ECE/CS 541
Computer System Analysis: Combinatorial Methods

Mohammad A. Noureddine
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Fall 2018
Learning Objectives

• Or what is this course about?

• At the start of the semester, you should have
  – Basic programming skills (C++, Python, etc.)
  – Basic understanding of probability theory (ECE313 or equivalent)

• At the end of the semester, you should be able to
  – Understand different system modeling approaches
    • Combinatorial methods, state-space methods, etc.
  – Understand different model analysis methods
    • Analytic/numeric methods, simulation
  – Understand the basics of discrete event simulation
  – Design simulation experiments and analyze their results
  – Gain hands-on experience with different modeling and analysis tools
Announcements and Reminders

• HW1 is out
  – Due on September 18, 2018 at the start of class

• Probability quiz on September 20, 2018
  – First 30 minutes of class

• Project Proposals due near the first week of October
  – List of possible projects and ideas on the website soon

• TA office hours: MW: 4:00 – 5:00 pm in CSL 231
Objectives for this Module

• Introduce combinatorial (non state-space) methods of modeling
• Develop and formulate models of system reliability
• Introduce different reliability formalisms
• Combinatorial models for improved testing research at Internet scale
  – Technique generated out of UC Santa Cruz and adopted by Netflix
Lecture Outline

• Reliability formalisms
  – Reliability block diagrams
  – Fault trees
  – Reliability graphs

• Case study
  – Automating Failure Testing Research at Internet Scale
A system comprises $N$ components, where the component failure times are given by the random variables $X_1, \ldots, X_N$. The system fails at time $S$ with distribution $F_S$ if:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>All components fail</td>
<td>$F_S(t) = \prod_{i=1}^{N} F_{X_i}(t)$</td>
</tr>
<tr>
<td>One component fails</td>
<td>$F_S(t) = 1 - \prod_{i=1}^{N} (1 - F_{X_i}(t))$</td>
</tr>
<tr>
<td>$k$ components fail, i.i.d</td>
<td>$F_S(t) = \sum_{i=k}^{N} \binom{N}{i} F_{X}(t)^i (1 - F_{X}(t))^{N-i}$</td>
</tr>
<tr>
<td>$k$ components fail, general case</td>
<td>$F_S(t) = \sum_{g \in G_k} \left( \prod_{X \in g} F_{X}(t) \right) \left( \prod_{X \not\in g} (1 - F_{X}(t)) \right)$</td>
</tr>
</tbody>
</table>
Reliability Formalisms

There are several popular graphical formalisms to express system reliability. The core of the solvers is the methods we have just examined.

In particular, we will examine

- Reliability Block Diagrams
- Fault Trees
- Reliability Graphs

There is nothing particularly special about these formalisms except their popularity. It is easy to implement these formalisms, or design your own, in a spreadsheet, for example.
Reliability Block Diagrams

- Blocks represent components.
- A system failure occurs if there is no path from source to sink.

**Series:**
System fails if any component fails.

**Parallel:**
System fails if all components fail.

**k of N:**
System fails if at least \( k \) of \( N \) components fail.
A NASA satellite architecture under study is designed for high reliability. The major computer system components include the CPU system, the high-speed network for data collection and transmission, and the low-speed network for engineering and control. The satellite fails if any of the major systems fail.

There are 3 computers, and the computer system fails if 2 or more of the computers fail. Failure distribution of a computer is given by $F_C$.

There is a redundant (2) high-speed network, and the high-speed network system fails if both networks fail. The distribution of a high-speed network failure is given by $F_H$.

