ABSTRACT

EARTHPERSON, ARJUN. A Data-Parallel Monte Carlo Framework for Large-Scale PRA using Probabilistic Circuits. (Under the direction of Dr. Mihai A. Diaconeasa).

Probabilistic risk assessment (PRA) for nuclear systems typically requires enumerating minimal cut sets or essential prime implicants to capture all possible sequences of component failures, an approach that grows exponentially more complicated for large models. Existing methods combat this complexity by imposing structural restrictions on logic gates, applying rare-event approximations, or using bounding techniques like the min-cut upper bound. In this dissertation, we propose a data-parallel Monte Carlo framework that directly estimates probability distributions over entire Boolean spaces, circumventing many of the constraints faced by standard cut set analyses. By sampling from component-level input distributions rather than symbolically evaluating each logic configuration, our approach quantifies both success and failure events in one pass. Our open-source framework, Canopy, packs component states into integer bit-vectors and uses vectorized bitwise operations to achieve high throughput across modern GPUs, multi-core CPUs, and FPGAs. As an extension, we introduce a technique for sampling partial-derivatives using bitwise operations, which provides a pathway towards future work on gradient-based updates and other learning-based tasks.

We verify our implementation against the Aralia dataset, comparing convergence and accuracy with the open source SCRAM tool. Preliminary results demonstrate that with the exception of rare-events, data-parallel Monte Carlo, which is primarily memory-bound, can sample a few hundred million events in under 300ms while fully saturating available resources, even on older-generation consumer GPUs. For fault trees with a hundreds of events, sampling achieves error margins comparable to exact solvers. However, specialized variance-reduction or importance-sampling strategies remain essential for capturing extremely rare events with adequate precision.

We anticipate ongoing work on benchmarking the generic pressurized water reactor PRA model, in comparison with industry-standard tools (CAFTA, FTREX, SAPHIRE, XFTA), to provide further insights into solver performance and bottlenecks as the PRA model scales.

An Informal Overview

Probabilistic risk assessment (PRA) for nuclear systems requires evaluating complex Boolean functions that represent failure scenarios across thousands of components in two main steps: identifying minimal cut sets (the smallest sets of conditional events leading to end states of interest) and estimating the likelihood of these end states given component reliabilities. Although additional steps exist, PRA quantification primarily revolves around these tasks, which are computationally burdensome because exact solutions involve exhaustive inclusion-exclusion calculations that grow exponentially with the number of components, making large-scale analyses intractable. In response, many approximations have emerged: restricting model logic to a limited set of gate-types, describing only the failure space, applying probability truncation schemes like the rare-event approximation, or employing bounding methods such as the min-cut upper bound. Yet, even with these strategies, PRA quantification remains unwieldy in practice, demanding careful model management and tempered expectations regarding both accuracy and runtime.

Core Rationale - Building a Probability Estimator

We propose a data-parallel Monte Carlo (MC) scheme that samples directly from input component-reliability distributions. Rather than generating every possible failure combination, we focus on random draws of the system's global state, then tally how often particular sequences lead to an end state of interest. This sampling-based approach trades the intrinsic explosion of inclusion-exclusion terms with an exponential number of sample draws. This approach can accommodate a wider range of logic gates and event dependencies while putting the analyst in control of convergence. To unify these ideas, we use the concept of probabilistic circuits. Here, probability flow is represented as a structured computation graph over Boolean logic, but crucially, each gate's input distributions come from random samplers rather than a symbolic enumeration of all possible states. This allows the circuit to model both failures and successes while keeping the model size manageable. However, MC approaches for PRA model quantification are not new. The key design principle rests on a massively parallel, hardware accelerated sampling scheme. By encoding component states in compact bit vectors, we exploit hardware-level parallelism. Boolean logic gates, including primitives and complex logic such as K/N translate into efficient bitwise operations. These operations are performed as a computation graph defined by the logic structure of the PRA model itself. We implemented the framework in an open-source tool named Canopy, which is built with SYCL to promote code portability across consumer GPUs, multi-core CPUs, or specialized accelerators like FPGAs.

The Role of Partial Derivatives

One notable benefit of modeling probability distributions over the entire Boolean space is the ability to compute partial derivatives using the Shannon decomposition. Conceptually, the Shannon decomposition defines the derivative of Boolean function f(x), with respect to a subset of inputs. It is efficiently evaluated as a combination of bitwise XOR operations. Our approach samples from computation graphs representing arbitrary Boolean functions, including partial derivatives or more complex operations. This has immediate use for computing not only the conventional Birnbaum importance measure (which indicates how sensitive a system's failure probability is to a component's reliability) but also expanding to multi-component subsets. Additionally, since derivatives allow for gradient-based optimizations, there is potential for future work bridging advanced PRA and emerging machine-learning techniques. By embedding an auto-differentiation layer in the MC pipeline, one can imagine learning gate probabilities or even partial system structures from operational data, with the same data-parallel routines driving updates to reliability estimates.

Limitations

Sampling Rare Events

Ensuring accurate estimates for extremely low-probability events demands careful convergence criteria. In typical nuclear safety analyses, component failures can be vanishingly rare, so polling enough samples to capture these tail probabilities can prove challenging. Techniques such as importance sampling or other variance-reduction strategies may be needed to mitigate the slow convergence that arises when event likelihoods are measured in the range of 10^{-7} or lower.

Sampling Correlated or Dependent Events

A related concern is the treatment of common-cause failures (CCFs) and component dependencies. Characterizing CCF related failures or performing dependency analysis are important PRA activities. Future extensions of this framework thus require sampling from joint distributions, where correlated events are generated as a block. Integrating such dependent draws into bitpacked logic operations is feasible in principle but will require additional data-processing layers to ensure that correlated assignments remain both realistic and efficient to evaluate in large batches.

A Data-Parallel Monte Carlo Framework for Large-Scale PRA using Probabilistic Circuits

by
Arjun Earthperson

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Nuclear Engineering

Raleigh, North Carolina
2025

APPROVED BY:

Dr. Aydin Aysu	Dr. Yousry Azmy
Dr. Nam Dinh	Dr. Mihai A. Diaconeasa Chair of Advisory Committee

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background & Motivation

Probabilistic risk assessment (PRA) aims to quantitatively evaluate the likelihood and severity of adverse events in safety-critical industries. Driven by seminal works such as WASH-1400 and subsequent regulatory guidance, PRA now serves as a cornerstone of risk-informed decision-making in nuclear engineering. A canonical feature of PRA is its reliance on Boolean logic structures (fault trees and event trees) that characterize sequences of component failures and human actions leading to top-level undesirable outcomes. While such structures ensure thoroughness, the computational complexity of enumerating all failure paths grows exponentially in the number of components. Even moderate-scale reactor models may involve tens of thousands of basic events, rendering naive calculation of end-state probabilities intractable.

Over decades, analysts have adopted a series of approximations and bounding schemes to handle this combinatorial explosion. Strategies include rare-event approximations (which assume minimal overlap between failure sets), min-cut upper bounds (which treat all minimal cut sets as mutually exclusive), and restrictions on gate types to keep expansions manageable. Tools such as CAFTA, FTREX, SAPHIRE, SCRAM, and XFTA implement these methods and remain widely used in industry. Nonetheless, these approximations can lead to conservative estimations.

In recent years, the continued growth of computing power has encouraged reassessment of how PRA calculations can be modernized. Specifically, massively parallel hardware (e.g., GPUs and multi-core CPUs) has prompted the exploration of data-parallel methods. Monte Carlo sampling is a natural fit for parallelization: since each sample is independent, thousands or millions of system-state draws can be processed simultaneously to build empirical estimates of key probabilities. Straightforward sampling from component failures (rather than enumerating

complex Boolean expansions) offers flexibility in modeling dependencies and higher-order correlations. The overarching purpose of this dissertation is to develop a data-parallel Monte Carlo framework for large-scale nuclear PRA, grounded in a GPU-friendly integer bit-packing approach and extended to advanced sensitivity analyses using partial derivatives via Shannon decomposition.

1.2 Scope

This work reexamines how to efficiently compute the failure probability of a large Boolean system while capturing a wide array of gate structures, potential dependencies, and partial derivatives for sensitivity. We concentrate on the following core questions:

- How can large-scale PRA models be quantified without explicit minimal cut set enumeration or strict reliance on model simplifications?
- Which data structures and numerical techniques allow us to exploit parallel hardware such as GPUs, multi-core CPUs, and field-programmable gate arrays (FPGAs)?
- What are the current limitations of Monte Carlo (e.g., rare-event estimation and common-cause failure sampling), and how might variance reduction or more sophisticated sampling schemes mitigate them?

While our emphasis centers on nuclear applications, the proposed techniques and software are equally suitable for other industries that manage complex risk scenarios (e.g., aerospace, chemical processing, or automotive safety). The dissertation does not attempt to unify every advanced PRA feature (e.g., dynamic simulations or large correlated uncertainties), but it lays the foundation for an extensible data-parallel approach that can incorporate such features in the future.

1.3 Outline and Contributions

The primary technical contributions of this dissertation can be summarized as follows:

1. **Data-Parallel Monte Carlo for Boolean Systems:** We introduce a new framework that estimates probabilities for all *success* and *failure* states in a single run of Monte Carlo sampling. By representing each random system state in a bit-packed data structure, we achieve high-throughput simulations where Boolean operators (AND, OR, k/n, etc.) map naturally to bitwise operations on GPUs or multi-core CPUs.

2. **Integration with Probabilistic Circuits:** To unify event/fault tree logic with more flexible gate structures, we embed the model in a *probabilistic circuit* representation. This perspective enables node-level factorization and sum mixtures, opening doors to advanced decomposition-based analyses while retaining parallel-friendly evaluation.

