ORNL/TM-2020/1554

A software package for modeling and simulating fault graphs

James Nutaro

August 4, 2020

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2020/1554

Computational Sciences and Engineering

A software package for modeling and simulating fault graphs

James Nutaro

Date Published: August 4, 2020

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ORNL	Oak Ridge National Laboratory
adevs	A Discrete EVent System simulator

ABSTRACT

This report describes a novel fault graph modeling language and a simulation tool for executing models specified in the language. The modeling language has three primary features that distinguish it from similar reliability analysis tools. These are (1) a random variable modeling several distinct outcomes of a single fault; (2) chains of faults in which one fault triggers another; and (3) time to fail sampled from probability distributions including positive normal, exponential, Weibull with a minimum, or immediate. These features are motivated by their use in a historical analysis of centrifuge reliability.

1. Introduction

There exist many software packages for conducting reliability studies and extensive surveys of the numerous methods for fault tree modeling and analysis; see, e.g., the reviews by Ruijters and Stoelinga [2015], Baklouti et al. [2017]. Nonetheless, it is possible to encounter unique analysis tasks requiring a collection of capabilities that are difficult to find in a single tool. It is this circumstance that motivates our new simulation program. The specific modeling constructs in our tool originate from an historical analysis of centrifuge reliability that had three unusual aspects.

1. There may be multiple outcomes possible for a single instigating event. For example, the failure of a motor may lead to a controlled shutdown or loss of machine with specific probabilities assigned to each outcome.
2. Failures can occur through chains of events. In a chain, the completion of one event sets in motion the next and each such event has a distinct time to occurrence.
3. Time to failure is modeled by normal and exponentially distributed random variables, and Weibull random variables with the addition of a minimum time to fail.

Item 1 in particular leads to the greatest deviation from a typical fault tree by introducing multiple end points, each with a distinct probability of occurrence. Instead of a fault tree, models constructed with our simulation program are directed, acyclic fault graphs. One such fault graph is shown in Figure 1.

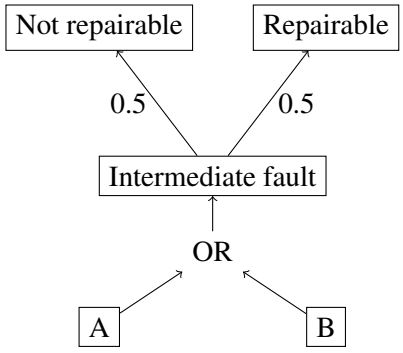

Figure 1. A fault graph with novel branching and intermediate events.

The rectangular nodes are potential faults, and each fault is assigned a random variable that models its time to occurrence. Only faults without inbound edges are are active at the start, and a simulation begins with each initially active fault sampling its time to occurrence. When this time expires, the fault generates an

event on one of its outward arcs. If there is a single outward arc, then that arc carries the event. When there are several outward arcs, then one arc is chosen at random according to the probabilities assigned to each.

A fault with an incoming arc is activated when it receives an event on any incoming arc. The time to occurrence is sampled when the first such event occurs, and the fault generates an event when this time expires. The outgoing event follows one outward arc as described above. Only one such event will be produced, and this event will be caused by the first incoming event on any arc. Subsequent incoming events have no effect.

Gates appear in the graph when some logical condition on the incoming events must be satisfied before the next fault is activated. The gate generates an event at the instant its logical condition is satisfied. For the OR gate in Fig. 1 the condition is that at least one event has arrived. After a gate's rule has been satisfied and the event propagated, the gate becomes unresponsive to new input and will not generate a second event.

Shown below is one possible sequence of events ending in system failure for the model illustrated in Fig. 1.

1. Fault A occurs at time 1.
2. The OR gate propagates this event to the Intermediate fault node, causing it to become active with a time to fail of 1 unit of time.
3. At time 2 the Intermediate fault occurs.
4. The left branch is selected causing the Not repairable fault to become active.
5. The Not repairable fault occurs immediately and the simulation terminates.

