

Article

Not peer-reviewed version

---

# Formally and Empirically Verified Methodologies for Scalable Hierarchical Full-Stack Systems

---

[Dong Liu](#)\*

Posted Date: 7 December 2025

doi: 10.20944/preprints202512.0579.v1

Keywords: formal verification; full-stack development; graph-based software engineering; hierarchical data systems; bitmask encoding; communicating sequential processes; linear temporal logic; empirical software engineering; industrial validation



Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a [Creative Commons CC BY 4.0 license](#), which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

# Formally and Empirically Verified Methodologies for Scalable Hierarchical Full-Stack Systems

Dong Liu

IBM Consulting; dliu@us.ibm.com

## Abstract

This paper introduces Primary Breadth-First Development (Pbfd) and Primary Depth-First Development (Pdfd)—formally and empirically verified methodologies for scalable, industrial-grade full-stack software engineering. Both approaches enforce structural and behavioral correctness through graph-theoretic modeling, bridging formal methods and real-world practice. Pbfd and Pdfd model software development as layered directed graphs with unified state machines, verified using Communicating Sequential Processes (CSP) and Linear Temporal Logic (LTL). This guarantees bounded-refinement termination, deadlock freedom, and structural completeness. To manage hierarchical data at scale, we present the Three-Level Encapsulation (TLE)—a novel bitmask-based encoding scheme. TLE operations are verified via CSP failures-divergences refinement, ensuring constant-time updates and compact storage that underpin Pbfd's robust performance. Pbfd demonstrates exceptional industrial viability through eight years of enterprise deployment with zero critical failures, achieving approximately 20× faster development than Salesforce OmniScript, 7–8× faster query performance, and 11.7× storage reduction compared to conventional relational models. These results are established through longitudinal observational studies, quasi-experimental runtime comparisons, and controlled schema-level experiments. Open-source Minimum Viable Product implementations validate key behavioral properties, including bounded refinement and constant-time bitmask operations, under reproducible conditions. All implementations, formal specifications, and non-proprietary datasets are publicly available.

**Keywords:** formal verification; full-stack development; graph-based software engineering; hierarchical data systems; bitmask encoding; communicating sequential processes; linear temporal logic; empirical software engineering; industrial validation

---

## 1. Introduction

### 1.1. Background

Modern Full-Stack Software Development (FSSD) integrates frontend interfaces, backend services, data models, and deployment tooling into cohesive, multi-tier applications. Popular stacks—such as MEAN, MERN, LAMP, and Spring Boot—provide standardized frameworks to support this integration across layers. The demand for full-stack developers has surged due to their ability to manage end-to-end development, a trend consistently reflected in workforce projections and training curricula [1–5].

Professional programs like IBM's Full Stack Developer Certificate now emphasize cloud-native architecture, AI integration, and DevOps practices [3], trends aligned with the broader shift toward scalable, AI-augmented full-stack workflows [1,2].

In practice, FSSD projects typically adopt a backend-first sequence, beginning with data modeling, API design, and business logic before frontend integration. This ordering aligns with Agile principles, which emphasize incremental delivery, stakeholder feedback, and adaptability [6]. Yet despite their flexibility, Agile approaches lack formal mechanisms for dependency modeling or correctness enforcement across layers [7,8]. Stojanovic et al. [9] and Mognon and Stadzisz [10] observe

that the de-emphasis on architectural specification in Agile environments introduces coordination overhead and increases integration risk in complex systems.

Existing literature on FSSD focuses largely on imperative workflows and technology stacks [11,12], with limited use of formal abstractions such as graph traversal, finite automata, or process algebra. The absence of mathematically grounded models hinders scalability, maintainability, and correctness in deeply interdependent systems. Without a unifying theoretical foundation, developers lack principled tools to reason about dependencies, enforce consistency, or optimize control flow across layers [13,14].

This need for rigor is echoed in recent work on orchestration and agent-based coordination, which has reinforced the importance of verifiable models in enterprise-scale environments [15]. These findings highlight the limitations of ad hoc sequencing and motivate the integration of formal semantics into full-stack workflows.

To address this gap, this paper introduces two methodologies—Primary Breadth-First Development (Pbfd) and Primary Depth-First Development (Pdfd)—that reframe FSSD as a formally verifiable workflow problem, expanding on a framework initially proposed in [16,17]. Grounded in graph theory, state machines, process algebra, and Linear Temporal Logic (LTL), Pbfd and Pdfd integrate with Agile practices while adding precision, scalability, and correctness guarantees. Although developed for FSSD, the models generalize to broader classes of hierarchical and dependency-aware systems (see Section 3).

## 1.2. Motivation

Enterprise-scale full-stack systems face escalating complexity, particularly in coordinating frontend, backend, and data layers. In the absence of formally specified workflows, development teams often rely on informal, tool-driven processes that suffice for small applications but break down under scale. This leads to fragmented dependencies, inconsistent state propagation, and growing technical debt—a well-documented challenge that affects both organizational outcomes and developer satisfaction [18,19].

### **Fragmented Dependency and Coordination Bottlenecks**

Disconnected workflows across layers result in duplicated validation logic and unpredictable system behavior. Kretschmer et al. [20] show that inconsistent state propagation arises when changes in one part of a system fail to trigger coordinated updates elsewhere, leading to architectural drift and regression. Tklich et al. [21] attribute frequent integration breakdowns in large-scale continuous engineering environments to the absence of formal dependency modeling. This problem is exemplified by one of our large claims processing platforms, where weak coordination between front-end states and backend APIs triggered cascading failures, requiring weeks of remediation.

