Calculating Reliability of Complex Systems Programmatically

Authors: Luke McDuff, Brad Van Wick, Tu Vũ

Abstract

Predicting the reliability of systems containing multiple components and sub-systems becomes challenging when the structures move beyond the scope of basic series and parallel combinations. Techniques for effectively predicting reliability of complex systems can be accurately performed by modeling these structures within the framework of computer software. For structures that contain many paths and extensive working combinations, and for system reliability predictions to be dependable, they must be based on data mined from an a array of test methods and programs. As a proof of concept, we developed and implemented a single test program that can predict theoretical system reliability for the general case, a system with N nodes of varying p reliability. A range of information is produced by the author's software including graphical representations of the system modeled, a detailed mathematical formula file of all calculations performed, and an overall prediction of the structure's reliability. As a proof of concept, the algorithm was designed not to perform operations extremely efficiently, but to display a clear working demonstration in order to provide a better understanding of the algorithm in a readable and understandable format. With this consideration, the algorithm performs accurately but slowly, yielding a solid method for understanding complex system modeling.

Modeling Reliability Systems as a Graph

Programming Language

For this implementation, Python 2.7 was chosen as the programming language to programmatically model connected systems and predict their reliabilities. Python was selected because of its straightforward syntax, which allowed for the majority of time to be focused on concept development and algorithm readability, and for its large selection of quality third-party modules and libraries.

Modules

To enhance system performance and usability a variety of modules was selected. NetworkX, a sophisticated graph modeling module, was selected to model input systems as directed graphs. Representing reliability systems a directed graph allowed us to experiment with known graph algorithms, and to better visualize the structures. MatPlotLib,a graphical plotting module,was selected to interface with the NetworkX module and plot the graphs distinctly. Graphic representations of connected systems allowed for not only enhanced visual understanding of concepts, but ensured verification of the program's calculations using traditional by-hand methods.

Algorithm

The academic community that studies complex system reliability has developed many algorithms that can accurately predict and analyze connected structures. Methods such as the Decomposition Method, Tie-Set and Cut-Set Method, Boolean Truth Table combined with Reduction Method, Path-Tracing Method and other variations such as the Factoring Method are employed in the task of predicting system reliability.[1] Each method employs a unique algorithm to predict system reliability. Research into improving these algorithms is a rich area of discourse and often seeks to combine different aspects of said methods in search of developing more efficient techniques of prediction. While not in the scope of this research, special systems that include unique structure patterns such as ladders, multi-state systems, systems where reliability is predicted over a set time, or system models with built-in redundancy require different algorithms to accurately model and predict these system reliabilities.

Event Space

An algorithm which solves complex systems with an effective and concise approach is the Event-Space Method. The Event-Space Method is, "based on listing all possible logical occurrence of the system."[2] In this scenario, all system components (or in author's case, nodes) are treated as though they are operational initially, and then allowed to fail in combinatorial order: individually, two at a time, three at a time and so on.[3] The system reliability can then be predicted by the union (probability law of addition[4]) of all successful occurrences. It is clear that the total number of either operational or failing occurrences is determined by the number of components (nodes) in the system. Since each node has the ability of being in either one of two states, the number of overall occurrences can be determined with the formula:

$$2^N,\ where\ N\ is\ the\ number\ of\ components\ (nodes)$$

$$For\ example,\ if\ N\ =\ 5:$$

$$2^5 = 32\ occurrences$$

[1] Elsayed, Elsayed A.. *Reliability Engineering (2nd Edition)*. Hoboken, NJ, USA: John Wiley & Sons, 123-124 (2012).
[2] Elsayed. *Reliability Engineering (2nd Edition)*, 123-124.
[3] Elsayed. *Reliability Engineering (2nd Edition)*, 123-124.
[4] Bertsekas, Dimitri P. and Tsitsiklis, John N.. *Introduction To Probability*. Belmont Massachusetts: Athena Scientific, 20 - 25 (2008).

