A Rainbow Net Simulator
with a Dependability Application

Michael A. Schoenfelder      William A. Rogers
University Of Texas
Computer Engineering Research Center
2201 Donley, Austin TX 78758

Abstract
Discrete event simulation is often required when modeling realistic systems. A powerful extension to Petri Nets, Rainbow Nets (RN) are a modeling technique that combine graphical and state variable based specification for discrete event simulation. This synergistic combination can capture both the spatial and procedural aspects of a system. This paper presents an efficient RN implementation that capitalizes on RN's multiple modeling paradigms, thus yielding a natural and convenient way of expressing system behavioral models.

RNs were designed especially for dependability modeling. Previous modeling techniques suffer from an explosion in the number of states required to model realistically-sized systems. The efficacy of the RN simulator is demonstrated by a dependability analysis of an n-modular redundant system. This model is shown to be constant in specification complexity, linear in simulation space, and polynomial in simulation time.

1 Introduction
A series of discrete events can describe the actions of many systems. The behavior of such systems is modeled by a discrete event model which can be used in either formal analysis or event simulation. While formal analysis may yield an elegant closed form solution, practical problems often require the use of simulation. Therefore, the goal of a practical discrete event modeling technique should be to exploit the power of simulation while maintaining the elegance of analytic models. This paper argues that the Rainbow Net modeling technique [1, 2] meets these requirements by demonstrating the viability of a Rainbow Net simulator implementation.

Discrete event models can be categorized as either graph based or state variable based. Two criteria for comparing these models are the method for model specification and susceptibility to state-space explosion.

Graph-based models are discrete event models whose states can be specified by a graph, e.g., Markov chains and Petri Nets [3]. These models can be represented graphically. While this representation property in no way enhances generality, it does make some models simpler for people to design because humans think about concepts such as organization and information flow from a spatial point of view. Spatial relationships are most easily depicted and manipulated graphically, thus making design simpler and increasing utility. Any modeling technique intended for interactive use should allow graphical representation of spatial concepts. The major disadvantage of graph based models is state space explosion. When the number of states grows exponentially with problem size, enumerating the states in a graph data structure becomes prohibitively expensive in space and often in time, depending on the algorithms applied to the state space.

On the other hand, state variable techniques specify states by means of formula variables rather than figures. These models do not suffer from an explosion in data storage requirements. In addition, complex rule-based mechanisms for determining the next state may be more easily specified by mathematical notation that is directly translatable into programming language statements. Unlike graphical models, the main disadvantage of state variables is their specification method often hides the relationships between states.

Obviously, a good modeling technique should combine both graph-based and state variable based specification. Inherently spatial concepts that do not suffer from space explosion should be specified graphically, and inherently formal and procedural concepts would be specified through a text-based language interface. Petri Nets and the subsequent extensions exhibit a trend toward this end. Rainbow Nets, the next step in this trend, are targeted specifically for practical, flexible simulation.

2 Petri Nets and Extensions
Rainbow Nets (RN), an extension to Petri Nets that combines graphical and state variable based specification, were developed by Allen Johnson, Jr. [2] as a new modeling technique for system evaluation and analysis. Assuming that the reader is familiar with basic Petri Net concepts, this section describes the Petri Net extensions leading to Rainbow Nets.

Petri Nets (PN) model systems whose actions are described by asynchronous events. Events occur when
certain preconditions are satisfied, and after the occurrence, certain postconditions hold. Specified by a bipartite graph [4], models consist of a finite set of places \( P \), a finite set of transitions \( E \), and a finite set of directed edges (arcs) \( A \) connecting places to transitions and transitions to places.

### 2.1 Timed Transitions

Since plain Petri Nets do not explicitly incorporate time, net theory was extended to model stochastic event times. Basic Petri Net transitions take zero time to fire and are thus called instantaneous transitions. One extension, which is fundamental to Rainbow Nets, is the addition of transitions with non-zero firing times. These transitions are termed timed transitions. When these transitions exhibit exponentially distributed firing times (but not instantaneous), then the resulting net is known as a Stochastic Petri Net (SPN) [5]. A net that combines stochastic and instantaneous transitions is called a Generalized Stochastic Petri Net (GSPN) [6]. In this case, if both an instantaneous and a timed transition are enabled and conflicting, then the instantaneous transition will fire.