### **Technical Debt and Productivity Loss**

Ad hoc implementation choices accumulate as technical debt in the absence of formal validation. Besker et al. [18] report that developers spend over 20% of their time addressing debt-related inefficiencies. Perera et al. [19] provide a systematic mapping of technical debt quantification approaches, revealing gaps in remediation strategies and highlighting the organizational cost of unmanaged debt. Behutiye et al. [22] further show that reduced productivity, system degradation, and increased maintenance cost are among the most significant consequences of technical debt in Agile environments. The same system we developed accumulated over 2,000 unresolved tickets due to ad hoc coordination, delaying milestones and increasing cost.

### **Performance and Scalability Constraints**

Legacy schema designs often prioritize readability or normalization over computational efficiency, leading to significant performance bottlenecks and storage overhead in enterprise-scale full-stack systems. Arulraj et al. [23] demonstrate that hybrid transactional and analytical workloads—common in full-stack architectures—suffer from high latency and poor throughput in traditional row-store schemas, highlighting a fundamental limitation of schema-first design without formal orchestration. In one of our enterprise-scale systems, relational schemas consumed 11.7× more

storage and exhibited  $O(n)$  query latency—causing responsiveness issues during peak operations (see Appendix 22 for a detailed case study).

### Cognitive Overhead and Developer Friction

Repeated transitions between backend schema updates and frontend logic introduce cognitive load and procedural friction. Meyer et al. [24] show that frequent context switching reduces developer productivity and erodes motivation, especially in systems lacking structural coherence. Etikyala and Etikyala [25] demonstrate how orchestrators such as Apache Airflow and Temporal reduce developer burden by managing dependencies and improving fault tolerance. Nevertheless, in the absence of such formalisms at the development workflow level, one of our mission-critical deliveries suffered from repeated context shifts that hindered team velocity and introduced regression defects, despite an experienced team.

To address these systemic limitations in dependency management, technical debt, performance, and cognitive overhead, we developed Primary Breadth-First Development (Pbfd) and Primary Depth-First Development (Pdfd). Building on prior exploratory work [16][17], the models presented in this paper aim to replace ad hoc sequencing and dependency management with principled, automation-ready solutions.

### 1.3. Contributions

This paper introduces a unified formal and practical framework that advances the rigor, scalability, and verifiability of full-stack software development through four primary contributions:

#### 1. Graph-Theoretic Formal Verified Development Framework

We formalize software development as graph traversal over layered directed acyclic graphs, represented with unified state machines and verified using Communicating Sequential Processes (CSP) and Linear Temporal Logic (LTL). Four foundational models (Directed Acyclic Development, Depth-First Development, Breadth-First Development, Cyclic Directed Development) are synthesized into two hybrid methodologies—Primary Breadth-First Development (Pbfd) and Primary Depth-First Development (Pdfd)—with provable properties including termination, deadlock freedom, dependency preservation, and finalization invariance.

#### 2. Three-Level Encapsulation for Hierarchical Data

We introduce Three-Level Encapsulation (TLE), a bitmask-based encoding pattern achieving  $O(1)$  hierarchical operations with 11.7x storage reduction and 85.7x smaller indexes compared to normalized relational schemas. TLE's correctness is established through CSP trace refinement and formal complexity proofs (Theorems A.10.1–A.10.4), enabling predictable, high-performance hierarchical data handling.

#### 3. Machine-Checked Formal Verification

All workflow semantics (DAD, DFD, BFD, CDD, Pbfd, Pdfd) and data operations (TLE: LOAD, READ, WRITE, COMMIT) are machine-checked using FDR4 refinement checker [26,27], establishing deadlock freedom, liveness, bounded refinement, and failures-divergences correctness.

#### 4. Rigorous Industrial Validation

Eight-year enterprise deployment with zero critical failures demonstrates 20x faster development cycles, 7–8x faster queries, and 11.7x storage reduction. Results are established through longitudinal observational studies (Appendix A.20), quasi-experimental runtime comparisons (Appendix A.21), and controlled schema experiments (Appendix A.22). Open-source MVPs [28–30] ensure reproducibility.

**Scholarly Impact:** Existing approaches—including agile feature delivery, low-code platforms, and normalized database schemas—lack formal guarantees for hierarchical systems. Pdfd and Pbfd establish the first graph-theoretic, formally verified foundation for full-stack development, uniting mathematical rigor with demonstrated industrial scalability.

## 2. Related Work

This section situates our work within the broader landscape of software engineering research, focusing on four interrelated research streams: (1) domain-driven and collaborative design, (2) formal development methods such as CSP and LTL, (3) state-based traversal and process-oriented methodologies, and (4) hierarchical data structures with encoded representations. We analyze the limitations of existing paradigms and highlight how Primary Breadth-First Development (PBFD), augmented by Three-Level Encapsulation (TLE), and Primary Depth-First Development (PDFD) integrate and extend these foundations to address a persistent gap in scalable, verifiable full-stack software engineering.

### *2.1. Domain-Driven Design, Collaborative Modeling, and Low-Code Platforms*

Domain-Driven Design (DDD) has significantly influenced software engineering by emphasizing alignment between software architecture and business domains through constructs like bounded contexts and ubiquitous language [31]. Collaborative practices such as EventStorming [32] extend this further by facilitating stakeholder workshops to build shared understanding. However, these approaches remain fundamentally heuristic: they lack executable semantics, formal operational guidance, and mechanisms to ensure consistency or correctness in the resulting models [33]. This often leads to ambiguity and significant challenges in scaling collaborative models to complex, hierarchical enterprise systems.

