illustrates the state-space associated with the processor-spare system similar to that described above. The failure rate of each processor is λ; the spares are "cold", meaning that they don't fail, but do require a mean time $1/\tau$ to be integrated into the system. We assume the repair time is exponentially distributed. The system fails if there are fewer than two working processors, or if a failure occurs while a repair is under way. The state is **(working,spares,status)**, where **working** is the number of functional processors, **spares** are

$$\begin{tikzcd}
(3,2,a) \arrow[r, "3\lambda"] & (2,2,r) \arrow[r, "2\lambda"] \arrow[d, "\tau"'] & \text{(Failed)} & & \\
& (3,1,a) \arrow[r, "3\lambda"] & (2,1,r) \arrow[r, "2\lambda"] \arrow[d, "\tau"'] & \text{(Failed)} & \\
& & (3,0,a) \arrow[r, "3\lambda"] & (2,0,a) \arrow[r, "2\lambda"] & \text{(Failed)}
\end{tikzcd}$$

Figure 1: Markov chain diagram of simple system with replacements

the number of spares, and **status** is 'a' (active) or 'r' (repair) to reflect whether the system is undergoing repair. We illustrate a system with three initial processors and two initial spares.

Reliability tools based on Markov models vary in how they allow a modeler to describe the state-space, and in how the necessary probabilities are computed. Very rarely is a state-space directly specified as a graph. Because state-spaces of interesting systems are quite large, most tools instead use some modeling formalism from which state-spaces can be derived automatically, e.g., a stochastic Petri Net. Whatever the formalism, given any configuration that maps to CTMC state i, one **must** be able to determine all configurations that map to all CTMC states j reachable from i in a single transition, and must be able to determine the associated transition rates. Given this ability, the entire state-space can (in principle) be constructed (at least, if it is finite). A list of states to explore is first initialized with an initial state, as is a table of previously discovered states. Then, one repeatedly removes a state from the list and generates all of its immediate descendents. The table is consulted for each descendent to determine whether it has already been generated. If not, the descendent is entered into the table, and is placed on the working list. The entire state-space is found in the table once the working list is empty.

Techniques vary for computing system unreliability from the state-space graph. They all share the drawback that state-spaces easily become huge. Even with virtual memory, solution speed plummets on large models. Recognizing this as the key obstacle to solving very large models automatically, we adopted an approach that uses dramatically less memory. Rather than build and analyze the state-space in two separate steps, we combine the two functions and retain only the amount of state information needed to support the analysis. The technique has made possible the analysis of models that simply could not be solved using other methods. The methods described in the remainder of this section are developed in detail in [3]. The basic ideas are sketched in the following paragraphs.

Suppose we are given a path p through the state-space, beginning at the initial state. If we let $\mathcal{P}_p$ be the event that the system chooses the transitions on that path (rather than any of the competitors) and let $\mathcal{T}$ be the event that it does so within time T, we have by definition of conditional probability

$$\Pr\{\mathcal{P}_p \text{ and } \mathcal{T}\} = \Pr\{\mathcal{P}_p\} \Pr\{\mathcal{T} \mid \mathcal{P}_p\}. \tag{1}$$

Now $\Pr\{\mathcal{P}_p\}$ is something we can easily calculate. Each transition on the path has some specific probability of being chosen over its competitors; $\Pr\{\mathcal{P}_p\}$ is just the product of all of these probabilities. We can also compute $\Pr\{\mathcal{T} \mid \mathcal{P}_p\}$, although it can be more involved. Given that the path was chosen, the probability that the system traverses it in time no greater than T is the probability that the sum of times spent in each state on the path is no greater than T. The time spent in each state is an exponential random variable, but the exponentials associated with the different states may have different means. The probability distribution constructed by summing heterogeneous exponentials is well understood, and one can numerically evaluate the probability needed. An alternative method is to use faster means of computing an upper bound on the probability [11]. The essential point is that given a path and all transition rates out of nodes on that path, we can compute the probability in equation (1). The only memory storage required is that needed to store the sequence of states, and the transition rates. Notice that the system unreliability is precisely the sum of probability (1) over all paths p whose terminal state is in $\mathcal{S}$.

We capitalize on the last observation by interweaving a depth-first state-space generation activity with model analysis activity. The algorithm creates and extends *path records*. A path record contains the terminal state on some path, plus all the rate information needed for computing probability (1). We initialize a working list of path-records with one that describes the initial state. In a loop, we remove the first element of the working list, and determine whether the terminal state is in $\mathcal{S}$. If so we compute probability (1) and add the result to a running total. If not, we generate all the descendents of the terminal state and their associated rates. For each one we create a path-record by extending the path-record of their common parent with additional transition information. Each new state replaces the parent as the path record's terminal state, and the list of new path-records is inserted at the front of the working list. This process continues until the working list is empty, at which point we know that every path from the initial state to some state in $\mathcal{S}$ has been explored and its contribution to system unreliability has been accounted for.

