NC STATE UNIVERSITY

FINAL TECHNICAL REPORT

An Open Source, Parallel, and Distributed Web-Based Probabilistic Risk Assessment Platform to Support Real Time Nuclear Power Plant Risk-Informed Operational Decisions

DOE-NEUP 21-24004

Probabilistic Risk Assessment Group
Department of Nuclear Engineering

NC STATE UNIVERSITY

DOE-NEUP 21-24004
Revision v0.0.4

An Open Source, Parallel, and Distributed Web-Based Probabilistic Risk Assessment Platform to Support Real Time Nuclear Power Plant Risk-Informed Operational Decisions

Final Technical Report

Manuscript Completed: June 16, 2025

Prepared by:
Arjun Earthperson, North Carolina State University,
Hasibul H. Rasheeq, North Carolina State University,
Egemen M. Aras, North Carolina State University,
Asmaa S. Farag, North Carolina State University,
Kirill D. Shumilov, North Carolina State University,
Stephen T. Wood, Idaho National Laboratory,
Jordan T. Boyce, Idaho National Laboratory,
Mihai A. Diaconeasa, North Carolina State University

Probabilistic Risk Assessment Group
Department of Nuclear Engineering
North Carolina State University

CONFIGURATION CONTROL

Document ID: DOE-NEUP 21-24004
Revision: v0.0.4
Source Repository: 10.5281/zenodo.15313673

This `git` version-controlled document was typeset using TEX. A complete revision history, with sources, is available from the repository link.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

ABI	Application Binary Interface.
AFW	Auxiliary Feedwater.
AI	Artificial Intelligence.
AIG	And-Inverter Graph.
ANF/RNF	Algebraic/Ring Normal Form.
ANS	American Nuclear Society.
API	Application Programming Interface.
ARS	Advanced Reactor Safety.
ASME	American Society of Mechanical Engineers.
BCF	Blake Canonical Form.
BDD	Binary Decision Diagram.
BE	Basic Event.
BICS	Boolean Indicated Cut Sets.
BN	Bayesian Network.
BSON	Binary-encoded JavaScript Object Notation.
CCCG	Common Cause Component Groups.
CCF	Common-Cause Failure.
CCW	Component Cooling Water.
CDF	Cumulative Distribution Function.
CI	Continuous Integration.
CLI	Command-Line Interface.
CLT	Central Limit Theorem.
CNF	Conjunctive Normal Form.
CPU	Central Processing Unit.
CRUD	Create-Read-Update-Delete.
CuDD	Colorado University Decision Diagram Package.
DAG	Directed Acyclic Graph.
d-DNNF	Deterministic Decomposable Negation Normal Form.
DLL	Dynamic Link Library.
DNF	Disjunctive Normal Form.
DNNF	Decomposable Negation Normal Form.
d-NNF	Deterministic Negation Normal Form.
DPMC	Data-Parallel Monte Carlo.
DPPL	Delphi Parallel Programming Library.
DUCG	Dynamic Uncertain Causality Graph.

EIP	Essential Prime Implicant.
EPI	Essential Prime Implicate.
EPRI	Electric Power Research Institute.
ET	Event Tree.
f-BDD	Free/Read-Once Binary Decision Diagram.
FDS	Frontier Development of Science.
FE	Functional Event.
FHE	Fully Homomorphic Encryption.
FIFO	First-In-First-Out.
FMEA	Failure Mode and Effects Analysis.
f-NNF	Flat Negation Normal Form.
FPGA	Field-Programmable Gate Array.
FPRM	Fixed Polarity Reed-Muller.
FT	Fault Tree.
FTA	Fault Tree Analysis.
FTREX	Fault Tree Reliability Evaluation eXpert.
GB	Gigabyte.
GPL	GNU General Public License.
GPU	Graphics Processing Unit.
G–PWR	Generic Pressurized Water Reactor.
GUI	Graphical User Interface.
HCL	Hybrid Causal Logic.
HCLA	Hybrid Causal Logic Analyzer.
HE	Flag/House Event.
HFE	Human Failure Event.
HPC	High Performance Computing.
HRA	Human Reliability Analysis.
HTTP	Hypertext Transfer Protocol.
HTTPs	Secure Hypertext Transfer Protocol.
i.i.d.	independent and identically distributed.
IMECE	International Mechanical Engineering Congress and Exposition.
INES	International Nuclear Event Scale.
INL	Idaho National Laboratory.
IP	Prime Implicant.
IRRAS	Integrated Reliability and Risk Analysis System.
ISLOCA	Interfacing Systems LOCA.
JIT	just-in-time.
JSCut	SAPHSOLVE MCS JSON.

JSCutIn	SAPHSOLVE MCS JSON, as Input.
JSInp	SAPHSOLVE Model Input JSON.
JSON	JavaScript Object Notation.
KAERI	Korea Atomic Energy Research Institute.
KIRAP	KAERI Integrated Reliability Analysis Code Package.
LLM	Large Language Model.
LLN	Law of Large Numbers.
LOCA	Loss of Cooling Accident.
LOOP	Loss of Offsite Power.
LWR	Light Water Reactor.
MAR–D	Models and Results Database.
MCS	Minimal Cut Sets.
MCUB	Min Cut Upper Bound.
MDP	Motor-Driven Pump.
MEF	Model Exchange Framework.
MHTGR	Modular High Temperature Gas-cooled Reactor.
MICSUP	Minimal Cut Sets Upward.
MIT	Massachusetts Institute of Technology.
MOCUS	Method for Obtaining Minimal Cut Sets.
NCSU	North Carolina State University.
NNF	Negation Normal Form.
NoSQL	No Structured Query Language.
NPP	Nuclear Power Plant.
NRC	US Nuclear Regulatory Commission.
O1-Pro	OpenAI's O1-Pro Agentic Assistant.
OBDD	Ordered Binary Decision Diagram.
OpenMP	Open Multi-Processing.
PC	Personal Computer.
PDAG	Probabilistic Directed Acyclic Graph.
PDF	Probability Density Function.
PGA	Peek Ground Acceleration.
PI	Prime Implicate.
PORV	Power-Operated Relief Valve.
PPRM	Positive Polarity Reed-Muller.
PRA	Probabilistic Risk Assessment.
PRNG	Pseudo Random Number Generator.
PSA	Probabilistic Safety Assessment.

PSDD	Probabilistic Sentential Decision Diagram.
PWR	Pressurized Water Reactor.
QRA	Quantitative Risk Assessment.
QRAS	Quantitative Risk Assessment System.
RAM	Random Access Memory.
REA	Rare-Event Approximation.
RegEx	Regular Expression.
REST	Representational State Transfer.
RHR	Residual Heat Removal.
RNG	Random Number Generator.
RoBDD	Reduced Ordered Binary Decision Diagram.
SAPHIRE	Systems Analysis Programs for Hands-On Integrated Reliability Evaluations.
SAPHSOLVE	SAPHIRE Solve Engine.
SAT	Boolean Satisfiability.
SDD	Sentential Decision Diagram.
sd-DNNF	Smooth/Structured Deterministic Decomposable Negation Normal Form.
s-DNNF	Smooth/Structured Decomposable Negation Normal Form.
SHA	Secure Hash Algorithm.
SHA-256	Secure Hash Algorithm 256-bit.
SIS	Safety Injection System.
SoP	Sum-of-Products/Sum-Product.
SP	Sum-Product/Sum-of-Products.
SPAR	Standardized Plant Analysis Risk.
SQL	Structured Query Language.
SRL	SetLib.
SSC	Systems, Structures, and Components.
STL	C++ Standard Template Library.
SWS	Service Water System.
SYCL	Formerly known as SYstem-wide Compute Language.
UCLA	University of California Los Angeles.
UCLA GIRS	The B. John Garrick Institute for the Risk Sciences, UCLA.
UI	User Interface.
UQ	Uncertainty Quantification.
URL	Uniform Resource Locator.
US	United States.
VOT	Voter (K-of-N).
VRAM	Video/Graphics RAM.

WHO	World Health Organization.
XAG	XOR-And-Inverter Graph.
XML	Extensible Markup Language.
XOR	Exclusive-Or.
ZDD	Zero-Suppressed Binary Decision Diagram.

1 INTRODUCTION

1.1 Project Motivation

The development and use of PRAs in the nuclear industry have revolutionized our approach to design, reliability, and safety. Since the release of WASH-1400 in 1975 and NUREG-1150 in 1990, the PRA applications have diversified and become more computationally demanding, nevertheless the tools that we use to perform such assessments have not kept up with the technological advancements in high-performance computing and computer science applications. Now, although there is a need for supporting real-time decisions at the nuclear plants using PRAs, this is not practical using the computationally strained legacy PRA tools currently available since they were developed and are maintained using technologies that are nowadays deprecated. The legacy PRA tools, although still perform well on internal events PRA models that have reasonable sizes, suffer greatly, both in terms of speed and memory requirements, when challenged by more sophisticated models such as single hazard PRAs, and especially multi-hazard PRAs. Therefore, a major redesign of the PRA tools is necessary starting from the computational engine capabilities, backend services to handle large PRA models, and a user-friendly PRA frontend that can automatically generate results and documentation necessary to inform non-PRA experts.

1.2 Project Scope

The scope of this project is to design, implement, and evaluate an open source, parallel, and distributed web-based PRA platform that addresses the limitations of legacy PRA tools. The platform is intended to support a wide range of PRA applications, from traditional internal events models to advanced multi-hazard and multi-unit scenarios. The project encompasses the development of a modular architecture that integrates a web-based client for model editing and visualization, RESTful backend APIs for data management and orchestration, and a distributed job queue system for high-throughput risk quantification. The platform is designed to be extensible, supporting integration with multiple quantification engines and enabling interoperability with industry-standard tools through the adoption of the OpenPRA schema and compatibility with the OpenPSA Model Exchange Format (MEF). The project also includes the development of automated benchmarking and model translation workflows, as well as the implementation of features to facilitate collaborative model development, transparent risk communication, and reproducible analysis. The ultimate goal is to provide a robust, scalable, and user-friendly solution that meets the evolving needs of PRA practitioners and stakeholders in the nuclear industry and beyond.

1.3 Project Objectives

The main objective of this work is to develop, demonstrate, and evaluate a probabilistic risk assessment (PRA) software platform needed to address the major challenges of the current legacy PRA tools, such as better quantification speed, integration of multi-hazard models into traditional PRAs, and model modi-

fication simplification and documentation automation. To achieve the main objective, we will first perform benchmarking and profiling of current PRA tools, such as SCRAM [1] and US NRC's SAPHIRE [2, Vol. 1] , to investigate the current bottlenecks in the quantification speed and memory requirements. Secondly, we will design, implement, and benchmark a PRA software platform based on a web-based stack using the latest technologies available to overcome the mentioned challenges. Finally, we will evaluate the performance gains of this framework by modeling and quantifying large PRA models that would have been too expensive to run using the legacy PRA tools.

1.4 Objectives of this Report

This report documents the design, implementation, and evaluation of the open source, parallel, and distributed web-based PRA platform developed under this project. The objectives of this report are to:

- Present the motivation, scope, and objectives of the project in the context of current challenges in PRA technology.
- Describe the architectural design and technical implementation of the platform, including its modular components and integration strategies.
- Summarize the benchmarking and evaluation of the platform against existing PRA tools, highlighting improvements in computational efficiency, scalability, and model interoperability.
- Identify current limitations and outline a roadmap for future research and development to address remaining gaps and advance the platform toward industry readiness.
- Provide guidance and best practices for PRA practitioners and stakeholders interested in adopting or contributing to the platform.

1.5 Project Outcomes

An important objective of this project is to disseminate accessible information and tools. In addition to the mandated deliverables, there are several notable outcomes from this project.

1.5.1 Software

- **@openpra-org/openpra-monorepo**: Mono repository for the OpenPRA web application [88]. Core code contributions for the distributed microservices developed in this study are integrated here.
- **@openpra-org/model-benchmarks**: Benchmarking suite for SCRAM, XFTA, FTREX, SAPHSOLVE using benchexec in Docker [53]. Continuous Integration pipelines for automated runs & report generation are available.
- **@openpra-org/PRAcciolini**: Automates the conversion, validation, & translation of PRA models. Provides interfaces for describing grammars & translation rules between schema. [48].
- **@openpra-org/model-generator**: CLI utility for creating stochastically generated synthetic event trees & fault trees. Supports OpenPSA, SAPHSOLVE, FTREX and OpenFTA schema. [54].

- **@openpra-org/mef-schema**: Schema definitions for OpenPRA supported model exchange formats including FTREX FTP, SAPHSOLVE JSInp, JSCut, OpenPSA XMLs, & canopy flatbuffers [55].
- **@openpra-org/multi-hazard-model-generator**: Generates multi-hazard event trees & fault trees in MAR–D from internal events PRA models [31].
- **@openpra-org/canopy-benchmarks**: Accuracy & performance benchmark scripts for canopy [47].

1.5.2 Datasets

- **Generic Pressurized Water Reactor (PWR) SAPHSOLVE Model**: Reference PWR model in SAPHSOLVE JSON (JSInp) format, supporting benchmarking & verification tasks [5].
- **Generic Pressurized Water Reactor (PWR) Open-PSA Model**: Equivalent PWR model in OpenPSA XML for cross-tool benchmark comparisons [4].
- **Generic Modular High Temperature Gas-cooled Reactor (MHTGR) Model**: Reference PRA model for a modular high temperature gas-cooled reactor, including event trees, fault trees, and basic event data in SAPHIRE and CAFTA formats [69].
- **Synthetic SAPHSOLVE Models**: Synthetically generated PRA models for benchmarking, quantification, & code verification [7].
- **Synthetic OpenPSA Models**: Synthetically generated OpenPSA XML format models for benchmark studies [6].
- **openpra-org/synthetic-models**: Centralized collection of stochastically generated PRA models in multiple supported formats, for validation & stress testing of quantification engines [19].
- **Benchmarking SAPHIRE, SCRAM & XFTA**: Dataset of synthetically generated fault trees with common cause failures, for head-to-head tool verification [50].
- **Generic OpenPSA Models: The Aralia Fault Tree Dataset**: Curated, large-scale synthetic PRA fault trees & event trees, compatible with the OpenPSA data model [51].
- **Post-processing Analysis & Supplementary Notes**: Excelsheets, plotting analysis, MATLAB scripts used for curating & analyzing the generated raw results. [49].

1.5.3 Reports

- Refining Processing Engines from SAPHIRE: Initialization of Fault Tree / Event Tree Solver, INL, 2023 [28].
- Diagnostics and Strategic Plan for Advancing the SAPHIRE Engine, INL, 2023 [27].

Internal Milestone Reports

- Milestone Report 1: Literature Review, Benchmarking, & Profiling of Current PRA Tools, 2022 [12].
- Milestone Report 2: Fine-Tuning Performance, Benchmarking, Profiling & Design Insights, 2023 [13].

Part I

Foundations

2 THE TRIPLET DEFINITION OF RISK

A central goal of risk analysis in nuclear engineering is to enable sound decision-making under uncertainty. To achieve this, risk must be defined in a way that is both rigorous and practically quantifiable. One widely accepted definition, tracing back to seminal work in [75, 65], frames risk as a *set of triplets*. Each triplet captures three essential dimensions:

1. *What can go wrong?*
2. *How likely is it to happen?*
3. *What are the consequences if it does happen?*

In more formal terms, let

$$R = \left\{ \langle S_i, L_i, X_i \rangle \right\}_c \tag{2.1}$$

where R denotes the overall risk for a given system or activity, and the subscript c emphasizes *completeness*: ideally, all important scenarios must be included. In this notation:

- S_i specifies the ith **scenario**, describing something that can go wrong (e.g. an initiating event or equipment failure). Typically, $S_i \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible scenarios.
- L_i (sometimes denoted p_i or ν_i) is the **likelihood** (probability or frequency) associated with scenario S_i. In other words, $L_i \in [0, 1]$ if modeled as a probability, or $L_i \in [0, \infty)$ if modeled as a rate/frequency.
- X_i characterizes the **consequence**, i.e. the severity or nature of the outcome if the scenario occurs. Consequences can range from radiological releases and economic cost to broader societal impacts. In some analyses, X_i is a single-valued metric in $\mathcal{X}$; in others, it may be treated as a distribution over possible outcomes in $\mathcal{X}$.

The notation $\{\cdot\}_c$ in Eq. (2.1) stresses that *all* substantial risk scenarios must be included. Omitting a significant scenario might severely underestimate total risk. One might ask, "What are the uncertainties?" In this report, uncertainties are embedded in each S_i implicitly, and evaluated in L_i and X_i explicitly via probability distributions.

2.1 A Scenario-Based Approach

A practical way to enumerate each triplet $\langle S_i, L_i, X_i \rangle$ is through logical decomposition of potential failures or disruptions, a process referred to as *scenario structuring*. Scenario structuring helps answer the question "*What can go wrong?*" in greater detail by dividing possible scenarios into commonly recognized classes. Each of these categories corresponds to a distinct family of *initiating events* (IE) that can trigger a chain of subsequent events or failures. At each node in the success scenario, we identify the IEs, which branch off from the initial success path S_0 into new pathways that may lead to undesirable states. Thus, each *scenario*

S_i can be interpreted as a distinct departure from the baseline success path, triggered by some IE that occurs at node i. From that point onward, a sequence of *conditional events* or barriers may succeed or fail, culminating in an end-state ES_i.

2.2 Definition of an Event Tree

An Event Tree (ET) unravels how a single *initiating event* (I) can branch into multiple possible *end-states* (X) through a sequence of *functional* (or conditional) events. Each branch captures the success or failure of an important *functional event* (e.g. a safety barrier or operator intervention). By following all possible paths, one can systematically account for each final outcome X_j. Figure 2.1 provides a schematic view of this process for an initiating event I and two subsequent functional events, F_1 and F_2. Each terminal node (leaf) corresponds to a distinct end-state, denoted $X_1, X_2, \ldots, X_n$. Though this illustration is intentionally simple, more complex systems may include numerous functional events, each branching into further outcomes.

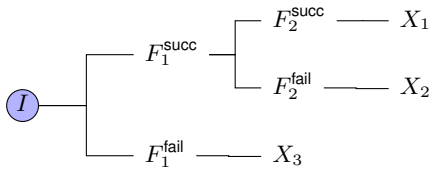


Figure 2.1: Illustrative event tree with an initiating event I, two functional events F_1 and F_2, and three end-states X_1, X_2, X_3.

At the highest conceptual level, an event tree is a collection of conditional outcomes. Let n be a positive integer, and let j range over some index set of end-states J. Then the scenario pathways can be defined as:

$$\Gamma \;=\; \Big\{\langle I,\, F_1,\, F_2,\, \ldots,\, F_n,\, X_j\rangle \,:\, j \in J\Big\}, \tag{2.2}$$

where:

- I is the **initiating event**. In a nuclear system, this could be an abnormal occurrence such as a coolant pump trip or an unplanned reactivity insertion.
- F_k ($k = 1, \ldots, n$) denotes the kth **functional (conditional) event**, which may succeed (F_k^{succ}) or fail (F_k^{fail}). Typically, each F_k depends on the outcomes $F_1, \ldots, F_{k-1}$.
- X_j is an **end-state**, describing the final outcome along a particular branch. End-states might indicate safe shutdown, fuel damage, or a radiological release.

Each tuple $\langle I, F_1, \ldots, F_n, X_j\rangle$ in Γ encapsulates a distinct scenario pathway. In the broader context of the risk triplet, such a pathway corresponds to S_i, the possibility of something going wrong, while the associated likelihood and consequences map directly to L_i and X_i.

2.3 Definition of a Fault Tree

A Fault Tree (FT) is a top-down representation of how a specific high-level failure can arise from malfunctions in the components or subsystems of an engineered system. It is typically drawn as a tree or a Directed Acyclic Graph (DAG) whose unique root node is the top event and whose leaves/basic events capture individual component failures or other fundamental causes. This hierarchical decomposition proceeds until all relevant failure modes are captured in the leaves or else grouped as undeveloped events.

Formally, the nodes of a fault tree can be divided into two main categories:

- **Events**, which denote occurrences at different hierarchical levels.
  - A *Basic Event (BE)* represents the lowest-level failures, typically single-component malfunctions or individual human errors. They are often depicted as circles or diamonds in diagrams.
  - *Intermediate events* indicate the outcome of one or more lower-level events. Though intermediate events do not change the logical structure of the FT analysis, they can greatly enhance clarity by grouping sub-failures into a meaningful subsystem label (e.g., Cooling subsystem fails). They are typically drawn as rectangles.
  - *Top event(TE)* is a single node, unique in the tree, that represents the high-level failure of interest (e.g., System fails).
- **Gates**, which describe how events combine to produce a higher-level event. Each gate outputs a single event (often an intermediate or the top event), based on one or more input events.

Because a fault tree traces failures up toward the top event, the overall structure becomes a DAG. If a particular event (basic or intermediate) is relevant to multiple subsystems, it can be shared among the inputs of different gates. Consequently, while many small FTs have a pure tree shape, complex systems generally produce shared subtrees, yielding a more general DAG. If a system is large, detailed modeling of every component may not be warranted. In such cases, one may simplify certain subsystems by treating their failures as single *undeveloped events*. An undeveloped event is effectively a basic event for analysis purposes, even though it may internally comprise several components. This method conserves complexity where the subsystem is either of negligible importance or insufficiently characterized to break down further.

A convenient formalization, explained in detail in [102], treats an FT as a structure $F = \langle \mathcal{B}, \mathcal{G}, T, I \rangle$ where the unique top event t belongs to $\mathcal{G}$, and:

- $\mathcal{B}$ is the set of basic events.
- $\mathcal{G}$ is the set of gates or internal nodes.
- $T \to$ GateTypes: assigns a gate type (AND, OR, k/n, etc.) to each gate in $\mathcal{G}$.
- $I \to \mathcal{P}(\mathcal{B} \cup \mathcal{G})$: specifies the input set of each gate, i.e. which events (basic or intermediate) feed into that gate.