3. **Sampling Techniques for Partial-Derivatives:** We develop a bitwise algorithm to approximate partial derivatives of the system's failure probability with respect to individual or clustered component reliabilities. By evaluating logical expressions under complementary assignments (as guided by the Shannon expansion), these derivatives can be computed in the same Monte Carlo pass. This capability facilitates advanced sensitivity and importance ranking in large models. It also opens a path towards integrating model evaluation with learning-based tasks.

4. **Benchmarking Against Industry Tools:** Through a series of case studies—most notably, the generic pressurized water reactor (PWR) reference model—we compare our approach with standard PRA tools (CAFTA, FTREX, SAPHIRE, SCRAM, XFTA). Results indicate that at comparable accuracy, our framework can surpass existing methods by orders of magnitude in runtime performance. We discuss how discrepancies in extremely low-probability events should be carefully monitored via convergence diagnostics.

5. **Prototype Implementation:** We present an open-source reference implementation named Canopy, built using the SYCL programming model. The code is portable across a variety of parallel architectures, including consumer GPUs and specialized accelerators. We provide usage examples and discuss future directions, such as unifying the approach with importance sampling to better handle rare events and building correlated sampling routines amenable to common-cause failure modeling.

1.4 Software Implementations

1.5 Related Publications

1.6 Organization of the Dissertation

Part I

Foundations

Chapter 2

Probabilistic Circuits

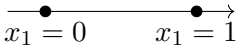
2.1 Boolean Functions

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a vector of n Boolean variables, where each $x_i \in \{0, 1\}$. A *Boolean function* is a map

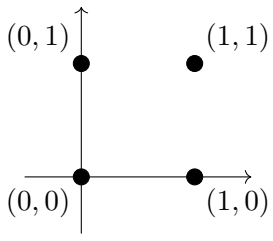
$$F(\mathbf{x}) \;=\; F(x_1, x_2, \ldots, x_n) \;\in\; \{0, 1\}. \tag{2.1}$$

This function takes each possible configuration of $\mathbf{x}$ (i.e., each element of $\{0, 1\}^n$) to a single binary output in $\{0, 1\}$. Boolean functions appear throughout digital logic, circuit design, and a wide range of computational applications.

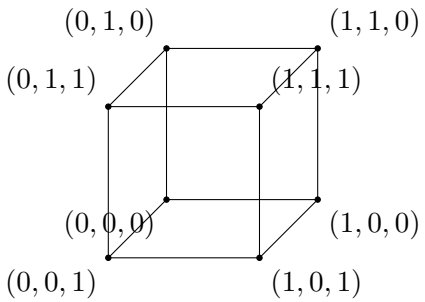
For illustration, we can visualize small Boolean functions based on the size of $\mathbf{x}$. For $n = 1$, there are two possible input states:



For $n = 2$, the four possible states can be positioned on a 2D lattice:



In three dimensions ($n = 3$), the eight possible states correspond to the vertices of a cube:



Boolean operators such as AND ($\wedge$), OR ($\vee$), and NOT ($\neg$) allow constructing a wide variety of logical relationships. For example, in a setting where a system fails if any one of two components fails, the Boolean function can be written as

$$F(x_A, x_B) \;=\; x_A \;\vee\; x_B.$$

Here, $F = 1$ precisely when $x_A = 1$ or $x_B = 1$, encompassing a failure event if either component A or B is in state 1. More complex systems with many interdependent components may require Boolean functions with numerous variables and deeply nested operators.

Although Boolean functions are crucial for representing logical configurations, they operate purely in a binary framework and do not directly encode probability distributions. To include probabilistic behavior, we can move to a more expressive framework called *probabilistic circuits*, which describe the distribution of variables in a directed acyclic graph (DAG). Such representations can capture both the combinatorial structure of system states and the uncertainty or likelihood associated with these states.

2.2 Definition and Structure

Consider a set of random variables $\mathbf{X} = (X_1, X_2, \ldots, X_n)$. A *probabilistic circuit* $\mathcal{C}$ is a DAG whose nodes consist of:

- **Input nodes (leaves):** Each leaf encodes a base distribution over some subset of $\mathbf{X}$. Often, these leaves correspond to univariate distributions $p(X_i)$ or constant/indicator functions.
- **Internal nodes (gates):** Each gate combines incoming distributions from its children using either:
  - **Sum-gates (mixture gates):** Weighted sums of child distributions, with non-negative weights summing to 1.

– **Product-gates:** Factorized products of child distributions, each child covering disjoint subsets of $\mathbf{X}$.

The acyclic nature of the graph ensures that information flows consistently from the leaves toward a designated *root* node.

2.2.1 Sum-Gates and Product-Gates

Let v be an internal node in $\mathcal{C}$. Denote the children of v by $\mathrm{ch}(v)$. Then:

- **Sum-gate:** Suppose v has children $u_1, \ldots, u_k$ with mixture weights $\{\theta_{v,u_i}\}_{i=1}^k$ satisfying $\sum_{i=1}^k \theta_{v,u_i} = 1$ and $\theta_{v,u_i} \geq 0$. The distribution encoded at v is

$$p_v(\mathbf{x}) \;=\; \sum_{i=1}^{k} \theta_{v,u_i}\, p_{u_i}(\mathbf{x}), \tag{2.2}$$

  where $p_{u_i}(\mathbf{x})$ is the distribution encoded by child node u_i.

- **Product-gate:** Suppose v has children $u_1, \ldots, u_k$, each covering disjoint subsets of $\mathbf{X}$. Let $\mathbf{X} = \bigcup_{i=1}^k \mathbf{X}_{u_i}$ and $\mathbf{X}_{u_i} \cap \mathbf{X}_{u_j} = \varnothing$ for $i \neq j$. Then the distribution at v is

$$p_v(\mathbf{x}) \;=\; \prod_{i=1}^{k} p_{u_i}\big(\mathbf{x}_{u_i}\big), \tag{2.3}$$

  where $\mathbf{x}_{u_i}$ is the restriction of $\mathbf{x}$ to the variables in $\mathbf{X}_{u_i}$.

2.2.2 Leaf Nodes and the Circuit Distribution

Each leaf node encodes a base distribution over its subset of variables (or a constant/indicator). Let v be a leaf node associated with $p_v(\mathbf{X}_v)$. When the circuit is evaluated, each leaf contributes its assigned distribution or constant term. By recursively composing sum-gates and product-gates, every node v in the circuit defines a distribution $p_v(\mathbf{x})$. The distribution of the entire circuit is given by evaluating its *root* node r:

$$p_r(\mathbf{x}) \;=\; (r \text{ evaluated from the leaves up}).$$

Probabilistic circuits unify structural and probabilistic modeling in a single formalism. They are widely used in fields such as artificial intelligence, machine learning, and automated reasoning, offering a tractable way to represent complex, high-dimensional probability distributions while preserving interpretable, compositional structure.

Chapter 3

Probabilistic Risk Assessment

3.1 The Triplet Definition of Risk

A central goal of risk analysis in nuclear engineering is to enable sound decision-making under large uncertainties. 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 Refs. **??**, 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

$$R \;=\; \left\{ \langle S_i,\, L_i,\, X_i \rangle \right\}_c \tag{3.1}$$

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. (3.1) 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 dissertation, uncertainties are embedded in each L_i (and sometimes X_i) via probability distributions.

3.1.1 Scenario Approach to PRA

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.

3.2 Definition of an Event Tree

Event trees unravel 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 3.1 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.

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 define

$$\Gamma \;=\; \Big\{ \langle I,\, F_1,\, F_2,\, \ldots,\, F_n,\, X_j \rangle \,:\, j \in J \Big\}, \tag{3.2}$$

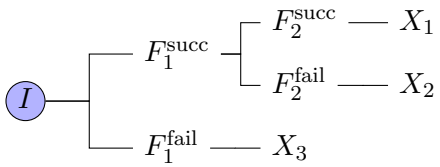


Figure 3.1: Illustrative event tree with an initiating event I, two functional events F_1 and F_2, and three end-states X_1, X_2, X_3.

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, core 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 probability and consequences map directly to L_i and X_i.

3.2.1 Probabilistic Representation

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

$$p(\omega_j) \;=\; p(I) \times \prod_{k=1}^{n} p\big(F_k^{\alpha_k} \mid I,\, F_1^{\alpha_1}, \ldots, F_{k-1}^{\alpha_{k-1}}\big). \tag{3.3}$$

The union of all such branches spans the full sample space of scenario outcomes generated by I and the subordinate functional events. Next, we show that every branch of an event tree can be represented by a product (logical AND) of the relevant Boolean variables for the initiating event and each functional event's success/failure. Collecting all branches via logical OR yields a disjunction of these products, precisely matching the standard structure of a Boolean expression in disjunctive normal form (DNF).

3.2.2 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*:

$$\omega_j \;\equiv\; \big(i = 1\big) \;\wedge\; \bigwedge_{k\in\alpha} \big(f_k^{\text{succ}} = 1\big) \;\wedge\; \bigwedge_{k\notin\alpha} \big(f_k^{\text{fail}} = 1\big). \tag{3.4}$$

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:

$$\Omega \equiv \omega_1 \vee \omega_2 \vee \cdots \vee \omega_m. \tag{3.5}$$

Substituting each branch's conjunction form (as in Eq. (3.4)) into Eq. (3.5) yields:

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

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

A standard DNF (sum-of-products) expression in Boolean algebra is

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

Each term in the sum (OR) is a logical AND of literals (variables or their negations). Comparing with Eq. (3.5), 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 Expressivity and Tractability of Event Trees

Within 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 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 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 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 negligiblly 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 SP−networks offer efficient linear-time evaluation with respect to the circuit size, the underlying circuit itself may have size exponential in n.