The algorithm above will explore the entire state-space. This is usually unnecessary. If the probability of a path p that does *not* end in $\mathcal{S}$ is lower than a *cut-off threshold*, we may choose not to expand it. Probability (1) for that path p is an upper bound on the probability of reaching $\mathcal{S}$ by any other path q which extends p by one or more states. Instead of expanding p, we add probability (1) to an accumulating pruning bound. At the end of the computation, the sum of the pruning bound with the computed system unreliability gives an upper bound on the true system unreliability.

This general approach has the consequence of recomputing a state i for every unpruned path that includes it. We have essentially exchanged the cost of storing the state-space with a cost of recomputing it. A rule of thumb is that if you have approximately n transitions out of each state, and a search depth of d, then $O(n^d)$ path-records will be generated. We have solved models where $n \approx 1000$ and $d \approx 3$; we have solved models where $n \approx 100$ and $d \approx 6$.

There are two additional mechanisms for analyzing even larger state-spaces. One is to use parallel processing. This is relatively easy to do, because the path-searches are disjoint. For example, one need only distribute the descendents of the initial state among processors. Each one initializes its working list with its assigned set of descendents, and then carries on independently of the others. At termination the processors aggregate their individual pruning bounds and unreliability probabilities. We have also implemented more sophisticated schemes that employ dynamic load-balancing.

A second mechanism is to forego exhaustive analysis, and accept statistical estimates of the system unreliability. This can be done with Monte Carlo style analysis. The idea here is to randomly sample a path through the state space until either pruning, or encountering a state in $\mathcal{S}$. Upon terminating the expansion of a path, probability (1) is computed and is used as one "sample". The process is repeated, each time starting from the initial state, to construct a set of statistically independent samples. Their sample mean is a point estimator of the system unreliability; standard statistical techniques are used to construct confidence intervals about this mean.

It is important to see that the techniques we've described for generating and analyzing Markov state-spaces are independent of the formalism from which the CTMC is derived. We can separate mathematical evaluation of system unreliability from the concerns of deciding what the state-vector is, what a state's descendents are, and whether a state is in $\mathcal{S}$. The Solution Engine Interface described next allows this separation.

3 Solution Engine Interface

The solution algorithm we've described interweaves state-space generation (which lies in the modeling formalism domain) and state-space analysis, which is not provided by the modeling formalism. We have developed a means of integrating an implementation of our solution method into general modeling tools, particularly discrete-event simulators. We accomplish this through a fully-specified message-passing protocol that sits between the modeling tool and our solution engine. The protocol sets up the solution engine as a "master" process that queries the modeling tool as a "slave" process. Clearly some integrating mechanisms must be resident in the modeling tool, a specific case of this is described in the next section. References [6, 7, 8] describe how the interface engine was integrated into an entirely different analysis package, ADEPT. The remainder of the present section discusses the interface protocol.

The solution engine does not have any notion of what the CTMC state represents. The modeling tool will be responsible for defining the CTMC states; to the solution engine a state is simply a block of memory. The solution engine does not have any idea of what constitutes a failure state, as that depends on the model. Similarly, the solution engine does not know how to find the descendents of a given state; that too is model dependent. What the solution engine *can* do is to query the modeling tool to provide answers to questions like "is this a death state?". The interface protocol defines a set of queries the solution engine may make, and the responses the modeling tool provides.

At the beginning of an analysis, the solution engine requires a number of initialization parameters. For instance, it needs to know whether the analysis will be exhaustive or Monte Carlo, it needs to know the mission time, it needs to know the cut-off threshold. it needs to be given a copy of the initial state. Once initialized, the solution engine enters a control loop whose body is repeated for every path-record pulled off the working list. First, the path-record's terminal state

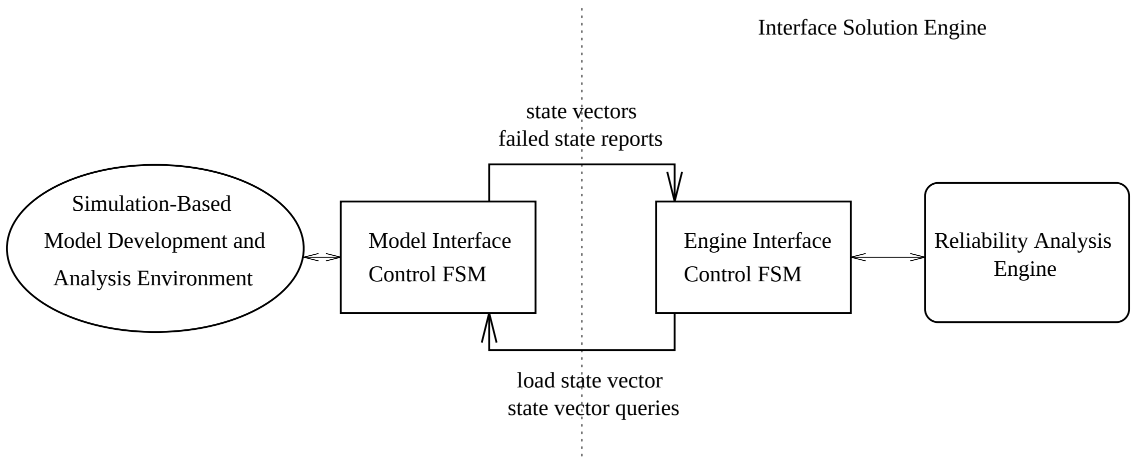
Interface Solution Engine
state vectors
failed state reports
Simulation-Based Model Development and Analysis Environment
Model Interface Control FSM
Engine Interface Control FSM
Reliability Analysis Engine
load state vector
state vector queries