The low-speed network is arranged similarly, with a failure distribution of $F_L$. 
RBD Example

\[
F_S(t) = 1 - \left(1 - \sum_{i=2}^{3} \binom{3}{i} F_C(t)^i (1 - F_C(t))^{3-i}\right) (1 - (F_H(t))^2) (1 - (F_L(t))^2)
\]
Probability all three systems survive to $t$

$$F_S(t) = 1 - \left( \min_{k \text{ of } N} \left( 1 - \sum_{i=2}^{3} \binom{3}{i} F_C(t)^i (1 - F_C(t))^{3-i} \right) \right)^{\max \left( 1 - (F_H(t))^2 \right) \left( 1 - (F_L(t))^2 \right)}$$
**RBD Example**

![Diagram of RBD Example]

Probability low speed network survives to $t$

$$F_S(t) = 1 - \left( 1 - \sum_{i=2}^{3} \binom{3}{i} F_C(t)^i (1 - F_C(t))^{3-i} \right) \left( 1 - \max (F_H(t))^2 \right) \left( 1 - \max (F_L(t))^2 \right)$$
RBD Example

Probability both components of low speed network fail by $t$

$$FS(t) = 1 - \left( 1 - \sum_{i=2}^{3} \binom{3}{i} FC(t)^i (1 - FC(t))^{3-i} \right) \left( \max \left( 1 - \left( FH(t) \right)^2 \right) \right) \left( 1 - \left( FL(t) \right)^2 \right)$$
Fault Trees

- Components are leaves in the tree, the system fails if the root is \textit{true}.
- **Explicit** representation of system decomposition and dependency of system operation on subsystems
- Fault tree expresses **logical** conditions necessary for system failure

**AND gates**

\textit{true} if all the components are \textit{true} (fail).

\begin{center}
\begin{tikzpicture}
  \node (and) {AND};
  \node[below left=0.5cm of and] (c1) {C1};
  \node[below right=0.5cm of and] (c2) {C2};
  \node[below right=1cm of c1] (c3) {C3};
  \draw[->] (and) -- (c1);
  \draw[->] (and) -- (c2);
  \draw[->] (c2) -- (c3);
\end{tikzpicture}
\end{center}

**OR gates**

\textit{true} if any of the components are \textit{true} (fail).

\begin{center}
\begin{tikzpicture}
  \node (or) {OR};
  \node[below left=0.5cm of or] (c1) {C1};
  \node[below right=0.5cm of or] (c2) {C2};
  \node[below right=1cm of c1] (c3) {C3};
  \draw[->] (or) -- (c1);
  \draw[->] (or) -- (c2);
  \draw[->] (c2) -- (c3);
\end{tikzpicture}
\end{center}

**\(k\) of \(N\) gates**

\textit{true} if at least \(k\) of the components are \textit{true} (fail).

\begin{center}
\begin{tikzpicture}
  \node (2of3) {	extbf{2 of 3}};
  \node[below left=0.5cm of 2of3] (c1) {C1};
  \node[below right=0.5cm of 2of3] (c2) {C2};
  \node[below right=1cm of c1] (c3) {C3};
  \draw[->] (2of3) -- (c1);
  \draw[->] (2of3) -- (c2);
  \draw[->] (c2) -- (c3);
\end{tikzpicture}
\end{center}
Fault Tree Example

- Consider the NASA example again
- How would we solve this fault tree?

Use $k$ of $N$ formula
Use max formula
Use min formula
Fault Trees – Further Analysis

\[ S_F = S_1 \lor S_2 \lor S_3 \]
\[ = (C_1 \land C_2) \lor (C_1 \land C_3) \lor (C_2 \land C_3) \lor (H_1 \land H_2) \lor (L_1 \land L_2) \]

Disjunctive Normal Form (DNF)

\[ S_1 = (C_1 \land C_2) \lor (C_1 \land C_3) \lor (C_2 \land C_3) \]

\[ S_2 = (H_1 \land H_2) \]

\[ S_3 = (L_1 \land L_2) \]
Fault Trees - Further Analysis

- **Explicit** representation of system decomposition and dependency of system operation on subsystems

\[ S_F = (C_1 \land C_2) \lor (C_1 \land C_3) \lor (C_2 \land C_3) \lor (H_1 \land H_2) \lor (L_1 \land L_2) \]

- Writing the tree in DNF gives us a sum (disjunction) of products (conjunctions)
  - Each product identifies sets of components, which when all of them fail, cause the system to fail

- We can convert any Boolean expression into its DNF

- We can further use the Boolean expressions to identify the minimum number of components needed for a system to fail
Reliability Graphs

- Reliability graphs are a more general way of representing complex interactions
  - RBDs and FTs general a special kind of graphs called “series-parallel” graphs

- The arcs (or edges) in the graph represent components and each has a failure distribution
  - A failure occurs if there is no path from the source to the destination

- We can represent series:

- We can represent parallel:
Reliability Graphs

- Reliability graphs can also capture more complex dependencies and interactions.
- For example, consider a network that fails when there is no path from the source to the destination.
Solving Reliability Graphs

- How do we approach solving the reliability graph of the network?