The graph is *acyclic* and has a unique root (the top event t) that is reachable from all other nodes. If an element is the input to multiple gates, it may be drawn once and connected multiple times or duplicated

visually; either way, the logical semantics remain the same.

Interpreting a fault tree involves examining which higher-level events fail when a subset S of basic events has failed. Let $\pi_F(S, e)$ denote the failure state (0 or 1) of element e when the set $S \subseteq \mathcal{B}$ of basic events has failed. Then:

- For each basic event $b \in \mathcal{B}$:

$$\pi_F(S, b) \;=\; \begin{cases} 1, & b \in S, \\ 0, & b \notin S. \end{cases}$$

- For each gate $g \in \mathcal{G}$ with inputs $\{x_1, \ldots, x_k\} \subseteq \mathcal{B} \cup \mathcal{G}$:

$$\pi_F(S, g) \;=\; \begin{cases} \bigwedge_{i=1}^{k} \pi_F(S, x_i), & \text{if } T(g) = \mathrm{AND}, \\ \bigvee_{i=1}^{k} \pi_F(S, x_i), & \text{if } T(g) = \mathrm{OR}, \\ 1 \;-\; \pi_F(S, x_i), & \text{if } T(g) = \mathrm{NOT} \text{ (single input)}, \\ \sum_{i=1}^{k} \pi_F(S, x_i) \;\geq\; k, & \text{if } T(g) = \mathrm{VOT}(k/n), \\ \Big(\sum_{i=1}^{k} \pi_F(S, x_i)\Big) \bmod 2, & \text{if } T(g) = \mathrm{XOR}, \end{cases}$$

The top event t (i.e., TE) is a gate in $\mathcal{G}$; it is common to write simply $\pi_F(S)$ to denote whether the top event fails under the set S of failed BEs.

2.3.1 Definition of k-of-n Voting Logic

A *k-of-n* gate, denoted $\mathrm{VOT}(k/n)$, outputs 1 if and only if *at least* k of its n inputs equal 1. Such a gate, often referred to as a *threshold* or *majority voting* gate, conveniently models partial redundancy and majority-vote mechanisms. Concretely, let each of the n input events be represented by a binary variable:

$$X_1,\, X_2,\, \ldots,\, X_n \;\in\; \{0, 1\}.$$

The gate's output Y is then defined by

$$Y \;=\; \begin{cases} 1, & \text{if } \sum_{i=1}^{n} X_i \;\geq\; k, \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

Equivalently, $Y = 1$ can be expressed as a disjunction of conjunctions:

$$Y \;=\; \Big[\sum_{i=1}^{n} X_i \;\geq\; k\Big] \;=\; \bigvee_{\substack{S \subseteq \{1,\ldots,n\} \\ |S| = k}} \Big(\bigwedge_{i \in S} X_i\Big), \tag{2.4}$$

meaning that at least one subset $S \subseteq \{1, \ldots, n\}$ of size k has all its corresponding X_i set to 1. Any larger subset $|S| > k$ naturally satisfies the same condition.

2.3.2 Common-Cause Failures

A Common-Cause Failure (CCF) is defined as the failure of multiple components due to a shared cause within a specified time interval. Unlike independent failures, CCFs can defeat redundancy-based safety systems, potentially leading to severe consequences in critical applications.

Common cause failures stem from three main categories of coupling factors [86]:

Hardware-based factors

- Hardware design (system layout, component internal parts)
- Manufacturing attributes
- Construction/installation characteristics

Operation-based factors

- Same operating staff
- Same operating procedures
- Same maintenance/test/calibration schedule or staff

Environment-based factors

- Same plant location exposing components to identical environmental stresses
- Same component location
- Same internal environment/working medium

Several parametric models have been developed to quantify CCF probabilities:

2.3.2.1 Basic Parameter Model (BPM)

Common Cause Component Groups (CCCG) are sets of components susceptible to the same CCF mechanisms. Basic parameter model decomposes the failure probability of a component in a CCCG into terms involving independent failure of the component and combinations of CCFs with the other components in the CCCG [98]. For a group of m components, the parameter Q_k represents the probability of a specific k-component failure combination. The total failure probability is then calculated as:

$$Q_T = \sum_{k=1}^{m} \binom{m-1}{k-1} Q_k^m$$

While conceptually straightforward, this model becomes impractical for large component groups due to the number of parameters required.

2.3.2.2 Alpha Factor Model (AFM)

The Alpha Factor Model develops CCF probabilities from set of failure ratios and the total component failure probability [98]. For a system with m components, the alpha factors are defined as:

$$\alpha_k = \frac{n_k}{\sum_{i=1}^{m} i \cdot n_i}$$

where n_k is the number of events with k failed components. The probability of a specific failure combination is then:

$$Q_k = \frac{\alpha_k}{\binom{m-1}{k-1}} \cdot Q_T$$

The Alpha Factor Model is particularly useful because its parameters have direct statistical meaning and can be estimated from operational data.

2.3.2.3 Multiple-Greek Letter (MGL) Model

Multiple Greek Letter model is used for systems with higher level of redundancy or k-out of-n logic. The multiple Greek letter model includes parameters for the conditional probabilities that the N+1-th subsystem fails given that N identical subsystems have already failed [74]. The key parameters are:

- β: Conditional probability that a component fails due to a common cause, given that another component has failed
- γ: Conditional probability that a third component fails, given that two components have failed due to a common cause
- $\delta, \epsilon, ...$: Additional parameters for larger component groups

For a three-component system, the common cause failure probabilities are:

$$Q_1 = Q_T(1-\beta)$$
$$Q_2 = Q_T\beta(1-\gamma)$$
$$Q_3 = Q_T\beta\gamma$$

2.3.2.4 Binomial Failure Rate (BFR) Model

The Binomial Failure Rate (BFR) model is based on the concept of two distinct types of shock events affecting components in a CCCG:

1. **Lethal shocks** occur at constant rate $\lambda^{(i)}$ and cause simultaneous failure of all components in the CCCG.
2. **Non-lethal shocks** occur at constant rate ν and cause each component to fail independently with probability p.

Each component in an CCCG has a total dependent failure rate of:

$$\lambda_c = \lambda^{(i)} + p\nu \tag{2.5}$$

The model uses binomial probability distribution to calculate the rate of failures with different multiplicities. When a non-lethal shock occurs, the probability of exactly k components failing follows the binomial distribution:

$$\mathsf{P}(k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{2.6}$$

Therefore, the rate of CCF events with exactly k components failing is:

$$\lambda_{G,k} = \nu \binom{n}{k} p^k (1-p)^{n-k} \tag{2.7}$$

The BFR model has the advantage of using physically interpretable parameters (ν and p) while requiring fewer parameters than the Basic Parameter Model.

2.3.2.5 Beta Factor Model

A simplified single-parameter model where all common cause failures are assumed to fail all components in the group. The model uses a single parameter β, representing the fraction of total failure probability attributed to common causes:

$$Q_I = Q_T(1-\beta) \quad \text{(independent failures)}$$
$$Q_m = Q_T\beta \quad \text{(complete common cause failure)}$$

This model is useful for initial screening and when data is limited.

2.3.2.6 Mapping Between Models

Relationships exist between the parameters of different CCF models, allowing conversion of parameters from one model to another. For example, the Alpha factors can be derived from MGL parameters as:

$$\alpha_1 = \frac{1-\beta}{1+\beta(\gamma-1)}$$
$$\alpha_2 = \frac{2\beta(1-\gamma)}{1+\beta(\gamma-1)}$$
$$\alpha_3 = \frac{3\beta\gamma}{1+\beta(\gamma-1)}$$

Industry databases such as the International Common Cause Failure Data Exchange (ICDE) [1] and the NRC's CCF database [83] provide valuable sources for parameter estimation.

2.4 Quantifying Event Sequences

Because risk analysis requires knowing how likely each branch in the tree is, event trees rely heavily on *conditional probabilities*. Let

$$p(I) \;\equiv\; \Pr(I)$$

be the probability (or frequency) of the initiating event. For each functional event F_k, define

$$p\big(F_k^{\text{succ}} \mid I, F_1, \ldots, F_{k-1}\big) \quad \text{and} \quad p\big(F_k^{\text{fail}} \mid I, F_1, \ldots, F_{k-1}\big),$$

which describe the likelihood of success or failure given all prior outcomes.

An *end-state* X_j arises from a particular chain of successes/failures:

$$\big(I, F_1^{\alpha_1}, F_2^{\alpha_2}, \ldots, F_n^{\alpha_n}\big) \longrightarrow X_j,$$

where each $\alpha_k \in \{\text{succ}, \text{fail}\}$. The probability of reaching X_j is the product of:

1. The initiating event probability $p(I)$.
2. The conditional probabilities of each functional event's success or failure.

Formally, if ω_j denotes the entire branch leading to end-state X_j, then

$$p(\omega_j) \;=\; p(I) \times \prod_{k=1}^{n} p\big(F_k^{\alpha_k} \mid I, F_1^{\alpha_1}, \ldots, F_{k-1}^{\alpha_{k-1}}\big). \tag{2.8}$$

Equation 2.8 shows that an event tree can be represented by a product (logical AND) of the relevant Boolean variables for the initiating event and each functional events success/failure. The union of all the branches spans the full sample space of scenario outcomes generated by I and the subordinate functional events. Collecting all branches via logical OR in this way yields a disjunction of these products, precisely matching the standard structure of a Boolean expression in Disjunctive Normal Form (DNF). This idea is explored further in Section 3.2.3.

2.4.1 Exact Methods for Calculating Probabilities

Beyond describing which combinations of basic events can fail, most Fault Tree Analysis (FTA) requires a quantitative assessment of the *likelihood* that the top event ultimately occurs. This section details how probabilities are embedded within the fault tree structure, how to compute the top event's failure probability (or system *unreliability*), and some common methods for handling large or dependent fault trees.

2.4.1.1 Assigning Probabilities to Basic Events

Let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be the set of basic events in the fault tree F. Each basic event b is associated with a *failure probability* $p(b) \in [0, 1]$. Interpreted as a Boolean random variable X_b, event b takes value 1 (failure) with probability $p(b)$ and value 0 (success) with probability $1 - p(b)$. Thus,

$$\Pr\big[X_b = 1\big] \;=\; p(b), \quad \Pr\big[X_b = 0\big] \;=\; 1 - p(b).$$

In the simplest *single-time* analysis, each basic event is either failed or functioning for the entire time horizon under study, and no component recovers once it has failed.

2.4.1.2 Event Probability Under Independence

If we assume that all basic events fail independently, then for any set $S \subseteq \mathcal{B}$ of failed basic events,

$$\Pr\big[\text{basic events in } S \text{ fail and others succeed}\big] \;=\; \prod_{b\in S} p(b) \;\times \prod_{b\notin S} [1-p(b)].$$

Recall from Section 2.3 that the top event T (also called TE) fails given S precisely if $\pi_F(S,T)=1$. Hence, the probability that the top event fails is the sum of these independent configurations S for which $\pi_F(S,T)=1$:

$$\Pr\big[X_t=1\big] \;=\; \sum_{S\subseteq\mathcal{B}} \Big[\pi_F(S,T) \prod_{b\in S} p(b) \prod_{b\notin S} \big[1-p(b)\big]\Big]. \tag{2.9}$$

A direct computation of (2.9) often becomes unwieldy for large FTs because of the exponential number of subsets S. However, if the fault tree is *simple* (no shared subtrees) and each gate in $\mathcal{G}$ has independent inputs, one may propagate probabilities *bottom-up* through the DAG using basic probability rules:

$$\Pr[g=1] \;=\; \prod_{x\in I(g)} \Pr[x=1], \qquad \text{if gate } g \text{ is AND,}$$

$$\Pr[g=1] \;=1\;-\; \prod_{x\in I(g)} \big[1-\Pr[x=1]\big], \qquad \text{if gate } g \text{ is OR,}$$

$$\Pr[g=1] \;=1\;-\; \Pr[x=1], \qquad \text{if gate } g \text{ is NOT (single input } x\text{),}$$

$$\Pr[g=1] \;=\; \sum_{j=k}^{|I(g)|} \;\sum_{\substack{A\subseteq I(g)\\ |A|=j}} \prod_{x\in A} \Pr[x=1] \prod_{x\in I(g)\setminus A} \big[1-\Pr[x=1]\big], \qquad \text{if gate } g \text{ is } \mathrm{VOT}(k/n),$$

$$\Pr[g=1] \;=\; \sum_{\substack{A\subseteq I(g)\\ |A| \text{ is odd}}} \prod_{x\in A} \Pr[x=1] \prod_{x\in I(g)\setminus A} \big[1-\Pr[x=1]\big], \qquad \text{if gate } g \text{ is XOR.}$$

2.4.1.3 Inclusion-Exclusion: Probability Computation under Independence

When each X_i is modeled as a Bernoulli random variable taking the value 1 with probability

$$p_i \;=\; P(X_i=1),$$

assumed independent of the other inputs, one can write for each k-element subset $S\subseteq\{1,\dots,n\}$:

$$A_S \;=\; \bigwedge_{i\in S} \{X_i=1\}, \quad P\big(A_S\big) \;=\; \prod_{i\in S} p_i.$$

The event $Y=1$ in Eq. (2.4) is the union of all such events A_S for $|S|=k$. Hence, in principle, to compute

$$P\Big(Y=1\Big) \;=\; P\Big(\bigvee_{|S|=k} A_S\Big),$$

one may apply the inclusion-exclusion principle. Specifically:

$$P\Big(\bigvee_{|S|=k} A_S\Big) \;=\; \sum_{|S|=k} P(A_S) \;-\; \sum_{1\le\alpha_1<\alpha_2\le M} P\Big(A_{S_{\alpha_1}} \cap A_{S_{\alpha_2}}\Big) \;+\; \dots \tag{2.10}$$

where $M = \sum_{j=k}^{n} \binom{n}{j}$ is the total number of events A_S of interest. Unfortunately, direct enumeration of these intersections can become infeasible for large n.

Worst-Case Example using 3-of-5 Voting Logic In many PRA tools, the $\mathrm{VOT}(k/n)$ gate is provided as a built-in element. Internally, these tools may expand Eq. (2.4) into an OR-of-ANDs or maintain a more compact representation. Under independence assumptions, standard failure-probability calculations proceed by summing over combinations of basic-event failures. However, an explicit expansion incurs the combinatorial factor

$$\binom{n}{k} \;+\; \binom{n}{k+1} \;+\; \dots \;+\; \binom{n}{n},$$

which can grow quickly as n increases. For example, consider the case where $k = 3, n = 5$. Using the notation from Section 2.3.1, the compressed form of the gate output is

$$Y \;=\; \mathrm{VOT}(3/5)\big(X_1, X_2, X_3, X_4, X_5\big) \;=\; \big[\, X_1 + X_2 + X_3 + X_4 + X_5 \;\ge\; 3\big].$$

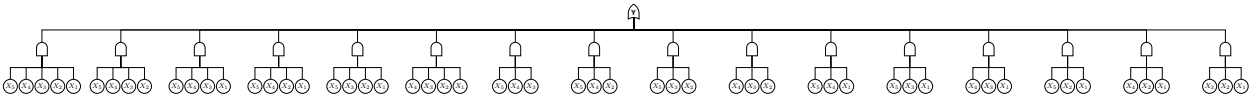


Figure 2.2: 3-of-5 voting logic, expanded as (AND-OR) Disjunctive Normal Form (DNF)

Since “at least 3 of 5” means any subset of size 3, 4, or 5 suffices, an explicit OR-of-ANDs expansion has the following terms:

$$\begin{aligned} Y \;=\; & \underbrace{X_1X_2X_3 \;\vee\; X_1X_2X_4 \;\vee\; X_1X_2X_5 \;\vee\; X_1X_3X_4 \;\vee\; X_1X_3X_5 \;\vee\; X_1X_4X_5 \;\vee\; X_2X_3X_4 \;\vee\; X_2X_3X_5 \;\vee\; X_2X_4X_5 \;\vee\; X_3X_4X_5}_{\text{subsets of size 3}} \\ & \vee\; \underbrace{X_1X_2X_3X_4 \;\vee\; X_1X_2X_3X_5 \;\vee\; X_1X_2X_4X_5 \;\vee\; X_1X_3X_4X_5 \;\vee\; X_2X_3X_4X_5}_{\text{subsets of size 4}} \\ & \vee\; \underbrace{X_1X_2X_3X_4X_5}_{\text{subset of size 5}}. \end{aligned} \tag{2.11}$$

Assume that each X_i is a Bernoulli random variable taking the value 1 with probability p_i, independently of the others. Then

$$p_i \;=\; P\big(X_i = 1\big), \quad i = 1, 2, 3, 4, 5.$$

We seek $P(Y = 1)$, i.e., the probability that at least three of the X_i are 1. A convenient way to write this is

to sum, for $j = 3, 4, 5$, the probabilities that exactly j of the inputs are 1:

$$P\big(Y = 1\big) \;=\; \sum_{j=3}^{5} \sum_{\substack{S \subseteq \{1,2,3,4,5\} \\ |S|=j}} \prod_{i \in S} p_i \prod_{i \notin S} (1 - p_i).$$

Exactly three of the five inputs are 1 (there are $\binom{5}{3} = 10$ such terms):

$$\begin{aligned}
&p_1 p_2 p_3 \,(1-p_4)\,(1-p_5) \;+\; p_1 p_2 p_4 \,(1-p_3)\,(1-p_5) \;+\; p_1 p_2 p_5 \,(1-p_3)\,(1-p_4) \;+ \\
&p_1 p_3 p_4 \,(1-p_2)\,(1-p_5) \;+\; p_1 p_3 p_5 \,(1-p_2)\,(1-p_4) \;+\; p_1 p_4 p_5 \,(1-p_2)\,(1-p_3) \;+ \\
&p_2 p_3 p_4 \,(1-p_1)\,(1-p_5) \;+\; p_2 p_3 p_5 \,(1-p_1)\,(1-p_4) \;+\; p_2 p_4 p_5 \,(1-p_1)\,(1-p_3) \;+ \\
&p_3 p_4 p_5 \,(1-p_1)\,(1-p_2).
\end{aligned}$$

Exactly four of the five inputs are 1 (there are $\binom{5}{4} = 5$ such terms):

$$\begin{aligned}
&p_1 p_2 p_3 p_4 \,(1-p_5) \;+\; p_1 p_2 p_3 p_5 \,(1-p_4) \;+\; p_1 p_2 p_4 p_5 \,(1-p_3) \\
&\quad+\; p_1 p_3 p_4 p_5 \,(1-p_2) \;+\; p_2 p_3 p_4 p_5 \,(1-p_1).
\end{aligned}$$

All five inputs are 1 (there is $\binom{5}{5} = 1$ such term):

$$p_1 p_2 p_3 p_4 p_5.$$

Summing all of the above terms yields $P(Y = 1)$, the fully expanded probability of a 3-of-5 vote under independence.

Complexity of the Inclusion-Exclusion Approach Observe that the union in Eq. (2.10) comprises M events, where

$$M \;=\; \sum_{j=k}^{n} \binom{n}{j}.$$

The inclusion-exclusion principle for a union of M events involves sums of intersections of size r, for $r = 1, \ldots, M$. The total number of terms is

$$\sum_{r=1}^{M} \binom{M}{r} \;=\; 2^M - 1,$$

which implies a worst-case computational complexity on the order of

$$\mathcal{O}\big(2^M\big).$$

Moreover, for $k \approx n/2$, the binomial coefficients $\binom{n}{k}$ become largest, so M can itself grow exponentially in n. Consequently, the inclusion-exclusion expansion may require up to $\mathcal{O}\big(2^{2^n}\big)$ operations for moderately large n.

2.4.2 Methods for Approximating Probabilities

2.4.2.1 The Rare-Event Approximation

When each basic event (b) has a sufficiently small failure probability (p(b)), the likelihood of multiple minimal cut sets failing simultaneously is often negligible. Under these conditions, one may approximate the failure probability of the top event by treating each Minimal Cut Sets as though it fails independently. Specifically, let $\mathrm{MCS}(T)$ denote the set of all minimal cut sets that either have cardinality up to some *truncation value* (T) or exceed a chosen probability threshold. Then one replaces the exact summation over all MCS by summing only over $\mathrm{MCS}(T)$, yielding

$$\Pr\big[X_t = 1\big] \approx \sum_{C \in \mathrm{MCS}(T)} \prod_{b \in C} p(b). \tag{2.12}$$

This approach is justified in highly reliable systems where multi-cut-set failures have low probability. Moreover, it reduces the computational burden by screening out higher-order or low-probability minimal cut sets. The choice of (T) typically adheres to engineering practice: for example, in a system designed to tolerate any single failure (often referred to as $(N-1)$ redundancy), all minimal cut sets of size up to $(N-1)$ might be considered while larger cut sets, deemed improbable, are excluded.

Error Bound for Truncated Approximations A natural way to measure the quality of a truncated approximation is by comparing it to the exact probability of top-event failure. Denote the exact probability by

$$\Pr\big[X_t = 1\big],$$

and the truncated approximation, which sums only over the minimal cut sets in $\mathrm{MCS}(T)$, by

$$\Pr_T\big[X_t = 1\big] = \sum_{C \in \mathrm{MCS}(T)} \prod_{b \in C} p(b).$$

Then the *error* associated with truncation up to (T) is

$$\Delta(T) = \Pr\big[X_t = 1\big] - \Pr_T\big[X_t = 1\big]. \tag{2.13}$$

Under the assumption that failures are sufficiently rare and interactions among higher-order minimal cut sets are negligible, an *upper bound* for this error may be obtained by summing the omitted terms:

$$\big|\Delta(T)\big| \leq \sum_{C \in \mathrm{MCS} \setminus \mathrm{MCS}(T)} \prod_{b \in C} p(b). \tag{2.14}$$

In practice, computing $\Delta(T)$ or its bound usually requires identifying and evaluating all minimal cut sets outside $\mathrm{MCS}(T)$, which may still be tractable if the omitted sets are large, unlikely, or both. Consequently, choosing a suitable truncation parameter, T (by size or probability threshold), ensures that the unaccounted failure modes contribute negligibly to the overall system unreliability.