This particular sequence of events produces the final failure at time two. Of the two initially active faults A and B, A occurs first. The OR gate is satisfied and generates an event that activates the Intermediate fault. Meanwhile, B may occur but its event will be discarded by the OR gate, which has already produced an event. Upon becoming active, the Intermediate gate selects its time to fail and generates an event at the appropriate time. Of the two possible outward arcs, the left is chosen by chance. The Not Repairable fault is activated and selects zero for its time to occurrence. Consequently, this fault occurs immediately and the simulation ends.

2. Preparing simulation input data

Input to the simulation is a text file that describes the fault graph and command line arguments specifying how many simulation runs to execute and what data to report. A fault graph is constructed from three types of objects:

1. Events, which model time to occurrence of a fault;
2. Processes, which model intermediate faults, their input, and the branching probabilities for their output; and
3. Gates, which appear in the input description of a process.

An event is described with a single line in the input file. This line begins with the keyword `event`, followed by a name for the event and the random variable that models time to occurrence. Four types of

random variables are supported. These are `immediate`, `exponential`, `normal`, and `weibull`. The type name is followed by parameters for the specific distribution.

The `immediate` distribution has no parameters. An event of this type occurs immediately upon being activated. Hence, its time to occurrence is zero.

The `exponential` distribution is characterized by a rate parameter, which specifies the mean number of occurrences per unit time. The rate inverse is mean time to occurrence. During a simulation, the time to occurrence is sampled from an exponentially distributed random variable with the indicated mean rate.

The `normal` distribution is characterized by a mean and standard deviation. Time to occurrence is generated by sampling from the specified normal distribution. A draw less than zero will cause the random variable to be sampled again so that the time to occurrence is non-negative.

The `weibull` distribution has a lifetime parameter, a shape parameter, and a minimum time to occurrence. During a simulation, time to occurrence is drawn from an appropriate Weibull distribution plus the given minimum time.

Specific examples of syntax for declaring an event are given below.

```
event MyImmediateEvent immediate
event MyExpEvent exponential rate 1
event MyNormalEvent normal mean 1 stdev 1
event MyWeibullEvent weibull shape 1 lifetime 1 minimum 1
```

By defining an event you create a template rather than a unique instance. A distinct instance of the event is created each time it appears in a process description.

A process comprises its name, input, gates, and output. Shown below is an example of a process description.

```
process Process
    outcome MyImmediateEvent 0.5
    outcome MyExpEvent 0.5
    and
        MyNormalEvent
        MyWeibullEvent
        or
            MyExpEvent
            MyNormalEvent
        end_or
    end_and
end_process
```

The name of this process is Process. Its input are four distinct instances of the previously defined events. Occurrences of the input events are combined with an `and` gate and `or` gate. When the `and` gate is activated, two outcomes are possible. With probability 0.5 an instance of MyImmedateEvent is generated, and with probability 0.5 an instance of MyExpEvent is generated. The fault graph corresponding to this process is shown in Fig. 2.

Below is a complete simulation program constructed from the above process and event definitions.