These limitations have contributed to the growing appeal of Low-Code Development Platforms (LCDPs) (e.g., Mendix, OutSystems, Microsoft Power Apps), which promise to accelerate development through visual modeling and automation [34]. While LCDPs operationalize domain concepts, they often do so with opaque orchestration logic, limited extensibility, and no formal guarantees of correctness [35]. They prioritize speed over verifiability, making them unsuitable for high-assurance systems.

PBFD and PDFD address these limitations by transforming collaborative modeling into a disciplined, verifiable process. Unlike DDD's reliance on emergent consensus or LCDPs' black-box automation, our methodologies provide algorithmically defined traversal strategies that enforce a rigorous sequence of development. For instance, PBFD's level-wise progression ensures domain patterns are finalized in an order that aligns with both stakeholder accessibility and architectural dependencies, while PDFD's depth-first refinement guarantees detailed feature completion before horizontal expansion. By embedding formal guarantees of termination, consistency, and correctness directly into the modeling lifecycle, PBFD and PDFD bridge the critical gap between collaborative design and a transparent, executable implementation.

### *2.2. Formal Methods, LTL, and Model-Driven Engineering*

Formal methods, including algebraic specification [36], Z [37], and Alloy [38], provide rigorous frameworks for specifying and verifying software systems. These approaches offer strong guarantees of soundness and precision but are often criticized for their steep learning curves and limited integration into practical, iterative development workflows [39]. Recent editorial perspectives emphasize that formal methods must be grounded in concrete modeling challenges to achieve broader impact in software and systems engineering [40].

Model-Driven Engineering (MDE) emerged to bridge this gap by elevating models to primary artifacts and automating implementation through model transformations [41]. However, MDE frequently struggles with aligning high-level models to evolving requirements, maintaining practicality in large-scale applications, and overcoming the "modeling bottleneck" [42,43]. Many MDE initiatives have failed to transition from academic research to widespread industrial adoption due to this complexity [44].

PBFD and PDFD integrate formal rigor directly into the development process without requiring practitioners to adopt entirely new specification languages or complex transformation frameworks. Our methodologies incorporate well-founded relations, inductive invariants, and process-algebraic semantics (e.g., CSP [45]) into the traversal logic itself. Additionally, Linear Temporal Logic (LTL) is

a cornerstone of model checking [46], providing a formal language to specify and verify temporal properties such as liveness, safety, and eventual completion. While traditional approaches apply CSP and LTL for system analysis, PBFD and PDFD elevate them to primary methods for governing the development process itself, enabling correctness verification as an inherent property of development workflows.

This integration lowers the adoption barrier by embedding verification into the operational semantics of development, rather than as a separate post-hoc phase. Consequently, PBFD and PDFD extend the MDE vision by offering formal correctness guarantees through pragmatic traversal strategies accessible to developers familiar with modern agile practices.

### 2.3. State-Based and Traversal-Oriented Approaches

State machines [47], Petri nets [48], and process algebras like CSP [45] provide foundational models for reasoning about concurrency, sequencing, and state transitions. These frameworks have profoundly influenced areas like verification, scheduling, and dependency analysis. More recently, traversal-based algorithms (e.g., BFS, DFS) have been incorporated into model checking [46] and dependency-aware development tools [49,50]. However, in existing work, these techniques are typically applied as auxiliary mechanisms for analysis rather than as primary, governing principles for structuring the entire development process. A key limitation is the general absence of built-in support for safe rollback and state recovery, which is crucial for managing iterative refinement in complex projects.

PBFD and PDFD advance this field by elevating traversal strategies to first-class citizens in software development methodology. Unlike traditional uses of BFS/DFS as support functions, our methodologies encode traversal logic directly into the state machine and process algebra that govern development progression. This allows properties like correctness, termination, and rollback safety to be derived directly from the traversal semantics. Beyond correctness, our approach supports rollback safety and iterative refinement—features often missing in traditional state-based models. By doing so, PBFD and PDFD establish a formal and practical bridge between classical state-based reasoning and the complexities of modern full-stack development, enabling a new paradigm of verifiable and scalable software construction.

### 2.4. Encoded Data Structures and Hierarchical Storage

Efficiently managing hierarchical data in relational systems has long been a challenge, typically relying on recursive mechanisms (e.g., Recursive CTEs on adjacency lists) that yield complexity proportional to the depth or size of the hierarchy, incurring substantial  $O(\log n)$  lookup costs and high query overhead [51,52]. This complexity directly contributes to the performance and scalability issues discussed in Section 1.2.

Our work is related to research in high-performance encoded data systems. Database designs like column-stores prioritize encoding and compression techniques to achieve faster query processing and reduced I/O [53–55]. The use of bitwise operations for fast filtering and lookup is a well-established principle in this domain. However, this work focuses on internal query optimization within the DBMS, whereas our Three-Level Encapsulation (TLE) model introduces a declarative bitmask-based schema pattern, a technique that uses bitwise operations to store and manipulate multiple Boolean states within a single integer field, externalizing optimization to the application layer.

In contrast, the TLE model enables  $O(1)$  lookup, update, and traversal while remaining fully compatible with standard relational platforms. By formalizing hierarchical semantics through bitmask encoding rather than traditional approaches like adjacency lists or nested sets, TLE bridges the gap between encoded data representations and application-level correctness—offering a formally verifiable alternative to materialized path or encoded columnar models not addressed in prior hierarchical storage research.

## 2.5. Synthesis and Positioning of PBFD/PDFD

As summarized in Table 1, existing research strands exhibit complementary strengths and limitations. DDD and collaborative modeling excel at fostering shared understanding but lack formal execution. Formal methods offer rigor but suffer from practicality issues. Traversal and state-based approaches provide analytical power but are rarely central to development methodologies. Encoded hierarchical storage approaches optimize performance but do not address formal correctness or integrated workflow management.