2.4.2.2 The Min-Cut Upper Bound

Another method for bounding the probability of a top event interprets system failure as the union of all minimal cut set (MCS) failures. The most direct upper bound is obtained by applying the union bound (Boole's inequality), which states that the probability of the union of events is no greater than the sum of their individual probabilities. For a set of minimal cut sets $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$, where each C_i is a set of basic events, the union bound yields:

$$\Pr[X_t = 1] \leq \sum_{C \in \text{MCS}} \prod_{b \in C} p(b), \tag{2.15}$$

where $p(b)$ is the probability of basic event b.

However, most modern PRA tools, including SAPHIRE [112], implement a slightly tighter upper bound, known as the *minimal cut set upper bound*, which is given by:

$$\Pr[X_t = 1] \leq 1 - \prod_{C \in \text{MCS}} \left[1 - \prod_{b \in C} p(b)\right]. \tag{2.16}$$

This expression accounts for the fact that the system fails if *any* cut set fails, and thus computes the probability that at least one cut set occurs, assuming independence between cut sets. It is always at least as large as the true probability, and is generally less conservative than the simple union bound, especially when cut sets overlap.

The min-cut upper bound is exact if and only if all minimal cut sets are mutually disjoint (i.e., share no basic events). In practice, cut sets often overlap, leading to double counting of shared failure modes and thus a conservative overestimate. This bound remains valid regardless of the magnitude of the basic event probabilities, unlike the rare-event approximation, which assumes that cut set failures are nearly disjoint and probabilities are small.

For illustration, consider a fault tree with three minimal cut sets [106]: $C_1 = \{\mathsf{A}, \mathsf{B}\}$, $C_2 = \{\mathsf{B}, \mathsf{C}\}$, and $C_3 = \{\mathsf{D}\}$, with $p(\mathsf{A}) = p(\mathsf{B}) = p(\mathsf{C}) = 0.7$ and $p(\mathsf{D}) = 0.5$. The min-cut upper bound is:

$$\begin{aligned}
\Pr(X) &\leq 1 - [1 - p(\mathsf{A})p(\mathsf{B})]\,[1 - p(\mathsf{B})p(\mathsf{C})]\,[1 - p(\mathsf{D})] && (2.17)\\
&= 1 - (1 - 0.49)(1 - 0.49)(1 - 0.5) && (2.18)\\
&= 1 - (0.51)(0.51)(0.5) && (2.19)\\
&= 1 - 0.13005 = 0.86995. && (2.20)
\end{aligned}$$

This value is a conservative estimate of the true top event probability.

2.4.3 Probability Estimation using Monte Carlo Sampling

Monte Carlo randomized algorithms or methods provide a versatile framework for approximating expectations, probabilities, and other quantities of interest by simulating random observations from an underlying distribution in which the efficiency is known with certainty. At its core, a Monte Carlo estimator uses repeated

random draws to approximate quantities such as

$$\mathbb{E}[f(X)] \;=\; \int f(x)\,p(x)\,\mathrm{d}x \;\;\approx\;\; \frac{1}{N}\sum_{i=1}^{N} f\big(x^{(i)}\big), \tag{2.21}$$

where $x^{(1)}, x^{(2)}, \ldots, x^{(N)}$ are independent and identically distributed (i.i.d.) samples drawn from p. The function f is a measurable function of the random variable X. In reliability and PRA contexts, f might be an indicator of a particular event (e.g., a system failure), in which case $\mathbb{E}[f(X)]$ becomes the probability of that event.

2.4.3.1 Convergence and the Law of Large Numbers

A central theoretical result underpinning Monte Carlo sampling is the Law of Large Numbers (LLN). In one of its classical forms, the Strong LLN states:

Theorem 1 (Strong Law of Large Numbers)**.** *Let $X_1, X_2, \ldots$ be a sequence of i.i.d. random variables with finite expectation $\mathbb{E}[X_1]$. Then, with probability 1,*

$$\lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^{N} X_i \;=\; \mathbb{E}[X_1].$$

Applied to the sample estimator in Eq. (2.21), the LLN implies that as the number of samples N grows large, the average of the function values $f\big(x^{(i)}\big)$ converges to $\mathbb{E}[f(X)]$. Thus, by simply drawing enough samples, one can approximate probabilities or expectations arbitrarily well (with probability 1).

2.4.3.2 Central Limit Theorem and Error Analysis

Another classical result is the *Central Limit Theorem (CLT)*, which indicates that the Monte Carlo estimator's distribution (around its true mean) approaches a normal distribution for large N. Specifically,

Theorem 2 (Central Limit Theorem)**.** *Suppose $X_1, X_2, \ldots$ are i.i.d. random variables with mean $\mu = \mathbb{E}[X_1]$ and variance $\sigma^2 = \mathbb{V}[X_1] < \infty$. Then the sample mean satisfies*

$$\sqrt{N}\left(\frac{1}{N}\sum_{i=1}^{N} X_i - \mu\right) \;\;\xrightarrow{\mathrm{d}}\;\; \mathcal{N}(0,\sigma^2),$$

where $\xrightarrow{\mathrm{d}}$ denotes convergence in distribution.

In practical terms, the Central Limit Theorem (CLT) implies that for sufficiently large N, the sampling fluctuations of the Monte Carlo estimator around the true mean are approximately normal. The variance of this normal distribution decreases with $1/N$. Therefore, one can estimate confidence intervals, standard errors, and convergence rates by tracking empirical variance across the sample.

The above principles remain valid even when f is an indicator of a Boolean event or a composite system failure embedded in an event/fault tree. One need only be able to draw samples $\big(x^{(i)}\big)$ from the system's joint distribution over basic events (or from any suitable representation of the PRA model) and then evaluate the function f to determine system success/failure for each sample. Subsequent chapters will expand on how

these samples can be generated for event trees, fault trees, or more complex DAG-based representations.

2.4.3.3 Generating Random Numbers

Monte Carlo estimators rely on the ability to generate random realizations from a given distribution. Computers, however, do not typically provide true randomness; instead, they use Pseudo Random Number Generator (PRNG)s to produce sequences of numbers that mimic realizations from a uniform distribution on $[0, 1]$. From these *uniform* samples, one can then derive samples from more general distributions using various transformations (e.g., the *inverse transform* method, acceptance-rejection, composition methods, or specialized sampling algorithms).

A PRNG is formally a deterministic function that, given an initial *seed*, generates a long sequence of values in $(0, 1)$. Popular choices include:

- *Linear Congruential Generators (LCG)*, which use a recurrence of the form

  $$X_{n+1} \;=\; (a\,X_n + c) \;\bmod\; m,$$

  then normalize $\frac{X_{n+1}}{m}$ to produce a pseudo-random variate in $(0, 1)$.
- *Mersenne Twister*, which generates high-quality pseudo-random numbers with a very long period (e.g., $2^{19937} - 1$).
- *Philox* or other counter-based methods that deliver high performance and reproducible streams across parallel computations.

While these methods provide deterministic sequences, strong design ensures that the resulting outputs pass numerous statistical tests for randomness. If the seed is chosen randomly (or from a secure source), these methods can approximate uniformity closely enough for most Monte Carlo studies.

Random Variates via Transformations Given access to uniform samples $U \sim \mathrm{Unif}(0, 1)$, one can construct samples from many other distributions. Two widely used techniques are:

1. **Inverse Transform Sampling:** Suppose a continuous variable X has Cumulative Distribution Function (CDF) $F_X(x)$. If $U \sim \mathrm{Unif}(0, 1)$, then $X = F_X^{-1}(U)$ follows the same distribution as X. More precisely,

   $$P\big[X \le x\big] \;=\; P\big[F_X^{-1}(U) \le x\big] \;=\; P\big[U \le F_X(x)\big] \;=\; F_X(x),$$

   provided F_X is continuous and strictly increasing.
2. **Acceptance-Rejection:** For certain distributions where the inverse CDF is not straightforward, one can sample from an easier *proposal distribution* $q(x)$ that bounds the targeted density $p(x)$. Specifically, if $p(x) \le M\,q(x)$ for all x, then:

   (a) Draw $Y \sim q(\cdot)$ and $Z \sim \mathrm{Unif}(0, 1)$.

   (b) Accept Y if $Z \le \frac{p(Y)}{M\,q(Y)}$. Otherwise, reject and repeat.

   The accepted sample Y follows distribution $p(x)$.

Boolean Events as Discrete Random Variables Many variables are *discrete*, often Bernoulli (success/-failure) or categorical (e.g. multiple failure modes). Generating $\{0, 1\}$-valued samples is then straightforward, since for each basic event b,

$$\Pr[b = 1] \;=\; p(b), \quad \Pr[b = 0] \;=\; 1 - p(b).$$

Given a uniform variate U, one sets

$$b \;=\; \begin{cases} 1, & U \le p(b), \\ 0, & \text{otherwise.} \end{cases}$$

This approach naturally extends to multi-categorical events. More complex dependencies among events can also be captured by specifying appropriate conditional distributions.

Extending Boolean Events to Continuous Random Variables A *continuous* random variable Y has a Probability Density Function (PDF) $f_Y(y)$ on a continuous domain $\mathcal{Y} \subseteq \mathbb{R}$. Common examples in reliability include:

- **Exponential Distribution**, often used to model times to failure under a constant hazard rate λ. Its PDF is
$$f_Y(y) \;=\; \lambda\, e^{-\lambda y}, \quad y \ge 0.$$
- **Weibull Distribution**, with flexible shape parameter $\beta > 0$ and scale parameter $\alpha > 0$. Its PDF is
$$f_Y(y) \;=\; \frac{\beta}{\alpha}\Big(\frac{y}{\alpha}\Big)^{\beta-1} \exp\Big[-\big(y/\alpha\big)^{\beta}\Big], \quad y \ge 0.$$
- **Lognormal Distribution**, where $\log(Y)$ follows a normal distribution. This is sometimes employed for components whose lifetimes span multiple orders of magnitude.

Continuous random variables typically arise when modeling the *time dimension*: for instance, the time until a valve sticks closed, or the moment when a pipe experiences a critical crack. One can then generate a Bernoulli indicator for whether the failure has occurred by time t using

$$\Pr[Y \le t] \;=\; \int_0^t f_Y(y)\,\mathrm{d}y \;=\; F_Y(t),$$

where F_Y is the Cumulative Distribution Function (CDF) of Y. Evaluating this probability at each Monte Carlo trial and comparing against a uniform random variate yields a discrete failure indicator. Hence, continuous distributions can be mapped to discrete states at any chosen time horizon.

2.5 Generating Cut Sets and Implicants

Once the fault tree or event tree model has been constructed, the *qualitative* phase of risk characterization begins. The objective is to identify unique combinations of basic events that are of interest because they either *cause* the top event (system failure) or *guarantee* its prevention (system success). These combinations are formalized by the logic concepts defined below.

Implicants are general combinations of events (either failure or success) that satisfy the top event. *Cut sets* are specialized implicants containing only failure events that cause system failure. *Path sets* are specialized implicants of the complement function ($\overline{T}$) containing only success events that ensure system operation. *Minimal cut sets* are prime implicants containing only failure events, representing the minimal ways the system can fail. *Maximal path sets* are prime implicants of $\overline{T}$ containing only success events, representing the maximal ways the system can operate successfully.

2.5.1 Implicants

Let $T(\mathbf{x})$ be the Boolean top event function of the model and $I \subseteq \{x_1, \ldots, x_n\}$ be a set of basic events. A set I is an *implicant* of T if

$$T(\mathbf{1}_I) = 1$$

where $\mathbf{1}_I$ is the truth vector that sets the basic events in I to TRUE. Implicants can contain both failure and success events depending on the analysis context.

- *Cut set*: An implicant containing only failure events.
- *Path set*: The complement of an implicant containing only success events.

2.5.1.1 Prime Implicants

A *prime* implicant P is an implicant that is not a subset of any other implicant. Formally, P is a prime implicant if:

1. $T(\mathbf{1}_P) = 1$
2. $\forall P' \subset P, T(\mathbf{1}_{P'}) = 0$

2.5.1.2 Essential Prime Implicants

An *essential* prime implicant π is a prime implicant that contains at least one minterm that is not covered by any other prime implicant, i.e., there exists at least one basic event $\mathbf{a}$ such that $T(\mathbf{a}) = 1$ and $\mathbf{a}$ satisfies π but does not satisfy any other prime implicant of T. This property guarantees that π must appear in every irredundant disjunctive normal form representation of T.

2.5.2 Path Sets

A *path set* P is a set of success events such that when all events in P occur, the system operates successfully. If $\overline{T}$ represents the complement of the top event function (system success), then:

$$\overline{T}(\mathbf{1}_P) = 1 \text{ where } P \text{ contains only success events}$$

2.5.2.1 Maximal Path Sets

A *maximal path set* is a path set to which no basic event can be added while still ensuring system success. A path set X is maximal if:

1. $\overline{T}(\mathbf{1}_X) = 1$

2. $\forall x \notin X, \overline{T}(\mathbf{1}_{X \cup \{x\}}) = 0$

Maximal path sets represent the largest combinations of component successes that guarantee system operation.

2.5.3 Cut Sets

A *cut set* C is an implicant containing only failure events such that when all events in C occur, the system fails. Formally, for a Boolean function T representing the top event:

$$T(\mathbf{1}_C) = 1 \text{ where } C \text{ contains only failure events}$$

2.5.3.1 Minimal Cut Sets

A *minimal cut set* is a cut set which causes the failure of the system, but when a basic event is removed from the cut set, it does not cause system failure anymore [3]. A cut set M is minimal if:

1. $T(\mathbf{1}_M) = 1$
2. $\forall M' \subset M, T(\mathbf{1}_{M'}) = 0$

Minimal cut sets represent the smallest combinations of component failures that can cause system failure.

2.5.3.2 Method for Obtaining Minimal Cut Sets (MOCUS)

Originally proposed by Fussell and Vesely in 1974 [64], the MOCUS algorithm remains one of the most widely deployed techniques for top-down generation of minimal cut sets in PRA tools. The procedure starts from the TOP event of a fault tree and repeatedly expands each logic gate until only basic events remain.

Let the following symbols be defined:

- w identifier of a logic gate
- φ identifier of a basic event
- $\rho_{w,i}$ *i*-th input to gate w
- λ_w fan-in (number of inputs) of gate w
- $\Delta_{x,y}$ entry located at row x, column y of the Boolean Indicated Cut Sets (BICS) matrix
- $x_{\max}$ ($y_{\max}$) current maximum row (column) index in Δ

Recursive expansion Starting with $\Delta_{1,1} = w_{\text{TOP}}$, every occurrence of a gate identifier is eliminated via the following transformation rules:

$$\Delta_{x,y} = \rho_{w,1}, \tag{2.22}$$

$$\Delta_{x,y_{\max}+1} = \begin{cases} \rho_{w,\pi}, & \text{if } w \text{ is an AND gate,} \\ \begin{cases} \Delta_{x,n}, & n = 1, \ldots, y_{\max},\ n \neq y, \\ \rho_{w,\pi}, & n = y, \end{cases} & \text{if } w \text{ is an OR gate.} \end{cases} \tag{2.23}$$

with $\pi = 2, 3, \ldots, \lambda_w$. Equation (2.22) seeds the matrix, while Equation (2.23) generates new columns (AND) or rows (OR) until all w symbols have been replaced by φ symbols, thereby producing the complete set of BICS.

Two refinement steps convert BICS to MCS:

1. Remove duplicate basic events within each row.
2. Discard any BICS that is a superset of another BICS.

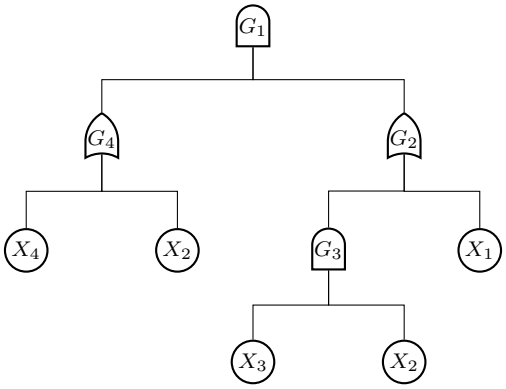


Figure 2.3: Sample fault tree for MOCUS demonstration [26].

Table 2.1 summarizes the sample fault tree by matching the formulas provided for the algorithm.

Table 2.1: Structural data for the sample fault tree in Figure 2.3.

w	Gate type	λ_w	$\rho_{w,i}$
G_1	AND	2	G_2 G_4
G_2	OR	2	X_1 G_3
G_4	OR	2	X_2 X_4
G_3	AND	2	X_2 X_3

Figure 2.4 schematically depicts the evolution of the $\Delta_{x,y}$ matrix for the fault tree in Figure 2.3.

After gate expansion, elimination of duplicates (e.g. event X_2) and removal of supersets (e.g. $\{X_2, X_4, X_3\}$)

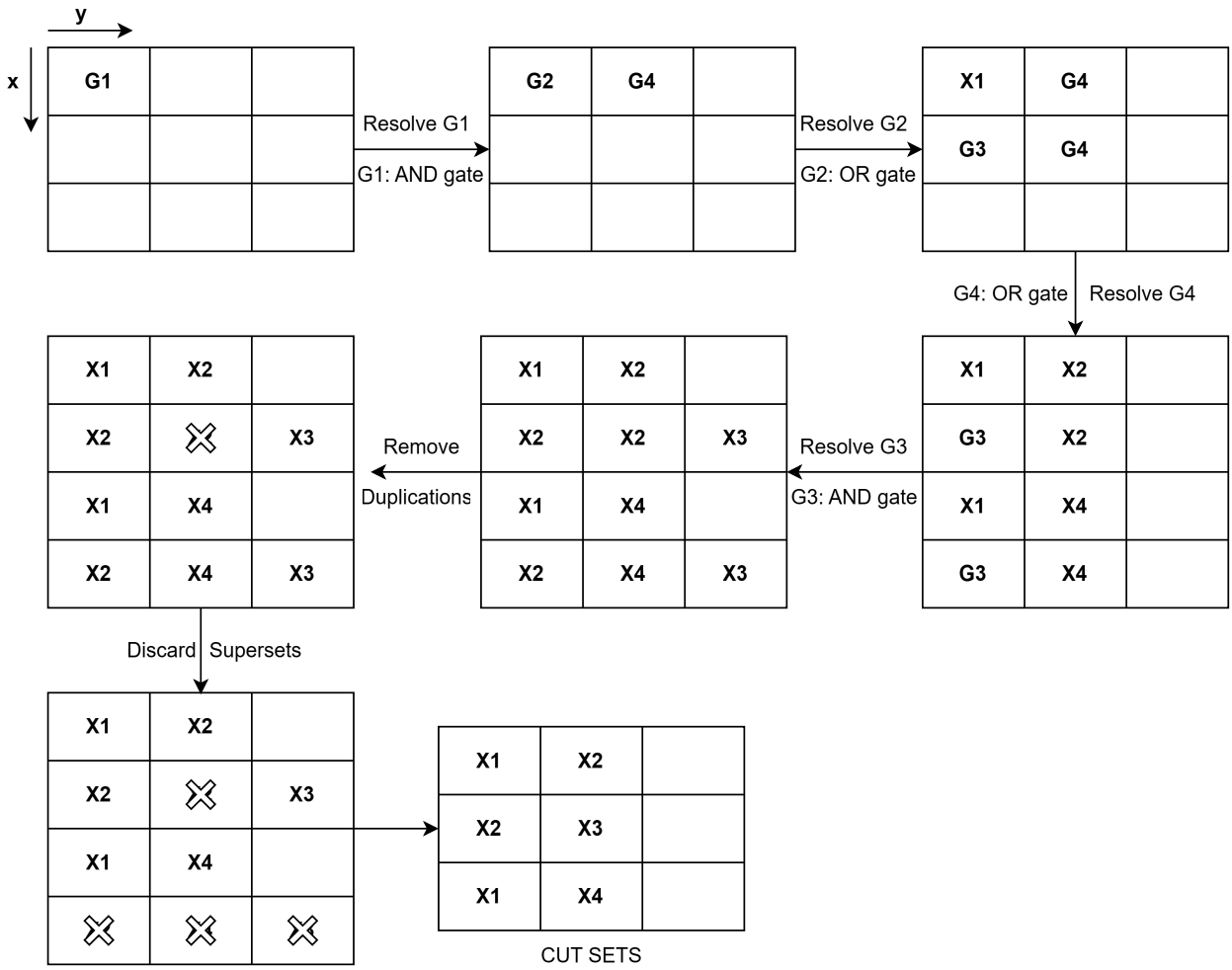


Figure 2.4: Schematic representation of MOCUS algorithm application to the sample fault tree [26].

yield the MCS family:

$$\big\{\{X_1, X_2\},\ \{X_2, X_3\},\ \{X_1, X_4\}\big\}.$$

Although algorithmically elegant, the recursive nature of MOCUS can incur considerable computational overhead for large, deeply nested fault trees. Nonetheless, virtually every PRA tool integrates some variant of MOCUS for minimal cut-set calculation.

2.6 Computing Importance Measures

Importance measures quantify the contribution of basic events to system reliability or risk. They provide insights into which components or events are most critical to system performance, guiding resource allocation for maintenance, design improvements, and risk management. PRA tools typically calculate the following importance measures to support comprehensive reliability analysis.

