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

Summary

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:

Condition	Distribution
All components fail	$F_S(t) = \prod_{i=1}^{N} F_{X_i}(t)$
One component fails	$F_S(t) = 1 - \prod_{i=1}^{N} (1 - F_{X_i}(t))$
k components fail, i.i.d	$F_S(t) = \sum_{i=k}^{N} \binom{N}{i} F_X(t)^i (1 - F_X(t))^{N-i}$
k components fail, general case	$F_S(t) = \sum_{g \in G_k} \left(\prod_{X \in g} F_X(t) \right) \left(\prod_{X \notin g} (1 - F_X(t)) \right)$

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

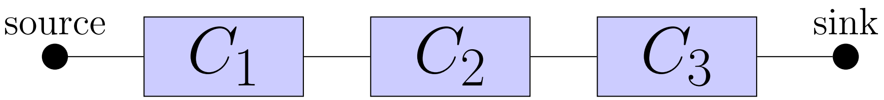
There is nothing particularly special about these formalisms except their popularity. It is easy to implement these formalisms, or design your own, in a spreadsheet, for example.

Reliability Block Diagrams

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

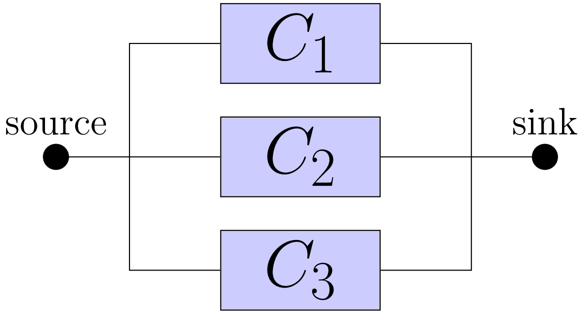
Series:

System fails if any component fails.



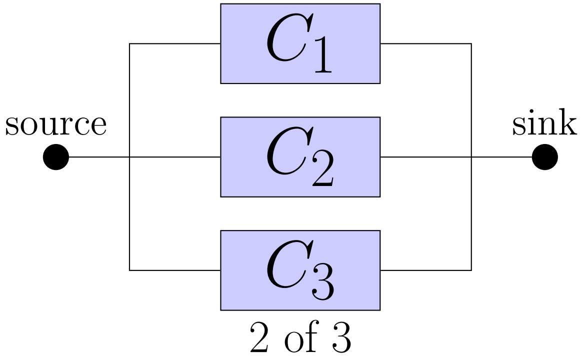
Parallel:

System fails if all components fail.



k of *N*:

System fails if at least *k* of *N* components fail.



Example

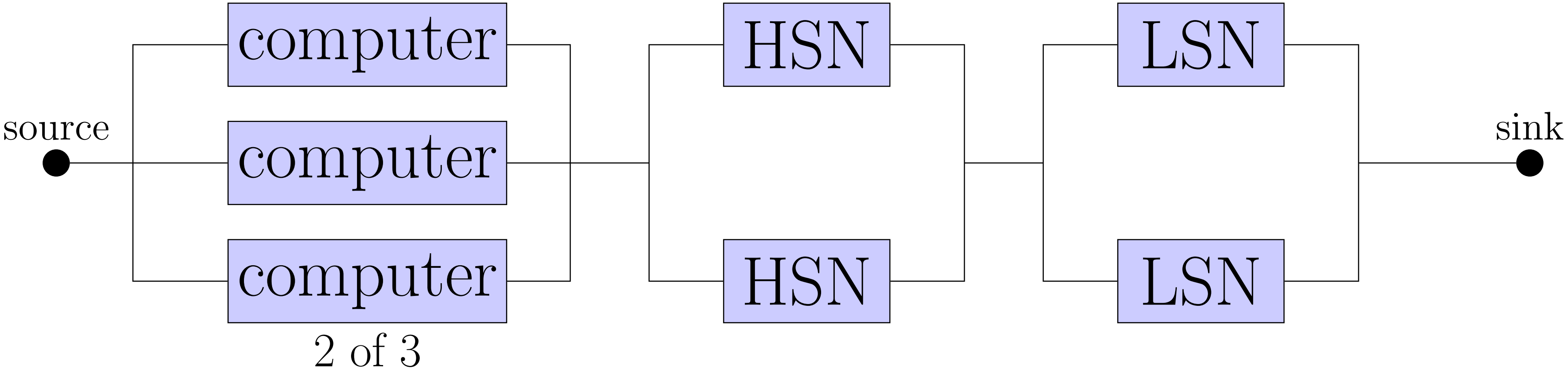
A NASA satellite architecture under study is designed for high reliability. The major computer system components include the CPU system, the high-speed network for data collection and transmission, and the low-speed network for engineering and control. The satellite fails if any of the major systems 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.

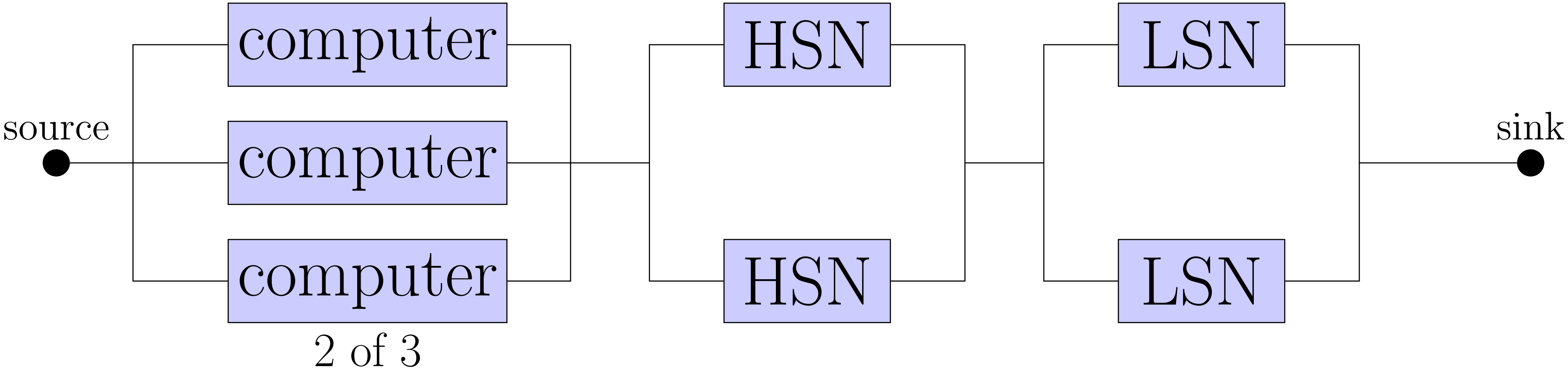
The low-speed network is arranged similarly, with a failure distribution of F_L.