**Table 1.** Positioning of PBFD and PDFD Against Existing Research Paradigms.

Research Area	Typical Limitations in Prior Work	PBFD/PDFD Contributions
Domain-Driven Design & Collaborative Modeling [31,32]	Heuristic, non-executable, lacks formal consistency guarantees	Formal semantics with executable workflow rules; ensures verifiable consistency
Formal Methods & LTL [39,40,44,48]	High abstraction, steep learning curve, limited integration with practice	Embedded rigor within accessible workflows; verification of temporal properties (liveness, safety, eventual completion)
State Machines & Traversal Algorithms [47,48]	Used as auxiliary tools, not primary development drivers	Traversal as a first-class development primitive; enables derivation of correctness properties, rollback safety
Model-Driven Engineering [41–44]	Struggles with evolving requirements, scalability, and industrial adoption	Pragmatic adaptability combined with formal foundation; scales to enterprise systems
Low-Code Development Platforms [34,35]	Opaque orchestration, limited extensibility, correctness not guaranteed	Transparent, graph-based orchestration; ensures structural correctness and extensibility
Encoded Data Structures, Columnar Encoding, Bitmap Indexes [52,54,55]	Encoding used internally by DBMS for query acceleration; hierarchical relations still require recursive/nested traversal ( $O(\log n)$ ); no formal semantics for hierarchy or correctness	Declarative bitmask-based hierarchical schema (TLE); $O(1)$ lookup/update/traversal; externalizes encoding at schema design level; preserves explicit hierarchical semantics and enables formal verification (CSP/LTL)

PBFD and PDFD synthesize these domains into a unified framework. Our methodologies leverage graph-based traversal as the core organizing principle for development, ensuring structured progression, formal verifiability, and practical adaptability. This integration addresses a persistent gap in the literature: the lack of a scalable, verifiable methodology that spans from collaborative design to full-stack implementation, while maintaining the rigor demanded by high-assurance systems (see Table 1).

Together, PBFD and PDFD provide a coherent foundation for automating, verifying, and scaling hierarchical full-stack systems, directly addressing the tensions between flexibility, rigor, and practicality that have long challenged the software engineering community.

## 3. Formal Framework and Methodologies

### 3.1. Introduction and Motivation

While Section 1 establishes the practical challenges of full-stack development, this section introduces a unified formal framework for reasoning about and comparing the software development methodologies that address them. Prior research has employed distinct formalisms—Petri nets for state modeling [56], process calculi for communication semantics [57], and temporal logic for property specification [46]—yet these techniques often operate in isolation, lacking systematic integration for cross-paradigm comparative analysis. This fragmentation persists despite calls for formal methods to engage with concrete modeling challenges to achieve lasting impact in software and systems engineering [40].

Our framework addresses this gap by formalizing development workflows as directed dependency graphs with traversal-driven development semantics. A software system under development is represented as a directed graph  $G = (V, E)$ , where vertices  $V$  denote Structural Entities—the units of development, refinement, or verification (e.g., modules, components, features, data schemas, or architectural layers)—and edges  $E \subseteq V \times V$  capture precedence constraints, semantic dependencies, or compositional relationships. Development follows systematic traversal of this graph, implementing either Primary Breadth-First Development (Pbfd) where nodes typically represent pattern instances, or Primary Depth-First Development (PDFD, where nodes may correspond to business data elements—such as countries, states, or schemas—depending on project constraints.

Methodologies are defined as systematic traversal strategies over this graph, governed by state machines that specify control flow, vertex selection rules, and refinement logic. This abstraction enables rigorous reasoning about critical correctness properties, including:

- **Termination** — The development process completes in finite time, visiting all reachable vertices.
- **Deadlock freedom** — No circular dependency chains prevent progress (i.e., the graph is acyclic or cycles are explicitly managed).
- **Dependency satisfaction**— All prerequisite vertices are processed before their dependents, respecting the partial order imposed by  $E$ .
- **Completeness**— All vertices representing required system components are eventually processed and verified.

To ensure rigor and verifiability [58][59], the framework integrates multiple complementary representational layers:

- Structural diagrams visualize workflow architecture and traversal paths.
- State machines define precise operational semantics and control logic.
- Unified transition tables specify deterministic rules linking states, conditions, and actions.
- Pseudocode encodes algorithmic logic for traversal, validation, and refinement.
- Communicating Sequential Processes (CSP) [45] model concurrent execution and inter-process communication, with execution traces serving as the semantic basis for temporal verification.
- Linear Temporal Logic (LTL) [60] specifies global temporal properties—such as liveness, termination, and rollback safety—to be proven over all possible CSP traces.

This hybrid approach supports both local reasoning (via state machines) and global verification (via CSP and LTL). Verification combines automated, instance-based model checking with generalizable correctness proofs derived from transition rules and graph-theoretic invariants. By embedding verification directly into workflow semantics, the framework transforms the design of methodologies such as Pbfd and PDFD from a largely heuristic practice into a formally grounded, reproducible engineering discipline [61].

### 3.2. Formal Notation and Communication Conventions

To support reproducibility and cross-methodology comparison, we standardize notation and communication across all representational layers. Formal definitions for logic symbols, state identifiers, and transition semantics are provided in Appendix A.1.