Conditional Importance The *Conditional Importance* (CI) of a basic event i measures the probability that event i contributes to system failure, given that the system has failed. It represents the fraction of system failures that involve the basic event.

$$\text{CI}_i = \frac{P(i \text{ contributes to system failure} \mid \text{system failure})}{P(\text{system failure})} = \frac{P(i \text{ critical})}{P(\text{system failure})}$$

This is calculated by identifying minimal cut sets containing the basic event and determining the proportion of failure probability attributed to these cut sets.

Marginal Importance The *Marginal Importance* (MI), also known as Birnbaum importance, measures the rate of change in system reliability with respect to the reliability of component i. It quantifies how sensitive the system failure probability is to changes in the failure probability of the component.

$$\text{MI}_i = \frac{\partial P(\text{system failure})}{\partial P(i \text{ fails})} = P(\text{system fails} \mid i \text{ fails}) - P(\text{system fails} \mid i \text{ succeeds})$$

This is computed by evaluating the difference in system failure probability when the component is assumed failed versus successful.

Potential Importance The *Potential Importance* (PI) measures the maximum possible reduction in system failure probability that could be achieved by improving component i. It indicates the potential benefit of perfect reliability for a component.

$$\text{PI}_i = P(\text{system failure}) - P(\text{system failure} \mid i \text{ succeeds}) = P(\text{system failure}) \times (1 - \frac{1}{\text{RRW}_i})$$

This is calculated using the system failure probability under current conditions compared to the system failure probability when the component is perfectly reliable.

Diagnostic Importance The *Diagnostic Importance* (DI) measures the probability that component i has failed, given that the system has failed. It is useful for fault diagnosis and identifying likely causes of system failure.

$$\text{DI}_i = \frac{P(i \text{ fails} \mid \text{system fails})}{P(i \text{ fails})} = \frac{P(i \text{ fails} \cap \text{system fails})}{P(i \text{ fails}) \times P(\text{system fails})}$$

This is calculated by determining the conditional probability of component failure given system failure, normalized by the component's failure probability.

Criticality Importance The *Criticality Importance* (CRI) combines the Marginal Importance with the probability of component failure. It measures the contribution of component i to the overall system failure probability.

$$\text{CRI}_i = \text{MI}_i \times \frac{P(i \text{ fails})}{P(\text{system fails})} = \frac{P(i \text{ fails}) \times [P(\text{system fails} \mid i \text{ fails}) - P(\text{system fails} \mid i \text{ succeeds})]}{P(\text{system fails})}$$

This is computed by multiplying the Marginal Importance by the ratio of component failure probability to system failure probability.

Risk Achievement Worth The *Risk Achievement Worth* (RAW) measures the factor by which system failure probability increases when component i is assumed to have failed. It indicates the importance of maintaining the current reliability of the component.

$$\text{RAW}_i = \frac{P(\text{system fails} \mid i \text{ fails})}{P(\text{system fails})}$$

RAW is computed by comparing the system failure probability when the component is assumed failed to the baseline system failure probability.

Risk Reduction Worth The *Risk Reduction Worth* (RRW) measures the factor by which system failure probability decreases when component i is assumed perfectly reliable. It indicates the potential value of improving component reliability.

$$\text{RRW}_i = \frac{P(\text{system fails})}{P(\text{system fails} \mid i \text{ succeeds})}$$

RRW is calculated by comparing the baseline system failure probability to the system failure probability when the component is assumed perfectly reliable.

3 QUANTITATIVE RISK ASSESSMENT AS KNOWLEDGE COMPILATION

The risk model-language has been developed for efficient and tractable model building and representation of risk and reliability of the systems at hand. The semantic distinctions between Initiating Events, Event Tree, and Fault Trees provides human-interpretable language that allows engineers to effectively reason about potential failure events and their consequences, describe failure propagation paths between them, and iteratively and systematically increase description granularity of these these paths. While this is approach facilities effective model building and allows quantification of small systems, it quickly becomes intractable even for moderately sized systems. Furthermore, an inherent inhomogeneity of data structures involved, though "human-readable", complicates qualitative and quantitative computational analysis of the model as a whole.

Therefore, a unified form of PRA model is required. Firstly, such form must be deterministically generated from the human-built data structures without loss of information. Secondly, it must be "computer-friendly" and facilitate optimized quantitative and qualitative risk analyses and operations such as probability estimation, minimal cut set generation, etc. Finally, it should allow iterative updates in correspondence to the evolution of the underlying PRA model.

In this section, we will show how these requirements can be achieved by viewing risk models as Probabilistic Directed Acyclic Graph (PDAG). We show how PDAG model maps on PDAG and how standard PRA methods can be viewed as operations on graphs. Furthermore, we show how PDAGs can be viewed as collections of propositional logic statements — knowledge bases. This view provides fundamental mathematical rigor to PRA formalism, by casting PRA methods as queries over and transformation of aforementioned knowledge bases. Not only this formalism provides strong computational bounds and guarantees for the algorithms of interest, but it also explicitly separates the preparation and analysis steps of PRA model, allowing for efficient separation and querying.

3.1 Risk Models as Probabilistic Directed Acyclic Graphs

Up to this point, we have introduced ETs to capture the forward evolution of scenarios and FTs to capture the top-down decomposition of system failures. In a full-scale PRA, many ETs and FTs are linked to form a single overarching model. The goal is to represent:

1. The branching structure of multiple event trees, which may feed into one another (ET$\rightarrow$ET),
2. Multiple fault trees that themselves can reference or be referenced by other FTs (FT$\rightarrow$FT),
3. Event trees that invoke fault trees to quantify key failure probabilities (ET$\rightarrow$FT), and similarly fault trees whose outcome may direct the next branch or state in an event tree.

All of these interconnections can be consolidated into a single PDAG. In broad terms, its nodes stand for

either (i) ET nodes (initiating or functional events), (ii) FT gates or intermediate events, or (iii) basic events.

3.1.1 Basic Structure and Notation

Let us denote:

- $\{\Gamma_1, \Gamma_2, \ldots, \Gamma_M\}$ as the *collection of event trees*, where each Γ_i may represent a different initiating event or system phase. Every Γ_i is itself structured as in Section 2.2, with a set of node events and directed edges (success/failure branches).
- $\{\Phi_1, \Phi_2, \ldots, \Phi_N\}$ as the *collection of fault trees*, each built according to Section 2.3. Every Φ_j has a unique top event, an acyclic arrangement of gates (AND, OR, voting, etc.), and a set of basic events.

In a large PRA, any ET Γ_i may:

- Lead to another ET Γ_k under certain branch outcomes (ET$\rightarrow$ET).
- Include a branch that requires computing "System X fails" via a fault tree Φ_j (ET$\rightarrow$FT).

Similarly, a fault tree Φ_j may:

- Contain the top event of another fault tree Φ_k as one of its inputs (FT$\rightarrow$FT).
- Generate an outcome (e.g. subsystem fails) that triggers a branch in some event tree Γ_i.

These inter-dependencies can be organized into a single PDAG, denoted

$$\mathcal{M} \;=\; \big(\mathcal{V},\, \mathcal{A}\big),$$

where:

- $\mathcal{V}$ is the set of *all* nodes in the unified model. Each node $v \in \mathcal{V}$ has a type indicating whether it belongs to an event tree (ET-node), a fault tree (FT-gate), or is a Basic Event (BE).
- $\mathcal{A} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of directed edges. Each $(u, v) \in \mathcal{A}$ signifies a logical or probabilistic dependency from node u to node v.

By design, $\mathcal{M}$ is acyclic: no path loops back through the same node. This condition prevents paradoxical definitions of probabilities or statements (e.g., an event that depends on itself).

3.1.2 Nodes and Their Inputs

We partition $\mathcal{V}$ into three principal categories:

1. **Basic Events (BEs).** Let $\mathcal{B} \subset \mathcal{V}$ be the set of all basic events across all FTs. Each $b \in \mathcal{B}$ is associated with a failure probability $p(b) \in [0, 1]$. A node b in the PDAG has no incoming edges (i.e., it is a leaf source for the rest of the logic).
2. **Fault Tree (FT) Nodes.** Let $\mathcal{G} \subset \mathcal{V}$ be the set of *internal FT-gates or intermediate FT-events*, unified across $\{\Phi_1, \ldots, \Phi_N\}$.
    - Each FT node $g \in \mathcal{G}$ has one output event g (the node itself in the PDAG).

- The set of inputs to g may include basic events $\mathcal{B}$ (e.g., component failures) and/or other FT nodes $\mathcal{G}$ (subsystem-level events). If g is the top event of Φ_j, then it may appear as an input into a *different* FT's gate or an ET node.
- As with standard FT logic, each gate has a type in $\{\mathrm{AND},\ \mathrm{OR},\ \mathrm{VOT}(k/n),\ \ldots\}$. Its output (failure state) is a Boolean function of its inputs' failure states.

3. **Event Tree (ET) Nodes.** Let $\mathcal{E} \subset \mathcal{V}$ be the set of *ET-nodes*, each referring either to an initiating event (IE) or to a functional event within some event tree Γ_i.
   - An ET node e in Γ_i can have multiple outgoing edges, each labeled by a particular outcome (e.g., success/failure). These edges may lead to another ET node (continuing the same event tree), to the root/top event of a fault tree (to evaluate subsystem reliability), or even to the root of a *different* event tree (ET→ET).
   - As in Section 2.2, each outgoing branch from e has an associated conditional probability, conditioned on the event e itself having occurred.

Hence,

$$\mathcal{V} \;=\; \underbrace{\mathcal{B}}_{\text{basic events}} \;\cup\; \underbrace{\mathcal{G}}_{\text{FT nodes}} \;\cup\; \underbrace{\mathcal{E}}_{\text{ET nodes}},$$

and every node v in $\mathcal{V}$ has an *input set*

$$I(v) \;\subseteq\; \mathcal{B} \cup \mathcal{G} \cup \mathcal{E} \;=\; \mathcal{V} \setminus \{v\},$$

indicating which nodes feed into v. By the PDAG property, v cannot be an ancestor of itself.

3.1.3 Edge Types and Probability Assignments

Each directed edge $(u, v) \in \mathcal{A}$ belongs to one of several categories:

- **ET $\rightarrow$ ET edges. [Transfers]** These edges connect an ET node $u \in \mathcal{E}$ to another ET node $v \in \mathcal{E}$ within the same tree Γ_i or leading to a subsequent tree Γ_k. In typical diagrams, these edges denote if u occurs, then with probability $\theta_{u\to v}$ we transition to v. Probabilities on all child edges of u sum to 1, reflecting the partition of possible outcomes.
- **ET $\rightarrow$ FT edges. [Functional Events]** These edges represent the case where an ET node $u \in \mathcal{E}$ designates Check if subsystem Φ_j has failed. Formally, the next node $v \in \mathcal{G}$ is the top event (or relevant subsystem event) in fault tree Φ_j. The probability of v failing is not assigned directly on the edge but is computed via the logical structure of Φ_j.
- **FT $\rightarrow$ FT edges. [Transfer Gates]** Such edges arise when the top event (or an intermediate gate) $u \in \mathcal{G}$ of one fault tree is input to a gate $v \in \mathcal{G}$ in another fault tree. For instance, if Φ_1 captures the failure mode of a pump and Φ_2 captures the failure of a coolant subsystem that includes that same pump's top event.
- **FT $\rightarrow$ ET edges. [Initiating Events]** Less common but still possible are edges that carry an outcome of a fault tree node $u \in \mathcal{G}$ to an ET node $v \in \mathcal{E}$. For instance, an initiating event might depend on

whether a certain subsystem fails, as computed by a separate FT Φ_j. Support system FTs are one such example.

- **BEs as sources (no incoming edges).** Each basic event $b \in \mathcal{B}$ has probability $p(b)$ of failing, so $\Pr[X_b = 1] = p(b)$. These do not have incoming edges because they represent fundamental failure modes, not dependent on other events within the model.

Denote by $\theta_{u \to v}$ the *conditional probability* weighting an ET-type edge $(u \to v)$. If node u splits into children $v_1, \ldots, v_k$, then

$$\sum_{i=1}^{k} \theta_{u \to v_i} \;=\; 1, \quad \theta_{u \to v_i} \;\geq\; 0.$$

By contrast, FT-type edges do not carry numerical probabilities directly. Instead, a gate node $v \in \mathcal{G}$ aggregates its inputs' *fail/success states* via Boolean logic (AND, OR, $\mathrm{VOT}(k/n)$, etc.) to yield $\pi_{\mathcal{M}}(S, v) \in \{0, 1\}$, the node v's failure state under a set S of basic-event failures.

3.1.4 Semantics of the Unified Model

A full *scenario* in $\mathcal{M}$ extends from a designated *initial node* (often an initiating event $I \in \mathcal{E}$) forward through whichever ET or FT edges are triggered. Because no cycles exist, every path eventually terminates in either (a) an ET leaf (end-state), (b) a top event that is not expanded further, or (c) a final subsystem outcome deemed not to propagate further risk.

3.1.4.1 Failure States in the Fault Trees.

For any subset $S \subseteq \mathcal{B}$ of basic events that fail:

1. Each $b \in \mathcal{B}$ fails iff $b \in S$.
2. Each FT node $g \in \mathcal{G}$ has failure state $\pi_F(S, g)$ determined by the usual fault tree semantics (Section 2.3).

That is, a node g in a fault tree Φ_j is in failure mode when its logical gate type indicates failure is activated by the failures of its inputs (which might be other gates or basic events).

3.1.4.2 Branching in the Event Trees.

Whenever an ET node $e \in \mathcal{E}$ is reached, outgoing edges $\{(e \to e_1), (e \to e_2), \ldots\}$ partially partition the scenario space. The choice of which child e_i is realized is probabilistic, with probabilities $\theta_{e \to e_i}$.

3.1.4.3 Event-Tree to Fault-Tree Links.

If an edge $(e \to g)$ connects an ET node e to a *fault tree top event* $g \in \mathcal{G}$, the scenario path triggers the question Does g fail? The probability that g is in failure, conditional on having arrived at node e, is determined by the set $S \subseteq \mathcal{B}$ of basic events that happen to fail in that scenario plus the gate logic of Φ_j.

Altogether, scenario outcomes in $\mathcal{M}$ thus combine:

- $\mathbf{X}_{\mathcal{B}} = \{ X_b : b \in \mathcal{B}\}$, where $X_b \in \{0, 1\}$ indicates whether basic event b fails or not, and

- A chain of ET decisions or fault-tree outcomes, traveling through the PDAG until reaching a terminal node.

If all basic events are assumed independent with probabilities $\{p(b)\}$, then the *global* likelihood of a specific path ω from an initiating event I to a final outcome (and with a particular pattern of success/failure across $\mathcal{B}$) factors into products of:

1. The product of $\prod_{b\in S} p(b) \times \prod_{b\notin S}[1-p(b)]$ for the relevant $S \subseteq \mathcal{B}$.
2. The product of all ET-edge probabilities $\theta_{u\to v}$ encountered along the path ($u \in \mathcal{E}$).
3. The logical constraints from each visited fault tree node $g \in \mathcal{G}$, which impose $\pi_F(S, g) \in \{0, 1\}$ in or out of failure.

Summing over all valid paths (or equivalently over all subsets $S \subseteq \mathcal{B}$ and the result of each ET/FT branching) yields the total system risk measure, such as the probability of a severe radiological release, or the probability that a certain undesired top event emerges.

3.1.5 Formal Definition of the Unified Model

Bringing these elements together, we propose the following definition:

Definition 1 (Unified PRA Model)**.** *A* unified PRA model *is a Probabilistic Directed Acyclic Graph (PDAG)*

$$\mathcal{M} \;=\; \Big\langle \mathcal{V} \;=\; \mathcal{B}\cup\mathcal{G}\cup\mathcal{E},\;\; \mathcal{A},\;\; p(\cdot),\;\; \pi_F \Big\rangle$$

with the following properties:

1. *$\mathcal{B}$ is the set of **basic events**, each $b \in \mathcal{B}$ failing with probability $p(b)$. These nodes have no incoming edges in $\mathcal{M}$.*
2. *$\mathcal{G}$ is the set of **Fault Tree nodes**, each representing a gate or intermediate event in a fault tree. For $g \in \mathcal{G}$, the function*

$$\pi_F(S, g) \;=\; \begin{cases} 1, & \text{if fault-tree logic declares } g \text{ fails under } S \subseteq \mathcal{B}, \\ 0, & \text{otherwise.} \end{cases}$$

3. *$\mathcal{E}$ is the set of **Event Tree nodes**, each having zero or more outgoing edges. If node e has children $\{v_1, \ldots, v_k\} \subset \mathcal{V}$, then the edges $\{(e \to v_i)\}_{i=1}^{k}$ carry probabilities $\theta_{e\to v_i} \geq 0$ summing to 1.*
4. *$\mathcal{A} \subseteq (\mathcal{V} \times \mathcal{V})$ is the set of directed edges. An edge $(u \to v)$ can be:*
   - $\mathrm{ET} \to \mathrm{ET}$*: event-tree branching,*
   - $\mathrm{ET} \to \mathrm{FT}$*: an event tree referencing a fault tree's top event,*
   - $\mathrm{FT} \to \mathrm{FT}$*: linking one fault tree node to another's input,*
   - $\mathrm{FT} \to \mathrm{ET}$*: an FT outcome passed back to an event tree node,*

- $\mathrm{BE} \to \varnothing$*: no edges emanate from a basic event node.*

5. *The graph $\mathcal{M}$ is acyclic: no path in $\mathcal{A}$ returns to a previously visited node.*

3.1.6 Operations on Probabilistic Directed Acyclic Graphs

In practice, we anticipate large nuclear PRAs to contain hundreds of event trees and thousands of fault trees, sharing many basic events or subsystem-level fault trees. By embedding them in $\mathcal{M}$, one can systematically compute probabilities for any high-level risk measure (e.g., core damage, large release) by enumerating scenario paths or using specialized algorithms (e.g. binary decision diagrams, minimal cut set expansions, or simulation-based techniques). The unified PDAG structure codifies both the forward scenario expansions (ET logic) and the top-down sub-component dependencies (FT logic) without creating contradictions or cycles. Definition 1 makes manifest *which* event tree references *which* fault tree, how fault trees are chained together, and how basic events ultimately feed every higher-level node. Once built, one may:

- **Traverse** each path from an initiating event to a final end-state, propelling forward along the ET edges and evaluating any FT nodes via π_F.
- **Find minimal cut sets and path sets** by traversing the PDAG and making cuts along the way.
- **Sum** (or bound, or approximate) scenario probabilities to quantify overall risk.
- **Perform sensitivity analyses** by biasing subsets of basic-event probabilities $p(b)$ or gate dependencies.

All standard PRA methods (minimal cut sets, Monte Carlo simulation, bounding formulas, etc.) remain applicable, but now from within a single unified PDAG representation.

3.2 Transformations

3.2.1 Knowledge Compilation

Casting PRA model as PDAG allows to view the model a set of boolean propositional statements. Thus, any question that can be asked about the system, can be seen as a general reasoning (queries or transformations) over the knowledge base, defined by the propositional statements. A common approach to dealing with such problems is knowledge compilation.

Knowledge Compilation has emerged as a significant direction of research for addressing the computational intractability inherent in general propositional reasoning tasks. This approach, which has a long tradition in reasoning Artificial Intelligence, was notably structured and analyzed in the work of Darwiche and Marquis. KC fundamentally involves splitting the reasoning process into two distinct phases:

1. An **off-line compilation phase**: In this initial phase, a knowledge base (represented, for instance, as a propositional theory or formula) is transformed or "compiled" into a different representation, referred to as a tractable form, or target language. The target language is specifically chosen because it supports certain desirable properties, such as tractability for specific queries (questions) or allowing polynomial time evaluation.

2. An **on-line query-answering phase**: Once compiled, the resulting target representation is utilized to answer queries efficiently. The goal is for these query answering procedures to be polynomial time with respect to the size of the compiled representation.

The primary rationalization behind this two-phase approach is to **shift as much of the computational overhead as possible into the off-line compilation phase**. While the compilation step itself can be computationally hard, this initial cost is amortized over the potentially large number of subsequent on-line queries. This amortized cost makes the overall reasoning process more efficient when multiple queries are anticipated on the same knowledge base.

3.2.2 Negation Normal Form (NNF)

The field of knowledge compilation operates primarily on a subset of boolean expression that conform to a set of properties. In general, imposition of more properties results increased tractability of the queries, i.e. answering questions becomes easier, however, at the cost of the size of boolean expression. The primary "workhorse" of Knowledge compilation is Negation Normal Form (NNF).

Boolean Negation Normal Form (NNF) is a syntactic restriction for Boolean formulas such that negations (NOT operators) are applied only directly to variables and not to compound subformulas. Formally, a Boolean formula is in NNF if it is built from variables, their negations, conjunctions (AND), and disjunctions (OR), where NOT appears only as part of literals.

A boolean expression in NNF can be represented as a rooted directed acyclic graph, DAG. The leaf of the graph correspond to constants (0, 1) or literals (a, $\neg b$), presented in the expression. The internal nodes of the DAG correspond to AND ($\wedge$) and OR ($\vee$) gates. Internal gates cannot be associated with NOT gates.

In the context of knowledge compilation, NNF serves as a foundational target language. Knowledge bases compiled into NNF allow efficient model checking and form the basis for further restricted normal forms like Conjunctive Normal Form (CNF), Disjunctive Normal Form (DNF), Decomposable Negation Normal Form (DNNF), or Deterministic Decomposable Negation Normal Form (d-DNNF). NNF enables subsequent transformations and reasoning tasks to be performed with well-bounded complexity, as it ensures logical negation is "pushed down" to the leaves of the formula's parse tree, simplifying subsequent manipulations.

3.2.2.1 Properties of NNF

NNFs can be classified based on adherence to particular properties/restrictions. The set of NNFs that adhere to a set of properties defines a representational language, L in Table 3.2.