RBD Example



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

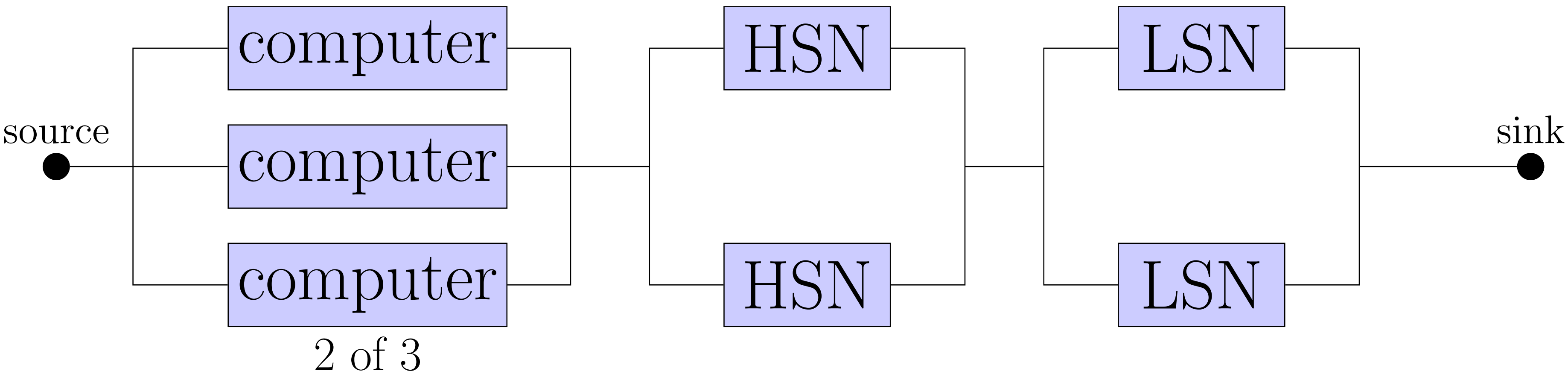
RBD Example



Probability all three systems survive to t

$$F_S(t) = 1 - \underbrace{\left(1 - \overbrace{\sum_{i=2}^{3} \binom{3}{i} F_C(t)^i (1 - F_C(t))^{3-i}}^{k \text{ of } N}\right)\left(1 - \overbrace{(F_H(t))^2}^{\max}\right)\left(1 - \overbrace{(F_L(t))^2}^{\max}\right)}_{\min}$$

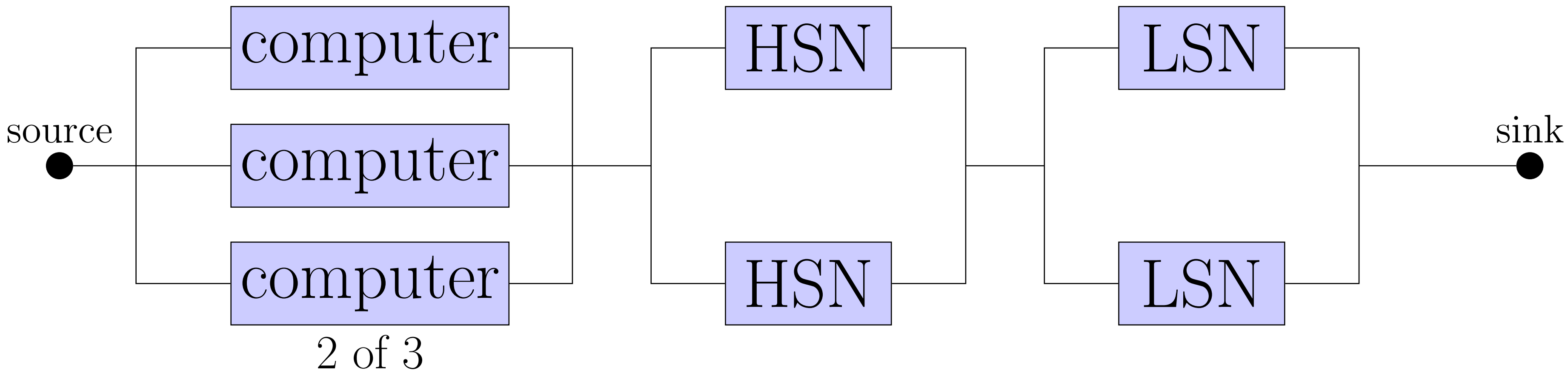
RBD Example



Probability low speed network survives to t

$$F_S(t) = 1 - \underbrace{\left(1 - \overbrace{\sum_{i=2}^{3} \binom{3}{i} F_C(t)^i (1 - F_C(t))^{3-i}}^{k \text{ of } N}\right) \left(1 - \overbrace{(F_H(t))^2}^{\max}\right) \left(1 - \overbrace{(F_L(t))^2}^{\max}\right)}_{\min}$$

RBD Example



Probability both components of low speed network fail by t

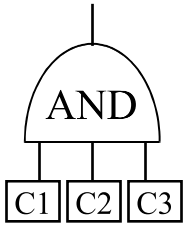
$$F_S(t) = 1 - \underbrace{\left(1 - \overbrace{\sum_{i=2}^{3}\binom{3}{i}F_C(t)^i(1-F_C(t))^{3-i}}^{k\text{ of }N}\right)\left(1 - \overbrace{(F_H(t))^2}^{\max}\right)\left(1 - \overbrace{(F_L(t))^2}^{\max}\right)}_{\min}$$

Fault Trees

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

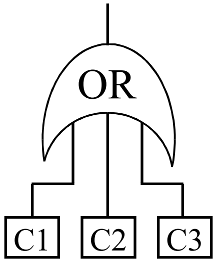
AND gates

true if all the components are *true* (fail).



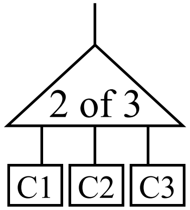
OR gates

true if any of the components are *true* (fail).