3.3 Definition of a Fault Tree

Whereas *event trees* begin with an initiating event and branch forward through possible outcomes, *fault trees* (FT) are 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 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.

3.3.1 Nodes in a Fault Tree: Events and Gates

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

- **Events**, which denote occurrences at different hierarchical levels.
  - *Basic events* (BEs) represent 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, large or intricate systems generally produce shared subtrees, yielding a more general DAG.

3.3.2 Common Gate Types in Fault Trees

FTs often use only a few canonical gate types (Figure 3.2), each describing a logical relationship among its inputs:

- **AND gate** – The output event occurs only if *all* input events occur.

$$\text{Output} = e_1 \wedge e_2 \wedge \ldots \wedge e_k.$$

- **OR gate** – The output event occurs if *any* input event occurs.

$$\text{Output} \;=\; e_1 \;\vee\; e_2 \;\vee\; \ldots \;\vee\; e_k.$$

- ***k*-out-of-*n* gate (Voting gate)** – The output event occurs if *at least* k of the n inputs fail. Denoted VOT(k/n), it can be expressed by a large OR of all possible subsets of size k, though notationally keeping it as one gate is much more concise.

$$\text{Output} \;=\; \Big[\sum_{i=1}^{n} e_i \;\geq\; k\Big].$$

- **INHIBIT gate** – The output event occurs if a specific trigger event happens *and* an additional conditioning event is present (often a privilege or a rarely active subsystem). This gate acts logically like an AND gate on two inputs, but is sometimes drawn differently for clarity.

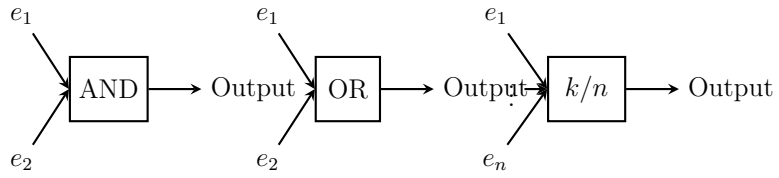


Figure 3.2: Examples of standard gate types in a fault tree: AND, OR, and k/n (voting).

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 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.
- $T : \mathcal{G} \to$ GateTypes assigns a gate type (AND, OR, k/n, etc.) to each gate in $\mathcal{G}$.

- $I : \mathcal{G} \to \mathcal{P}(\mathcal{B} \cup \mathcal{G})$ 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.

3.3.3 Fault Tree Semantics

To interpret a fault tree, we examine which higher-level events fail when a subset S of basic events have failed. Denote by $\pi_F(S, e)$ the *failure state* (0 or 1) of element e given a set $S \subseteq \mathcal{B}$ of failed basic events. 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 $\{x_1, \ldots, x_k\} \subseteq \mathcal{B} \cup \mathcal{G}$:

$$\pi_F(S, g) \;=\; \begin{cases} \bigwedge_{i=1}^{k} \pi_F(S, x_i), & \text{if } T(g) = \text{AND}, \\ \bigvee_{i=1}^{k} \pi_F(S, x_i), & \text{if } T(g) = \text{OR}, \\ 1 \;-\; \pi_F(S, x_1), & \text{if } T(g) = \text{NOT (single input)}, \\ \sum_{i=1}^{k} \pi_F(S, x_i) \;\geq\; k, & \text{if } T(g) = \text{VOT}(k/n), \\ \Big(\sum_{i=1}^{k} \pi_F(S, x_i)\Big) \bmod 2, & \text{if } T(g) = \text{XOR}, \end{cases}$$

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

3.3.3.1 Coherent vs. Noncoherent Fault Trees

FTs are usually *coherent*, meaning if a certain set of basic events causes a failure, then having more basic events fail cannot fix that failure—no NOT gates or other negations exist to

undo the top event. However, some advanced FTs allow thresholds or special conditions that introduce partial noncoherence (e.g., a system that can fail only if a valve is stuck *open* and a pump is *working* to overpressurize a vessel). Such cases add complexity and require extended gates or additional modeling logic.

3.3.4 Combining Shared Subtrees and Large Systems

In standard reliability analysis, certain fault tree computations—such as identifying *minimal cut sets* or calculating the probability of the top event—can be done by enumerating combinations of basic events that ensure system failure. However, if large shared subtrees exist or many gates gather multiple inputs, the expansion can become combinatorially challenging. Nevertheless, the DAG structure remains powerful for:

- **Clarity and Modularity:** Decomposing the system or subsystem failures into smaller, well-defined parts.
- **Reuse of Subtrees:** Representing components or subsystems that are relevant to multiple parts of the fault logic without arbitrarily duplicating them.
- **Analytical Efficiency:** Exploiting independence or bounding certain events, thereby limiting the combinatorial explosion.

3.3.5 Quantitative Analysis and Probability Estimation

Beyond describing which combinations of basic events can fail, most fault tree analyses (FTAs) require a quantitative assessment of the *likelihood* that the top event t ultimately occurs. This section details how to embed probabilities 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.

3.3.5.1 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.

3.3.5.2 Top 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 3.3 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 $\pi_F(S,t)=1$:

$$\Pr\big[X_t=1\big] \;=\; \sum_{S\subseteq\mathcal{B}} \Big[\pi_F(S,t) \prod_{b\in S} p(b) \prod_{b\notin S} \big[1-p(b)\big]\Big]. \tag{3.6}$$

A direct computation of (3.6) 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:

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

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

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

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

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

3.3.5.3 Dependence and Shared Subtrees

In many real-world fault trees, a single basic event b may feed into multiple gates, thereby making different branches of the tree dependent on one another. This violates the independence assumptions that simple bottom-up probability propagation requires. Below, we list several techniques that address or approximate these dependencies:

1. **Binary Decision Diagrams (BDDs).** Using BDDs involves encoding the entire Boolean structure of the fault tree into a single, canonical DAG. Sub-expressions within this diagram are cached, allowing an efficient evaluation of $\Pr[X_t = 1]$ even when basic events are shared across subtrees. In many cases, BDDs compress large FTs and facilitate fast reliability inference.

2. **Minimal Cut Sets (MCSs) and Approximations.** Many approaches characterize the top event in terms of its minimal cut sets. Let $\{\text{MCS}\}$ be the family of all minimal cut sets for the top event t. Each MCS $C \subseteq \mathcal{B}$ is a smallest set of basic events whose simultaneous failure brings down the system.

   - *Rare-Event Approximation.* When each $p(b)$ is small, the probability of two or more cut sets overlapping in their failures can be negligible. A common approximation is to sum the probabilities of each MCS as if they were mutually exclusive:

   $$\Pr[X_t = 1] \approx \sum_{C \in \{\text{MCS}\}} \prod_{b \in C} p(b). \tag{3.7}$$

   In highly reliable systems, this often yields a reasonable estimate.

   - *Min-Cut Upper Bound (MCUB).* A related bounding technique interprets the top event as the union of all MCS failures. By the union bound,

   $$\Pr[X_t = 1] \leq \sum_{C \in \{\text{MCS}\}} \prod_{b \in C} p(b). \tag{3.8}$$

   Known as the *min-cut upper bound*, this provides a guaranteed upper estimate for the probability of system failure. However, if multiple MCSs share common basic events of non-negligible failure probability, one might overestimate $\Pr[X_t = 1]$.

3. **Simulation-Based Methods.** Monte Carlo simulation bypasses many analytical complexities by directly sampling the failure state of each basic event from its distribution $p(b)$. After sampling, one evaluates the fault tree deterministically (i.e., checks whether $X_t = 1$). Repeating over many samples yields an empirical estimate of the top event probability. This approach is flexible, handles various dependencies, and easily extends to time-dependent or multi-state analyses. However, for extremely rare failures, simulation can require large sample sizes to achieve low-variance estimates.

Each of these techniques accommodates shared subtrees more effectively than naive bottom-up methods. In practice, analysts often deploy a combination of them (e.g., using

BDDs for certain gates, applying an MCS approximation for others) to balance accuracy, computational cost, and modeling complexity.

Chapter 4

Probability Estimation using Monte Carlo Sampling

4.1 Monte Carlo Fundamentals

Monte Carlo methods provide a versatile framework for approximating expectations, probabilities, and other quantities of interest by simulating random observations from an underlying distribution. At its core, a Monte Carlo estimator uses repeated random draws to approximate quantities such as

$$\mathbb{E}[f(X)] \;=\; \int f(x)\, p(x)\, \mathrm{d}x \;\;\approx\;\; \frac{1}{N}\sum_{i=1}^{N} f\big(x^{(i)}\big), \tag{4.1}$$

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.

4.1.1 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 4.1.1 (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. (4.1), 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).

4.1.2 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 4.1.2 (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 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.

4.2 Random Number Generation and Random Variates

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 generators (PRNGs) 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).

4.2.1 Pseudo-Random Number Generation

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 $U \sim \text{Unif}(0, 1)$, 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 $U \sim \text{Unif}(0, 1)$, 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 $p(x) \leq M\, q(x)$ for all x, then:

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

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

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

4.2.2 Boolean Events as Discrete Random Variables

In PRA contexts, 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

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

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

4.2.3 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
$$f_Y(y) \;=\; \lambda\, e^{-\lambda y}, \quad y \geq 0.$$

- **Weibull Distribution**, with flexible shape parameter $\beta > 0$ and scale parameter $\alpha > 0$. Its PDF is
$$f_Y(y) \;=\; \frac{\beta}{\alpha}\Big(\frac{y}{\alpha}\Big)^{\beta-1} \exp\Big[-\big(y/\alpha\big)^{\beta}\Big], \quad y \geq 0.$$

- **Lognormal Distribution**, where $\log(Y)$ follows a normal distribution. This is sometimes employed for components whose lifetimes span multiple orders of magnitude.