Figure 2: Interface between modeling tool and the Interface Solution Engine

(again, just a block of memory to the solution engine) is passed to the modeling tool with the command to LOAD the state, and to determine whether the state represents a "notable" state (e.g., one that may not indicate system failure, but is one for which wants probability estimates anyway). For its part, the modeling tool must load the system-state passed to it, and make the determination required. The corresponding response is passed back to the solution engine, and the appropriate probability calculation is made as a consequence. The interface engine then requests that the modeling tool determine whether the current system state represents system failure. If the answer is in the affirmative, the solution engine uses the path-record information to compute probability (1), and control returns to the top of the loop. Otherwise, the solution engine applies the pruning test: if the path is pruned then a calculation for the pruning bound is made, and control returns to the top of the loop. If the state passes both the death and pruning tests, the solution engine then asks the modeling tool to generate all descendents of the loaded state. The modeling tool's response to this query is to generate the descendent states, passing them back to the solution engine, each one tagged with the associated transition rate and (possibly) auxiliary information that is not necessary for the unreliability calculation. Receiving these, the solution engine creates a path-record for each, places the group of them at the front of the working list, and returns to the top of the control loop.

The interface between solution engine and modeling tool is evidently simple. The number of types of messages that pass between them is small. Figure 2 illustrates the interface. The protocol logic on both sides of the interface is simple. The solution engine is encapsulated as a C function. On each call a formated message is passed in. A finite state machine parses and processes the message, producing as output a message for the modeling tool. Since the whole of the solution engine is just a C function, it can be embedded in any modeling tool that allows integration of user defined C functions.

In addition to probability computations, the solution engine also provides a way of conducting automated failure modes and effects analysis (FMEA). For, a common model expansion paradigm is to essentially fail a component and track the effects through the model. The set of changes that transform a model can be stored as part of a path-record, and be appended as a path explores deeper into the state-space. In fact, many view this latter capability as more important than the reliability numbers themselves. Since the FMEA activity sits directly on top of the reliability calculation, one can actually order the importance of event sequences using the computed probability for that sequence, and store only the most probable ones. It should also be noted that this approach accounts for failures dependent on sequence dependencies (e.g. failure of component A followed later by failure of component B fails the system whereas failure of B followed by failure of A does not), and so the approach has more modeling power than more traditional solution methods—like fault trees—that do not.

The solution engine interface, as described, gives a way of processing a very general CTMC, provided that the CTMC is defined elsewhere. We turn next to a discussion of how a commercial modeling tool was used to provide that definition.

4 Integration into BONeS Designer

The Solution Interface Engine requires that the modeling environment be endowed with the capability to parse and respond to commands asking it to

- completely reset the state of its reliability model,
- analyze the existing state of the reliability model, and determine whether the system has failed, or has entered some "notable" state,
- generate all descendents of the current (non-failed) state.

Provision of these capabilities requires that two fundamental issues be addressed. One is a matter of control, the other a matter of representation of model state. As we approached BONeS Designer, we looked for features of the tool that would satisfy these requirements. To understand our solution, we must first sketch some relevant Designer characteristics.

BONeS Designer is a graphically oriented design tool where one describes a system in terms of interconnected "components", that appear as boxes or icons on the screen. Every component has a local environment which defines its interface to the rest of the model. The environment consists of

- input and output ports, through which data is passed to the module to activate an evaluation of the module,
- event-lists, which may also pass data and activate evaluations but through a less graphical linking process,
- parameters for passing constants into and out of a component (e.g., a service rate for a queue is passed in, an ASCII name for the component might be passed out by a separate parameter),

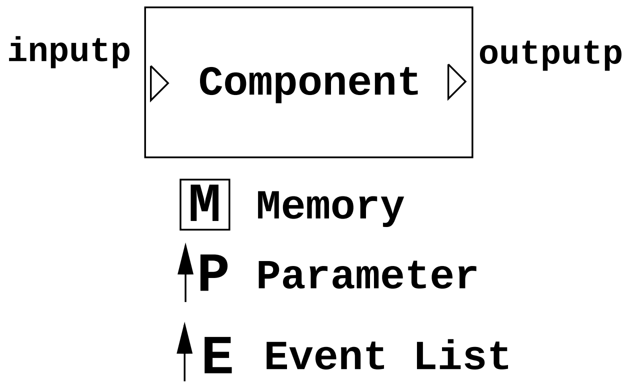

Figure 3: An example component in Designer

- memories, which are read/write locations.