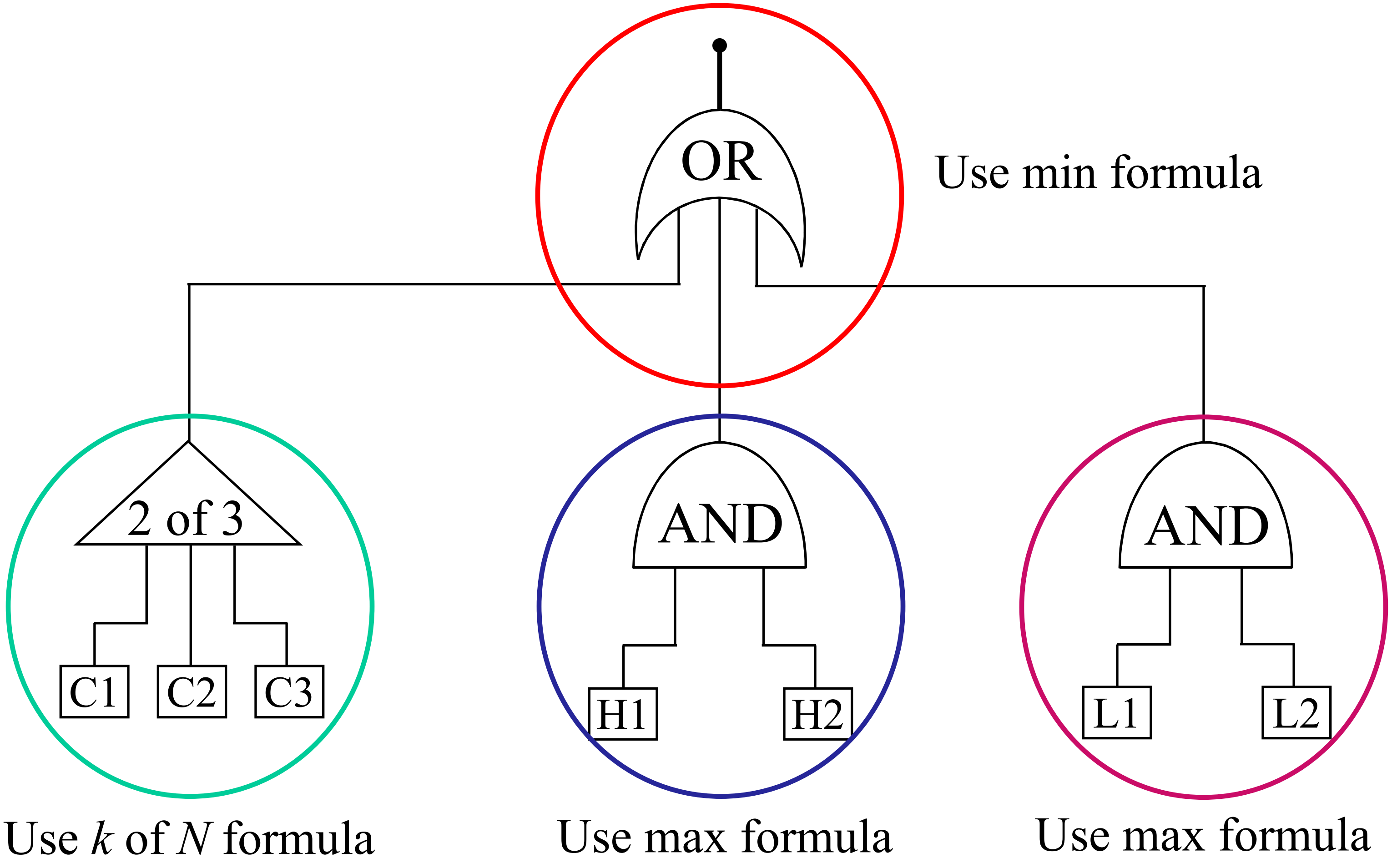
k of *N* gates

true if at least *k* of the components are *true* (fail).



Fault Tree Example

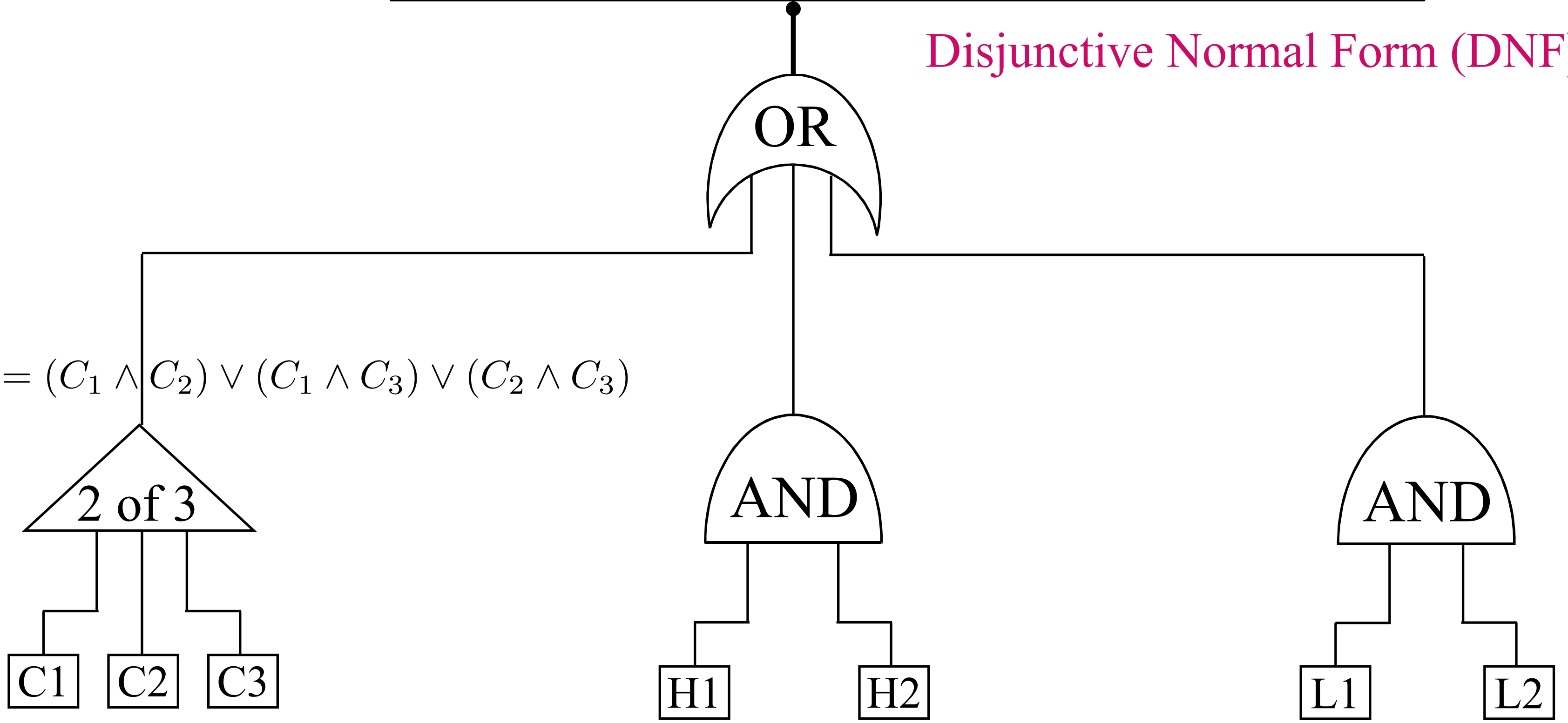
- Consider the NASA example again
- How would we solve this fault tree?