In a PRA context, 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

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

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.

Ensuring Reliable Sampling in High-Dimensional Boolean Spaces

When dealing with large-scale PRA models or deeply nested Boolean structures (multiple fault trees and event trees), a careful approach to random variate generation is needed:

- **Reusable Streams:** Use a consistent seeding and PRNG strategy to ensure reproducibility of results, especially when comparing multiple system configurations.
- **Parallel and Distributed Simulations:** Avoid overlapping random streams (i.e., ensure different parallel processes use uncorrelated seeds).
- **Validation of Randomness:** Use standard test suites (e.g. TestU01, Diehard) if the model's accuracy depends on fine-scale statistical properties.

Once random variates for each basic event are generated, higher-level logical structures (e.g. gates in a fault tree or branches in an event tree) can be evaluated deterministically. Subsequent sections will address how to form either a single *global* sample of all events in the system or to *factorize* the sampling process according to the structure of the DAG-based PRA model.

Chapter 5

Problem Statement

Part II

A Brute Force Approach

Chapter 6

Model Representation

6.1 PRA Models as 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→ET),

2. Multiple fault trees that themselves can reference or be referenced by other FTs (FT→FT),

3. Event trees that invoke fault trees to quantify key failure probabilities (ET→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 DAG. 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.

6.1.1 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 3.2, 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 3.3. 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→ET).
- Include a branch that requires computing "System X fails" via a fault tree Φ_j (ET→FT).

Similarly, a fault tree Φ_j may:

- Contain the top event of another fault tree Φ_k as one of its inputs (FT→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 DAG, 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).

6.1.2 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 DAG 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*[1], unified across $\{\Phi_1, \ldots, \Phi_N\}$.

[1] In some treatments, all FT internal nodes are called gates, even if no actual gate symbol is used. We adopt the broader term *FT nodes* for uniformity.

- Each FT node $g \in \mathcal{G}$ has one output event g (the node itself in the DAG).
- 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 {AND, OR, VOT(k/n), ...}. 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 3.2, 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 DAG property, v cannot be an ancestor of itself.

6.1.3 Edge Types and Probability Assignments

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

- **ET → 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 → 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, 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.

6.1.4 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.

6.1.4.1 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 3.3).

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).

6.1.4.2 Branching in the Event Trees.

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

6.1.4.3 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 DAG 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.

6.1.5 Formal Definition of the Unified Model

Bringing these elements together, we propose the following definition:

Definition 6.1.1 (Unified PRA Model)**.** A *unified PRA model* is a directed acyclic graph

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

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 **FT 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 **ET 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:

   - ET $\to$ ET: event-tree branching,
   - ET $\to$ FT: an event tree referencing a fault tree's top event,
   - FT $\to$ FT: linking one fault tree node to another's input,
   - FT $\to$ ET: an FT outcome passed back to an event tree node,
   - 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.

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 DAG structure codifies both the forward scenario expansions (ET logic) and the

top-down sub-component dependencies (FT logic) without creating contradictions or cycles. Definition 6.1.1 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 DAG 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 DAG representation.

6.2 Equivalent Canonical Representations for $\mathcal{M}$

6.2.1 Brief Overview of Equivalent Forms

6.2.2 Conversion Computation Complexity, Space & Time

6.2.3 Desirable Properties of an Equivalent Representation for $\mathcal{M}$

6.2.4 Homogeneity of Primitive Connectives - Kernel Simplification

6.2.4.1 Kernel Homogeneity & Recursion

6.2.4.2 Benefits of ANF: XOR & $\frac{\partial F}{\partial x}$

6.2.4.3 Benefits of DNF: Polynomial Time Satisfiability Checking

6.2.4.4 Potentially Fewer Literals

Chapter 7

Building a Data-Parallel Monte-Carlo Probability Estimator

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) 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 K-of-N 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. 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.

7.1 Layered DAG Topological Organization

Recall that a DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ 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 DAG: no gate or variable is computed until after all of its inputs are finalized.

7.1.1 Depth Computation and Node Collection

1. **Clear Previous State.** Any existing "visit" markers or stored depths in the DAG-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 DAG, 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 DAG from its root, recording each discovered node and adding it to a global list.

7.1.2 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

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

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.).

7.1.3 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.

7.2 Bitpacked Random Number Generator

Monte Carlo simulations, probability evaluations, and other sampling-based procedures benefit greatly from efficient, high-quality random number generators (RNGs). A large class of modern RNGs are known as *counter-based PRNGs*, 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, Philox 4×32-10 (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 (W32A $= 0x9E3779B9$ and 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.

7.2.1 Philox-4x32-10 Pseudocode

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 W32A $= 0x9E3779B9$ and 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).

Input : Four 32-bit counters (S_0, S_1, S_2, S_3), Key (K_0, K_1)
Output : Transformed counters (S_0, S_1, S_2, S_3)
Function `Philox_Round`$((S_0, S_1, S_2, S_3), (K_0, K_1))$**:**
  $P_0 \leftarrow M_{\mathrm{A}} \times S_0$;
  `// 64-bit product`
  $P_1 \leftarrow M_{\mathrm{B}} \times S_2$;
  `// 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 $(T_0, T_1, T_2, T_3), (K_0, K_1)$;
Function `Philox4x32_10`$((S_0, S_1, S_2, S_3), (K_0, K_1))$**:**
  for $i \leftarrow 1$ **to** 10 **do**
    $(S_0, S_1, S_2, S_3), (K_0, K_1) \leftarrow$ `Philox_Round`$(S_0, S_1, S_2, S_3, K_0, K_1)$;
  end
  return (S_0, S_1, S_2, S_3);

Algorithm 1: Philox-4x32-10

7.2.2 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 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 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.

Input : Probability $p \in [0, 1]$, 4 random 32-bit words (r_0, r_1, r_2, r_3)
Output : 4-bit integer bits, storing 4 Bernoulli draws
Function `FourBitPack`$(p, (r_0, r_1, r_2, r_3))$**:**

bits $\leftarrow 0$;
for $i \leftarrow 0$ **to** 3 **do**
  $u_i \leftarrow \frac{r_i}{2^{32}} \in [0, 1)$;
  $b_i \leftarrow \begin{cases} 1, & \text{if } u_i < p \\ 0, & \text{else} \end{cases}$;
  bits $\leftarrow$ bits $\parallel (b_i \ll i)$; `// Set bit` i `to` b_i
end
return bits;

Algorithm 2: Bitpacking of Four Bernoulli Samples Into a 4-Bit Block

In this procedure, $\parallel$ 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).

7.3 Preliminary Benchmarks

7.3.1 The Aralia Fault Tree Data Set

The Aralia dataset is a collection of 43 fault trees designed to exercise a wide array of logical features, problem sizes, and failure probability scales. In Table 7.1, each entry denotes a distinct fault tree with a different configuration of basic events, gates, and minimal cut sets, ultimately yielding broad diversity in both computational complexity and system reliability.

A prominent feature of the Aralia dataset is its wide range of fault-tree sizes and structures. On the smaller side, some references list only a few dozen basic events (e.g., `chinese` with 25 BEs or `isp9605` with 32 BEs). Others, such as `nus9601`, contain over 1,500 basic events, reflecting the scale of complex engineered systems or composite subsystems. Moreover, each fault tree employs a varied mix of `AND`, `OR`, `K/N` (voting), and occasionally `XOR` or `NOT` gates. This diversity of logical constructs makes Aralia an effective testbed for evaluating algorithms that parse and solve fault trees beyond the typical AND/OR structure.

Another key aspect of the Aralia models is the large spread in minimal cut-set counts and top-event probabilities. Some trees report only a few hundred or a few thousand minimal cut sets, while others claim tens of millions or more (reaching up to 8×10^{10} in the most expansive configurations). The top-event probabilities vary from very rare failures on the order of 10^{-13} to moderately likely failure rates above 0.7. This variance is crucial when assessing numerical stability and runtime performance: methods employing rare-event approximations can dramatically underestimate probabilities for the more frequent events, while computational overhead grows rapidly for cut-set expansions in highly interconnected models. All of the Aralia fault trees are provided in OpenPSA XML format.