Decomposability An NNF is decomposable if, at every conjunction ($\wedge$) gate, the sets of variables feeding into the subformulas of its children are pairwise disjoint. That is, for any $\wedge$-node with children, the sets of variables involved in the subformulas rooted at those children share no variables. Formally, for any $\wedge$-node with children $(\alpha_1, \ldots, \alpha_n)$,

$$\mathrm{Vars}(\alpha_i) \cap \mathrm{Vars}(\alpha_j) = \emptyset \quad \forall i \neq j$$

This property ensures tractable consistency checking and supports efficient computation on the representation. The language of DNNF comprises those NNFs that are decomposable.

Determinism An NNF is deterministic if, at every disjunction ($\vee$) gate, the sets of models (i.e., assignments that make the subformulas true) computed by its children are mutually exclusive–no single assignment satisfies more than one child. Zero-overlap of assignments between children guarantees tractable model counting. Determinism together with decomposability yields Deterministic Decomposable Negation Normal Form (d-DNNF), which enables polytime validity, implicant, and model counting queries. Sentential Decision Diagram (SDD)s are a strict subset of d-DNNF with additional structure.

Smoothness An NNF is smooth if, at every disjunction ($\vee$) gate, all children depend on the same set of variables (atoms). That is, for every OR-node, the set of variables in each child's subformula is identical. Smoothness simplifies certain operations in knowledge compilation and ensures that the models of disjuncts are over a fixed set of variables. Smoothness can always be enforced on any DNNF in polynomial time without affecting succinctness. Smooth/Structured Deterministic Decomposable Negation Normal Form (sd-DNNF) enforces decomposability, determinism, and smoothness.

Flatness An NNF is flat if the circuit/tree has height at most two: the root is an AND or OR, and the children are literals or simple conjunction/disjunctions of literals. Both CNF and DNF are examples of Flat Negation Normal Form (f-NNF): In CNF, the root is AND, its children are ORs of literals (clauses); in DNF, the root is OR, its children are ANDs of literals (terms). Flatness restricts structural complexity and further subclasses are defined by additional properties: for instance, CNF requires each clause to have no repeated variables, and DNF requires each term to have unique variables.

3.2.2.2 Summary

The following sections explore common target languages that find uses in Knowledge Compilation and can be used in PRA. For each selected language we note it's main properties, construction algorithms, and tractable queries available for this languages. We focus on the following:

1. CNF — the most well-studied form, ubiquitous in solving SAT problem.
2. DNF — a form that naturally renders itself for Event Tree analysis and acts as a precursor for more sophisticated essential prime implicant search.
3. BDD, ZDD, SDD — most tractable target languages, featured in the majority of practical applications.

This is not an exhaustive list. Its intention is to provide a summary of rigorous formalism that can find uses in Probabilistic Risk Assessment (PRA).

Table 3.1: Compiled target languages, acronyms defined.

Acronym	Full form
NNF	Negation Normal Form
XAG	XOR-And-Inverter Graph
AIG	And-Inverter Graph
ANF/RNF	Algebraic/Ring Normal Form
f-NNF	Flat Negation Normal Form
DNNF	Decomposable Negation Normal Form

Continued: Compiled target languages, acronyms defined.

Acronym	**Full form**
d-NNF	Deterministic Negation Normal Form
FPRM	Fixed Polarity Reed-Muller
CNF	Conjunctive Normal Form
DNF	Disjunctive Normal Form
s-DNNF	Smooth/Structured Decomposable Negation Normal Form
d-DNNF	Deterministic Decomposable Negation Normal Form
sd-DNNF	Smooth/Structured Deterministic Decomposable Negation Normal Form
PPRM	Positive Polarity Reed-Muller
PI	Prime Implicate
IP	Prime Implicant
BCF	Blake Canonical Form
EPI	Essential Prime Implicate
EIP	Essential Prime Implicant
BDD	Binary Decision Diagram
f-BDD	Free/Read-Once Binary Decision Diagram
OBDD	Ordered Binary Decision Diagram
SDD	Sentential Decision Diagram
RoBDD	Reduced Ordered Binary Decision Diagram

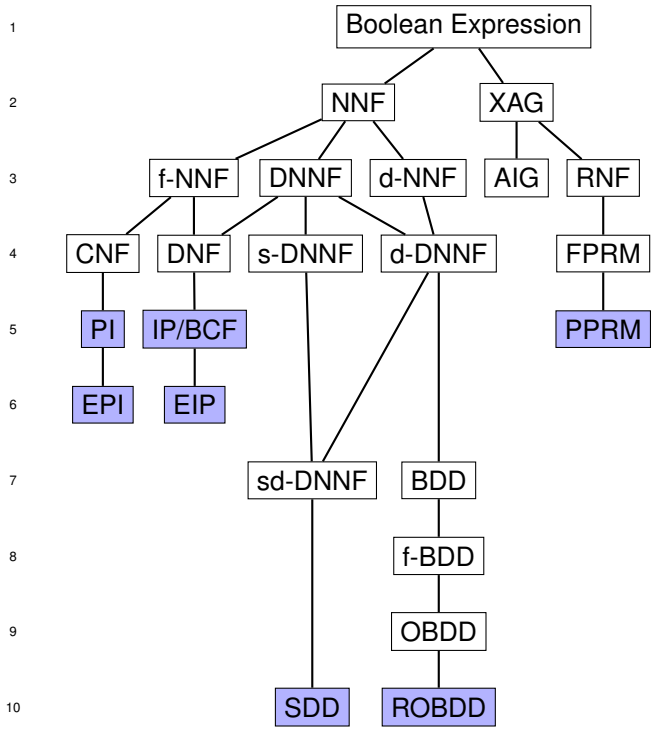


Figure 3.1: Hierarchy of compiled target languages. Blue nodes represent canonical forms.

Table 3.2: Properties of selected target languages.

Property	**NNF**	**CNF**	**DNF**	**DNNF**	**d-DNNF**	**SDD**	**ROBDD**
Decomposable			⊤	⊤	⊤	⊤	⊤
Structured Decomposable						⊤	⊤
Deterministic			⊤		⊤	⊤	⊤
Strong Determinism						⊤	⊤
Smooth						⊤	⊤
Flat		⊤	⊤				
Query Type							
Consistency (CO)	NP-c	coNP-c	polytime	polytime	polytime	polytime	polytime
Model Enumeration (ME)	NP-c	poly-delay	poly-delay	poly-delay	poly-delay	poly-delay	poly-delay
Model Counting (CT)	#P-hard	#P-hard	#P-hard	#P-hard	polytime	polytime	polytime
Equivalence (EQ)	coNP-c	coNP-c	coNP-c	coNP-c	coNP-c	polytime*	polytime†
Conditioning	polytime	polytime	polytime	polytime	polytime	polytime	polytime
Forgetting	coNP-c.	coNP-c.	coNP-c.	coNP-c.	coNP-c.	polytime	NP-hard
Boolean Combination	polytime	polytime	polytime	polytime	polytime	polytime	polytime

$*$: SDD equivalence: polytime when compressed with a fixed vtree.

†: OBDD equivalence: polytime with fixed variable order.

3.2.3 Disjunctive Normal Form (DNF)

3.2.3.1 Definition and Properties

Disjunctive Normal Form (DNF) is a classical representation language for propositional theories. A DNF formula is a disjunction of terms, with each term being a conjunction of literals (variables or their negations). Formally, a DNF formula has the form ($T_1 \vee T_2 \vee \cdots \vee T_m$), where each ($T_i = l_{i1} \wedge l_{i2} \wedge \cdots \wedge l_{ik}$), with l_{ij} — a literal. DNF is a subset of NNF and more specifically, a Flat Negation Normal Form (f-NNF). It is universal: any propositional theory has a DNF representation. DNF is not canonical; equivalent functions may have different DNF syntactic forms. Every DNF formula is also a DNNF but is not in general deterministic (and thus not always a d-DNNF).

3.2.3.2 Construction

To construct a DNF, rewrite the propositional theory as a disjunction of conjunctions of literals, with each term representing a possible partial assignment that satisfies the theory. Mechanical conversion from other forms (such as CNF) to DNF can require exponential space in the worst case. Applications DNF appears in:

1. Model-based diagnosis, where explicit representation of models is useful for enumerative reasoning.
2. Knowledge compilation pipelines as a source or target language and as an intermediary form in bottom-up compilation to tractable representations.
3. Problems requiring efficient model enumeration, such as explanation generation or solution space exploration.

3.2.3.3 Compilers and Implementations

Bottom-Up Compilation from DNF to OBDD/ SDD Each DNF term (conjunction of literals) is individually compiled into the target structure (OBDD or SDD). Terms are then combined using the “Apply” (disjunction) function, which is polynomial-time in the size of the operands.

1. OBDD Implementations: The CUDD package is a widely used library supporting OBDD construction and manipulation, including efficient Apply operations.
2. SDD Implementations: The SDD package, by the authors of SDD, is the primary implementation for Sentential Decision Diagrams, fully supporting bottom-up compilation and "Apply".

Top-Down Compilation Compilation algorithms initially designed for CNF, such as Decision- SDD compilers, can also be used directly on DNF input. These compilers operate recursively using principles from SAT solving, caching, and structural decomposition.

1. Actual Implementations: The SDD package and other SAT-based compilers (e.g., c2d for DNNF compilation) can process DNF input, although their performance is generally tuned for CNF.

No sources identify compilers specific to DNF-to- CNF conversion or tailored DNF-to-DNF simplification outside of general boolean function minimization algorithms (such as Quine-McCluskey or Espresso), which are not the focus of knowledge compilation pipelines.

3.2.3.4 Polynomial-Time Queries and Complexities

Consistency (CO) ($O(|\Delta|)$). Satisfiability is determined by checking that at least one term contains no complementary literals.

Model Enumeration (ME) Polynomial delay per model ($\mathcal{O}(|T_i|)$) per model for term (T_i). All models can be enumerated efficiently by expanding the terms.

Others All other standard queries (e.g., Validity [VA], Clausal Entailment [CE], Model Counting [CT], Equivalence [EQ], etc.): Intractable unless ($P = NP$) or $\#P = FP$ (e.g., Model Counting is $\#P$-hard).

1. Validity (VA): co-NP-complete;
2. Clausal Entailment (CE): co-NP-complete;
3. Model Counting (CT): #P-hard.

DNF is a flat, non-canonical, universal language supporting tractable consistency checking and model enumeration. It is commonly used as a source or intermediate representation in compilation pipelines targeting OBDD, SDD, or related languages, with mainstream libraries like CUDD and the SDD package implementing practical bottom-up compilation from DNF. All other semantic queries, including entailment and model counting, are computationally intractable.

Despite the aforementioned intractability, however, DNF find a unique place in PRA analysis. A pure Even-Tree Diagrams can be represented as sum-or-products boolean formulas.

3.2.3.5 Event Tree Structures as Sum-Product Networks

Consider a specific branch ω_j leading to the end-state X_j. By definition, ω_j occurs if and only if:

1. The initiating event I happens: $i = 1$.
2. For each functional event F_k, the branch specifies a particular outcome (success or failure). Suppose ω_j includes successes for some subset of indices $\alpha \subseteq \{1, \ldots, n\}$ and failures for the complementary indices. We can write this as:

$$\bigwedge_{k \in \alpha} \big(f_k^{\text{succ}} = 1\big) \quad \wedge \quad \bigwedge_{k \notin \alpha} \big(f_k^{\text{fail}} = 1\big).$$

Hence, the branch event ω_j is logically equivalent to a single *product term*:

$$\omega_j \;\equiv\; \big(i = 1\big) \;\wedge\; \bigwedge_{k \in \alpha} \big(f_k^{\text{succ}} = 1\big) \;\wedge\; \bigwedge_{k \notin \alpha} \big(f_k^{\text{fail}} = 1\big). \tag{3.1}$$

In standard Boolean notation, each literal (e.g., f_k^{succ}) is a variable that can be 0 or 1, and the branch is the $\wedge$ (AND) of those variables. An event tree describing all possible outcomes from I and the subsequent functional events can be viewed as the union (logical OR) of its disjoint branches:

$$\Omega \;=\; \omega_1 \;\cup\; \omega_2 \;\cup\; \cdots \;\cup\; \omega_m.$$

In Boolean terms, this is the $\vee$ (OR) of the product terms corresponding to each branch:

$$\Omega \equiv \omega_1 \vee \omega_2 \vee \cdots \vee \omega_m. \tag{3.2}$$

Substituting each branch's conjunction form (as in Eq. (3.1)) into Eq. (3.2) yields:

$$\Omega = \left[i \wedge \prod_{k \in \alpha_1} f_k^{\text{succ}} \wedge \prod_{k \notin \alpha_1} f_k^{\text{fail}} \right] \vee \left[i \wedge \prod_{k \in \alpha_2} f_k^{\text{succ}} \wedge \prod_{k \notin \alpha_2} f_k^{\text{fail}} \right] \vee \cdots$$

where each α_r is the set of functional events that succeed along branch r.

A standard DNF (SoP) expression in Boolean algebra is

$$(\text{literal}_1 \wedge \text{literal}_2 \wedge \cdots) \vee (\text{literal}_1' \wedge \text{literal}_2' \wedge \cdots) \vee \cdots$$

Each term in the sum (OR) is a logical AND of literals (variables or their negations). Comparing with Eq. (3.2), we see that an event tree is exactly a disjunction of terms, each term being a conjunction of the initiating event i (set to 1) and the success/failure indicators for each F_k. Since any negation can be encoded by stating whether F_k is succ ($f_k^{\text{succ}} = 1$) or fail ($f_k^{\text{fail}} = 1$), the entire event tree Ω is in DNF:

$$\Omega = \bigvee_{j=1}^{m} \Big[\bigwedge_{\ell \in \Lambda_j} (\text{appropriate literal}) \Big].$$

3.2.3.6 Tractability of Event Trees

Within $\mathrm{SP}-$networks (sum-product networks), a sum gate provides a weighted sum of child distributions, whereas a product gate factorizes them. An event tree can be cast as an $\mathrm{SP}-$network by feeding each branch's literal probabilities into product gates (one per branch), then summing over all branches with a sum gate. Once constructed, evaluating the resulting $\mathrm{SP}-$network at a specific configuration $\mathbf{x}$ or marginalizing out some of the variables is linear in the size of that network. Nevertheless, the tractability of event trees (and their circuit representations) heavily depends on their size and structure. We summarize several key considerations below:

1. *DNF size grows exponentially.*

   Suppose an event tree includes n functional events, each of which can succeed or fail. In the worst case, enumerating *all* possible outcome branches (i.e. each success/failure pattern) yields up to 2^n conjunction terms. Hence, the disjunctive normal form (DNF) representation can become exponentially large. Computing or marginalizing probabilities over such a large DNF may become prohibitively expensive if n is large enough.

2. *Evaluation linear in network size.*

   Even though the DNF itself may blow up exponentially, once the event tree is translated into an $\mathrm{SP}-$network, key inference tasks (such as evaluating it at a configuration or marginalizing over certain variables) proceed in time linear in the *compiled network size*. That said, if the underlying network has already reached exponential size in the number of events, the linear-time evaluation does not

necessarily improve the overall worst-case complexity.

3. *Approximations abound.*

   In practice, analysts often employ approximations to keep event trees tractable. One possibility is *decomposability*, a core principle behind tractable probabilistic circuits whereby each product gate operates on disjoint sets of variables. If the system decomposes (e.g. different safety barriers protect disjoint sets of equipment), one can evaluate probabilities without enumerating all branches. Another common approximation is to prune paths with very low probabilities or ignore paths that only negligibly contribute to the overall risk.

Fully enumerated event trees, regardless of being interpretable as DNF/SP networks, trade tractability for expressivity. The intuitive branching structure and conditional probability assignments make event trees easy to interpret. PRA analysts can read off and reason about the high-level scenario decomposition, incorporate domain knowledge, and analyze each branch explicitly. If the number of critical functional events is moderate, enumerating all branches remains tractable. As the depth and breadth of the tree grow, any brute-force probability computation over such a large DNF/SoP circuit is equally exponential in the worst case. Even though $\mathrm{SP}-$networks offer efficient linear-time evaluation with respect to the circuit size, the underlying circuit itself may have size exponential in n.

3.2.4 Conjunctive Normal Form (CNF)

Conjunctive Normal Form (CNF) is a specific subset of f-NNF. CNF further restricts this structure: a CNF formula is a conjunction of clauses, where each clause is a disjunction of literals. Thus, every CNF is an NNF where the formula has depth at most two: a top-level conjunction whose direct children are disjunctions of literals. Negations never apply to anything except individual variables.

Despite computational intractability of most queries, CNF is the dominant input language for SAT solvers, which employ highly optimized heuristics far surpassing brute-force complexity in practice. Many real-world verification, synthesis, and combinatorial search problems are encoded as CNF for this reason.

3.2.4.1 Key Properties

Flatness As a formula or DAG, a CNF has depth at most two. The root is a conjunction $\wedge$ whose direct children are disjunctions $\vee$ of literals (leaf nodes).

Simple-disjunction All disjunctions operate directly on literals–there are no nested or compound subformulas inside disjunctions.

Closure under conjunction The conjunction of two CNF formulas yields another CNF formula.

Non-uniqueness CNF is not canonical. Logically equivalent formulas can have very different CNF representations due to clause or literal redundancy, reordering, or other syntactic differences.

3.2.4.2 Construction and Compilation

Constructing CNF from arbitrary Boolean formulas using Tseitin encoding or distributive laws takes $(\mathcal{O}(|\varphi|))$ time and space, potentially with introduction of auxiliary variables for compactness. Any propositional formula can be converted to an equisatisfiable CNF using methods such as Tseitin encoding, which can be performed in $(\mathcal{O}(|\varphi|))$ time and size for a formula (φ). This transformation ensures that the original and resulting CNF formulas have the same satisfiability, though logical equivalence is not guaranteed.

3.2.4.3 Query Complexity

Implicant Query (IM) Checking whether a term implies a CNF formula can be done in $(\mathcal{O}(|\text{term}|+|\text{CNF}|))$ time.

Satisfiability (Consistency, CO) Determining satisfiability of a general CNF formula is NP-complete; the best algorithms run in time $(\mathcal{O}(2^n))$ in the worst case, where (n) is the number of variables, though modern SAT solvers perform much better in practice.

Validity (VA) Checking whether a CNF is a tautology is co-NP-complete. No polynomial-time algorithm is known unless $P = NP$.

Prime Implicant Generation Extracting a single prime implicant from a known model (an assignment satisfying the CNF) can be performed in polynomial time. This process involves iteratively attempting to remove each literal from the model and checking if the CNF remains satisfied. Each removal can be checked in time linear in the size of the CNF, and all removals together give a total complexity of $(\mathcal{O}(|\text{CNF}| \cdot |M|))$, where M is the model. With efficient algorithms, this task can be done in linear time.

1. Finding any model or implicant: Equivalent to SAT, and thus NP-complete.
2. Enumerating all prime implicants: Exponential time in the worst case; the number of prime implicants can be exponential in formula size.
3. Recognizing essential prime implicants: NP-complete.

Model Counting (CT) Counting the number of satisfying assignments is $\#P$-complete. For a CNF with primal treewidth (w) and (n) clauses, model counting via dynamic programming on a tree decomposition can be done in $(\mathcal{O}(n2^w))$ time when the decomposition is given. For a CNF with incidence treewidth (w) and (N) tree decomposition nodes, counting can be done in $(\mathcal{O}(2^w(kd+\delta)N))$, where (d) is maximum variable degree, (δ) is multiplication time.

Model Enumeration (ME) Enumerating all models is not feasible in polynomial time in general, due to potentially exponential output size.

Clausal Entailment (CE), Equivalence (EQ), Sentential Entailment (SE) All are co-NP-complete (or worse) in general for CNF.

3.2.4.4 CNF as a Source for Knowledge Compilation

CNF serves as the standard input for compilation into tractable reasoning languages:

CNF to DNNF With given decomposition tree of width (w) and (n) clauses, can be compiled in $(\mathcal{O}(nw2^w))$ time and space. Complexity is singly exponential in treewidth and linear in formula size when width is bounded.

CNF to d-DNNF Using tools like c2d and DSHARP, for CNF with incidence treewidth (k) and size (n), can be compiled into DNNF of size $(\mathcal{O}(2^k n))$.

CNF to SDD For a CNF with (n) variables and vtree of width (w), bottom-up compilation yields SDD of size $(\mathcal{O}(n2^w))$, and can be performed in $(\mathcal{O}(nw))$ time if the vtree is fixed.

CNF to OBDD Top-down approaches with caching result in size and time $(\mathcal{O}(2^{\text{pathwidth}}))$ for OBDD, where pathwidth is a width measure related to treewidth.

3.2.4.5 Compilers

1. DNNF/d-DNNF: c2d, dsharp (output size $(\mathcal{O}(2^{\text{width}} n))$)
2. RoBDD: CuDD, BuDDy (output size $(\mathcal{O}(2^{\text{pathwidth}}))$)
3. SDD: SDD package (output size $(\mathcal{O}(n2^w))$)

3.2.5 Decomposable Negation Normal Form (DNNF)

Decomposable Negation Normal Form (DNNF) is a central target language in knowledge compilation, representing a significant refinement of Negation Normal Form (NNF). In NNF, every propositional sentence is represented as a directed acyclic graph (DAG): leaves are labeled by literals (variables or their negations) or by the constants `true`/`false`, while internal nodes are labeled by conjunction ($\wedge$) or disjunction ($\vee$) operations. NNF allows unrestricted subformula structure as long as negations only appear at the literal level, but on its own does not provide tractability for key reasoning tasks unless P $=$ NP.