Fault Trees - Further Analysis

$$S_F = S_1 \vee S_2 \vee S_3$$
$$= (C_1 \wedge C_2) \vee (C_1 \wedge C_3) \vee (C_2 \wedge C_3) \vee (H_1 \wedge H_2) \vee (L_1 \wedge L_2)$$

Disjunctive Normal Form (DNF)



$S_1 = (C_1 \wedge C_2) \vee (C_1 \wedge C_3) \vee (C_2 \wedge C_3)$

$S_2 = (H_1 \wedge H_2)$

$S_3 = (L_1 \wedge L_2)$

Fault Trees – Further Analysis

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

$$S_F = (C_1 \wedge C_2) \vee (C_1 \wedge C_3) \vee (C_2 \wedge C_3) \vee (H_1 \wedge H_2) \vee (L_1 \wedge L_2)$$

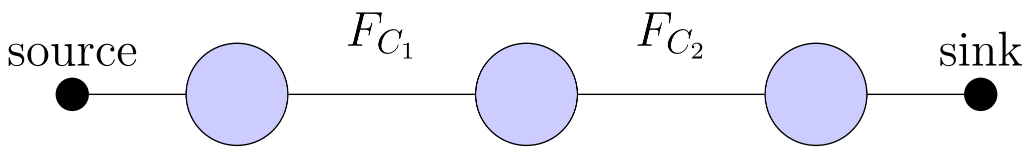
- Writing the tree in DNF gives us a sum (disjunction) of products (conjunctions)
  - Each product identifies sets of components, which when all of them fail, cause the system to fail

- We can convert any Boolean expression into its DNF

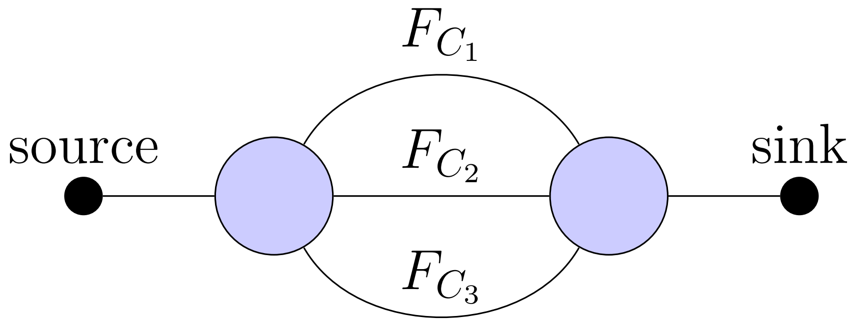
- We can further use the Boolean expressions to identify the minimum number of components needed for a system to fail

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:

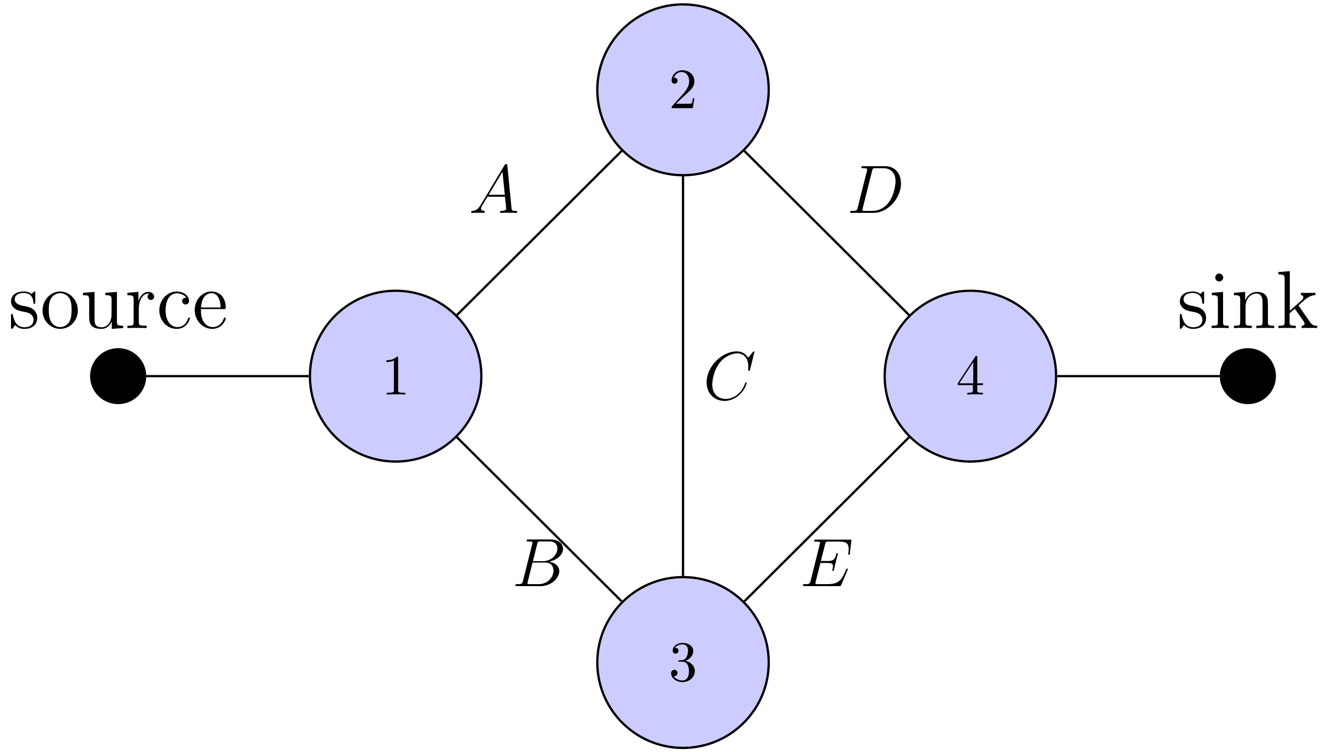


- We can represent parallel:



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

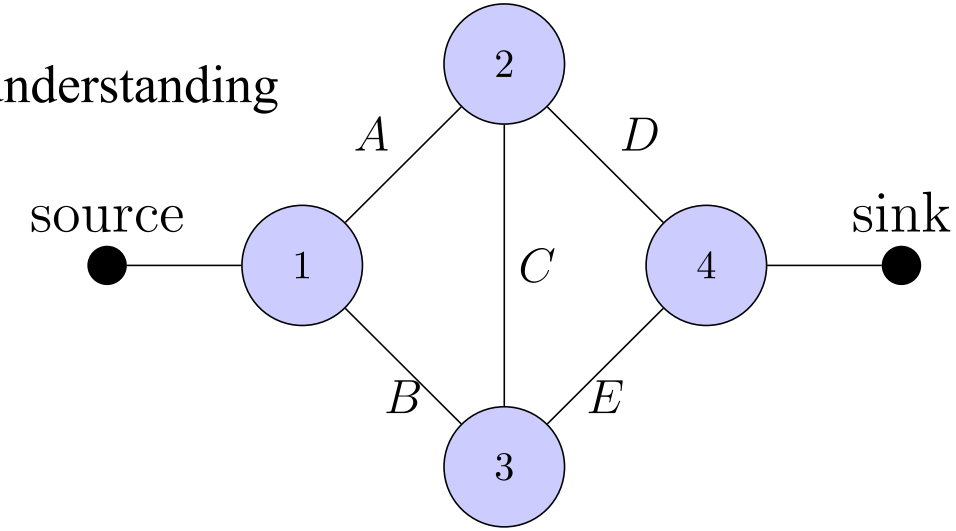


Solving Reliability Graphs

- 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

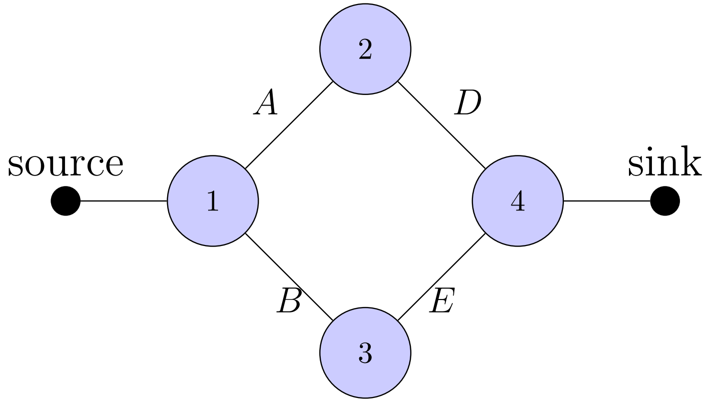


Solving Reliability Graphs

- By the law of total probability

$$P(S \le t) = \underbrace{P(S \le t \mid C \le t)}_{F_{S \mid C \ fails}} \times \underbrace{P(C \le t)}_{F_C(t)} + \underbrace{P(S \le t \mid C > t)}_{F_{S \mid C \ up}} \times \underbrace{P(C > t)}_{(1 - F_C(t))}$$

- First, let's condition on link *C* being down
- The system becomes the series *A* – *D* composed in parallel with the series *B* – *E*



- Can be solved using the standard tools we have developed so far
  - Max of two min's

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

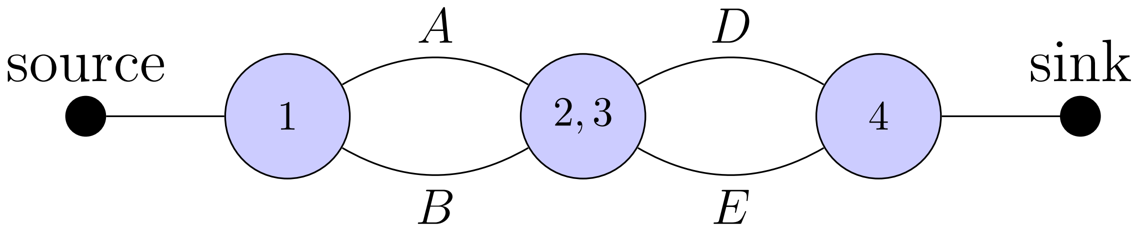
Series A – D   Series B – E

Solving Reliability Graphs

- By the law of total probability

$$P(S \le t) = \underbrace{P(S \le t \mid C \le t)}_{F_{S \mid C\ fails}} \times \underbrace{P(C \le t)}_{F_C(t)} + \underbrace{P(S \le t \mid C > t)}_{F_{S \mid C\ up}} \times \underbrace{P(C > t)}_{(1 - F_C(t))}$$

- 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

$$P(S \le t \mid C > t) = 1 - (1 - F_A(t)F_B(t))(1 - F_D(t)F_E(t)))$$

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

$$F_S(t) = F_{S|C \text{ Fail}}(t)F_C(t) + F_{S|C \text{ up}}(t)(1 - F_C(t))$$

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: $S = (A \wedge B) \wedge (B \vee C)$

- If B is down (i.e., B = 1), we get

$$S = (A \wedge 1) \wedge (1 \vee C) = A \wedge 1 = A$$

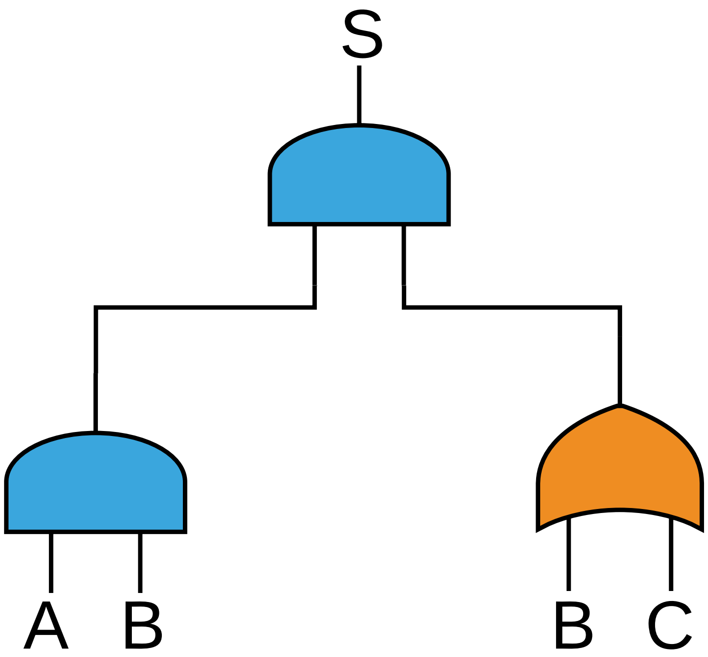
- So we can know that $F_{S|B\ \text{failed}}(t) = F_A(t)$

- If B is up (i.e., B = 0), we get

$$S = (A \wedge 0) \wedge (0 \vee C) = 0$$

- So we can know that $F_{S|B\ \text{up}}(t) = 0$

$$\boxed{F_S(t) = F_B(t) F_A(t)}$$



Example

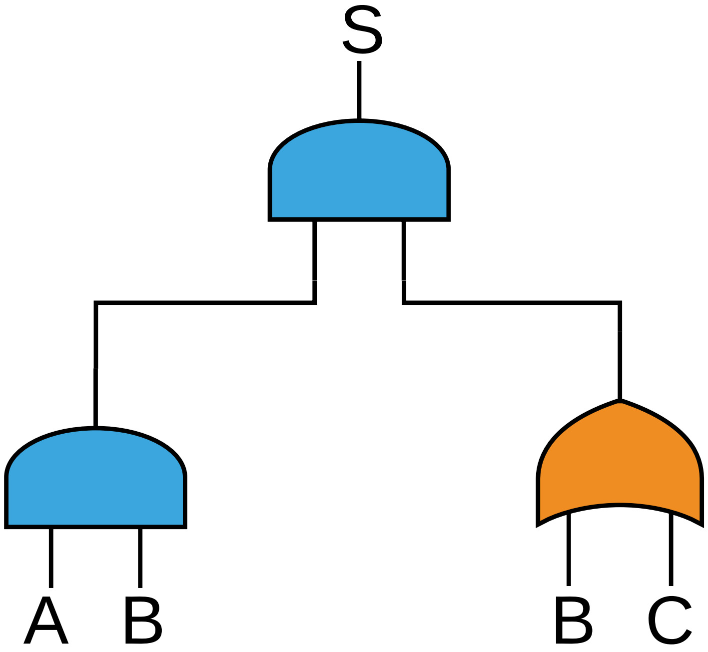
$$F_S(t) = F_B(t)F_A(t)$$

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

- We could see that from the expression for S:

$$\begin{aligned} S &= (A \wedge B) \wedge (B \vee C) \\ &= (A \wedge B \wedge B) \vee (A \wedge B \wedge C) \\ &= (A \wedge B) \vee (A \wedge B \wedge C) \\ &= (A \wedge B) \wedge (1 \vee C) \\ &= (A \wedge B) \end{aligned}$$

- Sanity check: Apply formula for max of two components

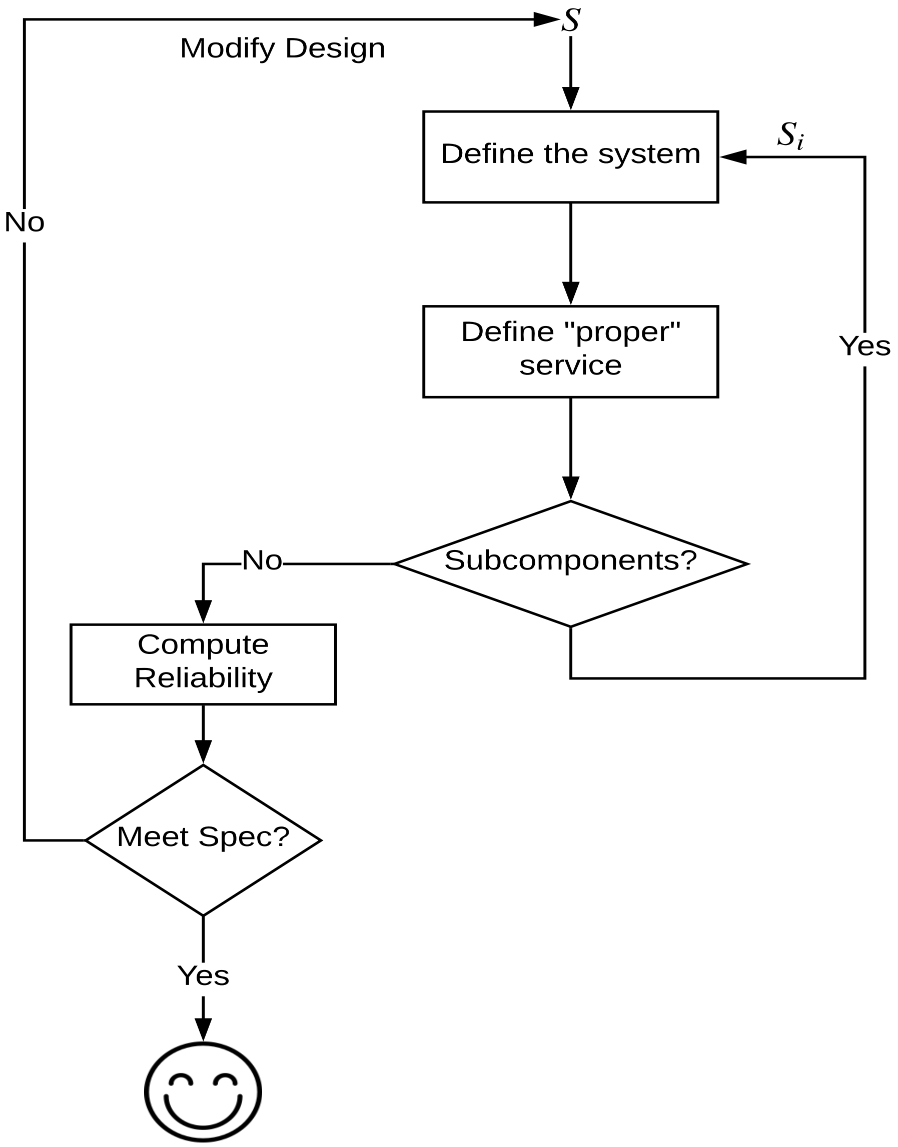


Reliability/Availability Tables

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 $A_{Xi}(t)$.

Condition	System Reliability	System Availability
system fails if all components fail	$R_S(t) = 1 - \prod_{i=1}^{n} (1 - R_{Xi}(t))$	$A_S(t) = 1 - \prod_{i=1}^{n} (1 - A_{Xi}(t))$
system fails if one component fails	$R_S(t) = \prod_{i=1}^{n} R_{Xi}(t)$	$A_S(t) = \prod_{i=1}^{n} A_{Xi}(t)$
system fails if at least k components fail, identical distribution	$R_S(t) = \sum_{i=k}^{N} \binom{N}{i} (1 - R_{Xi}(t))^i R_X(t)^{N-i}$	$A_S(t) = \sum_{i=k}^{N} \binom{N}{i} (1 - A_X(t))^i A_X(t)^{N-i}$
system fails if at least k components fail, general case	$R_S(t) = \sum_{g \in G_k} \left(\prod_{X \in g} (1 - R_X(t)) \right) \left(\prod_{X \notin g} R_X(t) \right)$	$A_S(t) = \sum_{g \in G_k} \left(\prod_{X \in G} (1 - A_X(t)) \right) \left(\prod_{X \notin g} A_X(t) \right)$