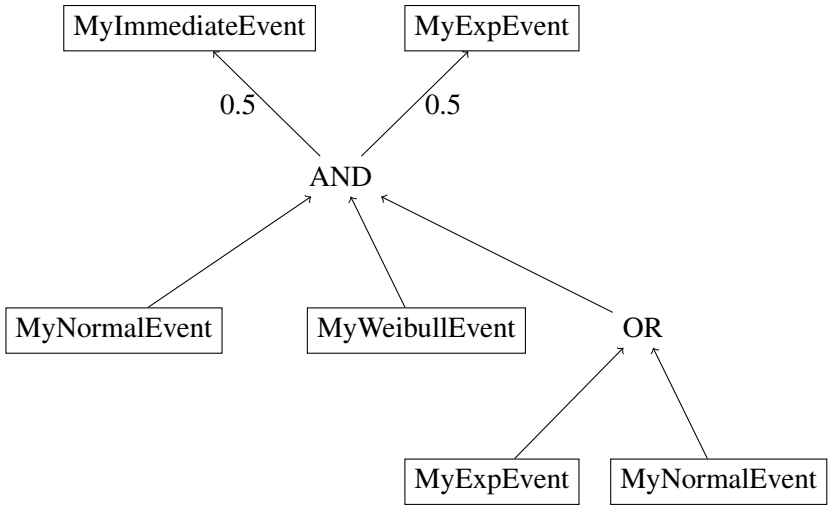

Figure 2. A process with two possible outputs and a four distinct inputs filtered through two gates.

```
event MyImmediateEvent immediate
event MyExpEvent exponential rate 1
event MyNormalEvent normal mean 1 stdev 1
event MyWeibullEvent weibull shape 1 lifetime 1 minimum 1

process Process
    outcome MyImmediateEvent 0.5
    outcome MyExpEvent 0.5
    and
        MyNormalEvent
        MyWeibullEvent
        or
            MyExpEvent
            MyNormalEvent
        end_or
    end_and
end_process
```

To execute this program once we run the command

```
fg -t -n 1 model.dat
```

where `model.dat` is the file that we have typed our model into, the switch `t` tells the simulator to save trace data, and `n` is the number of replications we wish to run. We describe one possible execution trace for this model. At the simulation start, each of the four leaf events determines its time to fire. At time 0.8 the `and` gate receives its first input from the first `MyNormalEvent` listed in the process description. The gate must receive all of its input before activating, and so it produces no event at this time.

At time 0.18 the `or` gate receives input from the leaf instance of `MyExpEvent`. The `or` gate requires only one input to become active, and so it generates an event for the `and` gate. Now the `and` gate has received two of its three input. The `or` gate, having been activated, will not generate any more events.

The final input for the and gate arrives from the MyWeibullEvent at time 1.8. Now all three inputs have arrived and the gate triggers one of the two outcomes. In this case, the outcome MyExpEvent is activated. At time 3.5 this final event occurs and the simulation terminates. The trace file generated by the simulator for this particular outcome is listed below.

```
0.0788023 MyNormalEvent @ Process.AND
0.177589 MyExpEvent @ Process.AND.OR
0.177589 Process.AND.OR @ Process.AND
1.75601 MyWeibullEvent @ Process.AND
1.75601 Process.AND @ Process
3.45871 Process.MyExpEvent
```

In this trace file we see the first input to the and gate on the first line. The activation of the or gate by MyExpEvent is listed on the second line. The third line shows the consequent input for the and gate.

Fault graphs can be stitched together by using the output from one process as input to another. An event originating from a process is indicated by originating process's name, a dot, and then the event name. For example the event MyImmediateEvent generated by process Process has the identifier Process.MyImmediateEvent.

When constructing a graph from parts there may be dangling outcomes; that is, outcomes not connected to a gate or subsequent fault. A dangling outcome that is activated will generate an event, but this event does not terminate the simulation. If a dangling outcome should be treated as a failure of the system, then the designation outcome must be replaced with terminal.

2.1 Putting it all together

Consider a room illuminated by two light bulbs and a skylight. The room performs some essential function and must remain illuminated at all times. During the day, the skylight provides sufficient light, but a failure of the electrical light still requires a repair before sunset. We'll call this undesirable outcome "Sunlight only". The electrical light may also fail after sunset, or without sufficient time to repair it before sunset. We'll call this undesirable event "Dark".

The electrical light can fail in three ways: both bulbs burn out, the switch that turns on the bulbs fails, or the electrical power fails. Failure of the electrical power happens in two stages: first the main power fails and then the backup power fails. To make things interesting, we'll include the possibility of a carbon monoxide leak when the backup power is operating. A leak forces the room to be evacuated even if it isn't dark. This outcome is called "Evacuated".