3.2.5.1 Definition of DNNF and Related Forms

A formula is in **DNNF** if it is in NNF and satisfies the *decomposability* property: at every conjunction ($\wedge$-node), the conjuncts mention disjoint sets of variables. Formally, if a conjunction node has children $\varphi_1, \ldots, \varphi_m$, then $\mathrm{Var}(\varphi_i) \cap \mathrm{Var}(\varphi_j) = \emptyset$ for all $i \neq j$. This property enables efficient independent evaluation of circuit branches.

- **Deterministic DNNF (d-DNNF):** Adds the *determinism* property. At every disjunction ($\vee$-node), disjuncts are mutually contradictory (i.e., the conjunction of any two child subcircuits is inconsistent). Determinism enables additional tractable queries, such as model counting.
- **Structured DNNF** / **Structured d-DNNF:** These subclasses further restrict DNNF/d-DNNF by requiring that decompositions reflect a hierarchical structure (typically enforced by a vtree specifying the

variable partitioning at each conjunction or disjunction). This *structured decomposability* allows additional tractability (notably, certain transformations), and forms the basis for languages like Sentential Decision Diagrams (SDDs), which are strict subsets of structured d-DNNF.

3.2.5.2 Construction and Compilation

Compiling an arbitrary propositional formula, often given in CNF or DNF, into DNNF or d-DNNF is a central algorithmic task. For CNF inputs, one algorithm uses a decomposition tree (related to variable treewidth). Given n clauses and treewidth w, d-DNNF can be compiled in time and space $O(nw2^w)$; thus, for bounded treewidth, linear-sized d-DNNF representations can be obtained in linear time. Practical compilers include `C2D`, `DSHARP`, and `d4`. `DSHARP`, leveraging #SAT technology, frequently exceeds the speed of `C2D` while producing similarly sized d-DNNF. OBDD representations for propositional theories can be converted into equivalent DNNF in linear time relative to OBDD size.

3.2.5.3 Succinctness

Succinctness compares the minimum size of representations of the same function across languages. The succinctness ordering (strict, unless the polynomial hierarchy collapses) is:

$$\text{DNNF} \ < \ \text{d-DNNF} \ < \ \text{FBDD} \ < \ \text{OBDD} \ < \ \text{CNF}/\text{DNF}$$

That is:

- DNNF (and its subclasses) are strictly more succinct than OBDDs.
- SDDs sit as a strict subset of structured d-DNNF.
- DNNF is generally more succinct than FBDDs, which are more succinct than OBDDs.
- CNF and DNF generally remain exponentially sized compared to DNNF for many functions.

Smooth deterministic DNNF (sd-DNNF) is as succinct as d-DNNF.

3.2.5.4 Supported Queries and Complexities

A principal utility of DNNF is to enable several queries in time polynomial to circuit size. The main queries and their tractability status for DNNF and derivatives:

- **DNNF:**
  - **Consistency (CO):** Polynomial time
  - **Clausal Entailment (CE):** Polynomial time
  - **Model Enumeration (ME):** Polynomial time; for sd-DNNF, output-linear time
- **d-DNNF/structured d-DNNF:**
  - All above, plus:
  - **Validity (VA):** Polynomial time

- **Model Counting (CT):** Polynomial time (linear for sd-DNNF)
- **Model-based Diagnosis:** Minimum-cardinality diagnosis, etc., in polynomial time
- **Implicant Checking (IM)**, **Model Minimization:** Polynomial time
- **Equivalence Testing (EQ):** Not known to be polynomial time for d-DNNF, unlike OBDD

3.2.5.5 Empirical Benchmarks

Empirical results show a clear advantage for DNNF and d-DNNF in both size and compilation efficiency versus OBDD on relevant AI tasks:

- Diagnoses compiled into DNNF are often orders of magnitude smaller than those into OBDD.
- Compilation time is generally faster with DNNF compilers (`DSHARP` is up to 27 times faster on average than `C2D`, and both outperform OBDD compilation for relevant model-based diagnosis inputs).
- Diagnostic and enumeration queries are more efficient on DNNF than OBDD, due to reduced circuit size and higher tractability.

DNNF and its subclasses, notably d-DNNF and sd-DNNF, provide a foundational set of tractable languages in knowledge compilation, balancing high succinctness with strong support for key inference queries. Their compilation is practical for many instances, and empirical results show clear advantages over OBDDs. SDDs and structured d-DNNFs offer further tractable transformations at some cost in succinctness.

3.2.6 Decision Diagrams

Decision diagrams provide a powerful, directed-graph-based representation of logical or Boolean functions. Their roots can be traced back to branching program ideas explored by Lee and Akers in the 1950s–1970s, but major refinement and widespread adoption occurred after Bryant's seminal work on *Ordered Binary Decision Diagrams (OBDDs)* in 1986. In reliability analysis, formal verification, and combinational circuit design, decision diagrams frequently offer more computationally tractable methods than naïve enumeration of all input patterns.

This section introduces the basic concepts of *Binary Decision Diagrams (BDDs)* and *Zero-Suppressed Decision Diagrams (ZDDs)*, along with the special class of *Ordered* and *Reduced* BDDs that guarantee a canonical (unique) form under fixed conditions. We emphasize:

- The structure and interpretation of BDDs as directed acyclic graphs (DAGs).
- The notion of an *ordered* BDD, imposing a strict arrangement on variable testing.
- Techniques for *reducing* BDDs into smaller yet equivalent graphs by merging or removing redundant parts.
- The main principles of Zero-Suppressed Decision Diagrams, designed for efficiently encoding sparse sets or combinatorial families.

Earlier, we saw that *event trees* and *fault trees* can be merged into a single directed acyclic graph to

represent complex system dependencies. BDDs and ZDDs, by contrast, focus more narrowly on Boolean functions, providing specialized node-splitting and merging operations to systematically capture logical behavior. Despite differing motivations, both families of DAG-based representations benefit from the avoidance of cycles and the ability to encode large models in a structured form.

3.2.6.1 Binary Decision Diagrams (BDD)

Let $f : \{0,1\}^n \to \{0,1\}$ be an n-variable Boolean function. A Binary Decision Diagram (BDD) is a directed acyclic graph whose internal nodes represent decisions on a single Boolean variable, and whose terminal (sink) nodes represent constant outputs (0 or 1).

Definition 2 (Binary Decision Diagram)**.** *A Binary Decision Diagram for f is a tuple $B = \big(N, n_0, V, E, T\big)$ with the following components:*

1. *N is a finite set of nodes, partitioned into* internal *nodes and* terminal *nodes.*
2. *$n_0 \in N$ is the* root node*, where evaluations begin.*
3. *$V = x_1, x_2, \ldots, x_n$ is the set of Boolean variables associated with the internal nodes.*
4. *$E \subseteq N \times 0, 1 \times N$ is the edge set. Each internal node u has two labeled edges, $(u, 0, v_0)$ and $(u, 1, v_1)$, indicating the next node in the diagram if $x_i = 0$ or $x_i = 1$ at node u.*
5. *T is a mapping that assigns the value 0 or 1 to each terminal node of B.*

For any input $a = (a_1, a_2, \ldots, a_n) \in \{0,1\}^n$ one identifies a unique path from n_0 to a terminal node by at each internal node following the edge labeled by the tested variable's value in a. The value of $f(a)$ is given by the terminal node reached, as encoded by T.

Interpretation. Each internal node corresponds to a variable test: if the variable is 0 (i.e. false), follow the *0-edge*, and if it is 1 (true), follow the *1-edge*. Eventually, one reaches a sink node labeled false $= 0$ or true $= 1$.

Example. For the three-variable function $f(a, b, c) = a \wedge \big(b \vee c\big)$, Figure 3.2 shows a small BDD. Each circular node tests one variable a, b, or c; the dashed and solid edges denote the 0- and 1-branches, respectively. Terminal nodes (squares) contain a 0 or 1 label.

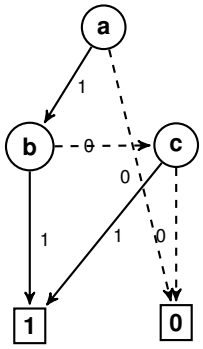


Figure 3.2: A Binary Decision Diagram (BDD) for $f(a, b, c) = a \wedge \big(b \vee c\big)$.

In practice, BDDs may experience large variations in size depending on how variables are tested as one traverses the graph. The *ordered* and *reduced* variants of BDDs are especially important, as they yield canonical forms for fixed variable orderings.

Definition 3 (Ordered Binary Decision Diagram (OBDD))**.** *An Ordered Binary Decision Diagram (OBDD) imposes a strict ordering π on the variables $x_1, \ldots, x_n$. For every path from the root to a terminal node, if the path encounters variables x_i and x_j, then x_i is tested before x_j whenever $i < j$ with respect to π. Equivalently, no path may test a higher-indexed variable and later test a lower-indexed one.*

OBDDs are also known as *read-once branching programs with an ordering restriction.* Bryant showed that under a particular variable order, the representation is often more compact than arbitrary BDDs and that many operations (e.g., equivalence checking, conjunction, disjunction) can be carried out efficiently.

Definition 4 ([Reduced Ordered Binary Decision Diagram (RoBDD))**.** *An OBDD is said to be* reduced *if it contains no isomorphic subgraphs and no node whose 0- and 1-branches lead to the exact same child. Equivalently, one applies two reduction rules:*

1. ***Elimination Rule:** If, for a given node v, the 0-edge and 1-edge both point to the same successor, remove v and connect its incoming edges directly to that successor.*
2. ***Merging Rule:** If two distinct nodes u and v test the same variable and have identical 0- and 1-successors, merge them into a single node.*

A Reduced Ordered Binary Decision Diagram (RoBDD) *respects a global variable order π and has been minimized via these rules.*

Canonical Representation. One of the principal advantages of ROBDDs is that, for a fixed variable ordering, every Boolean function has a unique representation. Consequently, checking whether two functions are identical reduces to testing whether their ROBDDs coincide as node- and edge-labeled graphs.

Theorem 3 (Canonical Form of RoBDDs)**.** *Let π be a fixed ordering on the variables $x_1, \ldots, x_n$. Then for any Boolean function f of n variables, its reduced OBDD with respect to π is unique.*

A direct consequence is that striving for reduced ordered forms both shrinks redundant structure and supports robust equivalence checks.

3.2.6.2 Zero-Suppressed Decision Diagrams (ZDD)

For certain applications, notably combinatorial itemset enumeration and other *sparse* set representations, Zero-Suppressed Binary Decision Diagram (ZDD) can be more compact than standard BDDs. Although ZDDs adhere to similar principles of node-based variable testing, they selectively *omit* many zero-branches that do not yield new information.

Key Distinctions. While ZDDs also enforce an ordering and can be reduced via isomorphism checks, the core difference lies in the zero-suppression mechanism:

- If following a 0-edge provides no meaningful distinction in the final outcome, that 0-edge and its corresponding node are pruned.

- The 1-branches are retained but merged where possible, much as in RoBDDs.

By removing portions of the diagram where "nothing interesting" (i.e. no new sets or subsets) occurs, the diagram remains compact.

Illustrative Example Revisiting $f(a, b, c) = a \wedge \big(b \vee c\big)$ from above, Figure 3.3 sketches a plausible ZDD. Note here:

- Node (a) splits into a 0-edge that immediately goes to a node (or directly to a 0-terminal) that is pruned if it carries no unique set representation.
- The 1-edge leads to further variable tests (b or c), but many 0-branches are again suppressed if they do not alter the final outcome distinct from an already-represented path.

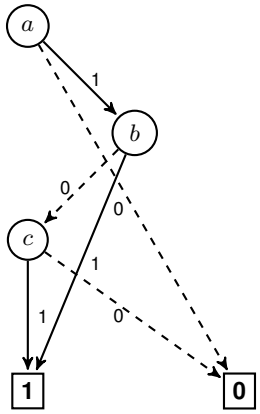


Figure 3.3: A Zero-Suppressed Binary Decision Diagram (ZDD) for $f(a, b, c) = a \wedge \big(b \vee c\big)$. Many zero-branches are pruned.

In general, ZDDs apply much the same merging rules as RoBDDs and can yield similarly unique structures for a given variable order. They tend to excel in representing large but sparsely populated families of subsets (e.g., all minimal cut sets in a reliability system) because superfluous 0-edges are systematically suppressed.

3.2.6.3 Probabilistic Sentential Decision Diagrams (PSDD)

A Probabilistic Sentential Decision Diagram (PSDD) is a tractable representation for a probability distribution over a set of propositional variables subject to logical constraints. In essence, a PSDD is a parameterized Sentential Decision Diagram (SDD) in which each node is assigned a well-defined local distribution. By construction, the PSDD's global distribution respects a given *base theory* (i.e., a propositional formula representing constraints), assigns zero probability to every assignment violating that theory, and factors the remaining assignments according to a hierarchical decomposition.

Definition 5 (Normalized SDD)**.** *Given a* vtree*, v, over variables $X_1, ..., X_n$, an normalized SDD over T is defined recursively as follows:*

1. *A* terminal *node is a literal X_i or $\neg X_i$*

2. *At an* internal vtree node *with left subtree V_l and right subtree V_r, a normalized SDD is a finite disjunction:*

$$\bigvee_{i=1}^{k} (P_i \wedge S_i)$$

where:

   (a) *For every i, P_i is a normalized SDD over the variables in V_l, and S_i is a normalized SDD over the variables in V_l.*

   (b) *The set $\{P_1, \ldots, P_k\}$ forms a partition of the space over V_l; that is, $\forall i \neq j : P_i \wedge P_j = \bot$ (mutually exclusive) and $\bigvee_{i=1}^{k} P_i = \top$ (exhaustive).*

   (c) *Each S_i is distinct (no two are equivalent).*

   (d) *Each prime-sub pair (P_i, S_i) is compressed: distinct primes never associate to equivalent subs and vice versa (no redundancy).*

Once the normalized SDD is fixed, one may introduce continuous parameters to obtain a PSDD. These parameters, in effect, turn each decision node into a *local mixture* of its prime components, while terminal nodes over variables become Bernoulli distributions.

Definition 6 (PSDD Syntax)**.** *Let n be an SDD node normalized for a vtree node v.*

- *If n is a terminal node:*

  1. *If it encodes a literal (e.g. X, $\neg X$) or the constant $\bot$, then its probability is fixed implicitly (e.g. $\neg X$ yields $\Pr(\neg X) = 1, \Pr(X) = 0$ for that node alone).*

  2. *If it is the constant $\top$ and v corresponds to some variable X, then we assign a parameter $\theta \in (0, 1)$ indicating $\Pr(X)$ at this node.*

- *If n is a decision node with k elements $[(p_1, s_1), \ldots, (p_k, s_k)]$, we assign nonnegative parameters $\theta_1, \ldots, \theta_k$ such that $\sum_{i=1}^{k} \theta_i = 1$. Furthermore, if $s_i = \bot$, then θ_i must be zero (no probability is allotted to a sub whose base is unsatisfiable).*

The resulting parameterized structure is called a PSDD*.*

Each node n in a PSDD induces a local distribution $\Pr_n(\cdot)$ on the variables of the vtree node n is normalized for. At a decision node, the probability of a complete assignment is given by multiplying:

1. The probability that we "choose" a particular prime p_i, labeled by θ_i.
2. The probability contributed by prime node p_i on its variables.
3. The probability contributed by sub node s_i on its (disjoint) variables.

Summing across all primes yields $\Pr_n$ at that node. By construction, the *root* node's distribution $\Pr_r$ then covers all variables and zeroes out any assignments that do not satisfy the base theory.

Theorem 4 (Base Property [41])**.** *If a PSDD node n is normalized for vtree node v, then $\Pr_n(x) = 0$ whenever x does not satisfy the SDD sub-formula $[n]$. At the root node r, $\Pr_r(x) > 0$ only if x satisfies the*

entire theory.

Parameter Semantics. A key property of PSDDs is that each parameter θ_i can be interpreted *locally* as a conditional probability given the *context* of the decision node. Formally, if node n has context γ_n (i.e., the partial assignment implied by traversing the SDD from the root to n), then:

$$\theta_i = \Pr\big([p_i] \mid [\gamma_n]\big)$$

where $[, p_i]$ is the logical content of prime p_i. This ensures that local parameters align with global $\Pr(\cdot)$ in a transparent, compositional way.

Context-Specific Independence. Due to the vtree-based factorization, PSDDs capture rich *context-specific independences* [34]. At high level, once we know the node's context (which is a partial assignment or formula), certain subsets of variables become conditionally independent of the rest. These independence statements can be read directly from the PSDD structure, generalizing common conditional-independence ideas in Bayesian or Markov networks.

Inference and Tractability. A central advantage of PSDDs is that computing $(\Pr_r(e)$ for any evidence e can be done in time linear in the size of the PSDD. This incremental algorithm proceeds bottom-up through each node, locally aggregating evidence contributions and summing accordingly. Moreover, once node-level evidence statistics are available, one can also efficiently compute single-variable or pairwise marginals using a second top-down pass.

Parameter Learning under Complete Data. Another appealing property is that *maximum-likelihood* parameters can be determined in closed form when every training example is a complete assignment of all variables. Specifically, if a PSDD node n with context γ_n has elements (p_i, s_i), one sets

$$\theta_i = \frac{\text{number of data points satisfying both } \gamma_n \text{ and } p_i}{\text{number of data points satisfying } \gamma_n}.$$

Parallel rules apply for terminal nodes representing $\top$. Because sub-contexts $\gamma_n \wedge p_i$ are pairwise disjoint, it suffices to tabulate data counts for each feasible sub-context. The outcome is a simple frequency-based update analogous to parameter estimation in Bayesian networks, yet here it respects the underlying SDD constraints exactly.

For any propositional distribution (and chosen vtree), there exists a corresponding PSDD whose root distribution matches it exactly. Furthermore, if the PSDD is kept *compressed* (meaning no redundant substructures), this representation is *unique* up to isomorphic details [41]. Thus, PSDDs can serve as canonical forms for distributions under logical constraints.

In contrast to classical graphical models, PSDDs operate at the confluence of tractable *Boolean* structure (via SDDs) and probability theory. They explicitly encode zero-probability assignments (via the SDD base) while ensuring all positive assignments factor through the decision nodes. Their parameter semantics aligns each local weight with a well-defined global conditional probability. In addition, closed-form parameter learning is possible in the complete-data setting. Hence, PSDDs provide a principled, canonical choice for modeling distributions when the domain is governed by complex logical constraints.

3.3 Queries

The primary choice that dictates the selection of target languages in PRA is availability of tractable/executable queries for a given language. The two most important query types that are of interest to PRA are Model Counting (CT) and Model Enumeration (ME).

3.3.1 Model Counting

Model Counting, a.k.a #SAT-problem, is a classic problem in Computability Theory and VLSI (Very Large Scale Integration) analysis. It strive to answer the following question about a given boolean expression (theory): “How many valid true/false variable assignments (models) satisfy the boolean expression?” This problem is central to PRA, as it the compuation algorithm behind Weighted Model Counting approach used for computing probabilities of top events in Fault Trees and End States in Even Sequence Diagrams.

Importance of this query has largely determined the relative scarcity of underlying implementations of PRA software. Tractability of this query in OBDD family of target languages is primarily reason for popularity of this data structure among PRA software vendors. However, despite the its perceived NP-complexity, CNF-based #SAT-solvers can also be used, due to their efficient implementations and long-running research. Finally, with new research showing still flowing, SDDs, identify themselves as a promising alternative due to their improved succinctness.

3.3.2 Model Enumeration

Model enumeration is arguably even more important query for PRA, as it responsible for implementations exhaustive Minimal Cutset search. This is one of the most intractable algorithms allowing for only polynomial-time delay between successive solution, as the primary determining complexity factor is exponential “blow-up” of the number of cutsets in the first place. Few implementations, offer truly efficient and tractable solutions, with CNFs and ROBDDs yet again taking the spotlight.

Part III

Proposed High-Throughput Compute Solution

10 BUILDING A DATA-PARALLEL MONTE CARLO BOOLEAN EVALUATIONS SOLVER

To handle massively parallel Monte Carlo evaluations of large-scale Boolean functions, we have developed a preliminary layered architecture that organizes computation in a topological graph. At the lowest level, each Boolean variable/basic event (e.g., a component failure mode) is associated with a random number generator to sample its truth assignment. We bit-pack these outcomes, storing multiple Monte Carlo samples in each machine word to maximize computational throughput and reduce memory footprint. Subsequent layers consist of logically higher gates or composite structures that receive the bit-packed results from previous layers and combine them in parallel using coalesced kernels. By traversing the computation graph topologically, dependencies between gates and events are naturally enforced, so kernels for each layer can run concurrently once all prerequisite layers finish, resulting in high kernel occupancy and predictable throughput.

In practice, each layer is dispatched to an accelerator node using a data-parallel model implement using SYCL. The random number generation pipelines are counter-based, ensuring reproducibility and thread-safety even across millions or billions of samples. Gates that go beyond simple AND/OR logic–such as VOT operators–are handled by specialized routines that can exploit native popcount instructions for efficient threshold evaluations. As we progress upwards through the layered topology, each gate or sub-function writes out its bit-packed output, effectively acting as an input stream to the next layer.

Throughout the simulation, online tallying kernels aggregate how often each node or gate evaluates to True. These tallies can then be turned into estimates of probabilities and sensitivity metrics on the fly in which the computational efficiency is certainly known. This approach also makes adaptive sampling feasible: if specific gates appear to dominate variance or are tied to particularly rare events, additional sampling can be allocated to their layer to refine estimates.

10.1 Layered Topological Organization

Recall that a PDAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ contains no cycles, so there is at least one valid *topological ordering* of its nodes. A topological ordering assigns each node a numerical *layer index* such that all edges point from a lower-numbered layer to a higher-numbered layer. If a node v consumes the outputs of nodes $\{u_1, \ldots, u_k\}$, then we require

$$\text{layer}(u_i) \; < \; \text{layer}(v) \quad \text{for each } i \in \{1, \ldots, k\}.$$

In other words, node v can appear only after all of its inputs in a linear or layered listing.