Each methodology is expressed through the following integrated representations:

- **Pseudocode:** Defined as Procedure [Name](...) with explicit inputs, outputs, and traversal logic.
- **CSP Specifications:** All formal models use synchronous channels to represent communication and control flow. Each specification is validated in FDR 4.2.7, with complete source code and verification scripts available in the corresponding appendices A.2–A.7 and linked GitHub repositories.
- **Unified Transition Tables:** Specify formal transition rules between states, including conditions, actions, and branching logic.

- **Structural Diagrams:** Mermaid-based diagrams visualize workflow structure and state transitions. Source code is provided in the respective appendices.
- **Cross-Representational Mappings:** Appendices A.2–A.7 include full mappings between pseudocode, CSP specifications, and transition tables, ensuring consistency and enable reproducibility across diverse implementation contexts.

The LTL properties defined for each methodology (e.g., termination, liveness, and dependency completeness) are evaluated over the observable traces of their verified CSP processes. For basic methodologies, representative properties are verified; for hybrid methodologies (Pbfd and Pdfd), all key temporal properties are formally proven in Appendix A.8. These properties are derived from each methodology's transition rules and foundational graph algorithms [62,63].

This layered formalism ensures that each methodology is both executable and verifiable across structural, operational, and temporal dimensions, providing a rigorous foundation for comparative reasoning and scalable adoption.

### 3.3. Basic Methodologies

The basic methodologies are rigorous graph-theoretic abstractions, each derived from a fundamental traversal or dependency structure. Rather than prescriptive software engineering practices, they serve as composable formal models that capture distinct workflow strategies:

- **Directed Acyclic Development (DAD):** Enforces strict, non-cyclic dependencies to ensure monotonic progress and traceability. Its full formal specification is provided in Appendix A.2.
- **Depth-First Development (DFD):** Derived from depth-first search (DFS). Prioritizes vertical exploration by completing deep dependency chains before addressing sibling units. Its full formal specification is provided in Appendix A.3.
- **Breadth-First Development (BFD):** Derived from breadth-first search (BFS). Promotes horizontal, level-wise traversal to maintain cross-component consistency at each stage. Its full formal specification is provided in Appendix A.4.
- **Cyclic Directed Development (CDD):** Based on cyclic directed graphs. Incorporates bounded feedback loops within otherwise acyclic workflows, supporting structured reprocessing for iterative refinement. Its full formal specification is provided in Appendix A.5.

Together, these methodologies establish the foundational traversal patterns and dependency constraints upon which hybrid approaches, such as Pdfd and Pbfd, are later defined.

#### 3.3.1. Directed Acyclic Development (DAD)

Directed Acyclic Development (DAD) is a hierarchical, dependency-driven methodology that organizes software construction around a strict-dependency chain. It ensures that a given node can only be processed once all of its direct dependencies ( $D(v)$ ) have been completed and validated. This approach guarantees logical correctness by enforcing that all foundational components are finalized before any dependent features are developed. The core of this methodology is derived from graph-based dependency analysis and a topological sort algorithm, ensuring a valid and predictable order of execution.

##### 1. Definition and Formalization

**Definition:** Directed Acyclic Development (DAD) structures development as a DAG  $G = (V, E)$ , where:

- Nodes represent components (e.g., modules, tasks).
- Edges represent irreversible dependencies ( $((u, v)$  means  $u$  must complete before  $v$ ).
- Acyclicity ensures no cycles exist, preventing deadlocks or circular dependencies.

**Formal Parameters:** The structural elements of DAD are defined in Table 2.

**Table 2.** Formal parameters for the DAD model.

Symbol	Description
--------	-------------

$G$	Directed Acyclic Graph with vertices $V$ and edges $E$
$D(v)$	Direct dependencies of node $v$ : $\{u \mid (u, v) \in E\}$

## 2. Key Characteristics

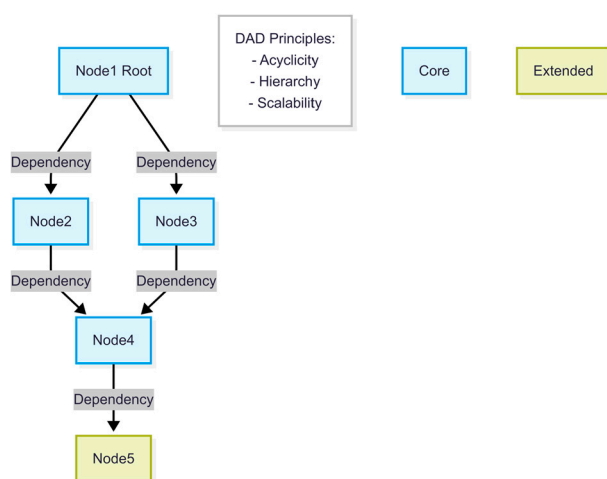
The essential features of DAD are summarized in Table 3.

**Table 3.** Key characteristics of DAD.

Characteristic	Description
Acyclic Enforcement	Ensures that the development dependency graph remains acyclic, preventing circular dependencies and infinite traversal loops
Scalability	Supports incremental addition of nodes and edges, provided that the overall graph preserves its acyclic structure

## 3. Workflow Representation

Figure 1 illustrates a five-node, four-level DAG model with modular parent-child dependencies and scalable extension at the leaf level. The corresponding MermaidJS source code is provided in Appendix A.2.1.



**Figure 1.** Structural workflow of the DAD model, highlighting acyclic dependencies, modular component relationships, and scalable node extension

## 4. State Descriptions

The states of the DAD process model are defined in Table 4.