The complete fault graph for this model is shown in Fig. 3. Each rectangular fault block has a corresponding event declaration in the simulation input file. Events that terminate the simulation happen immediately, as do the intermediate events that are merely descriptive. These immediate events are "Sunlight only", "Dark", "Evacuate", "Engage backup", and "Both bulbs fail". All other events have a non-negative time to occurrence sampled from the appropriate probability distribution. The event declarations in our model are shown below. Text bracketed by the # character is a comment to be ignored by the model compiler.

```
event Evacuate immediate # Escape carbon monoxide #
```

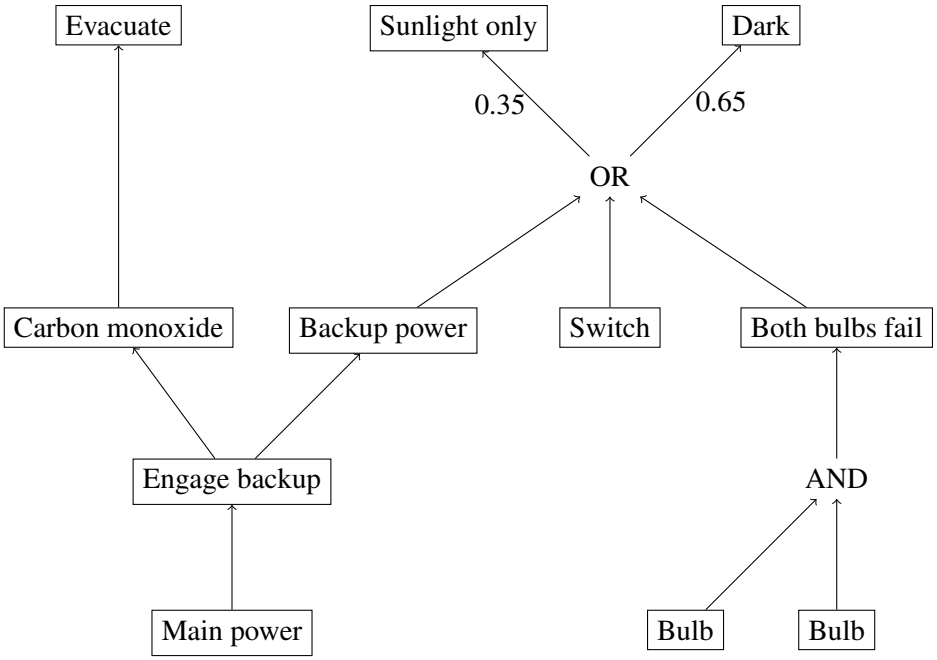

Figure 3. Fault graph for a lighting system comprising a skylight and two electrical bulbs.

```
event Sunlight_only immediate # No electrical light, but the sun is up #
event Dark immediate # No electrical light, sun is down #
event Both_bulbs_fail immediate # Both bulbs burnt out #
event Engage_backup immediate # Start backup power #
event Backup_power exponential rate 10 # Generator failed or out of fuel #
event Main_power normal mean 1 stdev 1.1 # Primary power lost #
event CarbonMonoxide weibull shape 1 lifetime 1E3 minimum 1 # Gas leak #
event Switch exponential rate 0.1E-3 # Light switch won't operate #
event Bulb exponential rate 1E1 # Light bulb burns out #
```

The graph shown in Figure 3 is built by assembling processes. We will create a process for the light bulbs, the electrical power including backup electrical power, carbon monoxide, and to encompass the illumination outcomes. Of course, other organizations of the graph into processes are possible. The part of the program defining processes is shown below. Note that the "Evacuate", "Sunlight only", and "Dark" events are `terminal` outcomes because these are failures of the system.

```
# Light bulbs fail #
process Bulbs
    outcome Both_bulbs_fail 1
    and
        Bulb # Bulb 1 #
        Bulb # Bulb 2 #
    end_and
end_process

# Main power #
```

```
process Power
    outcome Engage_backup 1
    and # Could be or but a gate is required #
        Main_power
    end_and
end_process

# Backup power fails #
process Backup
    outcome Backup_power 1
    and
        Power.Engage_backup
    end_and
end_process

# Gas #
process Gas
    outcome CarbonMonoxide 1
    and
        Power.Engage_backup
    end_and
end_process

# Evacuate! #
process Leave
    terminal Evacuate 1
    and
        Gas.CarbonMonoxide
    end_and
end_process

# Lighting fails #
process Lighting
    terminal Dark 0.65
    terminal Sunlight_only 0.35
    or
        Switch
        Backup.Backup_power
        Bulbs.Both_bulbs_fail
    end_or
end_process
```