- **Brute Force:**
  - Enumerate all possible scenarios
  - Check which ones lead to there not being a path
  - Compute probability distribution accordingly
    - Use independence assumption

- **“Smarter” approach:**
  - Link \( C \) seems to be important to understanding the network.
  - Condition on the status of link \( C \)
  - Use laws of probability
Solving Reliability Graphs

- By the law of total probability

\[ P(S \leq t) = P(S \leq t \mid C \leq t) \times P(C \leq t) + P(S \leq t \mid C > t) \times P(C > t) \]

- First, let’s condition on link \( C \) being down
- The system becomes the series \( A \rightarrow D \) composed in parallel with the series \( B \rightarrow E \)

\[
P(S \leq t \mid C \leq t) = [1 - (1 - F_A(t))(1 - F_D(t))] \times [1 - (1 - F_B(t))(1 - F_E(t))]
\]

- Can be solved using the standard tools we have developed so far
  - Max of two min’s
Solving Reliability Graphs

- By the law of total probability
  \[
P(S \leq t) = P(S \leq t \mid C \leq t) \times P(C \leq t) + P(S \leq t \mid C > t) \times P(C > t)
  \]

- Second, let’s condition on link \(C\) being up
- The system becomes the series of two parallels

Can be solved using the standard tools we have developed so far
  - Min of two max’s

\[
P(S \leq t \mid C > t) = 1 - (1 - F_A(t)F_B(t))(1 - F_D(t)F_E(t))
\]

Parallel A – B

Parallel D – E
Conditioning Fault Trees

- In more general cases, fault trees can be used to represent systems where a component appears more than once in the fault
  - Relaxing the independence assumption that we made initially

- One approach to deal with such cases is to also use conditioning

- Given a fault tree for a system S and component C that appears more than once in the tree
  - Use the law of total probability again

\[ F_S(t) = F_{S \mid C_{\text{Fail}}}(t)F_C(t) + F_{S \mid C_{\text{up}}}(t)(1 - F_C(t)) \]
**Example**

- Component B appears under both branches of the following fault tree
- \( P(S \leq t \mid B \leq t) = ? \)

- **Let’s look at the formula for S:** \( S = (A \land B) \land (B \lor C) \)

- If B is down (i.e., \( B = 1 \)), we get
  \[
  S = (A \land 1) \land (1 \lor C) = A \land 1 = A
  \]
  So we can know that \( F_{S|B\text{ failed}}(t) = F_A(t) \)

- If B is up (i.e., \( B = 0 \)), we get
  \[
  S = (A \land 0) \land (0 \lor C) = 0
  \]
  So we can know that \( F_{S|B\text{ up}}(t) = 0 \)

\[
F_S(t) = F_B(t)F_A(t)
\]
Example

\[
F_S(t) = F_B(t) F_A(t)
\]

- Component C is irrelevant, i.e., does not impact the reliability of the system.

- We could see that from the expression for S:

\[
S = (A \land B) \land (B \lor C)
\]

\[
= (A \land B \land B) \lor (A \land B \land C)
\]

\[
= (A \land B) \lor (A \land B \land C)
\]

\[
= (A \land B) \land (1 \lor C)
\]

\[
= (A \land B)
\]

- **Sanity check**: Apply formula for max of two components.
Reliability/Availability Tables

A system comprises $N$ components. Reliability of component $i$ at time $t$ is given by $R_{Xi}(t)$, and the availability of component $i$ at time $t$ is given by $A_{Xi}(t)$.