Another way to visualize this using the combination formula,
There is only one occurrence with no failure:

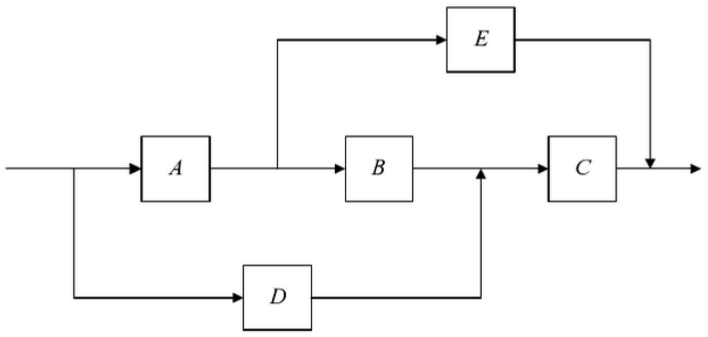
$$\left\{\binom{5}{0} = 1\right\}$$[5]

There are five occurrences with one failure:

$$\left\{\binom{5}{1} = 5\right\}$$[6]

As displayed by the above formulas, the occurrences increase exponentially for each new node added to the system. With as few as 20 nodes, there are over 1 million[7] possible occurrences of different states, which can present a challenge for developing an efficient algorithm to handle large quantities of combinations produced with this method.[8] The following example illustrates the use of the Event Space Method in estimating the reliability of a connected system.

For this example we will determine the reliability of a five node system modeled as a directed graph where each node represents a component with p probability (all similar probabilities for ease of notation).



[9]

Event Space Solution

There are five blocks in the system, which gives us $2^5 = 32$ occurrences.[10] According to the Event Space Method, the reliability of this system is the is the probability of the union of operational occurrences.

$$R = P(X_1 \cup X_2 \cup \ldots \cup X_7 \cup X_{10} \cup \ldots \cup X_{14} \cup X_{16} \cup X_{20} \cup X_{24})$$

[5] Bertsekas and Tsitsiklis. *Introduction To Probability*, 49.
[6] Bertsekas and Tsitsiklis. *Introduction To Probability*, 49.
[7] 1,048,576 total occurrences.
[8] See Program Performance Analysis, page 5.
[9] Elsayed. *Reliability Engineering* (2nd Edition), 123-124.
[10] See Table 1 for all operational occurrences, page 4.

If we then assume that all components of the system are disjoint (mutually exclusive)[11], we can write:

$$R = P(X_1) + P(X_2) + ... + P(X_7) + P(X_{10}) + ... + P(X_{14}) + P(X_{16}) + P(X_{20}) + P(X_{24})$$

Building the reliability formula is the summation of the product of occurrence failures and successes respectively: (if all components are identical and each has probability of p)

$$\sum_{i=0}^{n} P(X_i)\ ,\ \ (iff\ X_i\ does\ not\ cause\ system\ failure)$$

Where:

$$P(X_1) = P(ABCDE) = p^5$$
$$P(X_2) = P(X_3) = ... = P(X_6) = (1-p) \times p^4$$
$$P(X_7) = P(X_{10}) = ... = P(X_{14}) = P(X_{16}) = (1-p)^2 \times p^3$$
$$P(X_{20}) = P(X_{24}) = (1-p)^3 \times p^2$$

If all probabilities p are equivalent, the binomial formula can be used:

$$\sum_{i=0}^{n} \binom{n}{i} p^{(n-i)} \times (1-p)^{(i)}\ ,\ (iff\ nodes\ \binom{n}{i}\ does\ not\ cause\ system\ failure)$$ [12]

Determining which combinations produce system failures within the system (a path from source to target cannot be achieved) is an important aspect of the algorithm, and deserves a further breakdown.

Table 1: All Possible Failure/ Success Occurrences for Example Graph

Note: A*BCDE represents that A has failed. † AB*CD*E* represents that B,D,E have failed resulting in system failure.

Group 0 (no failures)	$X_1 = ABCDE$
Group 1 (one failure)	$X_3 = A^*BCDE, X_3 = AB^*CDE, X_4 = ABC^*DE$ $X_5 = ABCD^*E, X_6 = ABCDE^*$
Group 2 (two failures)	$X_7 = A^*B^*CDE, X_8 = \dagger AB^*C^*DE, X_9 = \dagger A^*BCD^*E, X_{10} = A^*BCDE^*,$ $X_{11} = AB^*C^*DE, X_{12} = AB^*CD^*E, X_{13} = AB^*CDE^*, X_{14} = ABC^*D^*E,$ $X_{15} = \dagger ABC^*DE^*, X_{16} = ABCD^*E^*$
Group 3 (three failures)	$X_{17} = \dagger ABC^*D^*E^*, X_{18} = \dagger AB^*CD^*E^*, X_{19} = \dagger AB^*C^*DE^*, X_{20} = AB^*C^*$ $X_{21} = \dagger A^*BCD^*E^*, X_{22} = \dagger A^*BC^*DE^*, X_{23} = \dagger A^*BC^*D^*E, X_{24} = A^*B^*CD$ $X_{25} = \dagger A^*B^*CD^*E, X_{26} = \dagger A^*B^*C^*DE$