Figure 3 illustrates how a component appears in a Designer model, and illustrates an example of each of these environment variables. "inputp" is an input port, "outputp" is an output port; the other examples are likewise given names reflecting function. The up-arrow on "Parameter" and "Event List" indicate that these entities are to be exported as part of the component's external environment. "Memory" is local, not visible outside of the component, as indicated by the square around its symbol. Connections are made explicitly by directing the contents of components' output ports to other components' input ports. The tool uses a data-flow evaluation paradigm. A signal (which may be a user-defined structured data type) enters a component at an input port, and may cause other signals to leave through output ports, possibly after some time delay. Events placed on event-lists may also pass data and trigger evaluation. Event lists, memories, and parameters may be made part of the module's externally visible environment, or may be strictly local to the component. Exported entities can be linked at higher levels in the model. For instance, if components A and B have a common parent component, that parent may have a memory to which a memory exported by A is linked, *and* to which a memory exported by B is linked.

The functionality provided by a component may be expressed itself as a network of interconnected components (and so the single iconic view is from a higher point in the hierarchy), or the component may be a "primitive" that does not decompose any further. BONeS Designer comes with a large component library. A basic library provides commonly used primitives, one may also purchase other libraries of already developed subsystems that are specialized for certain network types.

The functionality of every primitive is expressed as C or C++ functions that manipulate internal Designer data structures and call internal Designer routines. Designer supports development of custom primitives nicely; a user describes the interface of the primitive (e.g. input/output ports), and Designer generates a function template that serves as the basis for the customized primitive. A primitive has three functions to specify. One is called at initialization, when a simulation model is first run. Another is called whenever the component's environment calls for it to respond to some input, e.g., when a signal arrives on an input port. A third is called when the simulation terminates.

Custom primitives provide the hooks we need to integrate reliability analysis into BONeS Designer. First, consider the Interface Engine itself, and the logic that interfaces with it. It may be brought into a model with a single icon, which exports fewer than 10 event lists and parameters. The code represented by this icon is substantial, even though its interface to the rest of Designer model is very modest. But, from the Designer user's point of view, the reliability engine is brought in simply by placing an instantiation of this interface engine in the Designer model, and linking to the event-lists it exports.

Now consider the hooks within the actual Designer model, e.g., the system state vector. We need to be able to quickly push state vectors into and out of the Designer model. Our reliability library contains primitives for expressing state variables (single integers) and for reading and writing those variables. A user includes a **StateVariable** primitive wherever it is natural in the model to specify a component of the system state vector. To read from and write to that variable one includes the appropriate primitives, and links their exported memories to that of the **StateVariable**. That memory does not contain the actual state value, as one might think. Instead, it contains the *index* of that state variable in an array of integers that is unseen from within the Designer model. To copy in or out the entire system state vector we need only copy to or from that array. As it is a contiguous block of memory, this can be done very quickly. To read from or write to an individual element of the state vector, we need only to know its position in this array—the value found in the **StateVariable** memory.

Another key primitive in our library is **ScheduleTrigger**, one that exports an event-list, and contains one output port. This primitive's event list gets exported to be linked with one of several event-lists defined by the Interface Engine. Just as the **StateVariable** primitive provided the data hook into Designer, **ScheduleTrigger** provides the control hook. For instance, it is the modeler's responsibility to provide a computation that determines whether the system state is in a failed configuration. **ScheduleTrigger** provides the means by which the Interface Engine can initiate a death-state test evaluation. The Designer submodel for testing system status is made to depend upon receipt of a trigger signal to initiate it. Output from a **ScheduleTrigger** provides that trigger, and the primitive's event-list is linked to one exported by the Interface Engine primitives for the express purpose of initiating death-state testing logic. Naturally, that logic will include primitives to read **StateVariable** values to determine the system's status. The same type of mechanism is used for initiating evaluation of notable conditions, and for initiating generation of descendent states. Another set of primitives report the results of these tests to the logic that interfaces with the Interface Engine.

At initialization, each primitive executes its own initialization function. In the case of the **ScheduleTrigger** primitives, each ones' initialization places itself on the event-list to which it is linked. Later, when the Interface Engine schedules an event for an event-list, that event activates evaluation of *each* component that is linked to the event-list. For instance, a model may have a number of different notable state conditions. To have each one evaluated individually, the Interface Engine simply schedules an event on the event-list to which notable state **ScheduleTrigger** components have attached themselves. Each is executed in turn, and each reports whether its particular notable state condition is satisfied.

Owing to the intricacies of generating and reporting descendent system state-vectors, we have built higher level components from this set of primitives. These components hide almost entirely the details of system state manipulation. For instance, many reliability models contain representation of physical devices that are either operational or failed, and which fail with some device-specific failure rate. As with the example in section 2, we may model a collection of these with a single state variable that enumerates the number of working instances; at any instance there is a specific aggregate transition rate associated with one of those devices failing. Each such failure is a transition in the underlying Markov chain. We have further hidden the details of reliability analysis from the user by building several different types of **Device** components that model k-out-of-n subsystems (i.e., the subsystem is considered operational if at least k out of n devices are operational). Parameters to a device component include the number of devices n, the subsystem failure threshold (k), and the common device failure rate. For some versions of **Device** a user simply includes these components and links their exported event-lists to the appropriate Interface Engine event-lists. The implicit transitions and their interaction with the Interface Engine are hidden within the component design. There is a **StateVariable** and logic to report whether a transition is possible, and logic to decrement the **StateVariable** when a transition is triggered.