<table>
<thead>
<tr>
<th>Condition</th>
<th>System Reliability</th>
<th>System Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>system fails if all components fail</td>
<td>$R_S(t) = 1 - \prod_{i=1}^{N} (1 - R_{Xi}(t))$</td>
<td>$A_S(t) = 1 - \prod_{i=1}^{N} (1 - A_{Xi}(t))$</td>
</tr>
<tr>
<td>system fails if one component fails</td>
<td>$R_S(t) = \prod_{i=1}^{N} R_{Xi}(t)$</td>
<td>$A_S(t) = \prod_{i=1}^{N} A_{Xi}(t)$</td>
</tr>
<tr>
<td>system fails if at least $k$ components fail,</td>
<td>$R_S(t) = \sum_{i=k}^{N} \binom{N}{i} (1 - R_{Xi}(t))^i R_X(t)^{N-i}$</td>
<td>$A_S(t) = \sum_{i=k}^{N} \binom{N}{i} (1 - A_{Xi}(t))^i A_X(t)^{N-i}$</td>
</tr>
<tr>
<td>identical distribution</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system fails if at least $k$ components fail,</td>
<td>$R_S(t) = \sum_{g \subseteq G_i} \left( \prod_{X \in g} (1 - R_{X}(t)) \right) \left( \prod_{X \notin g} R_X(t) \right)$</td>
<td>$A_S(t) = \sum_{g \subseteq G_i} \left( \prod_{X \in g} (1 - A_{X}(t)) \right) \left( \prod_{X \notin g} A_X(t) \right)$</td>
</tr>
<tr>
<td>general case</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Reliability Modeling Process

1. Modify Design
2. Define the system
3. Define "proper" service
4. Subcomponents?
5. Compute Reliability
6. Meet Spec?
7. Yes

Yes

No

Yes

No

Yes

No
Combinatorial Methods in Practice

• “Automating Failure Testing Research at Internet Scale”
  – A collaboration between Netflix and UC Santa Cruz
  – Appeared in the 2016 ACM Symposium on Cloud Computing (SoCC’16)

• Based on a previous paper by the same author
  – “Lineage-Driven Fault Injection”
  – Appeared in the 2015 International Conference on Management of Data (SIGMOD’15)
Motivation

Linux kernel map

functionality layers
- user space interfaces
  - system calls and system files
- human interface
  - HI char devices
  - HI subsystems
    - video_device
    - audio_device
    - network_device
  - system run
    - kernel_thread
    - do_idle
    - schedulers
- system
  - interfaces core
    - socket
- processing
  - devices
    - driver
  - memory
    - memory access
- virtual
  - virtual memory
- networking
  - sockets access
  - virtual file system
  - protocol families
  - networking storage
  - networking devices

slides
Slide 29

ECE/CS 541: Computer System Analysis. Fall 2018.
Motivation: Distributed Systems

• Imagine this kernel running several services in a distributed large scale data center
  – Netflix, Amazon, Google, Facebook, etc.

• Large scale systems must be built to tolerate a variety of hardware and software faults
  – Mainly use replication to provide fault tolerance
    • Both at the software and hardware level
  – Building a static fault tree for the entire data center is infeasible
    • Server get upgraded, scaled up, etc.
    • Complex routing protocols
    • Multiple Sources of failures
  – Building a fault tree for a piece of distributed software is even worse!
Motivation: Chaos Engineering

- **Chaos Engineering:**
  - “experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production”
  - Netflix’s chaos monkey:
    - [https://github.com/Netflix/chaosmonkey](https://github.com/Netflix/chaosmonkey)

- Use automated tools to provide end-to-end tests for business-critical assumptions about the system
  - Inject failures and observes the system’s behavior and report

- “Confidence in the end-to-end behavior of the system is manufactured by experimenting with worst-case failure scenarios in the production, scaled-out system”
Chaos Engineering: How?

• But how do we choose which failures to inject?
  – Which hardware to fail?
  – Which links to fail?
  – Which software to crash?

• The combinatorial space of faults across a distributed system (the failure scenarios) grows exponentially in the number of potential faults

• Current approaches:
  – Random: Select a failure scenarios at random
    • Not good: Why?
  – Programmer-guided: Bring your developers together and use their intuition about the software they designed and implemented
    • Yeah, right?
Lineage Driven Fault Inject (LDFI)

• So far, we’ve been thinking about how our system might fail
  – How do we fail our system?
  – Building RBDs, fault trees, reliability graphs, etc.