2.2 Using simulation output

Every execution of the `fg` simulation program produces a file called `survival.txt`. Each line in the file describes one simulation run and there are five columns. The first column is the time at which the final, terminal event occurred. This is the time to fail for that simulation run. The second column is the percentage of simulation runs with time to fail greater than the current line. The third column is the number of the simulation run so that the percentage reported in the previous column is this number divided by the total number of simulation runs. The fourth column is the first event in the path through the fault graph that ended in the terminal event. This is the root cause of the failure. The final column is the terminal event that ended the simulation run. The entries in the file are sorted by time to fail in ascending order.

A sample of `survival.txt` generated by the model listing in Sect. 2.1 is shown below. This output was create with the command

```
fg -n 10 model.dat
```

where the flag `n` tells the program to run the model ten times and `model.dat` is the text file containing our model description.

```
0.0213127 0.9 9 Switch Sunlight_only
0.278945 0.8 8 Switch Dark
0.419607 0.7 7 Switch Dark
0.506423 0.6 6 Switch Dark
0.756008 0.5 5 Switch Dark
0.850469 0.4 4 Bulb.Bulb Sunlight_only
0.92752 0.3 3 Switch Sunlight_only
0.929913 0.2 2 Main_power Evacuate
1.11681 0.1 1 Main_power Dark
1.61516 0 0 Switch Dark
```

There are ten lines in the file, one for each of the requested simulation runs. The quickest failure happened after just 0.0213 units of time and the longest time to fail is 1.62 units of time. Seven of the ten failures have the switch as root cause; failure of the main power is the root cause in two cases; and burnout of both bulbs is the root cause in the remaining case. In six of the ten failures, the room is dark. In three instances only sunlight provides illumination. Carbon monoxide gas forces an evacuation just once.

3. Simulation internals

Within the simulation program, all elements of a fault graph are realized by components within a discrete event simulation model. The simulation portion of the program is implemented using A Discrete EVent System simulator (adevs*); see Nutaro [2011]. The model parser is realized simply with C++ code, though future versions of the software are anticipated to use a proper grammar and robust parser generator.

*Available online @ `https://sourceforge.net/projects/adevs/`

3.1 Definition of model components

The fundamental elements of the model can be given compact definitions in terms of the Discrete Event System Specification; see Zeigler et al. [2018] and Nutaro [2011]. There are three of these fundamental elements: leaf fault, process, and gate. The elements are organized into a directed graph precisely as indicated in the previous fault graph illustrations. The output of a leaf fault is coupled to the input of a gate. The output of a gate may be coupled to another gate or to a process embodying the intermediate faults that the gate may instigate. The output of the process may then be coupled to gates further along the graph.

For the purpose of specifying behavior we take input and output to be integer numbers that indicate which incoming or outgoing arc carries the event. The random variable **t** indicates time to occurrence and the random variable **a** indicates the outgoing arc. The number of items in the input bag x for a model is $|x|$.

The integer output of a model acts as a port in the coupling graph; see, e.g., the digraph models described in Zeigler et al. [2018]. In practice, additional information may be attached to the data produced on that port. For instance, to track the instigating event or the chain of events that have occurred. This additional information may be important for record keeping and reporting results, but it is not necessary for understanding how the model operates in time. Hence, this additional information is omitted from the model descriptions.

3.1.1 Leaf fault

A leaf fault has a state variable σ that is the time to occurrence and its initial value is $\sigma = \mathbf{t}$. The leaf is input free and has a single outgoing arc. Hence, it may be simply defined by

$$\delta_{int}(\sigma) = \infty \tag{1}$$

$$ta(\sigma) = \sigma \tag{2}$$

$$\lambda(\sigma) = 1 \tag{3}$$

3.1.2 Gate

In its present form, the model supports only AND gates and OR gates. For these two gates, it is sufficient to maintain a count p of inputs pending before the gate generates an event. For an OR gate $p = 1$ and for an AND gate p equals the number of incoming arcs. When $p \leq 0$ the gate generates an output, and the state variable f records whether this output has occurred. At the start $f = \mathrm{T}$ and upon producing output $f = \mathrm{F}$. The behvior of the model is defined by