However, some versions of **Device** are not completely transparent. For instance, there may be cases where one wants the execution of a **Device** transition to allow further system state processing beyond just modification of the single **StateVariable** in the **Device**. These variations export event-lists; evaluation of additional logic may thus be triggered by linking in a **ScheduleTrigger** primitive to the exported event-list. Other **Device** versions include input/output ports for more directed reporting of **Device** transition. For these, a signal is placed on all output ports when the subsystem the device represents transitions to a failed state (i.e., the number of operational devices there is less than k). Some run-time optimizations can take advantage of propagating this sort of information around.

An important aspect of the reliability model processing is that in no way is simulation time advanced. None of the components that are evaluated advance the simulation clock. To achieve this one must understand a number of subtleties involving the sequencing of component evaluations. We omit a discussion of these, as they are not essential to understanding generally how the Interface Engine is integrated into Designer.

5 Integrating Reliability and Performance

Reliability models of systems are usually expressed at a higher level of abstraction than are performance oriented models. The toolset we've developed supports the co-existence of a reliability model at one level of abstraction, and a performance model at another. For example, consider a highly reliable subsystem comprised of four identical but independent devices. A performance model of this system may go into some detail concerning interconnections between the four devices, in order

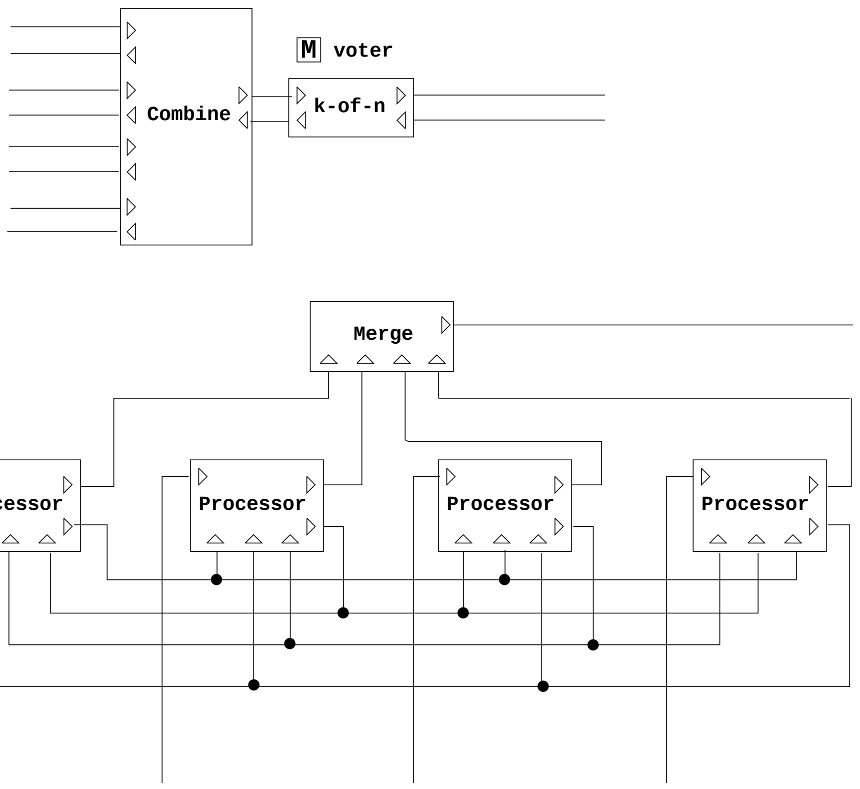

Figure 4: Joint reliability and performance models

to assess the cost/performance of different voting architectures. That level of detail is likely to be undesirable in the reliability model, where viewing the subsystem as a 2–of–4 device may suffice. Our tool extensions allow one to include both the reliability model, and the performance model, in the same place. This is illustrated in Figure 4. The upper portion is the simple reliability model. We illustrate here a version of the **k-of-n** module where one connects modules together bidirectionally in order to effect some automated state-space reduction techniques. The reliability model in the upper portion mirrors the performance model, but at a higher level of abstraction. Each model has four sets of inputs and one output, but the reliability model avoids the complications of how one actually builds a 2–of–4 device. Still, the two models coexist side-by-side in the same context. As the performance model changes, it is natural to change the reliability model correspondingly. If desired, one can link them, through the exported **StateVariable** memory. We have experimented also with merging the two representations more directly, attempting to use the same communication paths for both reliability and performance. However, BONeS Designer handling of connection types prohibited a general solution. One way of using a joint model of this type is to provide a "on-off" switch which enables or disables the Interface Engine processing, thereby selecting either performance or reliability analysis.

Another one of the benefits of embedding reliability modeling capability into a discrete-event simulation engine is that it allows one to use performance measures to describe notable events, system failure, or even state transition. For example, overly long control signal latency through a network could define system failure, e.g. in a fly-by-wire flight control system. To determine whether the system in a given "reliability" state is safe relative to a performance-based criterion, one can execute a discrete-event simulation whose behavior depends on that state to determine whether the performance criterion is met. Embedding the reliability model within a performance-oriented tool allows one to expand the description of system safety beyond what is typically provided in reliability tools.