Reliability Modeling Process



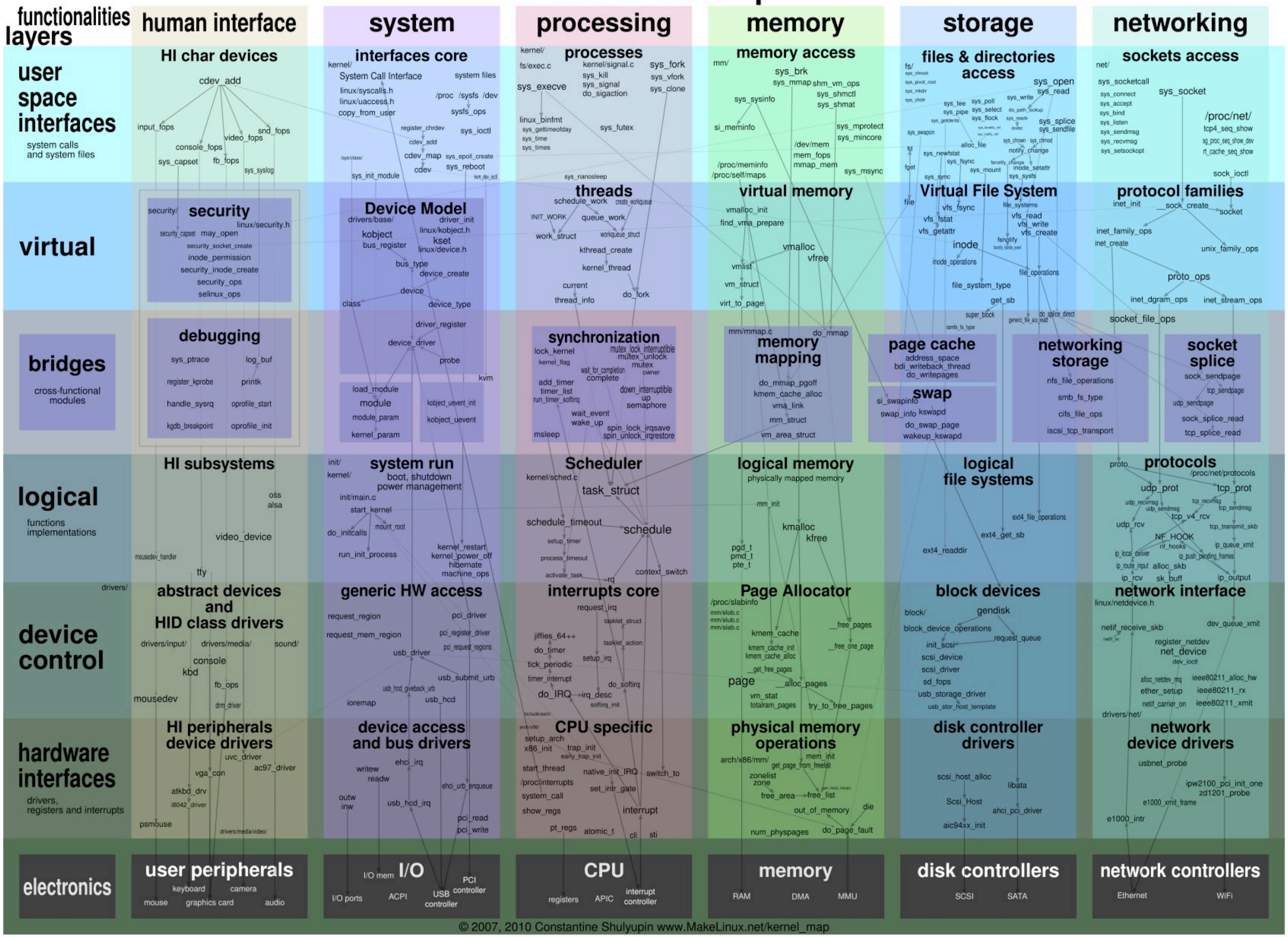
Combinatorial Methods in Practice

- "Automating Failure Testing Research at Internet Scale"
  - P. Alvaro, et al.
  - 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



Motivation: Distributed Systems

- Imagine this kernel running several services in a distributed large scale date 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
- 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

LDFI

- Lineage Driven Fault Tolerance is based on two insights
  - *Fault tolerance is redundancy*
    - Fault tolerance is achieve 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 the contributed to producing that good outcome
- Run this several time 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

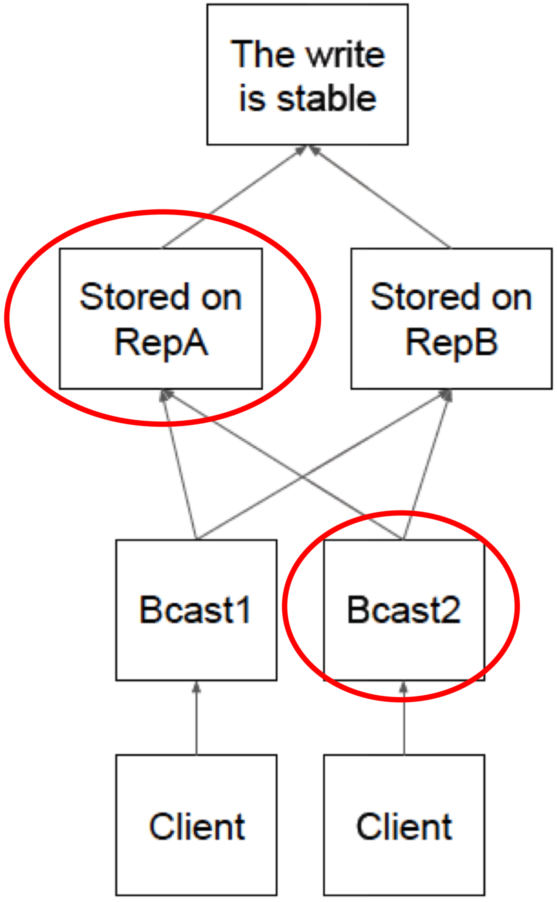


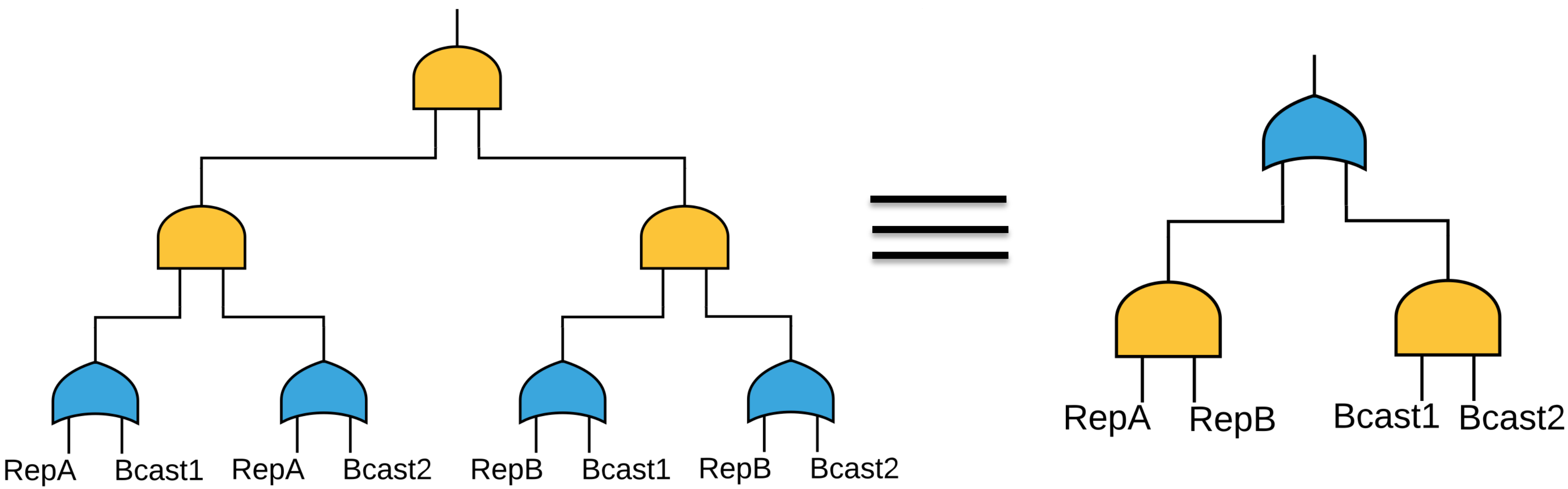
Figure 1. A simple lineage graph

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

$$(RepA \vee Bcast1) \wedge (RepA \vee Bcast2) \wedge (RepB \vee Bcast1) \wedge (RepB \vee Bcast2)$$