**Table 4.** State definitions in the DAD process model.

State ID	Phase	Description
$S_0$	Initialization	Load DAG $G$ and validate acyclicity
$S_1$	Node Processing	Process node $v \in V$ (e.g., develop component) and enqueue its children
$S_2$	Dependency Check	Verify the completeness of $v$ 's dependencies, $D(v)$
$S_3$	Graph Extension	Add new nodes or edges to resolve unmet dependencies while preserving acyclicity
$T$	Termination	Final validation and workflow conclusion

## 5. Unified State Transition Table

The formal transition rules, with conditions expressed in first-order logic, are defined in Table

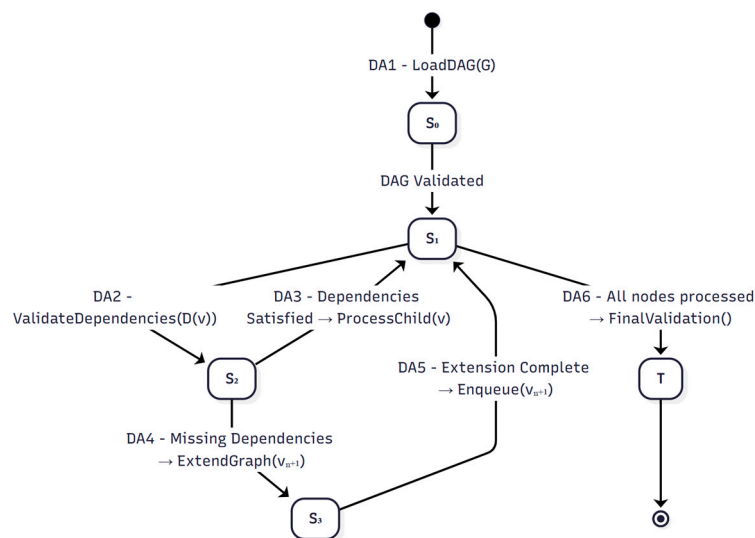
5.

**Table 5.** Formal state transitions and workflow operations in DAD.

Rule ID	Source State	Target State	Condition	Operational Step
A1	S <sub>0</sub>	S <sub>1</sub>	DAG G is loaded and validated as acyclic.	Initialize processing queue with the root node
A2	S <sub>1</sub>	S <sub>2</sub>	A node v is dequeued for processing.	Initiate a check for all dependencies D(v)
A3	S <sub>2</sub>	S <sub>1</sub>	$\forall u \in D(v): \text{processed}(u)$ (All dependencies are resolved).	Enqueue the dependencies of v for processing
A4	S <sub>2</sub>	S <sub>3</sub>	$\exists u \in D(v): \neg \text{processed}(u)$ (An unresolved dependency exists).	Extend the DAG by adding a new node v <sub>n+1</sub> or edge
A5	S <sub>3</sub>	S <sub>1</sub>	DAG extension is complete and acyclicity is preserved.	Enqueue the new node v <sub>n+1</sub> for processing
A6	S <sub>1</sub>	T	$\forall v \in V: \text{processed}(v)$ (All nodes are processed).	Perform final validation and terminate the workflow

## 6. State Machine Diagram

The state machine model for DAD, reflecting transitions DA1–DA6 from Table 5, is shown in Figure 2. The corresponding MermaidJS source code is available in Appendix A.2.2, and the function definitions are in Table A.2.1.



**Figure 2.** State machine model of DAD showing transitions DA1–DA6, corresponding to the development and extension process.

## 7. CSP Formal Verification Results and Guarantees for DAD

This section confirms that the CSPM model (See Appendix 2.4) of the Directed Acyclic Development (DAD) pipeline satisfies the formal properties verified using the FDR model checker. The verification demonstrates that the concrete DAD implementation adheres to behavioral constraints, dependency-first processing, and liveness requirements expressed in the DAD specification.

The results below show that DAD’s dependency-first mechanism—specifically its topological node handling, dependency validation, and ordered graph extension—is formally correct (see Table 6).

**Table 6.** Summary of verification results.

Property	CSP Assertion	FDR Result	Engineering Significance
Core Safety	DAD :[deadlock free [F]]	✓ Passed	Ensures no circular dependencies or blocking states during processing

Property	CSP Assertion	FDR Result	Engineering Significance
Core Liveness	DAD :[divergence free]	✓ Passed	Confirms absence of infinite loops or $\tau$ -cycles in dependency checking
Determinism	DAD :[deterministic [F]]	✓ Passed	Guarantees predictable topological execution order
Dequeue-Process Sequencing	DequeueThenProcess [T= DAD_Core]	✓ Passed	Ensures dequeued nodes are immediately processed (local atomicity, DA2)
Process-Validate Sequencing	ProcessThenValidate [T= DAD_Core]	✓ Passed	Verifies that processing a node triggers dependency validation (DA2 $\rightarrow$ DA3/DA4)
Dependency Completion Logic	DepsProcessedThenGenerate [T= DAD_Core]	✓ Passed	Enforces children generation only after all dependencies completed (DA3)
Child Enqueueing Logic	GenerateThenEnqueue [T= DAD_Core]	✓ Passed	Ensures generated children are properly scheduled for processing (DA3)
Graph Extension Control	MissingDepThenExtend [T= DAD_Core]	✓ Passed	Triggers DAG extension for missing dependencies while maintaining acyclicity (DA4 & DA5)
Final Validation Timing	AllProcessedThenValidate [T= DAD_Core]	✓ Passed	Confirms final validation occurs after all nodes are processed (DA6)
Termination Guarantee	TerminationAllowed [T= DAD_Core]	✓ Passed	Ensures system can always reach a successful or error termination state

### Interpretation & Contributions

#### Dependency-first execution guarantees

Assertions `DequeueThenProcess`, `ProcessThenValidate`, `DepsProcessedThenGenerate`, and `GenerateThenEnqueue` collectively verify DAD's dependency-first processing:

- Nodes are processed immediately after being dequeued (DA2).
- Dependency validation occurs immediately after processing (DA2  $\rightarrow$  DA3/DA4).
- Children are generated only once all dependencies are completed (DA3).
- Generated children are properly enqueued for subsequent processing (DA3).