$$\delta_{int}((p, f)) = (p, \mathrm{T}) \tag{4}$$

$$\delta_{ext}((p, f), e, x) = (p - |x|, f) \tag{5}$$

$$\delta_{con}((p, f), x) = (p - |x|, \mathrm{T}) \tag{6}$$

$$ta((p, f)) = \begin{cases} 0 & p \leq 0 \text{ and } f = \mathrm{F} \\ \infty & \text{otherwise} \end{cases} \tag{7}$$

$$\lambda((p, f)) = 1 \tag{8}$$

3.1.3 Process

A process has a single input that it receives from its uppermost gate. For each outgoing arc $1, \ldots, n$, there is a random variable $\mathbf{t}_1, \ldots, \mathbf{t}_n$ describing the time to an output on that arc. The process has a state variable a which is the selected outgoing arc and time to next event σ for the selected arc.

As with the gate, the state variable f is equal to T if the process has generated an output and F otherwise. Initially, f = F and the time to occurrence $\sigma = \infty$. The variable $a' = \mathbf{a}$ is an arc selected at random and $t'_a = \mathbf{t}'_a$ the time until the arc generates an event, thereby propagating a fault.

$$\delta_{int}((a, f, \sigma)) = (a, \mathrm{T}, \infty) \tag{9}$$

$$\delta_{ext}((a, f, \sigma), e, x) = \begin{cases} (a, f, \infty) & f = \mathrm{T} \\ (a', f, t'_a) & f = \mathrm{F} \text{ and } \sigma = \infty \\ (a, f, \sigma - e) & \text{otherwise} \end{cases} \tag{10}$$

$$\delta_{con}((a, f, \sigma), x) = \delta_{int}((a, f, \sigma)) \tag{11}$$

$$ta((q, f, \sigma)) = \sigma \tag{12}$$

$$\lambda((a, f, \sigma)) = a \tag{13}$$

3.2 Fault graph generation

The parser makes a single pass through the input file, processing each event definition, process, and gate as they are discovered. When a line defining an event is found, the corresponding data is stored in a table. When a process is discovered, a process model, as described in Sect. 3.1.3, is instantiated and stored in a table. A list of outcomes and their parameter values are attached to the process.

Having done encountered a process and stored it in the table, the first gate in that process is parsed. A gate of the appropriate type is created, as described in Sect. 3.1.2, and this gate is coupled to the input of the process. If the gate is an OR gate, then $p = 1$. Otherwise, p is increment each time we encounter a child of the gate.

When a child of the gate is an basic event, a leaf event model is created, as described in Sect. 3.1.1, and this leaf event is coupled to the input of the gate. When a child of the gate is the output of a process, we extract the appropriate process instance from the table of processes and the appropriate output port is coupled to the input of the gate. If the child is a gate, we recursively call the procedure for parsing a gate.

The end result of this process is a flat digraph model; see, e.g., Zeigler et al. [2018]. One example of a digraph model is shown in Figure 4. The program text that encodes this model is listed below.

```
event EventB exponential rate 1
event EventA exponential rate 1

process ProcessB
   outcome EventB 0.5
   outcome EventA 0.5
   or
       EventA
```