This idea[1] is illustrated in Figure 5, where three **k-of-n** subsystems are in series. Each one models some part of a network, and for the purposes of illustration we suppose that the delay time of a message through a subsystem is inversely proportional to the number of working components. All blocks shown are from the standard Designer library except for **ScheduleTrigger**, **ReadState**, **k-of-n**, **ReportDeathState** and **ReportNoDeathState**. The Interface Engine schedules evaluation of **ScheduleTrigger** to initiate the death state test. The trigger generated propagates to three components. One of these, **TNow**, places the value of the simulation clock on its output, a value which is then read and stored by the **RealLocalMem** component. The trigger also prompts **AbsDelay**, but its response requires two inputs, the second of which has not yet arrived. The trigger also activates **ReadState**, which reads the state of the first **k-of-n** system, and sends it to **cvt**, which converts the integer into a Real, then passes that value to **1/R** which inverts the value and passes it as the delay parameter to **AbsDelay**. After the specified delay, this module produces a triggering output to the next stage. In this fashion a trigger is delayed three times, each delay being a function of the reliability state. As the trigger emerges from the last stage, it prompts a remeasurement of the current simulation clock, and a recall of the measurement made when the trigger was first generated. The two clock values are differenced and sent to a comparison module which prompts **ReportDeathState** if the difference exceeds the safety threshold, and prompts **ReportNoDeathState** if not.

While this example is too simple to be practically useful, it does illustrate the central point—the definition of reliability itself may depend on performance aspects, and in a setting such as ours such problems can be approached.

6 Optimizations

The tool described in this paper is actually a second generation integration of Designer and our Interface Engine. The first generation tool was used several times in Boeing to evaluate experimental flight control architectures. System sizes were large, several hundred components. Within that setting a particular form of system evaluation was common. The system architects wanted to know the probability that a control surface was not under control of the pilot. This condition could be translated into the existence of electrical paths between key points in the architecture. Consider Figure 6, which provides a very much simplified view of a fly-by-wire flight control system. There is redundancy in sensors, in actuators, and in data transport. There is also considerable redundancy

[1]The example discussed here is not yet supported in our tool extensions. The mechanisms needed to supported it are understood, and our next version of the extensions will contain them.

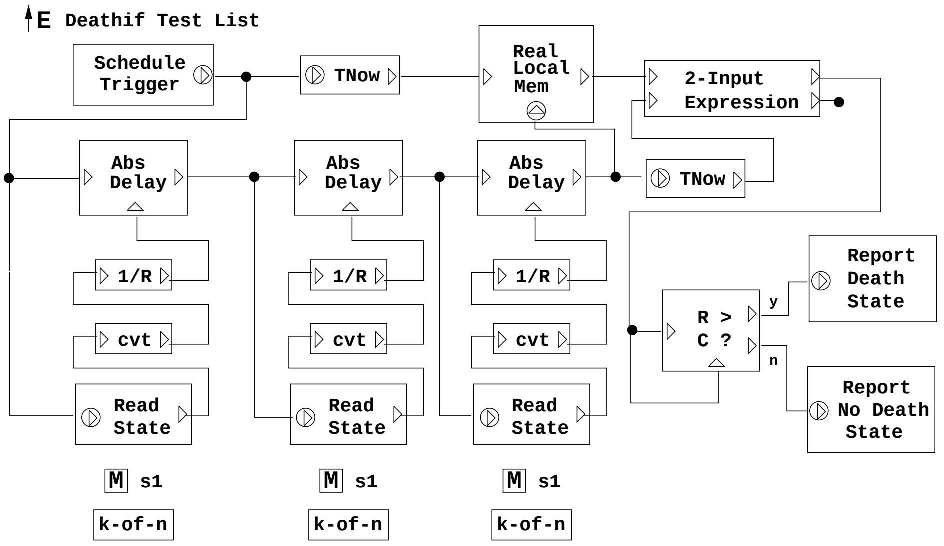

Figure 5: Reliability's dependence on performance

hidden within the Primary Flight Computer (PFC). As system components fail, paths involving the failed components cannot establish needed connections. The system is operational if at least one sensor can communicate with the PFC, the pilot input can communicate with the PFC, and the PFC can communicate with at least one actuator. Taking this sort of criteria as a guide we've developed a set of optimizations that are embedded in some versions of the **Device** primitives.

The optimization library includes additional primitives. The first is a **PathPoint**. These are placed in the model to create reference points for description of paths. In Figure 6 one **Path Point** would be inserted between the sensors and their connections to the buses, another between the Pilot inputs and their connection with the buses, another one between the buses and the PFC, yet another between the PFC and the buses, and a final one between the buses and the actuators. A **PathPoint** exports a memory containing its (unique) identity. A **PathConnect** primitive exports two memories, ordered (source and destination), which are linked to two **PathPoint** memories. Under stimulation by an initiation signal, the **PathConnect** primitive indicates whether there exists at least one path from the source **PathPoint** to the destination. Responses from **PathConnect** primitives can then be used to assess whether the system is operational (or is in some notable state).