[11] Bertsekas and Tsitsiklis. *Introduction To Probability*, 20 - 25.
[12] Bertsekas and Tsitsiklis. *Introduction To Probability*, 49.

Group 4 (four failures)	$X_{27} = \dagger A B^* C^* D^* E^*$, $X_{28} = \dagger A^* B C^* D^* E^*$, $X_{29} = \dagger A^* B^* C D^* E^*$, $X_{30} = \dagger A^* B^* C^* D E^*$, $X_{31} = \dagger A^* B^* C^* D^* E$
Group 5 (five failures)	$X_{32} = \dagger A^* B^* C^* D^* E^*$

By examining the original equation of reliability, it is now apparent why various occurrences are not included in the formula. For example, the majority of occurrences in Group 3, Group 4 and Group 5 cause system failure and thus are not included in the summation formula. Determining which occurrences cause system failure is straightforward.

Consider the occurrence: $X_8 = \dagger A B^* C^* D E$

Where all possible paths of the above graph are:

$$All\,Paths = [\ [A,\ B,\ C],\ [D,\ C],\ [A, B]\ ]$$

The failure of nodes B and C affects the success of each path in the collection of all paths, thus causing system failure.

$$All\,Paths = \dagger [\ [A,\ B^*,\ C],\ [D,\ C^*],\ [A, B^*]\ ]$$

While having a uniform probability of p provides for concise formulas, for sake of understanding, a sample output from our software provides a breakdown of the formulas. Consider that the reliabilities of A,B,C,D,E,F are 0.9, 0.8, 0.7, 0.5, 0.9, 0.9 respectively.

```
  (0.9) * (0.8) * (0.7) * (0.5) * (0.9) * (0.9)             +      (1 - 0.9) * (0.8) * (0.7) * (0.5) * (0.9) * (0.9)
+ (1 - 0.8) * (0.9) * (0.7) * (0.5) * (0.9) * (0.9)         +      (1 - 0.7) * (0.9) * (0.8) * (0.5) * (0.9) * (0.9)
+ (1 - 0.5) * (0.9) * (0.8) * (0.7) * (0.9) * (0.9)         +      (1 - 0.9) * (0.9) * (0.8) * (0.7) * (0.5) * (0.9)
+ (1 - 0.9) * (1 - 0.7) * (0.8) * (0.5) * (0.9) * (0.9)     +      (1 - 0.9) * (1 - 0.5) * (0.8) * (0.7) * (0.9) * (0.9)
+ (1 - 0.8) * (1 - 0.7) * (0.9) * (0.5) * (0.9) * (0.9)     +    (1 - 0.8) * (1 - 0.5) * (0.9) * (0.7) * (0.9) * (0.9)
+ (1 - 0.8) * (1 - 0.9) * (0.9) * (0.7) * (0.5) * (0.9)     +    (1 - 0.7) * (1 - 0.5) * (0.9) * (0.8) * (0.9) * (0.9)
+ (1 - 0.7) * (1 - 0.9) * (0.9) * (0.8) * (0.5) * (0.9)     +    (1 - 0.9) * (1 - 0.7) * (1 - 0.5) * (0.8) * (0.9) * (0.9)
+ (1 - 0.8) * (1 - 0.7) * (1 - 0.9) * (0.9) * (0.5) * (0.9)
Reliability = 0.81243
```

Program Performance Analysis

Performance

The main drawback of the performance of the Event Space Method is noticeable in two main areas, the exponential growth of the number of occurrences that must be processed and the computation time required to check whether each occurrence will cause failure of all possible paths in the connected system.