Table 7.1: Aralia Fault Tree Dataset

#	Fault Tree	Basic Events	Logic Gates					Minimal Cut Sets	Top Event Probability
			Total	AND	K/N	XOR	NOT		
1	baobab1	61	84	16	9	-	-	46,188	1.01708E-04
2	baobab2	32	40	5	6	-	-	4,805	7.13018E-04
3	baobab3	80	107	46	-	-	-	24,386	2.24117E-03
4	cea9601	186	201	69	8	-	30	130,281,976	1.48409E-03
5	chinese	25	36	13	-	-	-	392	1.17058E-03
6	das9201	122	82	19	-	-	-	14,217	1.34237E-02
7	das9202	49	36	10	-	-	-	27,778	1.01154E-02

Table 7.1: Aralia Fault Tree Dataset

#	Fault Tree	Basic Events	Logic Gates					Minimal Cut Sets	Top Event Probability
			Total	AND	K/N	XOR	NOT		
8	das9203	51	30	1	-	-	-	16,200	1.34880E-03
9	das9204	53	30	12	-	-	-	16,704	6.07651E-08
10	das9205	51	20	2	-	-	-	17,280	1.38408E-08
11	das9206	121	112	21	-	-	-	19,518	2.29687E-01
12	das9207	276	324	59	-	-	-	25,988	3.46696E-01
13	das9208	103	145	33	-	-	-	8,060	1.30179E-02
14	das9209	109	73	18	-	-	-	8.20E+10	1.05800E-13
15	das9601	122	288	60	36	12	14	4,259	4.23440E-03
16	das9701	267	2,226	1,739	-	-	992	26,299,506	7.44694E-02
17	edf9201	183	132	12	-	-	-	579,720	3.24591E-01
18	edf9202	458	435	45	-	-	-	130,112	7.81302E-01
19	edf9203	362	475	117	-	-	-	20,807,446	5.99589E-01
20	edf9204	323	375	106	-	-	-	32,580,630	5.25374E-01
21	edf9205	165	142	30	-	-	-	21,308	2.09351E-01
22	edf9206	240	362	126	-	-	-	385,825,320	8.61500E-12
23	edfpa14b	311	290	70	-	-	-	105,955,422	2.95620E-01
24	edfpa14o	311	173	42	-	-	-	105,927,244	2.97057E-01
25	edfpa14p	124	101	42	-	-	-	415,500	8.07059E-02
26	edfpa14q	311	194	55	-	-	-	105,950,670	2.95905E-01
27	edfpa14r	106	132	55	-	-	-	380,412	2.09977E-02
28	edfpa15b	283	249	61	-	-	-	2,910,473	3.62737E-01
29	edfpa15o	283	138	33	-	-	-	2,906,753	3.62956E-01
30	edfpa15p	276	324	33	-	-	-	27,870	7.36302E-02
31	edfpa15q	283	158	45	-	-	-	2,910,473	3.62737E-01
32	edfpa15r	88	110	45	-	-	-	26,549	1.89750E-02
33	elf9601	145	242	97	-	-	-	151,348	9.66291E-02
34	ftr10	175	94	26	-	-	-	305	4.48677E-01
35	isp9601	143	104	25	1	-	-	276,785	5.71245E-02
36	isp9602	116	122	26	-	-	-	5,197,647	1.72447E-02
37	isp9603	91	95	37	-	-	-	3,434	3.23326E-03
38	isp9604	215	132	38	-	-	-	746,574	1.42751E-01

Table 7.1: Aralia Fault Tree Dataset

#	Fault Tree	Basic Events	Logic Gates					Minimal Cut Sets	Top Event Probability
			Total	AND	K/N	XOR	NOT		
39	isp9605	32	40	8	6	-	-	5,630	1.37171E-05
40	isp9606	89	41	14	-	-	-	1,776	5.43174E-02
41	isp9607	74	65	23	-	-	-	150,436	9.49510E-07
42	jbd9601	533	315	71	-	-	-	150,436	7.55091E-01
43	nus9601	1,567	1,622	392	47	-	-	unknown	unknown

7.3.2 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-IR-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, ensuring repeatability and independence for large sample counts.

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.

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.

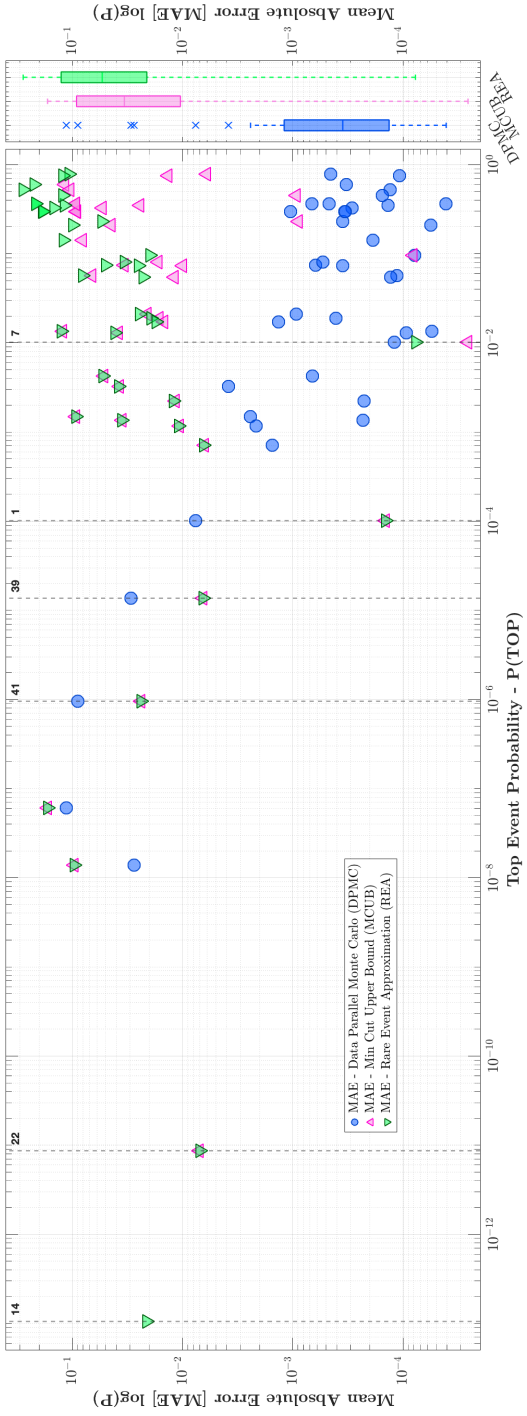


Figure 7.1: Mean Absolute Error – Exact (BDD) vs Approximate Methods

7.3.3 Results on Aralia Fault Trees: Comparative Accuracy and Runtime

Table 7.2: Mean Absolute Error vs Log-Probability (Approximate Methods)

#	Fault Tree	Mean Absolute Error - log(P)			MC Samples	Runtime [sec]
		REA	MCUB	Monte Carlo		
1	baobab1	$1.451\,56\times10^{-}$	1.45156E-04	7.61880E-03	2.5E+08	0.262
2	baobab2	$6.486\,28\times10^{-3}$	6.34705E-03	1.54436E-03	2.5E+08	0.209
3	baobab3	$1.215\,09\times10^{-2}$	1.16701E-02	2.24843E-04	2.4E+08	0.259
4	cea9601	$9.361\,95\times10^{-2}$	9.32207E-02	2.41802E-03	1.2E+08	0.262
5	chinese	$1.087\,42\times10^{-2}$	1.06354E-02	2.14601E-03	9.4E+08	0.277
6	das9201	$1.266\,49\times10^{-1}$	1.22765E-01	5.49963E-05	2.3E+08	0.279
7	das9202	$7.727\,43\times10^{-}$	2.57596E-05	1.20232E-04	5.2E+08	0.295
8	das9203	$3.590\,19\times10^{-2}$	3.55935E-02	2.31768E-04	5.2E+08	0.292
9	das9204	$1.680\,86\times10^{-1}$	1.68087E-01	1.13495E-01	6.1E+08	0.292
10	das9205	$9.638\,25\times10^{-2}$	9.63725E-02	2.76190E-02	3.3E+09	0.958
11	das9206	$5.435\,61\times10^{-2}$	8.89660E-04	3.51548E-04	2.0E+08	0.269
12	das9207	$1.184\,86\times10^{-1}$	2.45492E-02	1.36519E-04	9.5E+07	0.282
13	das9208	$4.128\,08\times10^{-2}$	3.81968E-02	9.34017E-05	2.5E+08	0.307
14	das9209	$2.112\,42\times10^{-}$	1.70245E+01		-	-
15	das9601	$5.292\,85\times10^{-2}$	5.19122E-02	6.67174E-04	1.1E+08	0.256
16	das9701	$5.028\,04\times10^{-2}$	3.37565E-02	6.22978E-04	2.3E+07	0.273
17	edf9201	$1.480\,12\times10^{-1}$	5.36182E-02	2.88906E-04	1.8E+08	0.315
18	edf9202	$1.071\,81\times10^{-1}$	6.05976E-03	4.53900E-04	7.8E+07	0.271
19	edf9203	$2.221\,46\times10^{-1}$	1.17293E-01	3.27993E-04	8.0E+07	0.302
20	edf9204	$2.795\,31\times10^{-1}$	1.05591E-01	1.31416E-04	8.7E+07	0.298
21	edf9205	$9.943\,39\times10^{-2}$	4.46260E-02	5.60146E-05	1.9E+08	0.284
22	edf9206	$6.987\,97\times10^{-}$	7.07775E-03	-	-	-
23	edfpa14b	$1.855\,74\times10^{-1}$	9.15983E-02	1.04767E-03	9.4E+07	0.267
24	edfpa14o	$1.864\,82\times10^{-1}$	9.18665E-02	3.39049E-04	9.8E+07	0.275
25	edfpa14p	$3.400\,10\times10^{-2}$	1.66283E-02	5.35099E-04	2.1E+08	0.294
26	edfpa14q	$1.856\,09\times10^{-1}$	9.15366E-02	3.33292E-04	9.6E+07	0.282
27	edfpa14r	$2.480\,88\times10^{-2}$	2.09729E-02	9.33865E-04	2.1E+08	0.294
28	edfpa15b	$2.163\,29\times10^{-1}$	9.37065E-02	4.67881E-04	1.1E+08	0.283

Table 7.2: Mean Absolute Error vs Log-Probability (Approximate Methods)