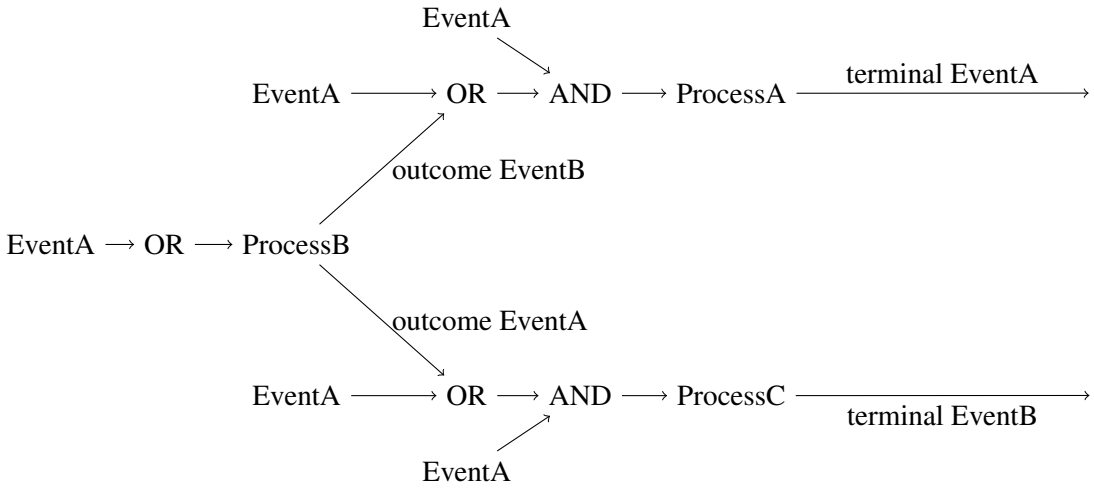

Figure 4. A digraph model generated by the recursive parsing algorithm.

```
   end_or
end_process

process ProcessA
    terminal EventA 1
    and
        EventA
        or
            EventA
            ProcessB.EventB
        end_or
    end_and
end_process

process ProcessC
    terminal EventB 1
    and
        EventA
        or
            EventA
            ProcessB.EventA
        end_or
    end_and
end_process
```

4. Designing experiments for model validation

Suppose we wish to gather evidence of a model's validity. To do so we will operate N instances of the system simultaneously for a time T. When a unit fails, it is replaced immediately. The experiment stops at time T. By using the survival function $s(t)$ stored in the `surival.txt` file produced by the simulation we may calculate the likelihood $P(m; N, T)$ of seeing exactly m failures in a simulation of this experiment.

It is typical when modeling reliability to be conservative so that the number of failures seen in practice is less than what is anticipated by the model. When true, the time to fail observed in the real experiment will generally be less than what is seen in the simulated experiment. On the other hand, if the real system performs much worse than expected, then we will see more failures and a smaller time to fail than is indicated by the model.

To make this precise, let $0 < \alpha < 1$ quantify "much worse". We would like to be able to detect a circumstance where the real survival curve satisfies $r((1-\alpha)t) \leq s(t)$. That is, the lifetime of the real system is shorter by a factor α than the lifetime anticipated by the model. If we assume $r((1-\alpha)t) = s(t)$ then the data in `survival.txt` with time scaled by $1-\alpha$ may be used to calculate the probability $P^*(m; N, T)$ of observing exactly m failures of this poorly performing machine.

The hypothesis that the model is valid will be rejected if the number of failures observed in the real experiment is sufficiently unlikely. If β quantifies "sufficiently unlikely" and we observe m failures, then

$$\sum_{k=m}^{\infty} P(m; N, T) < \beta \tag{14}$$

is sufficient to reject the model. In statistical terms, β is the likelihood of committing a type I error by rejecting the model when it is valid. There is a smallest number of failures m^* for which Eqn. 14 is satisfied for given N, T.

To avoid accepting the model when it is invalid, we would like to ensure that the experiment produces fewer than m^* failures with a likelihood γ. Hence, if the machine actually performs poorly, where poor is quantified with α above, then with probability $1-\gamma$ we observe sufficient failures to reject the model. Therefore, an acceptable experiment design has N, T such that

$$\sum_{m=0}^{m^*} P^*(m; N, T) \leq \gamma \tag{15}$$

Given $s(t)$, α, β, and γ our aim is to select suitable N and T, with shorter and smaller experiments being more desirable than longer and larger experiments. A map of acceptable N and T pairs can be created with simulations. Given feasible limits for N and T we sample the space inside that limit marking each point as acceptable or not acceptable.

The acceptability of a point is determined as follows. First, we simulate the experiment using $s(t)$ to determine time to fail and use this data to estimate the cumulative distribution function for the number of failures. We select m^* as the smallest for which the cumulative probability is greater than $1-\beta$.