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

$$\{\{repA, repB\}, \{Bcast1, Bcast2\}\}$$

- 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

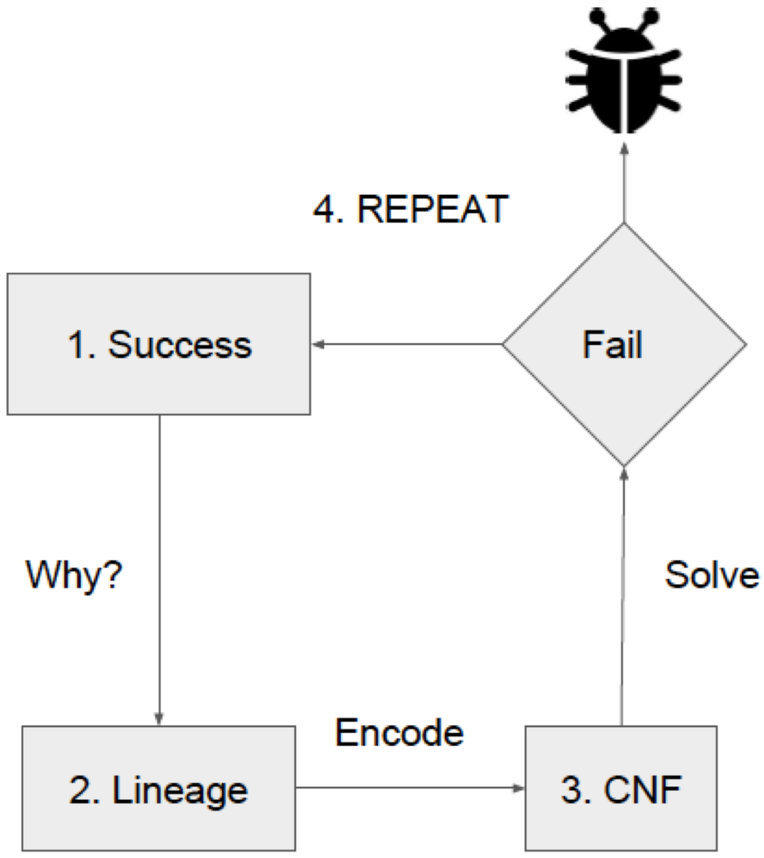


Figure 2. Overview of LDFI.

LDFI Process

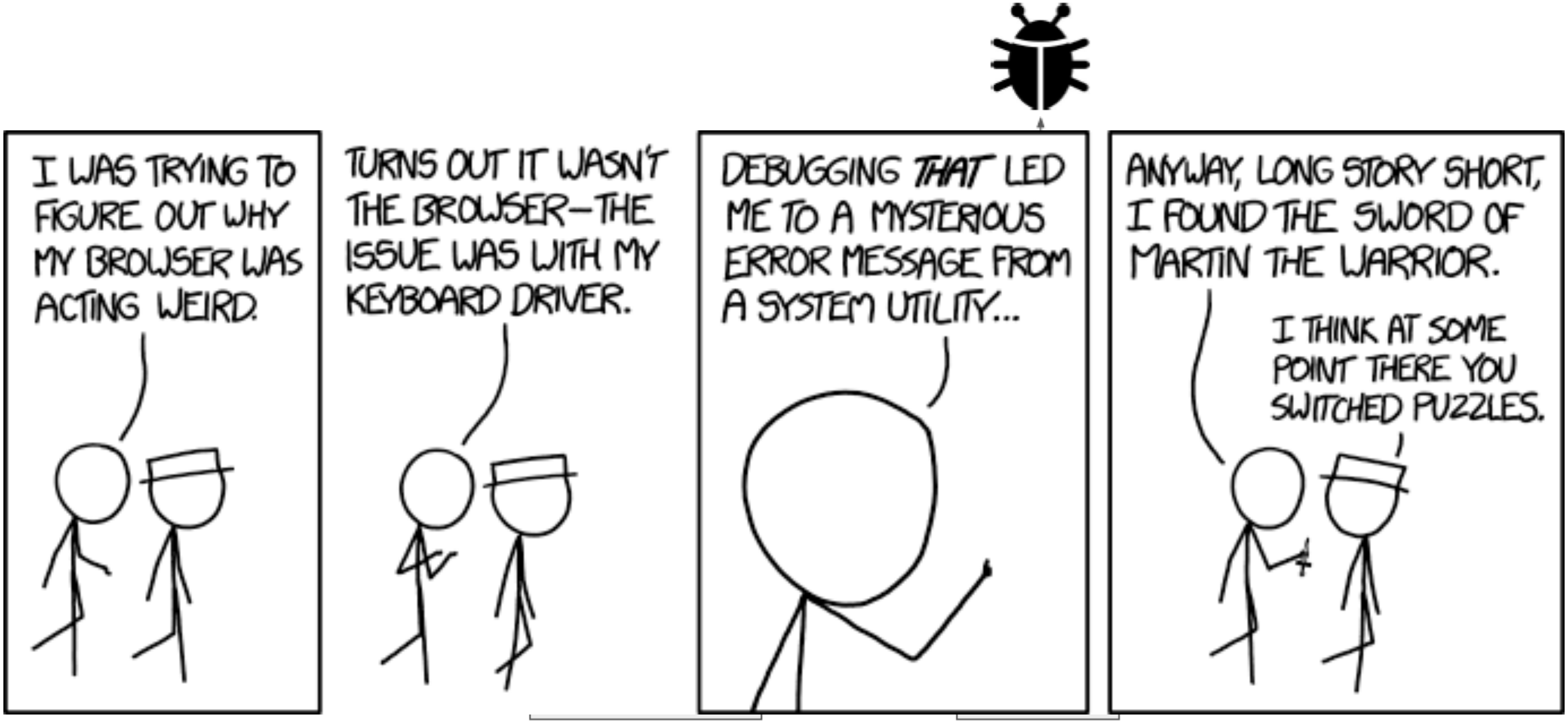


Figure 2. Overview of LDFI.

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!!!