### 2.2 Colored Tokens

In contrast to the basic black tokens of Petri Nets, colored or individual tokens increase the descriptive power of nets [7, 8, 9]. Concepts that may have been modeled by many separate places can now be modeled by many different colored tokens and transition rules that apply only to certain colors. For example, if one wished to model the activity of each individual processor in a 16 processor network, a single-color net would need 16 identical subgraphs, one for each processor. On the other hand, a net with 16 token colors only needs one such subgraph. Since each processor is represented by a different color, its activity can be easily differentiated from the others.

### 3 Rainbow Nets

"Rainbow Nets provide new extensions to Petri Nets which enable the avoidance of an explosion in the number of places and transitions as the size and complexity of the problem increases. [1]" This is accomplished by increasing the modeling power (generality) of both the tokens and the transitions. Further extensions in the area of temporal modeling allow more realistic modeling of time-dependent systems. This section informally discusses Rainbow Net elements and properties. For a more formal description, see Johnson and Malek [1].

#### 3.1 RN Tokens

Rainbow Net tokens include all previously defined tokens plus an extension of colored tokens. As shown in Figure 1, Rainbow Net tokens are divided into two classes—individual tokens with identity and generic tokens without identity. The first type of generic token is termed clear and is equivalent to the traditional black Petri Net token; it has no discriminating characteristics. The second type of generic token, hue, is similar to colored tokens. Although hues have a discriminating characteristic, their hue, they do not have an individual identity; tokens of the the same hue cannot be differentiated. In contrast, item tokens are uniquely identifiable, that is they have an individual identity or unique color. Items represent the entity which is being modeled, e.g., a particular computer. Items may possess features or conditions, e.g., a computer may have a fault in it. Features usually represent transitory entities which may come and go during the lifetime of an item. When a feature is separated from an item token, it becomes a token in its own right and is termed a feature token. Features, which may assume different states (shades), are always associated with one or more items and are predisposed to combine with their associated item. Items and features are known collectively as token attributes.

#### 3.2 RN Transitions

Arbitrarily complex, RN transitions can fire due to the presence/absence of input tokens, according to logic functions, or as a result of some specific precondition algorithm. RN transitions are classified as either simple or complex. Retaining the relative simplicity of Petri nets, simple transition preconditions depend only on the presence of tokens in the input places. In contrast, complex transitions, represented as three-dimensional bars, depend not only on the presence of tokens in the input places but also on the attributes of those tokens.

Complex transitions perform their work in three phases. In the first phase, precondition, a transition decides whether or not it is enabled. If it is enabled, then the transition enters the timing phase, where it calculates the time at which it will fire. At the scheduled firing time, the transition enters phase three, action, and acts on the participating tokens.

**Preconditions:** The precondition rules are arbitrary—from the simple "a token in every input place" rule to some complex function of token attributes and time. A simple model may only slightly
extend the basic Petri Net rules. For example, a \( m \)-out-of-\( n \) transition modifies the basic rule to read: an \( m \)-out-of-\( n \) transition is enabled if \( m \) or more input places contain tokens. A transition of this type can drastically reduce the number of graph elements needed to model a system. Compare the Petri Net model of Figure 2 to the TMR system modeled with an \( m \)-out-of-\( n \) transition, shown in Figure 3.

Not only is this new model easier to draw, its simulation is more efficient. For the Petri Net model with inhibitor arcs, each of the 4 transitions has to examine 3 input places for a total of 12 tests. The number of required arcs and transitions, and therefore tests, grows exponentially with the number of processors. On the other hand, the \( m \)-out-of-\( n \) transition only needs to examine each place once for a total of 3 tests. The number of tests performed by the \( m \)-out-of-\( n \) transition grows linearly.

**Firing Times:** The transition firing time is the time at which the transition plans to fire and can be of arbitrary distribution—from immediate to exponentially distributed to fixed time. They can be time dependent or be a function of token state. The parameters governing firing times can be uniform over all tokens or can be individualized for each token. For example, a particular transition may be assigned an exponentially distributed firing rule with unspecified firing rate. Each individual token would then carry the rate parameter to be used by that transition. This parameter can be a function of the combination of features and their shades.