In any given system state we have associated a directed graph of components; this graph already reflects the full effect on connectivity of all previous component failures. A graph node representing a **k-of-n** component is thought to encapsulate n parallel components, each receiving all of the components inputs and each contributing to all of the component's outputs. Now imagine a series of m components such that all have exactly one input arc and one output arc, and such that the next transition for each will cause it to be failed (i.e., any **k-of-n** component has exactly k operational subcomponents). The next transition undergone by any one of the components in the series removes from the graph any path that passes through that series. The aggregate transition

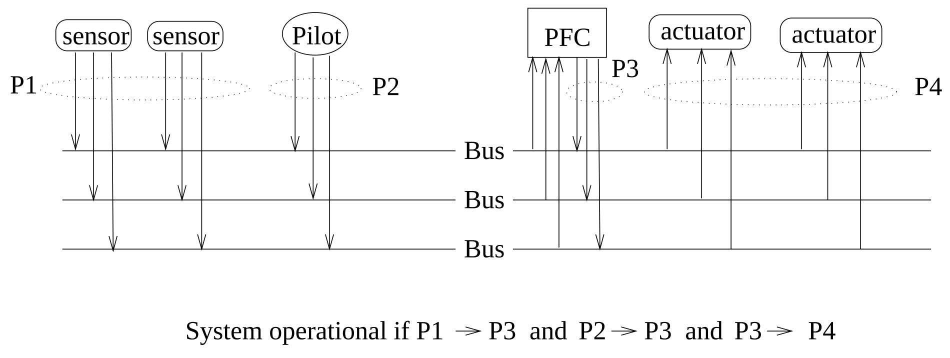

Figure 6: Simplified model of flight control architecture

rate of that series is equal to the sum of the transition rates of its constituents. It is well known that a single component with the aggregate series transition may replace the series in the graph, without altering the probabilistic structure. Our optimizations essentially do that, but they also do it dynamically. The substitution is not one that is done once on the initial graph structure, for that structure will not contain all of the series it may develop as failures occur. Instead, when a **Device** transition is caused by the Interface Engine and the device is pushed into a failed state, report of this failure is sent to all of the device's predecessors and descendents in the graph. The signal is propagated by any recipient that (i) needs only one further transition to be failed itself, and (ii) is in series with the report source. Continuing in this fashion, each of the graph nodes that is in series with the failing **Device** recognizes this, each of them alters it **StateVariable** to represent failure (precluding further transition), and each of them contributes its transition rate to an aggregate.

In an earlier approach [5] we determined whether the required sequence of paths exists by actively pushing signals out from the sources, and noting whether any such signal reached the necessary destination. This tactic was very costly computationally, since in a highly reliable system, the needed paths are usually present. Another optimization we developed (whose details are beyond the scope of this paper) approached the problem differently. Now, when a component fails, the effect of that failure on path connectivity is automatically propagated. Instead of trying to see if a path exists at every death-state test, we do a small amount of computation with every component failure, and eliminate paths that are no longer viable. At the time of the death-state test, all of the information concerning path-connectivity is immediately available. As we will see, this optimization has profound benefits.

7 Experiments

We now briefly describe some experiments that illustrate the processing power of our combined tool. The problem domain is the fly-by-wire flight control architecture described earlier. We are interested here principally in processing speed, and in the effectiveness of Monte Carlo estimation.

The fly-by-wire models we considered earlier determined system safety during a death-state

cut-off threshold	exc. time (secs)	path-records/sec	unreliability bound
1e-10	20	811	7.5e-9
1e-12	36	763	5.51e-9
1e-14	145	759	1.92e-11
1e-16	874	709	3.5e-13

Table 1: Processing rates for model of fly-by-wire architecture

test by broadcasting signals and noting connectivity. This proved to be one of the more severe bottlenecks in the processing. Another was that much of the state-space analysis was programmed within the BONeS Designer graphical language. A substantial performance boost is obtained by conducting that analysis in C, within a primitive. The processing rate of the tool was about 30 path-records per second. In order to analyze very large models, parallel processing had to be used, for very long running times.

The model we consider now is similar in structure, but uses the approach and the optimizations already described. The model is smaller than those used in the actual architectural studies, but in our experience the path-record processing rate is relatively insensitive to problem size (there is of a course a dependency on the length of the state-vectors that are copied, but this cost is relatively insignificant compared to others). Table 1 summarizes the data, collected from runs on a 55 MHz Sparc 10, on a largely dedicated processor with 128Mb of memory. As we decrease the cut-off threshold, the search process generates and explores more of the underlying state-space. Naturally then, the execution time increases, but the unreliability bound decreases. There is a factor of forty difference in time between the run at the cut-off threshold and that of the lowest threshold, there is a corresponding increase in the number of path-records generated. The relatively small decrease in path-record processing rate may be attributed to the fact that at deeper levels in the state-space, the cost of maintaining path connectivity information is a little higher, owing to more extensive propagation of component failure effects on the model. Only at the deepest cut-off level shown (1e-16) did the sum of measured death-state probabilities exceed the sum of pruned state probabilities; the value shown, 3.5e-13, is not larger than the true system unreliability by more than 10%. This happens because the searching is finally deep enough to undercover dominant failure states. Finally, note that the path-record processing rates are over 200 times faster than those we achieved in the first generation tool.