• But we have a treasure trove of our system did not fail
  – i.e., how our system gave us “good outcomes”

• Transformation the question from “could a bad thing ever happen”
  – Use narrower “how did this good thing happen?”

• Answers can provide rich information about the different paths that a successful request can take within our system
  – Use the answers to prune out scenarios that do not really matter
Lineage Driven Fault Tolerance is based on two insights

- **Fault tolerance is redundancy**
  - Fault tolerance is achieved if a system can provide *alternative ways* in which one can obtain the same outcome
  - If we had perfect information about all the possible ways in which a system can service a request, we can determine which faults it can tolerate and which it cannot
- Usually we moved forward: start from an initial state and explore the space of possible executions
  - It would be more efficient for identifying fault tolerance bugs to *work backwards*
  - Start from a successful execution and move your way back
    - From effects to causes
  - What combination of fault could have prevented the good outcome
LDFI: How it works?

• Begin with a correct outcome and ask:
  – How did this outcome occur?

• Obtain a lineage graph
  – Captures all the computations and data that contributed to producing that good outcome

• Run this several times and it would reveal the implicit redundancy in your deployment
  – What are the alternative computation paths that are sufficient to produce a certain good outcome

• Now it becomes tractable to reason about important failures for that good outcome you are trying to achieve
Example

- Consider the following example:
  - “Good outcome” = all acknowledged writes are durably stored.

- Consider a write that was durably stored
  - Q: Why was that write durably stored?
  - A: because it is stored on two replicas: repA and repB.

- Keep going backwards
  - Q: Why was the write stored on repA
  - A: because the client issued one or more broadcast requests to store a write

- Identified 4 important events that contributed to the good outcome of a durable write

\[ E \equiv \{RepA, RepB, Bcast1, Bcast1\} \]
Lineage Graph

- Backward reasoning brings us to a lineage graph for that durable write.

- Space of possible failure scenarios is $2^E$:
  - But not all are interesting.
  - Failing RepA and Bcast2 tells us nothing.

- Random strategy cannot tell us that!

- Which failure scenarios are then interesting?
  - Build a fault tree.

Figure 1. A simple lineage graph.
Build the Fault Tree

- They don’t actually build the fault tree
  - They build the equivalent *Conjunctive Normal Form* (CNF) expression
  - CNF: product of sums

\[
\begin{align*}
(RepA \lor Bcast1) & \\
\land (RepA \lor Bcast2) & \\
\land (RepB \lor Bcast1) & \\
\land (RepB \lor Bcast2)
\end{align*}
\]

Path in lineage graph
Min set of useful scenarios

• We can now obtain the **minimal solution to the CNF formula** that we generated
  – Use off-the-shelf SAT solvers

• We see that the **only two scenarios** that we care about are
  \[ \{\{\text{repA, repB}\}, \{Bcast1, Bcast2\}\} \]

• Outcome of one execution **might not reveal all the dependencies**
  – Run the failure scenario, one of two things will happen
    • A **new execution path will be revealed**
      – Update the fault tree and rerun
    • System fails and you have uncovered a **fault tolerance bug**
LDFI Process

Figure 2. Overview of LDFI.
LDFI Process

Figure 2. Overview of LDFI.
Results

• Implemented at Netflix to find fault tolerance bugs

• Paper provide interesting details about the challenges they faced and how they overcame them
  – I do recommend reading the paper

• LDFI at Netflix covered the failure space after doing 200 experiments
  – Number of possible scenarios in considered case study is $2^{100}$

• Revealed 11 new critical failures that could prevent a customer from loading the initial Netflix homepage
Further Reading

• Systems are becoming large, distributed and complex

• Our reliability process is not scalable to such systems

• So how do we build fault trees
  – Let the computers do it – Use machine learning

• **LIFT: Learning Fault Trees from Observational Data**
  – Meike Nauta et al.
  – Appeared at QEST 2018
  – Available on the course website

• Use failure datasets to generate fault trees and use them for analysis

• Interesting project ideas!!!