Given this m^*, we simulate the experiment again with $r((1-\alpha)t)$ to calculate the cumulative distribution function for the number of failures. If the probability of seeing less than m^* failures is greater than γ then the point N, T is unacceptable. Otherwise, we accept it.

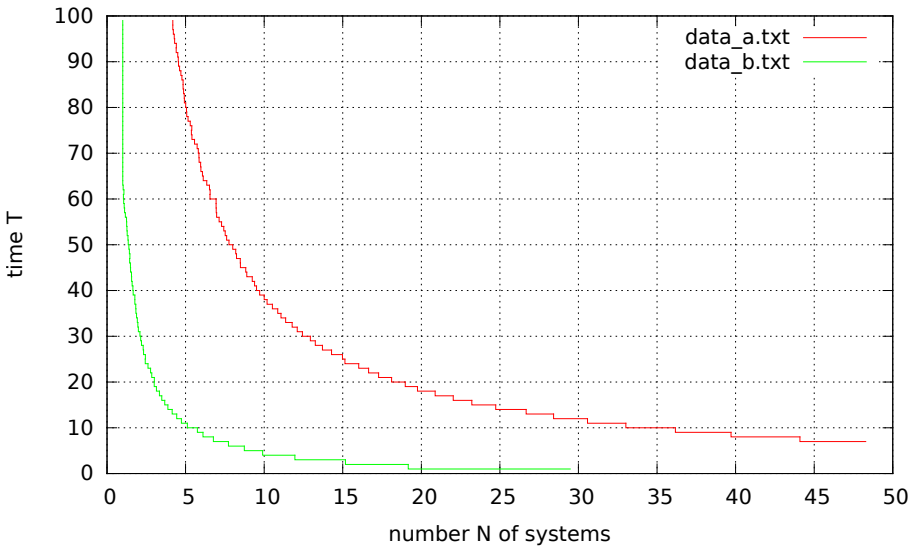

Figure 5. Output of the experiment design program.

The program `size` performs the analysis described above. It accepts eight command line arguments. In order, these are α, β, γ, the upper and lower limits of N (inclusive), the upper and lower limits of T (inclusive), and the number of sample points to use when estimating the probability distribution of the number of failures. For the sample model in Sect. 2.1 we performed the analysis twice. The first analysis is executed with the command

```
size 0.1 0.025 0.025 1 100 1 50 20000 | tee data_a.txt
```

which uses $\alpha = 0.1, \beta = \gamma = 0.025$, $N \in [1, 100]$, $T \in [1, 50]$, and 20,000 samples of the failure count distributions. The results are stored as two columns in the file `data_a.txt`. The first column contains N and the second columns is the smallest T that satisfy the experiment criteria α, β, and γ. The second analysis is executed with the command

```
size 0.2 0.05 0.05 1 100 1 50 20000 | tee data_b.txt
```

which is identical to the first except for $\alpha = 0.2$ and $\beta = \gamma = 0.05$ with the data stored in file `data_b.txt`. This experiment is less sensitive (larger 'badness' α) and less definitive (larger probabilities β and γ of an inconclusive experiment) but should require less time and fewer systems.

The product of this analysis is plotted in Fig. 5. Acceptable points are to the right of the curve of interest. As expected, the less sensitive, less conclusive experiment B has an acceptable boundary to the left of the more exacting experiment A. Hence, experiment B requires fewer systems and less time than experiment A. In both cases, the experimenter is offered a trade between the number of systems and the length of the experiment to be run.

References

A. Baklouti, N. Nguyen, J. Choley, F. Mhenni, and A. Mlika. Free and open source fault tree analysis tools survey. In *2017 Annual IEEE International Systems Conference (SysCon)*, pages 1–8, 2017.

James Nutaro. *Building software for simulation*. Wiley, 2011.

Enno Ruijters and MariÃńlle Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review*, 15-16:29 – 62, 2015.

Bernard Zeigler, Alexandre Muzy, and Ernesto Kofman. *Theory of Modeling and Simulation, 3rd edition*. Academic Press, 2018.