These behaviors confirm correctness of the S1 (Node Processing) and S2 (Dependency Check) states and DA2–DA3 rules.

#### Graph integrity and termination guarantees

Assertions `MissingDepThenExtend`, `AllProcessedThenValidate`, and `TerminationAllowed` verify:

- Missing dependencies properly trigger DAG extension while preserving acyclicity (DA4 & DA5).
- Final validation occurs only after complete processing (DA6).
- System can always reach a successful or error termination state.

These ensure proper state flow through S2/S3 and eventual workflow completion.

#### Practical significance

Collectively, the results show that DAD:

- Supports correct dependency-first construction of hierarchical software components
- Ensures topological order execution and integrity of the DAG
- Allows incremental graph extension while maintaining acyclic structure
- Avoids deadlocks, livelocks, and nondeterministic processing

#### 8. LTL Properties

The global properties of DAD, expressed in LTL [60] and proven manually from the transition rules, are given in Table 7.

Table 7. LTL properties of DAD ensuring correctness and termination.

Property	Formal Specification	Description
Acyclicity Invariant	$\square(\forall v \in V, \nexists \text{cycle}(v_0, \dots, v_k))$	No cycles are introduced during operation. Rule DA4 triggers graph extension, which is implemented by the <code>ExtendGraph</code> function (Appendix A.2.3) to guarantee acyclicity is preserved.

Property	Formal Specification	Description
Dependency	$\square(\text{processed}(v) \Rightarrow \forall u \in D(v), \text{processed}(u))$	A node is processed only after all its dependencies are processed (Rules DA2, DA3).
Completeness	$\square(\text{dequeue}(v) \Rightarrow \diamond \text{process}(v))$	Every dequeued node is eventually processed (Enabled by DA2-DA5 and the acyclicity invariant).
Liveness of Processing	$\square \forall v \in V, \diamond \text{processed}(v)$	Every node in the graph is eventually processed (Guaranteed by DA6 and the exhaustive traversal semantics).
Fairness (No Starvation)	$\square(\text{start}(\text{DAD}) \Rightarrow \diamond \text{terminate}(\text{DAD}))$	The process eventually terminates for any finite DAG (Rule DA6).

## 9. Advantages

The benefits of applying DAD are summarized in Table 8.

**Table 8.** Advantages of DAD in dependency-aware systems.

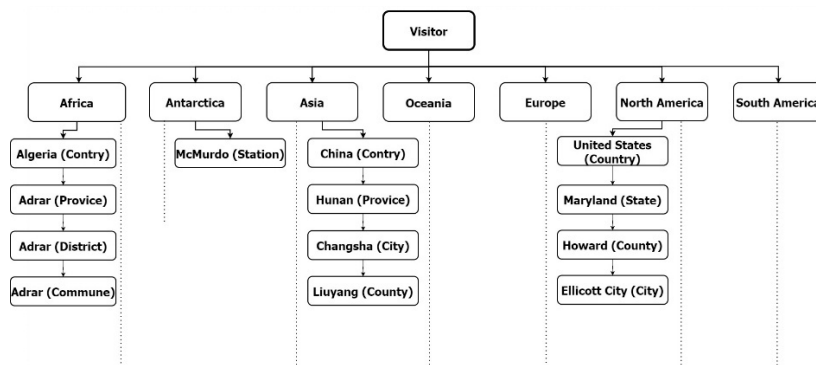
Property	Advantage
Cycle Prevention	Eliminates circular dependencies and development deadlocks
Dependency Isolation	Isolation of branch changes
Incremental Scaling	Supports evolutionary system growth
Impact Analysis	Traceable dependency chains aid debugging and planning

## 10. Example Use Case

A geospatial logging system can be modeled using DAD:

- Root: Continent (e.g., "Africa")
- Hierarchy: Country → Province → Commune
- Termination: Process completes at leaf nodes (communes)
- Dependencies: Unidirectional (e.g., Africa → Algeria → Adrar Province)

Figure 3 illustrates this DAD-based structure, with ellipses indicating unexpanded branches.



**Figure 3.** Geospatial DAD-based model for logging visited places, where each level (continent, country, province, commune) represents a hierarchical dependency enforced by Directed Acyclic Development.

The full formal specification for DAD is provided in Appendix A.2.

### 3.3.2. Depth-First Development (DFD)

Depth-First Development (DFD) organizes software construction around a single, vertical progression. The methodology ensures that a complete feature or branch of the system is fully processed and validated down to its deepest nodes before backtracking to explore new or alternative branches. This approach facilitates early end-to-end integration and provides a holistic view of a single system slice. The operational model of DFD is based on the Depth-First Search (DFS) graph traversal algorithm, which systematically explores, completes, and validates one path before moving on to the next.

#### 1. Definition and Formalization

**Definition:** Depth-First Development (DFD) is a software development methodology that traverses a semantic dependency tree  $Tr$  (e.g., representing domain hierarchies or functional prerequisites) in a depth-first order. Derived from the depth-first search (DFS) algorithm [63], it prioritizes the completion of vertical dependency chains before horizontally exploring sibling branches, using backtracking to ensure exhaustive coverage.