As previously described the number of occurrences can be calculated by the formula 2^N where N represents the number of nodes. By the time 20 nodes have been added to the system, over 1 million possible occurrences exist.[13] For computer systems with large hardware specifications, this type of processing may not present an unnecessary burden on system resources. However, with the use of consumer laptops and desktops, performing the Event Space Method requires an overwhelmingly large portion of system resources, and for graphs with more than 25 connected nodes, processing times can be calculated in hours rather than minutes.

While the techniques of the Event Space Method do require large amounts of processing due to large quantities of occurrences created, our specific algorithm developed to check whether each occurrence caused system failure proved to be a significant weak link in the overall efficiency of program performance. In order to find all working occurrences, the entire set of occurrences must be iterated through and compared with each possible path. If there is a failed node from an occurrence within each of the possible paths, the system is in a state of failure and therefore not included within the reliability summation. This sub-process as a whole is estimated at being a Big-O of N^4 or $O(N^4)$.[14] This easily becomes a processing bottleneck as the number of nodes increase. It is this weakness in the program that keeps it from being able to provide use for modeling large, real world connected systems, but is rather better suited for purposes of academic understanding and testing of small scale systems.

Improving the Algorithm

While the exponential growth of occurrences with the introduction of each unique node is defined by the Event Space Method and cannot be augmented, the algorithm for checking which occurrences cause system failure could be improved and Big-O order of magnitude reduced with more time and research. A possible method could be to eliminate the checking of certain occurrences. For example, occurrences that contain more failed nodes than the longest path and fewer working nodes than the shortest path could be excluded prior to checking. Another example of improvements pertains to the methods of checking for system failure. The algorithm was developed using simple iterations with equality checks to see if an occurrence will break a given path. If the paths were stored in Sets, faster comparisons without iterations could be achieved since Sets use a foundation of hash tables where lookups can be as little as O(1) Big-O. Looking for intersections could reduce the order of magnitude by a factor of N. While the algorithm Big-O could be reduced, the overall computation time would still be too large to prove a

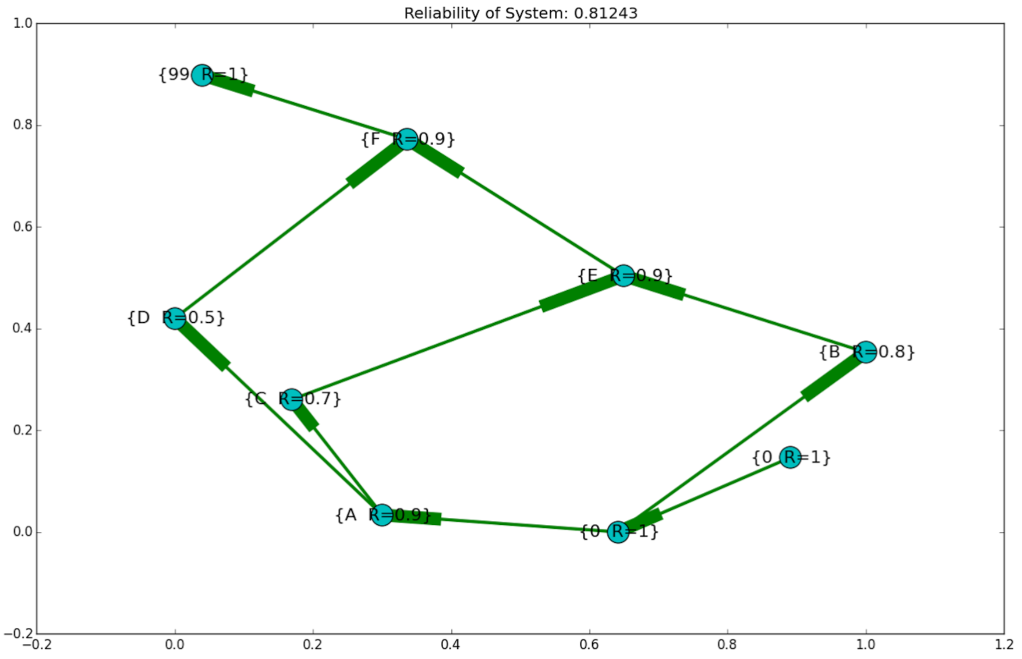
[13] 1,048,576 total occurrences.

[14] See Source Code methods: will_it_break_graph_flow(), build_master_data_structure().

viable option for modeling large complex system. Rather it would be consigned to working on complex models of a recommended node count of less than 20.