The essential steps to build and traverse these layers are:

1. *Compute Depths via Recursive Analysis:* Each node's depth is found by inspecting its children (or inputs). If a node is a leaf (e.g., a `Variable` or `Constant` that does not depend on any other node), its

depth is 0. Otherwise, its depth is one larger than the maximum depth among its children.

2. *Group Nodes by Layer:* Once each node's depth is computed, nodes of equal depth form a single *layer*. Thus, all nodes with depth 0 are in the first layer, those with depth 1 in the second layer, and so on.

3. *Sort Nodes within Each Layer:* Within each layer, enforce an additional consistent ordering: (i) variables appear before gates, (ii) gates of different types can be grouped to facilitate specialized processing. This step is not strictly required for correctness, but it can streamline subsequent stages such as kernel generation or partial evaluations.

4. *Traverse Layer by Layer:* A final pass iterates over each layer in ascending order. Because all inputs of any node in layer d lie in layers $< d$, the evaluation (or "kernel build") for layer d can proceed after the entire set of layers $0, \ldots, d-1$ is processed.

This structure ensures a sound evaluation of the PDAG: no gate or variable is computed until after all of its inputs are finalized.

10.1.1 Depth Computation and Node Collection

1. **Clear Previous State.** Any existing "visit" markers or stored depths in the PDAG-based data structures are reset to default values (e.g., zero or -1).

2. **Depth Assignment by Recursion.** A `compute_depth` subroutine inspects each node:

   (a) If the node is a `Variable` or `Constant`, it is a leaf in the PDAG, so depth $= 0$.

   (b) If the node is a `Gate` with multiple inputs, the procedure first recursively computes the depths of its inputs. It then sets its own depth as

$$\text{depth}(\texttt{gate}) \;=\; 1 \;+\; \max_{\ell \in \text{inputs of gate}} \Big[\text{depth}(\ell)\Big].$$

3. **Order Assignment.** Each node stores the newly computed depth in an internal field. This numeric value anchors the node to a layer. A consistent pass over the entire graph ensures correctness for all nodes.

After depths are assigned, gather all nodes, walking the PDAG from its root, recording each discovered node and adding it to a global list.

10.1.2 Layer Grouping and Local Sorting

Begin by creating:

- A global list of all nodes, each with a valid depth,
- A mapping from node indices to node pointers,

Then, sort the global list by ascending depth. Let order(n) be the depth of node n. Then

$$\text{order}(n_1) \;\leq\; \text{order}(n_2) \;\leq\; \ldots \;\leq\; \text{order}(n_{|\mathcal{V}|}).$$

Finally, partition this list into contiguous *layers*: if the deepest node has a depth $\delta_{\max}$, then create sub-lists:

$$\{\text{nodes s.t. depth} = 0\}, \quad \{\text{nodes s.t. depth} = 1\}, \quad \ldots, \quad \{\text{nodes s.t. depth} = \delta_{\max}\}.$$

Within each layer, sort nodes to ensure that `Variable` nodes precede `Gate` nodes, and `Gate` nodes may be further sorted by `Connective` type (e.g., `AND`, `OR`, `VOT`, etc.).

10.1.3 Layer-by-Layer Kernel Construction

Apply the layer decomposition to drive *kernel building* and *evaluation*:

1. **Iterate over each layer in ascending depth**. Because every node's dependencies lie in a strictly lower layer, one is guaranteed that those dependencies have already been assigned memory buffers, partial results, or other necessary resources.
2. **Partition the layer nodes into subsets by node type**. Concretely, `Variable` nodes are batched together for *basic-event sampling* kernels, while `Gate` nodes are transferred into *gate-evaluation* kernels.
3. **Generate device kernels**. For `Variable` nodes, create Monte Carlo sampling kernels. For `Gate` nodes, it constructs logical or bitwise operations that merge or transform the sampled states of the inputs.

Once kernels for a given layer finish, move on to the next layer. Because of the topological guarantee, no node in layer d references memory or intermediate states from layer $d+1$ or later, preventing cyclical references and ensuring correctness.

10.2 Bitpacked Random Number Generator

Monte Carlo simulations, probability evaluations, and other sampling-based procedures benefit greatly from efficient, high-quality Random Number Generator (RNG)s. A large class of modern RNGs are known as *counter-based Pseudo Random Number Generator (PRNG)s*, because they use integer counters (e.g., 32-bit or 64-bit) along with a stateless transformation to produce random outputs. The *Philox* family of counter-based PRNGs is a well-known example, featuring fast generation, high period, and good statistical properties. In this section, we discuss the general principles of counter-based PRNGs, explain how Philox fits into this paradigm, analyze its complexity, and present a concise pseudocode version of the Philox 4×32-10 variant. Subsequently, we detail the bitpacking scheme used to reduce memory consumption when storing large numbers of Bernoulli samples.

A counter-based PRNG maps a user-supplied *counter* (plus, optionally, a *key*) to a fixed-size block of random bits via a deterministic function. Formally, if

$$\mathbf{x} \;=\; (x_1, x_2, \ldots, x_k)$$

is a vector of one or more 32-bit or 64-bit counters, and

$$\mathbf{k} \;=\; (k_1, k_2, \ldots, k_m)$$

is a key vector, then a counter-based PRNG defines a transformation

$$\mathcal{F}(\mathbf{x}, \mathbf{k}) \;=\; (\rho_1, \rho_2, \ldots, \rho_r),$$

where each ρ_j is typically a 32-bit or 64-bit output. Different increments of the counter $\mathbf{x}$ produce different pseudo-random outputs ρ_j. The process is stateless in the sense that advancing the RNG amounts to incrementing the counter (e.g., $\mathbf{x} \mapsto \mathbf{x} + 1$).

Compared to older recurrence-based RNGs such as linear congruential generators or the Mersenne Twister, counter-based methods offer more straightforward parallelization, reproducibility across multiple streams, and strong structural simplicity: no internal state must be updated or maintained. This is particularly valuable in distributed Monte Carlo simulations or GPU-based sampling, where each thread or work-item can be assigned a different counter. Philox constructs its pseudo-random outputs by applying a small set of mixed arithmetic (multiplication/bitwise) rounds to an input *counter* plus *key*. In particular, $\mathrm{Philox}\,4 \times 32$-10 (often shortened to "Philox-4x32-10) works on four 32-bit integers at a time:

$$\mathbf{S} = (S_0, S_1, S_2, S_3), \qquad \mathbf{K} = (K_0, K_1).$$

The four elements $\{S_0, S_1, S_2, S_3\}$ collectively represent the counter, e.g., (x_0, x_1, x_2, x_3). The two key elements (K_0, K_1) are used to tweak the generator's sequence. A single invocation of Philox-4x32-10 transforms $\mathbf{S}$ into four new 32-bit outputs after ten rounds of mixing. At each round, the algorithm:

1. Multiplies two of the state words by fixed "magic constants to create partial products.
2. Takes the high and low 32-bit portions of those 64-bit products.
3. Incorporates the round key to shuffle the words.
4. Bumps the key by adding constant increments $(\mathrm{W32A} = 0x9E3779B9$ and $\mathrm{W32B} = 0xBB67AE85)$.

After ten rounds, the final (S_0, S_1, S_2, S_3) is returned as the pseudo-random block. A new call to Philox increases the counter $\mathbf{S}$ by one (e.g., $S_3 \mapsto S_3 + 1$) and re-enters the same function. The Philox-4x32-10 algorithm is designed so that each blocking call requires a *constant number* of operations, independent of the size of any prior "state. Specifically, each round involves:

$$\mathcal{O}(1) \;\text{ arithmetic operations},$$

and there are $\mathrm{R} = 10$ rounds. Thus, each Philox invocation is asymptotically constant time $\mathcal{O}(\mathrm{R}) = \mathcal{O}(1)$. The total cost to generate 128 bits (4 words $\times$ 32 bits) is therefore constant time per call.

10.2.1 The 10-round Philox-4x32

Our implementation follows the standard 10-round approach for generating one block of four 32-bit random words, also called Philox-4x32-10. Let $M_{\mathrm{A}} = 0xD2511F53$, $M_{\mathrm{B}} = 0xCD9E8D57$ be the multipliers, and let

(K_0, K_1) be the key which is updated each round by $\mathrm{W32A} = 0x9E3779B9$ and $\mathrm{W32B} = 0xBB67AE85$. The function Hi$(\cdot)$ returns the high 32 bits of a 64-bit product, and Lo$(\cdot)$ returns the low 32 bits. Because each call produces four 32-bit pseudo-random words, Philox-4x32-10 is particularly convenient for batched sampling. If only a single 32-bit word is needed, one can still call the function and discard the excess words; however, many applications consume all four outputs (e.g., to produce four floating-point variates).

Algorithm 4 Philox-4x32-10

Require: Four 32-bit counters (S_0, S_1, S_2, S_3), key (K_0, K_1)
Ensure: Transformed counters (S_0, S_1, S_2, S_3)

procedure PHILOX_ROUND((S_0, S_1, S_2, S_3), (K_0, K_1))
  $P_0 \leftarrow M_{\mathsf{A}} \times S_0$ ▷ 64-bit product
  $P_1 \leftarrow M_{\mathsf{B}} \times S_2$ ▷ 64-bit product
  $T_0 \leftarrow \mathrm{Hi}(P_1) \oplus S_1 \oplus K_0$
  $T_1 \leftarrow \mathrm{Lo}(P_1)$
  $T_2 \leftarrow \mathrm{Hi}(P_0) \oplus S_3 \oplus K_1$
  $T_3 \leftarrow \mathrm{Lo}(P_0)$
  $K_0 \leftarrow K_0 + \mathrm{W32A}$
  $K_1 \leftarrow K_1 + \mathrm{W32B}$
  return $\big((T_0, T_1, T_2, T_3),\ (K_0, K_1)\big)$
end procedure

procedure PHILOX4X32_10((S_0, S_1, S_2, S_3), (K_0, K_1))
  for $i \leftarrow 1$ **to** 10 **do**
    $\big(S_0, S_1, S_2, S_3\big),\ (K_0, K_1) \leftarrow$ PHILOX_ROUND((S_0, S_1, S_2, S_3), (K_0, K_1))
  end for
  return (S_0, S_1, S_2, S_3)
end procedure

10.2.2 Bitpacking for Probability Sampling

It takes exactly one bit to represent the outcome of a trial. If these If outcomes are stored naively, each one occupies a full 8-bit byte. Hence, only $\frac{1}{8}$ of the allocated space is used for actual data. By instead packing up to w indicators into a w-bit machine word, the memory usage can be reduced by a factor of up to 8 (in the simplest scenario of 8-bit groupings). In more general terms:

$$\text{Memory usage } M_{\text{naive}} \;=\; N \times 8 \text{ bits}, \qquad \text{Memory usage } M_{\text{pack}} \;=\; \left\lceil \frac{N}{w} \right\rceil \times w \text{ bits}.$$

In our implementation, each call to Philox-4x32-10 yields 128 bits of randomness. We use those bits to draw exactly 128 Bernoulli outcomes at once, then combine them into a $\mathrm{bitpack}$ of two 64-bit integers. For instance, if we choose a batch size of 4-bits to represent four Bernoulli samples in a single chunk, we can:

1. Generate a block $\{r_0, r_1, r_2, r_3\}$ of four 32-bit random integers from Philox.
2. Convert each r_i into a uniform $[0, 1)$ floating-point value by dividing by 2^{32}.
3. Compare each to the target probability p.

4. Form a 4-bit integer, each bit set to 1 if the corresponding comparison succeeded, or 0 otherwise.

Repeating these steps for multiple rounds of 4 bits each can fill a 16-bit or 32-bit $\mathrm{bitpack}$ variable with many Bernoulli indicators. Then it can be stored into an array at a single index, reducing memory overhead by constant factor of N.

Algorithm 5 Bit-packing of four Bernoulli samples into a 4-bit block

Require: Probability $p \in [0,1]$, random 32-bit words (r_0, r_1, r_2, r_3)
Ensure: 4-bit integer bits containing the four Bernoulli draws

```
procedure FOURBITPACK(p, (r_0, r_1, r_2, r_3))
    bits ← 0
    for i ← 0 to 3 do
        u_i ← r_i / 2^32                                   ▷ u_i ∈ [0, 1)
        if u_i < p then
            b_i ← 1
        else
            b_i ← 0
        end if
        bits ← bits | (b_i ≪ i)                            ▷ set bit i to b_i
    end for
    return bits
end procedure
```

In this procedure, $\|$ denotes a bitwise OR, and $\ll$ denotes a left shift. One then repeats the above call to accumulate multiple 4-bit blocks (e.g., for a total of 16 bits, one calls FourBitPack four times and merges the results with the appropriate shifts).

10.3 Tallying Layer Outputs

At every Monte-Carlo iteration the simulator produces, for each logic node $v \in \mathcal{V}$, a bit-packed buffer encoding

$$\mathbf{Y}_v^{(t)} \;=\; \big(y_{v,1}^{(t)}, y_{v,2}^{(t)}, \ldots, y_{v,N}^{(t)}\big) \in \{0,1\}^N, \quad t = 1, \ldots, T,$$

where $N = B \times P \times \omega$ is the number of Bernoulli trials per Monte-Carlo *iteration*:

- B - number of *batches*,
- P - bit-packs per batch,
- $\omega = 8 \cdot \mathrm{sizeof}(\text{bitpack\_t})$ - bits per pack.

Because the buffers are overwritten at the next iteration, a separate *tally layer* accumulates summary statistics that persist for the entire simulation. The present section formalises that process and outlines an implementation-agnostic, data-parallel algorithm that realises it on modern accelerators.

10.3.1 Statistical Objectives

For every node v we wish to estimate, after T Monte-Carlo iterations,

$$\widehat{p}_v \;=\; \frac{1}{T\,N}\sum_{t=1}^{T}\sum_{j=1}^{N} y_{v,j}^{(t)} \;=\; \frac{s_v}{T\,N}, \qquad s_v \;=\; \text{total \# of one-bits observed for node } v.$$

Under the usual independence assumptions, the sampling distribution of $\widehat{p}_v$ is asymptotically

$$\mathcal{N}\!\left(p_v,\; \tfrac{p_v(1-p_v)}{T\,N}\right)$$

Hence

$$\widehat{\sigma}_v \;=\; \sqrt{\frac{\widehat{p}_v\,(1-\widehat{p}_v)}{T\,N}}$$

is an unbiased estimator of the standard error, giving the $(1-\alpha)-$level normal confidence interval

$$\left[\widehat{p}_v - z_{1-\alpha/2}\,\widehat{\sigma}_v,\; \widehat{p}_v + z_{1-\alpha/2}\,\widehat{\sigma}_v\right], \qquad z_{1-\alpha/2} \in \{1.96,\, 2.58, \dots\}.$$

The tally routine therefore needs to maintain only the scalar s_v while the simulation is running; the derived statistics can be updated in-place whenever a user requests intermediate results or at a fixed cadence.

10.3.2 Parallel Accumulation Algorithm

The accumulation kernel is invoked on a three-dimensional `nd_range`, chosen such that

$$\begin{aligned} \text{global}_x &\;\geq\; V, \\ \text{global}_y &\;\geq\; B, \\ \text{global}_z &\;\geq\; P. \end{aligned}$$

Work-item (i_x, i_y, i_z) is responsible for *exactly one* bit-pack:

$$\text{node } v = i_x, \quad \text{batch } b = i_y, \quad \text{pack } p = i_z.$$

Local workflow of a work-item

1. Load the p^{th} bit-pack of batch b from `buffer`.
2. Compute $c = \mathrm{popcount}(\text{bitpack})$.
3. Reduce the c's belonging to the same work-*group* in shared memory (tree reduction or `reduce_over_group`).
4. One designated leader performs `atomic_add`(`num_one_bits`, group_sum).

The reduction ensures only one atomic operation per group, greatly reducing contention when P is large.

We present platform-neutral pseudocode that encapsulates the above logic while remaining agnostic to the underlying API. After each Monte-Carlo iteration the host enqueues TALLYKERNEL with a fresh `iteration` counter. When either (i) a user requests intermediate statistics or (ii) a pre-set reporting interval is reached, the host reads back `num_one_bits` and executes the purely serial routine shown in Algorithm 6.

Algorithm 6 Post-processing of a single node's tally

Require: s - total one-bits, T, B, P, ω - run parameters
Ensure: $\widehat{p}$, $\widehat{\sigma}$, two symmetric CIs
1: $N \leftarrow B \cdot P \cdot \omega$
2: $\widehat{p} \leftarrow s/(T\,N)$
3: $\widehat{\sigma} \leftarrow \sqrt{\widehat{p}(1-\widehat{p})/(T\,N)}$
4: **for each** $z \in \{1.96,\ 2.58\}$ **do**
5: CI $\leftarrow \left[\max(0, \widehat{p} - z\widehat{\sigma}), \min(1, \widehat{p} + z\widehat{\sigma})\right]$
6: **end for**

The above normal approximation is valid provided $T\,N\widehat{p}$ and $T\,N(1-\widehat{p})$ both exceed roughly 10; otherwise an exact Clopper-Pearson interval can be substituted with no change to the running sum logic.

10.3.3 Correctness and Complexity

Work-item cost. Each work-item performs one $\mathrm{popcount}$ and participates in an $O(\log L)$ intra-group reduction ($L =$ local_range), yielding an overall $O(\log L)$ instruction count.

Global cost. The total number of work-items launched per iteration is $V \cdot B \cdot P$. Because each bit-pack contains ω Bernoulli trials, the cost *per trial* shrinks as ω^{-1}.

Memory traffic. Every work-item reads exactly one machine word and no writes occur except the single atomic addition per work-group. Hence the algorithm is memory-bandwidth bound only at extremely low arithmetic intensity ($P \approx 1$).

Linear scalability. All tally nodes are independent. Increasing V therefore scales the total runtime linearly until either (i) the device saturates its occupancy or (ii) atomic contention becomes non-negligible; the group-level reduction mitigates the latter.

10.4 Preliminary Benchmarks on Aralia Fault Trees

10.4.1 Runtime Environment and Benchmarking Setup

All experiments were performed on a consumer-grade desktop provisioned with an NVIDIA® GeForce GTX 1660 SUPER graphics card (1,408 CUDA cores, 6 GB of dedicated GDDR6 memory) and a 10th-generation Intel® Core™ i7-10700 CPU (2.90 GHz base clock, with turbo-boost and hyperthreading enabled). The code implementation relies on SYCL using the AdaptiveCpp (formerly HipSYCL) framework, which employs an LLVM based runtime and just-in-time (JIT) kernel compilation.

Monte Carlo Sampling Strategy

Each fault tree model was evaluated through a single pass (one iteration), generating as many Monte Carlo samples as would fit into the GPU's 6 GB memory. A 64-bit counter-based Philox4x32x10 random number generator was applied in parallel to produce the basic-event realizations. Note, with the exception of `das9205`, for which 5 passes were performed (in ≈ 0.96 seconds), all inputs were quantified using just one pass. We specifically chose `das9205` since its overall event probability is quite low, and requires many naive Monte Carlo samples.

Bit-Packing and Data Types

To reduce memory usage and increase vectorized throughput, every batch of Monte Carlo results was bit-packed into 64-bit words. Accumulated tallies of successes or failures were stored as 64-bit integers, while floating-point calculations (e.g., probability estimates) used double precision (64-bit floats). These design decisions are intended to maintain numerical consistency and make use of native hardware operations (such as population-count instructions for threshold gates).

Execution Procedure

Upon launching the application, the enabling overhead (host-device transfers, JIT compilation, and kernel configuration) was included in the total wall-clock measurement. Each benchmark was compiled at the `-O3` optimization level to ensure efficient instruction generation. Every experiment was repeated at least five times, and measured runtimes were averaged to reduce the impact of transient background processes or scheduling variations on the host system.

10.4.2 Assumptions and Constraints

The primary objective was to gauge runtime across a set of fault trees that vary widely in size, logic complexity, and probability ranges within a typical Monte Carlo integration workflow. The experiments assume independent operation of the test machine, with no significant other processes contending for GPU or CPU resources. All sampling took place within a single pass, so the measured wall times incorporate initial kernel launches, memory copies, and statistical collection of gate outcomes. No specialized forms of hardware optimization beyond the data-parallel approach (e.g., pinned memory or asynchronous streams) were used.

10.4.3 Comparative Accuracy and Runtime

Table 10.1 and Figure 10.1 summarize the accuracy of three approximate quantification methods Rare-Event Approximation (REA), Min Cut Upper Bound (MCUB), and our GPU-accelerated Monte Carlo by listing each approach's mean relative error in the log-probability ($\log p$) domain, alongside the total MC samples and runtime. Although each fault tree exhibits its own complexities, several broad trends emerge:

1. **REA accuracy strongly depends on the *actual* top-event probability.**

    - For trees with very low-probability failures (e.g., `baobab1`, `das9202`, `isp9605`), where individual component failures rarely coincide, REA's mean error often remains near or below 10^{-2} in log space. This indicates that summing only the first-order minimal cut sets–assuming higher-order

intersections contribute negligibly–can be valid when the system is indeed dominated by single-component or few-component events.

- However, for fault trees with moderate or higher top-event probabilities ($\gtrsim 10^{-2}$), REA's inaccuracy tends to grow (for instance, up to 10^{-1} in `edf9203`, `edf9204`, and `edfpa15b`). In these cases, ignoring the overlap of multiple cut sets leads to a visible systematic error.

2. **Min-Cut Upper Bound (MCUB) often mirrors REA but with exaggerated errors in certain overlapping cut configurations.**

   - In many models (e.g., `cea9601`, `baobab3`, `das9601`), MCUB closely tracks REA, suggesting that higher-order combinations remain negligible in those systems.