#	Fault Tree	Mean Absolute Error - log(P)			MC Samples	Runtime [sec]
		REA	MCUB	Monte Carlo		
29	edfpa15o	$2.165\,02\times10^{-1}$	9.37627E-02	4.06846E-05	1.1E+08	0.282
30	edfpa15p	$2.525\,68\times10^{-2}$	1.00382E-02	3.54344E-04	2.6E+08	0.299
31	edfpa15q	$2.163\,29\times10^{-1}$	9.37065E-02	6.74736E-04	1.1E+08	0.284
32	edfpa15r	$1.946\,93\times10^{-2}$	1.62668E-02	4.04924E-04	2.5E+08	0.290
33	elf9601	$1.981\,07\times10^{-2}$	8.08925E-05	7.86600E-05	2.3E+08	0.274
34	ftr10	$1.220\,76\times10^{-1}$	9.27268E-04	1.54844E-04	2.1E+08	0.297
35	isp9601	$8.083\,92\times10^{-2}$	6.63074E-02	1.13264E-04	1.8E+08	0.271
36	isp9602	$1.745\,72\times10^{-2}$	1.47782E-02	1.35280E-03	2.3E+08	0.281
37	isp9603	$3.823\,37\times10^{-2}$	3.74815E-02	3.82344E-03	2.7E+08	0.278
38	isp9604	$1.208\,89\times10^{-1}$	8.14313E-02	1.88665E-04	1.4E+08	0.280
39	isp9605	$6.573\,44\times10^{-}$	6.57032E-03	2.93472E-02	5.0E+08	0.262
40	isp9606	$2.278\,11\times10^{-2}$	1.18983E-02	1.30307E-04	3.4E+08	0.289
41	isp9607	$2.388\,80\times10^{-}$	2.38880E-02	1.28136E-01	3.8E+08	0.282
42	jbd9601	$1.220\,01\times10^{-1}$	1.35343E-02	1.08116E-04	5.7E+07	0.279
43	nus9601		-	-	1.6E+07	0.289

Table 7.2 summarizes 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 absolute 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. **Rare Event Approximation (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 ≈ 0.2–0.3 s for most fault trees, rarely exceeding 1 s (see, for instance, row 10 with 3.3 B samples and ~ 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 erratic 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, often 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.

Part III

Refinements

Chapter 8

Variance Reduction

8.1 Dealing with Rare Events using Importance Sampling

8.1.1 Interplay between ultra-rare and ultra-frequent events - how they affect convergence

8.2 Sampling Correlated Events

Part IV

Learning

Chapter 9

Towards Parameter Learning

PRAs invariably involve uncertainty. When explicitly modeled, these uncertainties can be updated or inferred from evidence, engineering judgments, or reliability targets. We refer to such systematic updating of probability or frequency distributions across the PRA model as form of parametric learning.

Recall from (Section 6.1) that we represent a PRA model as a PDAG. Let $\boldsymbol{\theta}$ be the collection of parameters governing all relevant probabilities/frequencies in this PDAG. For an end-state S_j, the model-based prediction under $\boldsymbol{\theta}$ is

$$P_{\mathcal{M}}\big(S_j \mid \boldsymbol{\theta}\big).$$

If one also has observed or target frequencies $\big\{p_j^{\text{obs}}\big\}$, parametric learning seeks to reconcile this information with the model's predictions by updating $\boldsymbol{\theta}$. In a Bayesian setting, one may specify a prior distribution over $\boldsymbol{\theta}$ and update this prior to a posterior distribution via the likelihood of observed end-state frequencies or other system-level evidence. Alternatively, one may adopt an optimization-based approach: define a loss or cost function that measures the discrepancy between $\{p_j^{\text{obs}}\}$ and $\{P_{\mathcal{M}}(S_j \mid \boldsymbol{\theta})\}$, then minimize this loss with respect to $\boldsymbol{\theta}$. Both perspectives aim to systematically adjust the PRA model's probabilistic parameters so that end-state frequencies (or other risk metrics) remain consistent with available data or requirements.

In the next section, we show how parametric learning over the PDAG can be setup as a constrained optimization problem.

9.1 Parameter Learning as Constrained Optimization

Each node X_i in the PDAG has an associated parameter θ_i, gathered into a vector

$$\boldsymbol{\theta} \;=\; (\theta_1,\; \theta_2,\; \ldots,\; \theta_n).$$

For a set of end-states $\{S_j\}_{j=1}^{m}$, the model's predicted probability under $\boldsymbol{\theta}$ is

$$p_j^{\text{pred}}(\boldsymbol{\theta}) \;=\; P_{\mathcal{M}}\big(S_j \mid \boldsymbol{\theta}\big).$$

Suppose observed or target frequencies $\left\{p_j^{\text{obs}}\right\}$ are given. A discrepancy measure

$$d\big(p_j^{\text{obs}},\, p_j^{\text{pred}}(\boldsymbol{\theta})\big)$$

compares the model's predictions to these values. One can also add a regularization term $\Psi(\boldsymbol{\theta})$ to encode additional constraints such as engineering limits or prior information. Let Ω denote the feasible set for $\boldsymbol{\theta}$, enforcing domain-specific requirements (e.g., probability normalization). Parameter learning then becomes the following constrained optimization problem:

$$\min_{\boldsymbol{\theta}\,\in\,\Omega} \quad \sum_{j=1}^{m} d\Big(p_j^{\text{obs}},\, p_j^{\text{pred}}(\boldsymbol{\theta})\Big) \;+\; \Psi(\boldsymbol{\theta}).$$

A solution $\boldsymbol{\theta}^*$ in Ω is sought that minimizes overall discrepancy while respecting any additional constraints. Gradient-based methods (when d is differentiable) or other solvers can be employed.

9.2 Case Study: EBR-II Liquid Metal Fire Scenario

We apply the proposed optimization method to an event tree from the Experimental Breeder Reactor-II (EBR-II) Level I PRA (Chang 2018). The potential initiating event is a leak in the piping loop of the reactor's shutdown cooler, which uses sodium-potassium (NaK) coolant. Air intrusion near NaK can cause fire hazards. The event tree, shown in Figure 9.1 enumerates whether (i) the liquid-metal fire is detected in time (LMFD), (ii) a reactor scram is successfully initiated (RFIR), (iii) the fire is classified as severe or limited (LLRF), (iv) a plant-level fire suppression system fails or succeeds (SSSD), and (v) critical secondary systems remain operational (SYSO). These conditional events interact to form multiple end-states, labeled SDFR-0 through SDFR-8. Some end-states represent minimal impact (e.g., immediate

fire detection and promptly executed scram), whereas others lead to more severe conditions (e.g., no detection and system failures yielding potential core damage).

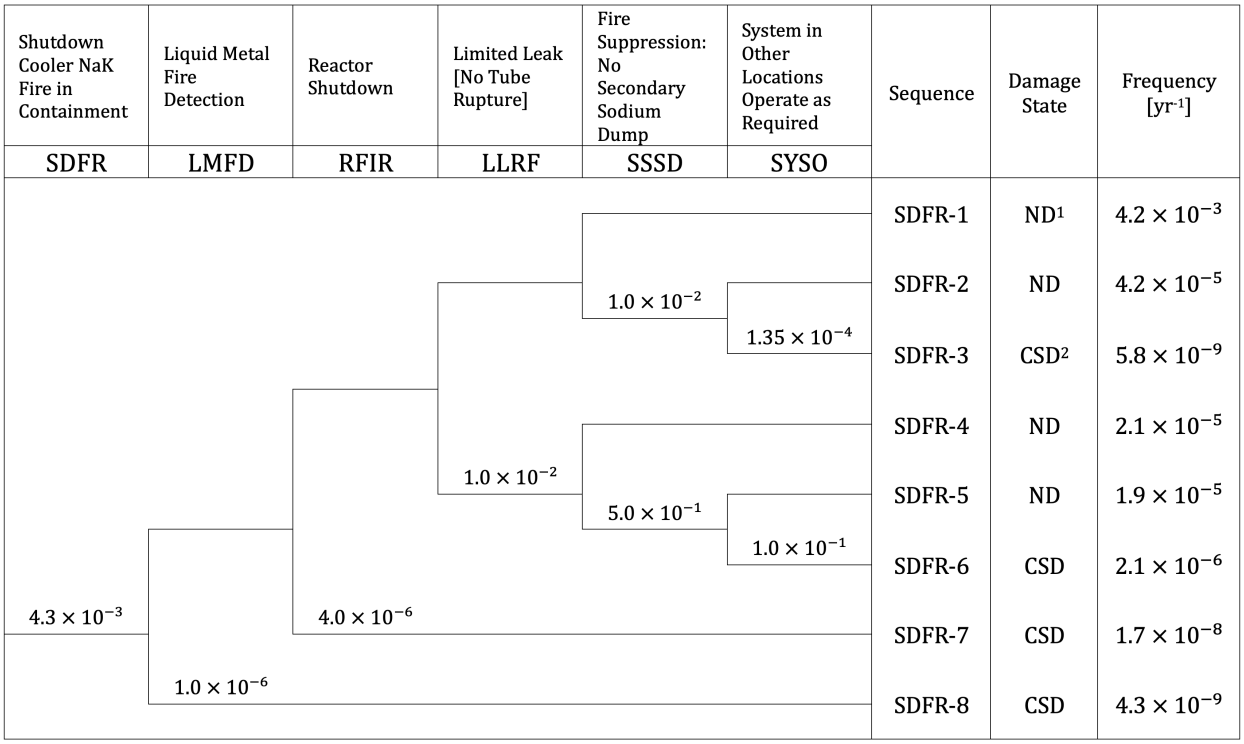


Figure 9.1: EBR-II Shutdown Cooler NaK Fire in Containment

9.2.1 Event Tree Structure and Problem Setup

Following the notation from Section 3.2, each end-state S_j arises from a particular path of success/failure outcomes across the conditional events. Let $\{X_1, \ldots, X_n\}$ be the events (e.g., $\text{LMFD}, \text{RFIR}, \ldots$), and let $y_{ji} \in \{0, 1, \text{NaN}\}$ indicate whether X_i fails, succeeds, or is not applicable for path S_j. The probability of end-state S_j is

$$P(S_j) \;=\; \prod_{i=1}^{n} P\big(y_{ji}\big), \tag{9.1}$$

where

$$P\big(y_{ji}\big) \;=\; \begin{cases} P\big[X_i = 1\big], & \text{if } y_{ji} = 1, \\ 1 - P\big[X_i = 1\big], & \text{if } y_{ji} = 0, \\ 1, & \text{if } y_{ji} = \text{NaN}. \end{cases} \tag{9.2}$$

Thus, one may represent each end-state S_j by multiplying the associated conditional event probabilities along its branch of the tree.