Source Material

Program Output: Graphical Display



Formula

```
+ (0.9) * (0.9) * (0.8) * (0.7) * (0.5) * (0.9)  -----> = 0.20412

+ (1 - 0.9) * (0.9) * (0.8) * (0.7) * (0.5) * (0.9)  -----> = 0.2268

+ (1 - 0.8) * (0.9) * (0.9) * (0.7) * (0.5) * (0.9)  -----> = 0.27783

+ (1 - 0.7) * (0.9) * (0.9) * (0.8) * (0.5) * (0.9)  -----> = 0.36531

+ (1 - 0.5) * (0.9) * (0.9) * (0.8) * (0.7) * (0.9)  -----> = 0.56943

+ (1 - 0.9) * (0.9) * (0.9) * (0.8) * (0.7) * (0.5)  -----> = 0.59211

+ (1 - 0.9) * (1 - 0.7) * (0.9) * (0.8) * (0.5) * (0.9)  -----> = 0.60183

+ (1 - 0.9) * (1 - 0.5) * (0.9) * (0.8) * (0.7) * (0.9)  -----> = 0.62451

+ (1 - 0.8) * (1 - 0.7) * (0.9) * (0.9) * (0.5) * (0.9)  -----> = 0.64638

+ (1 - 0.8) * (1 - 0.5) * (0.9) * (0.9) * (0.7) * (0.9)  -----> = 0.69741

+ (1 - 0.8) * (1 - 0.9) * (0.9) * (0.9) * (0.7) * (0.5)  -----> = 0.70308
```