### 3.3 Places

Places do not actively modify tokens, but rather passively allow tokens to combine and merge according to a set of rules. When entities combine, they are removed from their carrier tokens and placed into a single token. Although combined, the entities retain their identity and may, at some later time, divide into separate tokens. When tokens merge, all distinction between participants is lost; the resulting token consists of a single entity.

### 3.4 Generators

Items and features are generated through a special combination of a place and transition. These combinations are called item generators and feature generators. The item generator consists of a timed transition with one input and one output. When the transition fires, it generates a unique item token with a specific birthtime and lifetime. Similarly, a feature generator generates features. In addition to a birthtime and lifetime, a feature is also assigned a disposition to a specific item.

### 3.5 RN TMR Model

As a final illustration of the suite of RN extensions, consider the RN model of the N-modular redundancy (NMR) system shown in Figure 4. The SYSGEN item generator produces items modeling NMR systems. The FGEN feature generator produces faults that combine with systems. The transition \( T_F \) determines whether or not the systems are operational. Note that the topology of the RN is no longer constrained to reflect the number of processors in a system and therefore models a NMR rather than a TMR. To simulate a TMR, the modeler need only to define four types of features and specify the rules for \( T_F \). Three of the feature types (colors) represent faults to the three processors, respectively, and the remaining feature type models a fault to the voter. The \( T_F \) precondition is true when a particular system contains two different types of processor features (faults) or one voter feature (fault).

For a TMR simulation example, consider the following scenario. At time 0, SYSGEN generates system number 1 composed of processors A, B, and C. At time 5000, FGEN generates a fault feature of type System 1/Processor A, representing a failure of processor A after 5000 hours of operation. This fault fea-
4 RN Simulator Design Issues

4.1 Division of Labor

One goal of this research was to provide several levels of simulation generality. As the levels increase in generality, they correspondingly increase in complexity. Consequently, users are expected to model at the level appropriate to their responsibilities, needs, and capabilities. The lowest level is, of course, the general-purpose high-level programming language in which the simulator is written.

The second most complex level is the specification of the rules for transitions and places. These specifications, procedural in nature, provide the core of a simulation design and can most likely be used over and over again in similar simulations. Following good modular design practices, simulation designers are expected to develop transition function libraries for classes of applications such as dependability modeling. At yet a higher level, the topology of the Rainbow Net is specified. Inherently graphical in nature, the topology describes the interconnection of the RN components and the rules to which the components are bound. It is at this level that a simulation designer would model a system architecture, for example a network of moderately-coupled star-connected processors. Finally, at the highest level, a simulation user would model specific instances and states of the system by supplying parameters to a pre-designed topology and set of rules. These parameters might include component failure rates or the number of processors in a homogeneous network.

This multi-level rule-based modeling technique allows users to easily separate invariant system rules from policies employed by the system, a feature useful for policy comparison as described by Pittl [10]. Pittl suggests that simulators accept parameters at run-time to avoid having to recompile the program for each experiment. Not only does the RN simulator implement such parameter input, it allows topologies and even the modeling rules themselves to be input during run-time.

4.2 Language Considerations

Dividing the simulation problem into levels allows different specification languages to be used for each level. This is especially beneficial because each level of RN specification is best suited to differing types of languages. The rationale behind the choice of each language follows.

The simulator executive program is written in the C programming language. C is a time and space efficient language very suitable for writing the relatively low-level routines found in simulation executives [11]. Furthermore, C provides a clean access to the Unix operating system running on the workstations on which the simulator was developed.

As previously motivated, one would use the RN topology to model the inherently graphical concepts of a system. These concepts most often are behavioral in nature, but, of course, could also reflect the physical organization of the system. Because of the spatial thought process used when designing this type of model, a graphical user interface to this portion of the simulator would be appropriate. Although still in progress, a few of its major features will be mentioned here. First, the interface will provide a "drawing program" paradigm. The user will place icons representing each of the RN components into the drawing area and then connect them with the appropriate arcs. Clicking an icon will pop up a screen in which the user may select that object’s specific function from a list of standard functions and user libraries, or from a "rules" editor. Ideally, the graphical interface will also provide simulator control and animation.