In this case study, each $P\big[X_i = 1\big]$ is assigned a (truncated) log-normal parameterization, reflecting the fact that event probabilities can span several orders of magnitude. Let μ_i, σ_i denote the log-space mean and standard deviation of event X_i. Under truncation rules (e.g., restricting $\mu_i \in [10^{-10}, 1]$ and $\sigma_i \in [10^{-10}, 10^4]$), the resulting probability stays in $(0, 1)$ and avoids extreme instabilities. These are plotted in Figure 9.2 as normalized kernel density estimatesTerrell and Scott (1992).

9.2.2 Loss Function Definition

Given a set of target or observed end-state frequencies $\{p_j^{\text{obs}}\}_{j=1}^m$, the task is to infer $\{\mu_i, \sigma_i\}_{i=1}^n$ so that the predicted frequencies

$$p_j^{\text{pred}} \;\equiv\; P\big(S_j \mid \{\mu_i, \sigma_i\}\big)$$

match p_j^{obs} as closely as possible. Denoting $\boldsymbol{\theta} = (\mu_1, \sigma_1, \ldots, \mu_n, \sigma_n)$ for all events, the optimal parameters solve a constrained minimization:

$$\min_{\boldsymbol{\theta} \in \Omega} \quad \mathcal{L}\big(\boldsymbol{\theta};\, \{p_j^{\text{obs}}\}\big) \quad \text{subject to truncation and system constraints,} \tag{9.3}$$

where Ω encodes bounds (e.g., $\mu_i, \sigma_i \geq 10^{-10}$), and $\mathcal{L}$ is a loss function. Here, one defines $\mathcal{L}$ via a Normalized Relative Logarithmic Error (NRLE), which balances discrepancies in both the predicted end-state frequencies and the tails of the distributions. A simplified version of NRLE is:

$$\text{NRLE} \;=\; \frac{1}{m} \sum_{j=1}^{m} \frac{1}{2} \Big(\text{MAE}\big(\log[p_j^{\text{obs}} + \epsilon],\, \log[p_j^{\text{pred}} + \epsilon]\big) \;+\; \text{MAE}\big(\sigma_j^{\text{obs}},\, \sigma_j^{\text{pred}}\big) \Big), \tag{9.4}$$

where MAE denotes mean absolute error, and ϵ is a small positive constant to avoid $\log(0)$. The terms σ_j^{obs} and σ_j^{pred} refer to log-space standard deviations for the respective distributions of (or mapped from) end-states or functional events. By design, this objective penalizes deviations of both central tendencies and spread. A gradient-based algorithm (e.g., Adam

(Zhang 2018)) then iteratively refines $\{\mu_i, \sigma_i\}$, using automatic differentiation with respect to $\mathcal{L}$.

9.2.3 Results & Discussion

The parameter estimation recovered target distributions and corresponding end-state frequencies with near-accurate fidelity, indicating that the method is capable of approximating underlying probabilities from limited inputs. The predicted end-state frequency estimates are plotted in Figure 9.5.

Specifically, end-state frequencies estimated under the constrained optimization process diverged from reported references by small margins: on average, the mean values were recovered with an error of about $(1.08 \pm 0.96)\%$, the 5th percentile with $(4.39 \pm 7.09)\%$, and the 95th percentile with $(3.82 \pm 5.91)\%$. Such deviations suggest that the overall approach captures the central tendencies of event probabilities reasonably well, while still exhibiting moderate scatter in both lower and upper distribution tails. Recurrence of larger discrepancies in selected events (e.g., certain fire detection or suppression paths) emphasizes the known difficulty of accurately modeling rare failure or success probabilities—particularly when the choice of distribution (e.g., log-normal) imposes strong structural assumptions on the shapes of these probability curves.

Despite these promising quantitative metrics, two issues warrant discussion. First, although end-state frequencies are reproduced within small mean errors, there is a real possibility of overfitting to the specified targets. The optimization-driven procedure can finely tune parameters to minimize a chosen loss function; however, doing so may lead to calibrated event probabilities that reflect artifacts of the objective rather than a physically robust representation. This risk is heightened when dealing with low-probability events (e.g., a rare liquid metal fire condition combined with other system failures)—situations that often exhibit limited empirical data.

Second, the truncation and bounds on the log-normal parameterization, while necessary for numerical stability, can restrict the feasible solution space in unintended ways. Large or extremely small event probabilities, particularly in tail regions, must fit within these truncated distributions. If the true system behavior lies outside the assumed bounds, the resulting estimates may systematically under- or overestimate important tail events. This possibility is underscored by the modest underestimation observed at the 95th percentile for certain functional events in the demonstration.