+ (1 - 0.7) * (1 - 0.5) * (0.9) * (0.9) * (0.8) * (0.9) -----> = 0.79056

+ (1 - 0.7) * (1 - 0.9) * (0.9) * (0.9) * (0.8) * (0.5) -----> = 0.80028

+ (1 - 0.9) * (1 - 0.7) * (1 - 0.5) * (0.9) * (0.8) * (0.9) -----> = 0.81

+ (1 - 0.8) * (1 - 0.7) * (1 - 0.9) * (0.9) * (0.9) * (0.5) -----> = 0.81243

Source Code

```
import networkx as nx
import matplotlib.pyplot as plt
import itertools as it
from GraphExamples import *
import copy

def get_all_combinations(nodes):
    """return all combinations of nodes from N-N to N"""
    all_combinations = []

    for num in range (len(nodes) + 1):
        all_combinations.extend([combo for combo in it.combinations(nodes,num)])

    return all_combinations

def build_master_data_structure(all_combinations, all_nodes, all_paths):
    """Builds master data structure of all working and non working combinations
       If A,B,C,D,E are nodes the structure will look similar to:
       [ [["+ or -"],["A","B"],["C","D","E"]],
         [["+ or -"],["A","D"],["B","C","E"]],......]
       Where:
       [[Subtract or add to Reliability],[Broken Nodes],[Working Nodes]]

    """
    master = []

    for line in all_combinations:

        broken_nodes = [nde for nde in line]

        working_nodes = [nde for nde in all_nodes if nde not in broken_nodes]

        if will_it_break_graph_flow(broken_nodes, all_paths):   #append broken
            master.append([["-"], broken_nodes, working_nodes])
        else:
            master.append([["+"], broken_nodes, working_nodes]) #append working

    return master
```

```
def will_it_break_graph_flow(ndes, all_paths):
    """Checks to see if selected Broken nodes will break all flow in graph"""
    temp_paths, count = copy.deepcopy(all_paths), 0

    for val in ndes:

        for index in range(len(temp_paths)):
            if val in all_paths[index]:
                temp_paths[index].append("broken")

    for path in temp_paths:
        if "broken" in path:
            count += 1

    if count >= len(temp_paths):
        return True
    else:
        return False


def get_reliability(master_struct, fileHandle):
    """Return the probability of the system"""
    Total_P = 0
    for line in master_struct:
        if "+" in line[0]:

            prob_b, formula = 1, ""
            for val in line[1]: #broken combinations
                prob_b *= (1 - val.P)
                formula += " * (1 - " + str(val.P)+ ")"

            prob_w = 1
            for val in line[2]: #working combinations
                prob_w *= val.P
                formula += " * ("+ str(val.P) + ")"

            Total_P += (prob_b * prob_w)
            formula += "  -----> = " + str(Total_P)

            write_formula_file(str(prob_b * prob_w), fileHandle)

    write_formula_file(str(Total_P), fileHandle)

    return Total_P

def display_graph(G, reliability_of_system ):
    """Displays graph visual"""
    nx.draw_networkx(G,
                     node_size = 400,
                     font_size = 16,
                     width = 3.0,
                     edge_color = 'g',
                     node_color = 'c')

    plt.title("Reliability of System: " + str( reliability_of_system))

    plt.show()
```

```
def write_formula_file(formula_line, file_handle):
    """writes a mathematically correct visual representation of formula to file"""
    formula_line = " + " + formula_line[3:]

    file_handle.write(formula_line)

    file_handle.write('\n\n')

#### MAIN ####

file_handle = open("formula.txt","w")

Graph, src, trgt = graph6()

#create a list of path sets in graph
all_paths = [[nde for nde in path if type(nde.Num) != int]
             for path in nx.all_simple_paths(Graph, source = src, target = trgt)]

nodes = [nde for nde in Graph.nodes() if type(nde.Num) != int]

combos = get_all_combinations(nodes)

master = build_master_data_structure(combos, nodes, all_paths)

reliability_of_system =  get_reliability(master, file_handle)

file_handle.close()

print "Reliability of System:", reliability_of_system

display_graph(Graph, reliability_of_system)
```

```
# example Graph from text example
import networkx as nx

def graph6():
    G = nx.DiGraph() #create graph object
    #define nodes
    n1 = node(0,1)
    n2 = node("A",0.9)
    n3 = node("B",0.8)
    n4 = node("C",0.7)
    n5 = node("D",0.5)
    n6 = node("E",0.9)
    n7 = node("F",0.9)
    n8 = node(99,1)

    #map edges
    t = [(n1,n2),(n2,n4),(n4,n6),(n6,n7),(n7,n8),(n1,n3),(n3,n6),(n2,n5),(n5,n7)]
    G.add_edges_from(t)

    #return graph, source, target
    return G, n1, n8
```

```
#node class object

class node(object):
    def __init__(self,number,p):
        self.Num = number       #node number or letter
        self.P = p              #defined probability of node
    def __repr__(self):
        return "{"+str(self.Num) +"  R=" + str(self.P) +"}"
    def __str__(self):
        return "{"+str(self.Num) +"  R=" + str(self.P)+"}"
```

References

Bertsekas, Dimitri P. and Tsitsiklis, John N.. *Introduction To Probability.* Belmont Massachusetts: Athena Scientific, 2008.

Birolini, Alessandro. Reliability Engineering : Theory and Practice. Dordrecht: Springer, 2013.

Elsayed, Elsayed A.. *Reliability Engineering (2nd Edition)*. Hoboken, NJ, USA: John Wiley & Sons, 2012.

O'Connor, Patrick D. T. *Practical reliability engineering*. New York: Wiley, 1995.

Online:

Basics of System Reliability Analysis. ReliaWiki. 2014. Accessed April 2015. http://reliawiki.com/index.php/Basics_of_System_Reliability_Analysis

Calculating the Reliability of Series/Parallel and Non Series/Parallel Systems V 1.26. Department of Electrical Engineering and Computer Engineering, University of Massachusetts, Amherst. 2001. Accessed April 2015. http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems/simulator/NonSerPar/nsnpframe.html

Computer Systems Engineering/Reliability models. WikiBooks. Accessed April 2015. http://en.wikibooks.org/wiki/Computer_Systems_Engineering/Reliability_models#Series-Parallel_System_Reduction

MatPlotLib. John Hunter, Darren Dale, Eric Firing, Michael Droettboom. 2014. Accessed April 2015. http://matplotlib.org/

NetworkX. NetworkX developer team. 2014. Accessed April 2015. http://networkx.github.io/

Python 2.7. Python Software Foundation. 2015. Accessed April 2015. https://www.python.org/

RBDs and Analytical System Reliability. ReliaWiki. 2014. Accessed April 2015. http://reliawiki.org/index.php/RBDs_and_Analytical_System_Reliability

Reliability Engineering - Part 12. Accessed April 2015. http://d577289.u36.websitesource.net/articles/ReliabilityEng-Part12.pdf