To keep the major portions of the simulator self-contained and to provide an intermediate storage medium, the graphical interface communicates the RN specification to the simulator executive via a text-based language. To allow RN specifications to be written directly in this language, it is designed to be human readable and writable.

The rules for transitions, decision nodes, and places are also written in the C programming language. Here, C was chosen for two main reasons. First, C, being compiled, is much more efficient than an interpreted home-grown language. Second, rules written in C can be separately compiled and loaded into a running simulator, thus combining the advantages of separate rules files and compiled code. In addition, C has proven to be a reasonable modeling language, not only because of its rich supply of operators, but because it is a widely known language—users do not have to learn yet another simulation language [11, 12]. Admittedly, one of the drawbacks of using the dynamically loaded rules technique is that users have the ability to crash the simulator due to a bug in their code. This hazard can
be mitigated by restricting the functions a user may invoke and enforcing strict type checking. On a brighter note, casual users, who are most likely to write buggy code, will only interact with the topology and executive levels of the simulator and not write any rules code themselves.

5 Dependability Application

To provide an example of Rainbow Net simulation, this paper now presents the results of a TMR dependability analysis. A designer can estimate or measure the reliability of a system’s components. The designer also knows how the system reacts in the presence of faults to its component parts. The purpose of developing a system dependability model is to determine the dependability of the entire system based on knowledge of the component parameters and system behavior.

Traditionally, dependability has been modeled by analytic techniques which include combinatorial methods, such as reliability graphs and fault trees, and state-based methods, such as Markov chains and basic Petri Nets. A variety of evaluation tools have been developed using these techniques; these tools, as well as their modeling methods, are discussed by Johnson and Malek in [3]. These tools have practical disadvantages. The combinatorial methods do not incorporate the concept of time, so they are inadequate for realistic dependability modeling. Although state-based methods include time, they suffer from a combinatorial explosion when attempting to model larger systems, as previously shown. In addition, even if these tools manage to model a system, their analysis may be analytically intractable and require numerical simulation. Therefore, it is most appropriate that Rainbow Nets, a technique that overcomes many of these limitations, be used to evaluate dependability.

One way to achieve dependability is to incorporate fault tolerance into a system design. Because of the complicated methods by which fault-tolerant systems handle faults, a powerful modeling technique is required to model such systems. The complex transitions of RNs help simplify the modeling of fault tolerance, thus making RNs an appropriate modeling technique [1]. As an example of a fault-tolerant design, consider the replicated processors of a multiprocessor. The inherent parallelism of fault-tolerant multiprocessors makes them especially amenable to RN modeling. The RN’s use of individual tokens with feature attributes allows increasing numbers of processors to be specified with a constant model complexity. Moreover, it will be shown that the simulation time only increases as a polynomial function of processor quantity. In contrast, completely state-based modeling techniques such as Petri Nets and Markov chains suffer from combinatorial explosion as the number of modeled processors (and thus states) increases.

5.1 TMR Systems

The Triple Modular Redundancy (TMR) system is a typical example of a fault-tolerant multiprocessor configuration. A TMR system is considered to be functioning properly if at least two of the three processors and the voter module are working. Given that the reliability of each processor is \( R \) and the reliability of the voter is \( R_V \), the system reliability is \( R_{TMR} \).

\[
R_{TMR} = [R^3 + \left( \frac{3}{2} \right) R^2(1-R)]R_V = (3R^2 - 2R^3)R_V
\]

In this formula, \( R^0 \) represents the probability that all three processors are working and the other term represents the probability that any two out of three are operating correctly [3].

The following two examples model a TMR system with and without repair capabilities. The goal of the model is to determine system dependability parameters, given the mean time to failure (MTTF) and mean time to repair (MTTR) of the components. MTTF can be measured, or if the reliability function \( R(t) \) is known, can be calculated as \( MTTF = \int_0^\infty R(t)dt \). The models assume that processor failures are independent and that failure times have an exponential distribution with parameter \( \lambda_P \), \( R(t) = e^{\lambda_P t} \). With this assumption, \( \lambda_P = \frac{1}{MTTF} \).