   - Yet, in a few cases involving heavy cut-set overlap (e.g., `das9209`, row 14), MCUB soars to a mean log-probability error of ~ 17, dwarfing REA or Monte Carlo. This highlights the well-known pitfall: if multiple cut sets are not genuinely "rare" and substantially overlap, the union bound becomes extremely loose.

3. **Monte Carlo yields more consistent and often dramatically lower numerical errors for most moderate- to high-probability top events.**

   - For example, in `das9201` (row 6) and `edf9203` (row 19), the Monte Carlo error is well below 10^{-3}, whereas both REA and MCUB can exceed 10^{-1}. In these situations, ignoring or bounding higher-order intersections proves inadequate, while direct sampling naturally captures all overlaps.

   - However, for fault trees with extremely small top-event probabilities, Monte Carlo's variance can become harder to control. For instance, some rows (`das9204`, `das9205`, `isp9605`, `isp9607`) show that roughly 10^8–10^9 samples are required to constrain the error within a few tenths in $\log p$. Those entries either exhibit a slightly higher Monte Carlo error than REA/MCUB or demonstrate that we needed a disproportionately large sample count (and thus more runtime) to compete with simple rare-event approximations.

4. **Sampling scale and runtime remain surprisingly feasible, even for up to 10^9 draws.**

   - Despite some test cases sampling in the hundreds of millions or billions, runtimes remain $\approx$ 0.2–0.3 s for most fault trees, rarely exceeding 1 s (see, for instance, row 10 with 3.3 B samples and $\sim$ 0.96 s). This indicates that the bit-packed, data-parallel Monte Carlo engine is highly optimized, making large-sample simulation a viable alternative to purely analytical approaches for many real-world PRA problems.

   - By contrast, the bounding methods (REA and MCUB) typically run in negligible time but deliver inconsistent accuracy depending on each tree's structure. In practice, a hybrid strategy may emerge: apply bounding methods for quick estimates, then selectively invoke large-sample Monte Carlo for trees or subsections where the bounding approximation diverges.

5. **Omitted or Extreme Cases.**

   - Rows where Monte Carlo entries are missing (e.g., `das9209` and `edf9206`) indicate difficulty in converging to a useful estimate within a fixed iteration budget. Conversely, MCUB shows er-

ratic jumps in some of those same cases, underlining the fact that both bounding and sampling approaches can struggle in certain outliers.

- Model `nus9601` (row 43) lacks all three error columns since no reference solution was available, reflecting a scenario where direct verification remains pending or inapplicable. Nevertheless, the completion time of ~ 0.29 s for a partial exploration suggests that the structural overhead of large fault trees can still be handled efficiently.

These results affirm that Monte Carlo methods, when equipped with high throughput sampling, can achieve the most robust accuracy across a broader spectrum of top-event probabilities, particularly in configurations where standard cut set approximations fail to capture significant event dependencies. At the same time, rare-event with exceptionally small probabilities can pose challenges for naive sampling, revealing the potential need for adaptive variance-reduction techniques or partial enumerations. In practice, analysts may combine bounding calculations (REA/MCUB) for quick screening or preparatory checks, then use hardware-accelerated Monte Carlo to refine those domains most susceptible to underestimation or overestimation by simpler approximations. Alternatively, for very large models, where exact solutions may be unavailable, data-parallel Monte Carlo can still estimate event probabilities without building minimal cut sets.

Table 10.1: Relative error (Log-probability), Data-Parallel Monte Carlo (DPMC) vs Min Cut Upper Bound (MCUB) and Rare-Event Approximation (REA).

#	Fault Tree	Relative Error $\left\lvert\log_{10}\left(\frac{P_{\text{approx}}}{P_{\text{exact}}}\right)\right\rvert$			Samples	Runtime (s)
		REA	MCUB	DPMC		
1	baobab1	1.46×10^{-4}	1.46×10^{-4}	7.62×10^{-3}	2.6×10^{8}	0.27
2	baobab2	6.49×10^{-3}	6.35×10^{-3}	1.55×10^{-3}	2.6×10^{8}	0.21
3	baobab3	1.22×10^{-2}	1.17×10^{-2}	2.25×10^{-4}	2.5×10^{8}	0.26
4	cea9601	9.37×10^{-2}	9.33×10^{-2}	2.42×10^{-3}	1.3×10^{8}	0.27
5	chinese	1.09×10^{-2}	1.07×10^{-2}	2.15×10^{-3}	9.5×10^{8}	0.28
6	das9201	1.27×10^{-1}	1.23×10^{-1}	5.50×10^{-5}	2.4×10^{8}	0.28
7	das9202	7.73×10^{-5}	2.58×10^{-5}	1.21×10^{-4}	5.3×10^{8}	0.30
8	das9203	3.60×10^{-2}	3.56×10^{-2}	2.32×10^{-4}	5.3×10^{8}	0.30
9	das9204	1.69×10^{-1}	1.69×10^{-1}	1.14×10^{-1}	6.2×10^{8}	0.30
10	das9205	9.64×10^{-2}	9.64×10^{-2}	2.77×10^{-2}	3.4×10^{9}	0.96
11	das9206	5.44×10^{-2}	8.90×10^{-4}	3.52×10^{-4}	2.1×10^{8}	0.27
12	das9207	1.19×10^{-1}	2.46×10^{-2}	1.37×10^{-4}	9.6×10^{7}	0.29
13	das9208	4.13×10^{-2}	3.82×10^{-2}	9.35×10^{-5}	2.6×10^{8}	0.31
14	das9209	2.12×10^{-2}	1.71×10^{1}			
15	das9601	5.30×10^{-2}	5.20×10^{-2}	6.68×10^{-4}	1.2×10^{8}	0.26
16	das9701	5.03×10^{-2}	3.38×10^{-2}	6.23×10^{-4}	2.4×10^{7}	0.28
17	edf9201	1.49×10^{-1}	5.37×10^{-2}	2.89×10^{-4}	1.9×10^{8}	0.32
18	edf9202	1.08×10^{-1}	6.06×10^{-3}	4.54×10^{-4}	7.9×10^{7}	0.28
19	edf9203	2.23×10^{-1}	1.18×10^{-1}	3.28×10^{-4}	8.1×10^{7}	0.31
20	edf9204	2.80×10^{-1}	1.06×10^{-1}	1.32×10^{-4}	8.8×10^{7}	0.30

Continued: Relative Error (Log-Probability), DPMC vs MCUB, REA.

#	Fault Tree	Relative Error $\left\lvert\log_{10}\left(\frac{P_{\text{approx}}}{P_{\text{exact}}}\right)\right\rvert$			Samples	Runtime (s)
		REA	MCUB	DPMC		
21	edf9205	9.95×10^{-2}	4.47×10^{-2}	5.61×10^{-5}	2.0×10^{8}	0.29
22	edf9206	6.99×10^{-3}	7.08×10^{-3}			
23	edfpa14b	1.86×10^{-1}	9.16×10^{-2}	1.05×10^{-3}	9.5×10^{7}	0.27
24	edfpa14o	1.87×10^{-1}	9.19×10^{-2}	3.40×10^{-4}	9.9×10^{7}	0.28
25	edfpa14p	3.41×10^{-2}	1.67×10^{-2}	5.36×10^{-4}	2.2×10^{8}	0.30
26	edfpa14q	1.86×10^{-1}	9.16×10^{-2}	3.34×10^{-4}	9.7×10^{7}	0.29
27	edfpa14r	2.49×10^{-2}	2.10×10^{-2}	9.34×10^{-4}	2.2×10^{8}	0.30
28	edfpa15b	2.17×10^{-1}	9.38×10^{-2}	4.68×10^{-4}	1.2×10^{8}	0.29
29	edfpa15o	2.17×10^{-1}	9.38×10^{-2}	4.07×10^{-5}	1.2×10^{8}	0.29
30	edfpa15p	2.53×10^{-2}	1.01×10^{-2}	3.55×10^{-4}	2.7×10^{8}	0.30
31	edfpa15q	2.17×10^{-1}	9.38×10^{-2}	6.75×10^{-4}	1.2×10^{8}	0.29
32	edfpa15r	1.95×10^{-2}	1.63×10^{-2}	4.05×10^{-4}	2.6×10^{8}	0.30
33	elf9601	1.99×10^{-2}	8.09×10^{-5}	7.87×10^{-5}	2.4×10^{8}	0.28
34	ftr10	1.23×10^{-1}	9.28×10^{-4}	1.55×10^{-4}	2.2×10^{8}	0.30
35	isp9601	8.09×10^{-2}	6.64×10^{-2}	1.14×10^{-4}	1.9×10^{8}	0.28
36	isp9602	1.75×10^{-2}	1.48×10^{-2}	1.36×10^{-3}	2.4×10^{8}	0.29
37	isp9603	3.83×10^{-2}	3.75×10^{-2}	3.83×10^{-3}	2.8×10^{8}	0.28
38	isp9604	1.21×10^{-1}	8.15×10^{-2}	1.89×10^{-4}	1.5×10^{8}	0.29
39	isp9605	6.58×10^{-3}	6.58×10^{-3}	2.94×10^{-2}	5.1×10^{8}	0.27
40	isp9606	2.28×10^{-2}	1.19×10^{-2}	1.31×10^{-4}	3.5×10^{8}	0.29
41	isp9607	2.39×10^{-2}	2.39×10^{-2}	1.29×10^{-1}	3.9×10^{8}	0.29
42	jbd9601	1.23×10^{-1}	1.36×10^{-2}	1.09×10^{-4}	5.8×10^{7}	0.28
43	nus9601				1.7×10^{7}	0.29

10.4.4 Memory Consumption

As mentioned previously, the memory was set to the maximum allocatable 6GB, constrained by the NVIDIA GTX 1660 SUPER GPU's VRAM. Figure 10.2a plots the actual consumed memory, as a function of PDAG input size and total number of bits sampled per node (gate or basic-event) per pass. Since there are multiple types of preprocessing steps, all of which affect the final size of the pruned PDAG, those have been plotted in Figure 10.2b for completeness. Since the nature of the actual pruning logic is not being benchmarked here, we named these v1, v2, v3 respectively. The key takeaways are that while some trees are more compressible than others, nearly all computations were performed by saturating available VRAM. As a zoomed out version of Figure 10.2a , Figure 10.3 projects trends for the sampled bits count, as a function of model size, for varying amounts of available RAM.

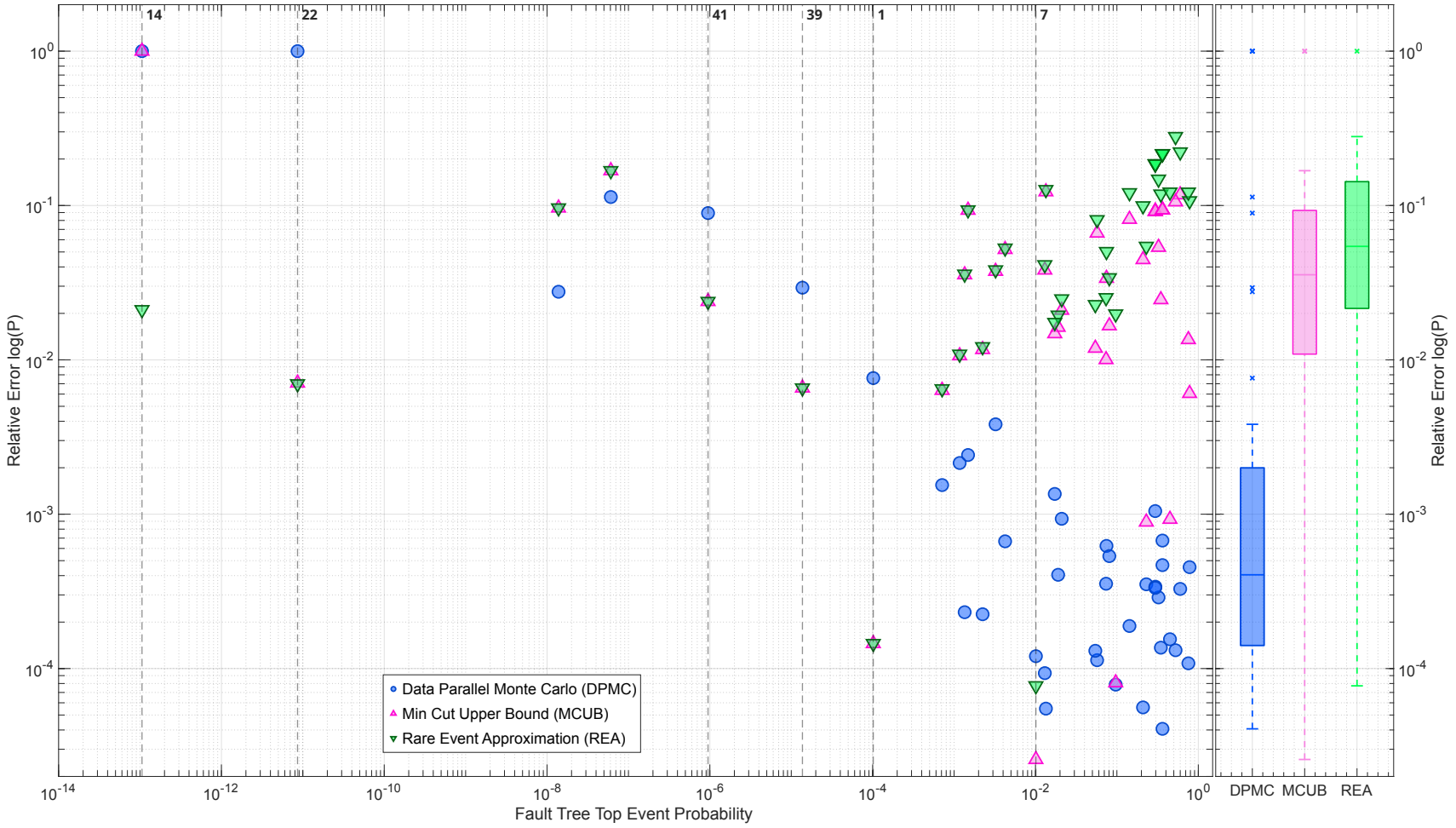
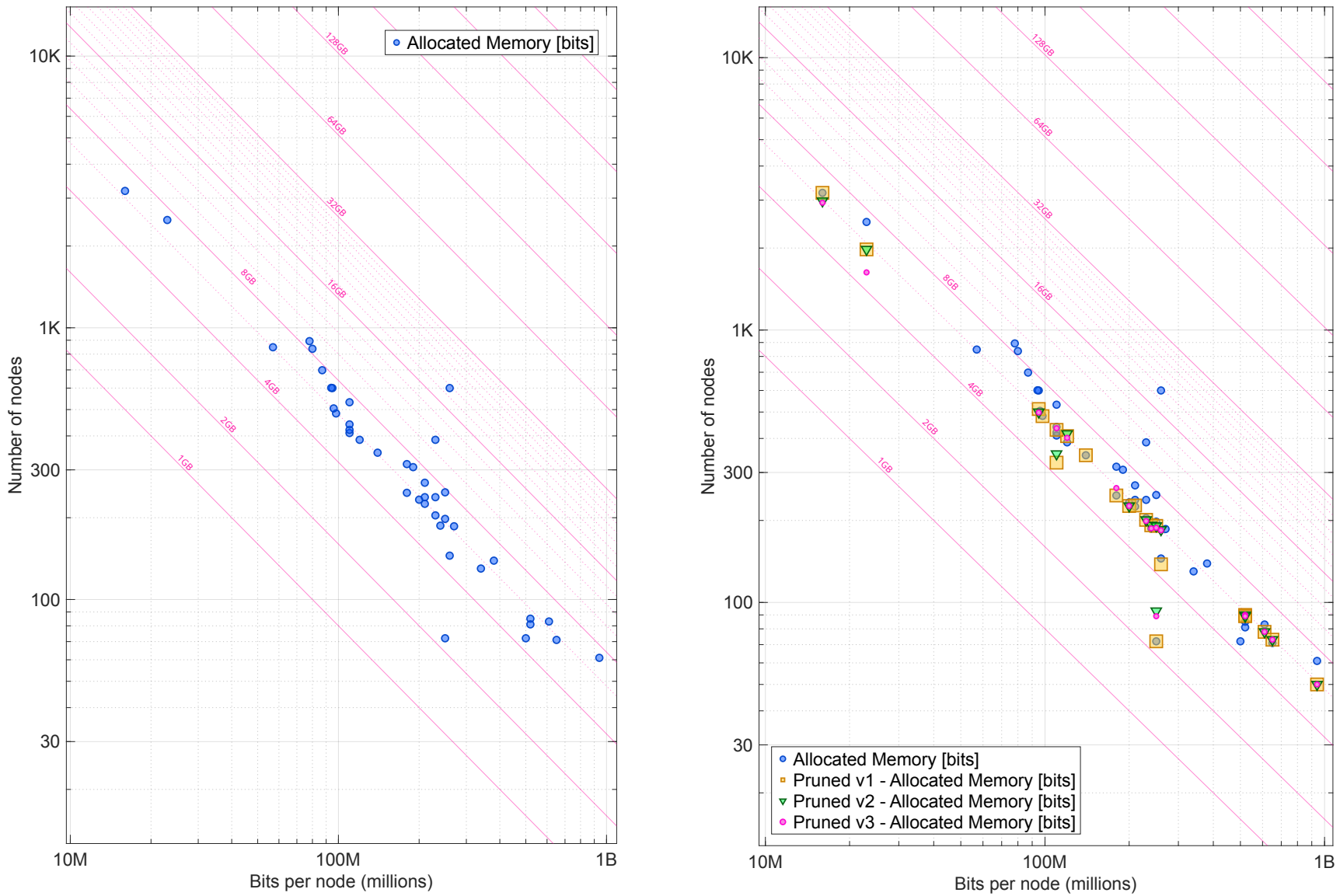


Figure 10.1: Relative error (Log-probability) for Data-Parallel Monte Carlo (DPMC) vs Min Cut Upper Bound (MCUB) and Rare-Event Approximation (REA)



(a) Sampled Bits per Event per Iteration

(b) Sampled Bits per Event per Iteration

Figure 10.2: Comparison of sampled bits per event per iteration.

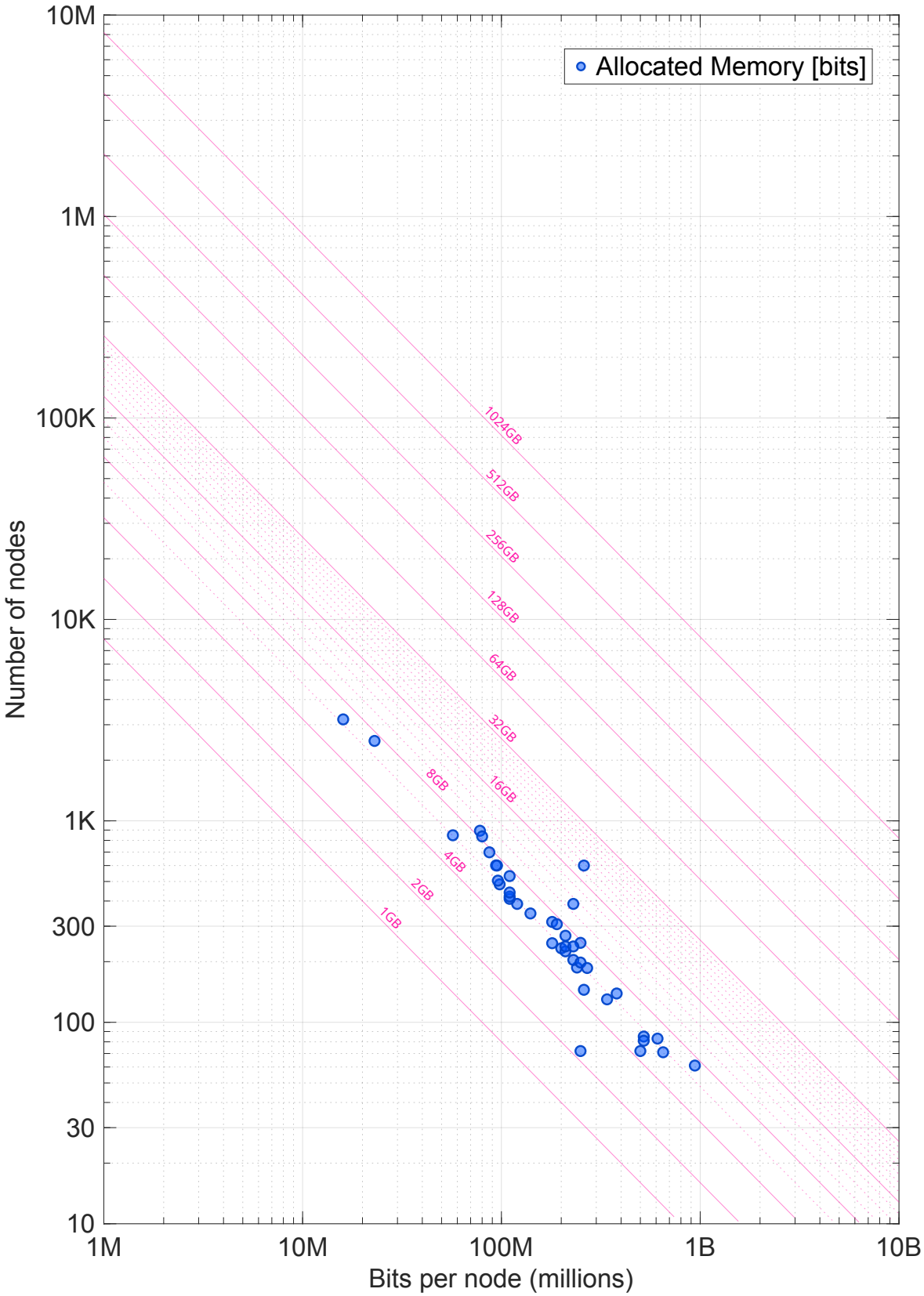


Figure 10.3: Sampled bits per event per iteration