Processing rates of this order bring into reach some goals for analysis capability that have been privately expressed to us by potential users. For instance, an architecture with 1000 components, analyzed to three failure levels, gives rise (without optimizations) to approximately 10^9 path-records. At a rate of 10^3 path-records per second, one million seconds (11.5 days) processing time would be required. One can push this back by an order of magnitude or two, using parallel processing. A day's execution time for a large complex architecture is deemed acceptable. Even so, an interactive capability would be desirable, as would be a capability of handling larger still models. For this we turn to Monte Carlo analysis.

The technical details of how we perform Monte Carlo analysis are found in [3]. The general idea is to repeatedly choose independent sample paths through the state-space, each of which terminates in a death-state. On reaching the death-state, we compute the probability of reaching that state within the mission time. This provides a statistical estimate of the overall unreliability. Repeating this process and averaging the estimates provides a point sample of the unreliability, about which confidence intervals may be constructed in the standard fashion. Provision is also made to use importance sampling, when fast recoveries are part of the model.

On the specific problem whose timings we report in Table 1 we used 10 replications (requiring 10 seconds of processing time), 100 replications (requiring 97 seconds of processing time), and 4000 replications (requiring 420 seconds of processing time). The 10 replication run estimated the unreliability to be no more than 8e-13, with 99% confidence. The 1000 replication run lowered that estimate to 4.4e-13, with 99% confidence, and the 4000 replication run gave 3.7e-13 with 99% confidence. We see here the usual situation with statistical estimation, that to tighten the confidence interval by a factor of f requires a factor of f^2 more samples. The key thing to note is the amazingly good answer that is provided by Monte Carlo after only 10 seconds of processing time. To achieve that same level of result in the exhaustive approach requires almost two orders of magnitude more processing time.

8 Conclusions

We have developed a reliability analysis engine suited for integration into existing modeling tools. We have accomplished this integration in the specific context of BONeS Designer, a commercial simulation modeling tool. This paper discusses the overall integration approach, and specifics concerning integration into BONeS Designer. We point out how our approach supports integrated performance and reliability analysis, we discuss some run-time optimizations that provide fast execution times, and discuss performance on a prototypical model. We show that our approach brings into reach the hope of analyzing systems with hundreds of components to three failure levels. Monte Carlo analysis allows analysis of problems with substantially larger size.

We have demonstrated the feasibility of joint and integrated performance and reliability analysis in a common, industrial quality modeling tool. The tool extensions we've developed are experimental. We are continuing to explore ways of expanding the modeling power of the combined approach, and to further optimize run-time efficiency.

References

[1] G. Ciardo, J. Gluckman, D. Nicol. Distributed State-Space Generation of Discrete-State Stochastic Models. *ICASE Technical Report* 95-75, 1995.

[2] P. Heidelberger. Fast Simulation of Rare Events in Queueing and Reliability Models. *ACM Trans. on Modeling and Computer Simulation*, vol 5, no. 1, Jan. 1995, pp. 165-202.

[3] D. Nicol and D. Palumbo. Reliability Analysis of Complex Models using SURE Bounds. *IEEE Trans. on Reliability*, vol 44, no. 1, March 1995, pp. 46–53.

[4] D. Nicol, D. Palumbo, and A. Rifkin. REST: A Parallelized Reliability Estimation System. *Proceedings of the 1993 IEEE Annual Reliability and Maintainability Symposium*, Atlanta, Georgia, Jan. 1993, pp. 436-442.

[5] D. Nicol, D. Palumbo, and M. Ulrey. A Graphical Tool for Reliability and Failure-Mode-Effects Analysis. *Proceedings of the 1995 IEEE Annual Reliability and Maintainability Symposium*, Washington, D.C. Jan. 1995, pp. 74-81.

[6] R. Rao, A. Rahman, and B. Johnson. Integrated Performance and Dependability Analysis Using the Advanced Design Environment Prototype Tool ADEPT. *Proceedings of the AIAA Computing in Aerospace Conference*, San Antonio, Texas, March 28-30, 1995, pp. 285-300.

[7] R. Rao, B. Johnson, J. Aylor, and R. Williams. Integrated Performance and Reliability Evaluation Using Information Flow Models. *Proceedings of the Second Annual Conference on Rapid Prototyping of Application Specific Integrated Circuits (RASSP)*, Washington, DC, July 24-27, 1995, pp. 109-114.

[8] R. Rao, A. Rahman, B. Johnson. Reliability Analysis Using the ADEPT-REST Interface. *Proceedings of the 1996 IEEE Annual Reliability and Maintainability Symposium*, Las Vegas, Nevada, January 22-25, 1996, to appear.

[9] J. Swain. *Simulation Survey. OR/MS Today*, 64-79, August 1995.

[10] M. Ulrey, D. Palumbo, and D. Nicol. Case Study: Safety Analysis of the NASA/Boeing Fly-By-Light Airplane Using a New Reliability Tool.

[11] A.L. White. Reliability estimation for reconfiguration systems with fast recovery. *Microelectronics and Reliability*, **26**(6):1111–1120, 1986.