The examples also assume that there is a repairman for every fault and that repair times are also exponentially distributed with parameter \( \mu_P \), repair\( (t) = e^{\mu_P t} \). With this assumption, \( \mu_P = \frac{1}{MTTR} \).

For these examples, let \( MTTF_P = 10000 \) hours and \( MTTR_P = 100 \) hours.

5.2 TMR System Without Repair

5.2.1 Theoretical Analysis

Using the formula for TMR system reliability and the processor reliabilities calculated above, \( R_{TMR} \) is given as follows.

\[
R_{TMR} = 3e^{-2\lambda_P t} - 2e^{-3\lambda_P t}
\]

Therefore, the system mean time to failure is calculated as \( MTTF_{SYS} = \frac{3}{\lambda_P} \). With the given component parameters, \( MTTF_{SYS} \) is 8333 hours.

5.2.2 Simulation Analysis

A TMR system without repair was simulated using both of the TMR models developed earlier. These models are shown in Figures 4 and 3. All the models produced results that were close to the theoretical \( MTTF_{SYS} \) of 8333 hours. For example, two runs of 2000 repetitions yielded \( MTTF \) of 8609 and 8381 hours. These two figures differ from the theoretical result of 8333 hours by 0.432% and 0.576%, respectively.
5.3 TMR System With Repair

5.3.1 Theoretical Analysis

When a TMR system is repairable, either Markov analysis or simple approximations can be used to calculate availability. SHARPE, an analytic dependability evaluation tool from Duke University [13], was used to calculate the steady state availability of a repairable TMR system. The result was \( MTTF_{SYS} = 0.999708 \), i.e., the system is expected to be available 99.9708% of its lifetime. In a \( k \)-of-\( n \) system, steady state system availability can be approximated using the following formulas.

\[
MTTR_{SYS} = \frac{MTTR}{n - k + 1}
\]

\[
MTTF_{SYS} = MTTF \left( \frac{MTTF}{MTTR} \right)^{n-k} \left( \begin{array}{c} n \\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ k \end{array} \right)
\]

\[
AVAIL_{SYS} = \frac{MTTF_{SYS}}{MTTF_{SYS} + MTTR_{SYS}}
\]

Using these formulas, availability was calculated as \( AVAIL_{SYS} = 0.999700 \).

5.3.2 Simulation Analysis

The RN model shown in Figure 5 was used to determine dependability results for a repairable TMR system. The transitions are the same as in the non-repairable system, except for the addition of repair transitions \( TDR \) and \( TCR \) and restart transition \( TR \). Transition \( TCR \) models the repair of processors while the system is still operational, whereas \( TDR \) models processor repair while the system is down. \( TR \) fires when enough processors have been repaired to meet the functionality requirement—at least 2 out of 3 processors working, in the TMR case. Note that the precondition of \( TR \) is merely the logical complement of the precondition of \( TV \).

Token Structure: The entire TMR system is represented by one item token. Faults to processors are represented by features with a disposition to combine with the one item token. There are three types of features corresponding to faults on the three processors. Additional fields in the item hold statistics for use in availability calculations.

Rules: The voter transition follows a 2 out of 3 rule; it fires when there are two differing faults present in the item. The repair transitions, \( TDR \) and \( TCR \), are scheduled to fire when a \( Processor\_Faulty \) feature is detected in a TMR item token. When the transition fires, the feature that caused the true precondition is destroyed, that is, the fault is repaired and removed from the system. Note that these rates are local firing rates, so a repair may have to be rescheduled. This rescheduling happens in the following scenario. Consider an operational system with one faulty processor. At the time that the fault for that processor was attached to the system, transition \( TCR \) scheduled a repair for it. If a second processor fails before the repair transition fires, then the system token moves into the \( System\_Failed \) place. At this time, transition \( TDR \) schedules repairs for all fault features. The scheduled firing of transition \( TCR \) now becomes invalid, even if the system token happens to return to the \( System\_Operational \) place. When the scheduled firing time arrives, the transition will recheck its precondition function, find that the token state has changed, and abort the firing.