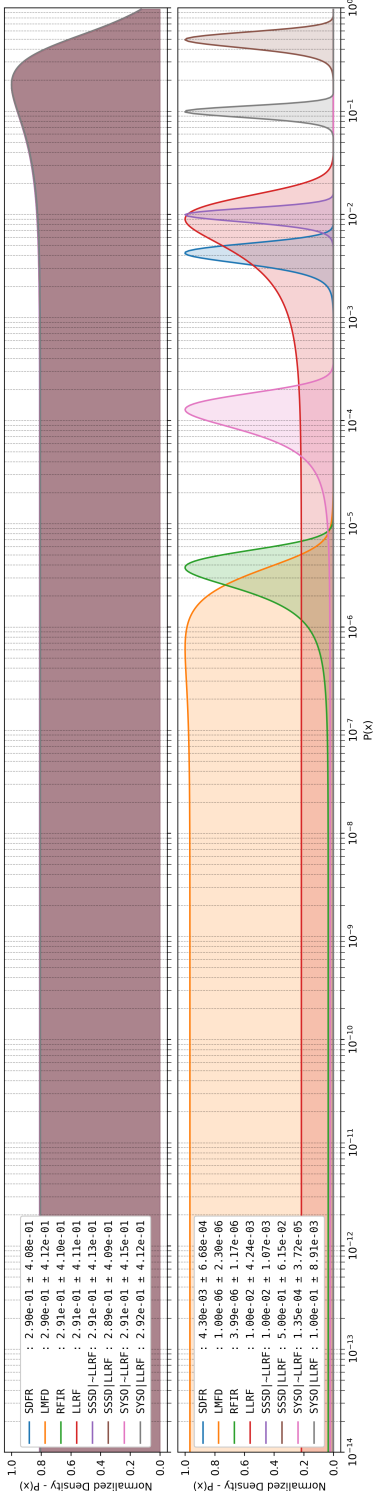


Figure 9.2: Initial vs Target Functional Event Probability Distributions

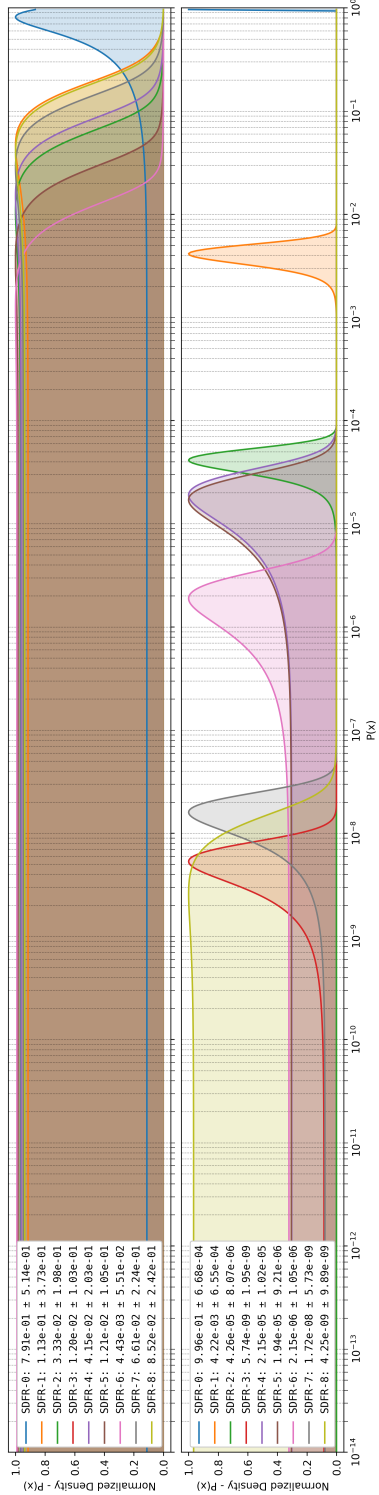


Figure 9.3: Initial vs Target End-State Frequency Distributions

Table 9.1: Estimated vs Target Functional Event Probabilities Summarized

Event	5^{th} Percentile			Mean			95^{th} Percentile		
	Estimated	Target	Error[1]	Estimated	Target	Error[1]	Estimated	Target	Error[1]
SDFR	3.28×10^{-3}	3.30×10^{-3}	-2.30×10^{-5}	4.24×10^{-3}	4.30×10^{-3}	-5.97×10^{-5}	5.37×10^{-3}	5.47×10^{-3}	-1.07×10^{-4}
LMFD	4.39×10^{-8}	4.29×10^{-8}	9.85×10^{-10}	1.00×10^{-6}	9.99×10^{-7}	7.13×10^{-10}	3.70×10^{-6}	3.70×10^{-6}	-5.09×10^{-9}
RFIR	2.49×10^{-6}	2.39×10^{-6}	9.28×10^{-8}	4.13×10^{-6}	4.00×10^{-6}	1.29×10^{-7}	6.32×10^{-6}	6.15×10^{-6}	1.71×10^{-7}
LLRF	4.73×10^{-3}	4.72×10^{-3}	1.94×10^{-5}	9.93×10^{-3}	9.99×10^{-3}	-6.36×10^{-5}	1.77×10^{-2}	1.79×10^{-2}	-2.25×10^{-4}
SSSD | $\overline{\text{LLRF}}$[2]	8.72×10^{-3}	8.35×10^{-3}	3.68×10^{-4}	1.01×10^{-2}	1.00×10^{-2}	5.37×10^{-5}	1.15×10^{-2}	1.19×10^{-2}	-3.31×10^{-4}
SSSD | LLRF	4.93×10^{-1}	4.06×10^{-1}	8.74×10^{-2}	4.94×10^{-1}	5.00×10^{-1}	-5.48×10^{-3}	4.96×10^{-1}	6.07×10^{-1}	-1.11×10^{-1}
SYSO | $\overline{\text{LLRF}}$	8.24×10^{-5}	8.33×10^{-5}	-9.54×10^{-7}	1.36×10^{-4}	1.35×10^{-4}	8.31×10^{-7}	2.07×10^{-4}	2.03×10^{-4}	3.85×10^{-6}
SYSO | LLRF	8.54×10^{-2}	8.60×10^{-2}	-6.49×10^{-4}	9.89×10^{-2}	1.00×10^{-1}	-1.11×10^{-3}	1.14×10^{-1}	1.15×10^{-1}	-1.66×10^{-3}

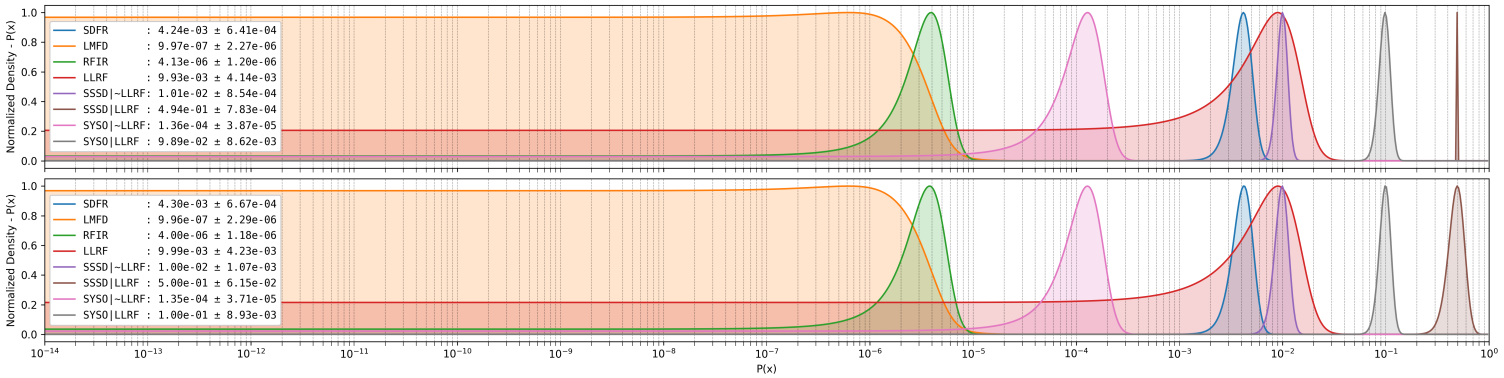


Figure 9.4: Estimated vs Target Functional Event Probability Distributions

[1][Estimated - Target], negative values represent underestimates.

[2]$A \mid \overline{B}$: event A conditional on the non-occurrence of event B.

Table 9.2: Estimated vs Target End-State Frequencies Summarized

Event	5^{th} Percentile			Mean			95^{th} Percentile		
	Estimated	Target	Error[3]	Estimated	Target	Error[1]	Estimated	Target	Error[1]
SDFR-0[4]	9.95×10^{-1}	9.95×10^{-1}	1.08×10^{-4}	9.96×10^{-1}	9.96×10^{-1}	5.98×10^{-5}	9.97×10^{-1}	9.97×10^{-1}	2.30×10^{-5}
SDFR-1	3.21×10^{-3}	3.23×10^{-3}	-2.26×10^{-5}	4.16×10^{-3}	4.22×10^{-3}	-5.86×10^{-5}	5.27×10^{-3}	5.37×10^{-3}	-1.06×10^{-4}
SDFR-2	3.12×10^{-5}	3.07×10^{-5}	5.81×10^{-7}	4.22×10^{-5}	4.26×10^{-5}	-3.56×10^{-7}	5.52×10^{-5}	5.69×10^{-5}	-1.70×10^{-6}
SDFR-3	3.15×10^{-9}	3.15×10^{-9}	-7.36×10^{-12}	5.73×10^{-9}	5.74×10^{-9}	-1.31×10^{-11}	9.33×10^{-9}	9.35×10^{-9}	-2.39×10^{-11}
SDFR-4	9.62×10^{-6}	9.22×10^{-6}	4.07×10^{-7}	2.13×10^{-5}	2.15×10^{-5}	-1.90×10^{-7}	3.94×10^{-5}	4.07×10^{-5}	-1.33×10^{-6}
SDFR-5	8.47×10^{-6}	8.31×10^{-6}	1.66×10^{-7}	1.88×10^{-5}	1.94×10^{-5}	-5.87×10^{-7}	3.47×10^{-5}	3.67×10^{-5}	-2.04×10^{-6}
SDFR-6	9.10×10^{-7}	9.07×10^{-7}	3.35×10^{-9}	2.06×10^{-6}	2.15×10^{-6}	-9.10×10^{-8}	3.85×10^{-6}	4.13×10^{-6}	-2.86×10^{-7}
SDFR-7	9.84×10^{-9}	9.58×10^{-9}	2.64×10^{-10}	1.75×10^{-8}	1.72×10^{-8}	3.08×10^{-10}	2.82×10^{-8}	2.79×10^{-8}	3.17×10^{-10}
SDFR-8	1.81×10^{-10}	1.80×10^{-10}	1.85×10^{-12}	4.18×10^{-9}	4.24×10^{-9}	-5.50×10^{-11}	1.56×10^{-8}	1.58×10^{-8}	-2.53×10^{-10}

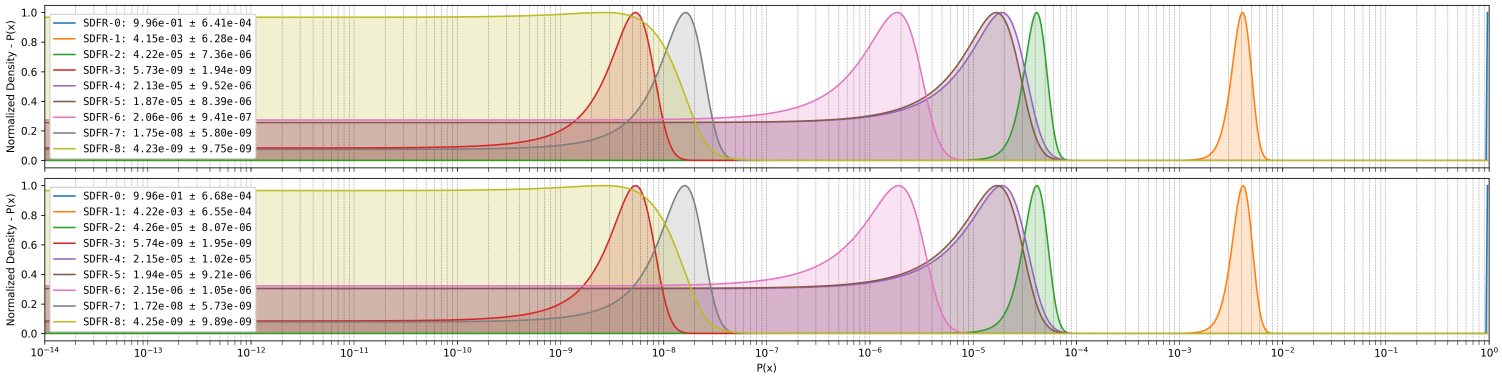


Figure 9.5: Estimated vs Target End-State Frequency Distributions

[3][Estimated - Target], negative values represent underestimates.

[4]The likelihood of no SDFR. Computed by subtracting all end-state frequencies from the total probability.

BIBLIOGRAPHY

Chang, Y. (2018). Experimental Breeder Reactor II (EBR-II) Level 1 Probabilistic Risk Assessment. Technical Report ANL-NSE-2, Argonne National Lab. (ANL), Argonne, IL (United States).

Terrell, G. R. and Scott, D. W. (1992). Variable Kernel Density Estimation. *The Annals of Statistics*, 20(3):1236–1265. Publisher: Institute of Mathematical Statistics.

Zhang, Z. (2018). Improved Adam Optimizer for Deep Neural Networks. In *2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, pages 1–2, Banff, AB, Canada. IEEE.

APPENDICES