Both the voter and restart transitions update the item statistics fields. Each time a system is restarted, the restart transition prints out the amount of time in the failed state. These numbers are used to calculate the mean time to repair. Likewise, the voter transition prints out system failure times, used in calculating \( MTTF \). Since the steady state availability is only valid at the end of the system's lifetime, it is only calculated at the end of the simulation by the formula.

\[
AVAIL_{SYS} = \frac{1}{MTTF} \times \frac{AVAIL_{SYS} \times MTTF_{SYS} + MTTR_{SYS}}{MTTF_{SYS} + MTTR_{SYS}}
\]

Simulator Input: The RN simulator input for this model is shown in Figure 6. The \( TOPOLGY \) block specifies the topology that is graphically depicted in Figure 5. The \( RNMODEL \) block binds the transition rules to the topological elements. The rules themselves are not presented here. Finally, the \( JOB \) block defines parameters used to run the simulation.

Simulation Results: Table 1 displays the results of a single simulation iteration from the above input file and model. To validate the simulator results, they are compared to the theoretical approximations given in Section 5.3.1. The simulation result of 0.999704 differed from the SHARPE result of 0.999708 by 1.37%.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>RN Simulator</th>
<th>Formulas</th>
<th>% Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>( MTTF_{SYS} )</td>
<td>160082</td>
<td>169667</td>
<td>1.45</td>
</tr>
<tr>
<td>( MTTR_{SYS} )</td>
<td>50.095</td>
<td>50.095</td>
<td>0.19</td>
</tr>
<tr>
<td>( AVAIL_{SYS} )</td>
<td>0.999704</td>
<td>0.999700</td>
<td>1.33</td>
</tr>
</tbody>
</table>

Table 1: Summary of Repairable TMR Analysis
TOPOLOGY tnrprop;
DEFAULT FUNCTION rules/defrules.o; /* default RN rules */
PLACE
ICENVGEN (TGEN);
FICENVGEN (TGEN);
ICENVGEN [low=0, high=10000] (WORKING);
WORKING:comb (TVOTER, TCONCREP);
FAILED (RESTART, TDOWNREP);
TRANSITION
TGEN igen (WORKING);
TGEN igen [low=0, high=10000] (WORKING);
TVOTER (FAILED);
RESTART (WORKING);
TDOWNREP:repairs [time=200, rep=100.0] (FAILED);
TDOWNREP:repair [time=200, rep=100.0] (WORKING);
END tnrprop;

RMODEL tnrmod;
APPLY TOPOLGY tnrprop;
FUNCTION rules/rules.o; /* special rules */
FUNCTION rules/tmgarmc.o; /* generator rules */
END tnrmod;

JOB run1;
APPLY RMODEL tnrmod;
REPEAT 100; /* run 100 simulations */
INITIALIZE ();
RUN 50000000; /* Simulate for 50 million hours */
END run1;

Figure 6: Input File for TMR Repair Model

6 Conclusions

Although formal analysis has the advantage of closed form solutions and accurate results, simulation is often required when modeling realistic systems. Johnson's Rainbow Nets are a good candidate modeling technique for such systems because they incorporate both graphical and state-variable based modeling techniques. The Rainbow Net Simulator provides a flexible and effective implementation of the RN concept. It capitalizes on RNs multiple modeling paradigms yielding a natural and convenient way of expressing system behavioral models.

To demonstrate the feasibility of both Rainbow Nets and their simulator, a TMR dependability problem was modeled and simulated. Previous models of this system proved to be exponential in complexity. In these models, specification of system behavior becomes increasingly complicated as the size of the system increases. In contrast, the Rainbow Net model was shown to be constant in specification complexity, linear in simulation space, and polynomial in simulation time. These examples illustrate the viability of Rainbow Nets as a complexity-reducing modeling technique. Furthermore, these examples demonstrate the feasibility and efficiency of the Rainbow Net simulator. Rainbow Nets and their simulator promise to yield previously unobtainable results not only in the field of dependability analysis, but with any system amenable to discrete event modeling.

Acknowledgement

The authors would like to thank Allen Johnson, Jr. and Miroslaw Malek for their valued input to the design of the Rainbow Net Simulator.

References