**Formal Parameters:** The structural elements of DFD are defined in Table 9.

**Table 9.** Formal parameters for the DFD model

Symbol	Description
$Tr$	Rooted, finite, acyclic tree structure with nodes $V$ and edges $E$
$D(v)$	Direct dependencies of node $v$ : $\{u \mid (u, v) \in E\}$
$C_i$	The current node being processed in the traversal
$B_j$	A backtrack point (a node on the current path with unvisited siblings)

## 2. Key Characteristics

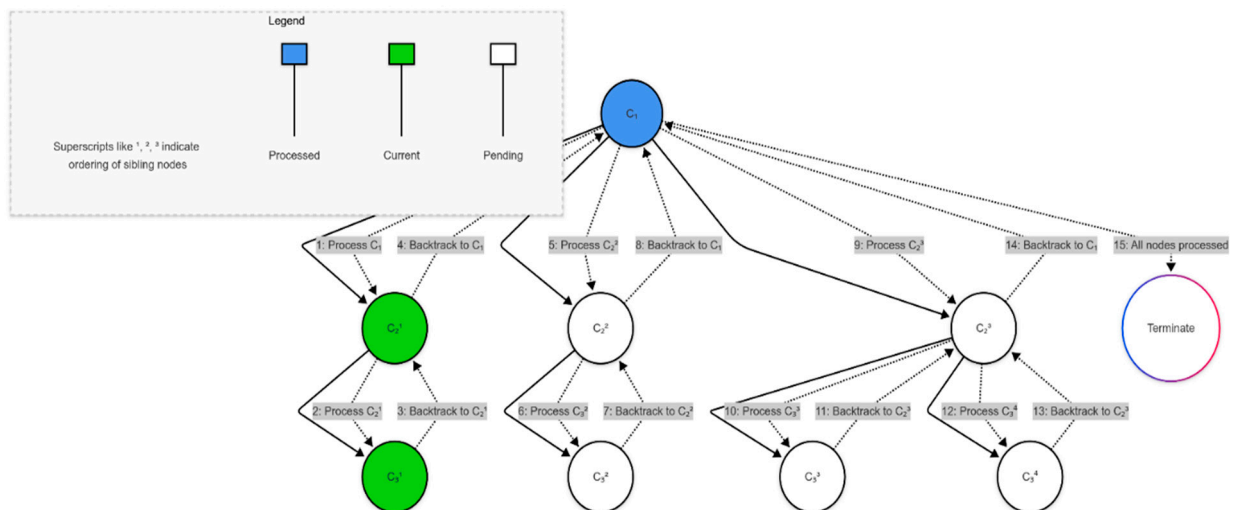
These structural limitations are manifested in Table 10.

**Table 10.** Key characteristics of DFD.

Characteristic	Description
Vertical Progression	Prioritizes traversing a single dependency path to its deepest point before exploring other branches
Exhaustive Traversal	Ensures all nodes and their subtrees are eventually visited and processed by combining vertical progression and backtracking
Backtracking Enablement	Allows returning to a parent node to explore unvisited sibling branches after a path is completed

## 3. Workflow Representation

Figure 4 illustrates the conceptual flow of an eight-node, three-level DFD model, emphasizing depth-first exploration and controlled backtracking. The corresponding MermaidJS source code is provided in Appendix A.3.1.



**Figure 4.** Structural workflow of DFD traversal highlighting depth-first exploration and backtracking

#### 4. State Descriptions

The states of the DFD process model are defined in Table 11.

**Table 11.** State definitions in the DFD process model.

State ID	Phase	Description
$S_0$	Initialization	Load tree $Tr$ and initialize stack with root node
$S_1$	Vertical Processing	Process current node $C_i$ and push its direct dependencies onto the stack
$S_2$	Backtracking	Return to a parent node ( $B_j$ ) after processing a leaf or a completed branch
$S_3$	Validation	Validate the fully explored subtree rooted at the current backtrack point
T	Termination	Final state after all nodes are processed and validated

#### 5. Unified State Transition Table

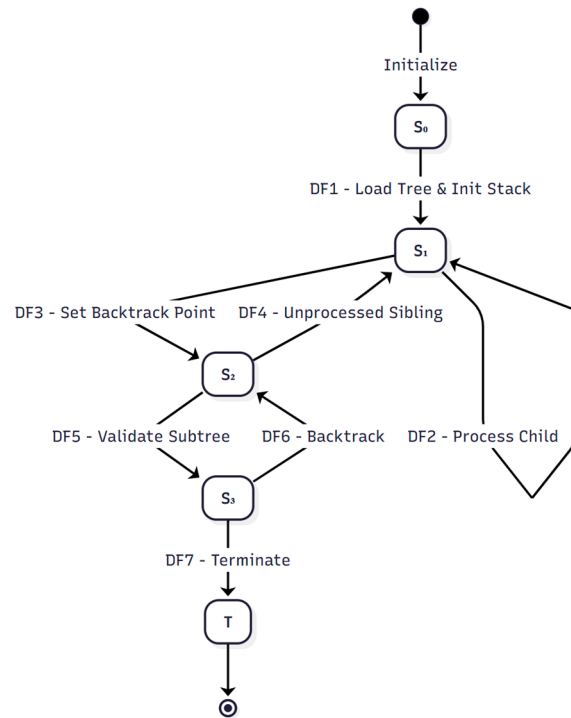
The formal transition rules are defined in Table 12.

**Table 12.** Formal state transitions and workflow operations in DFD.

Rule ID	Source State	Target State	Condition	Operational Step
F1	$S_0$	$S_1$	Tree $Tr$ is loaded and valid.	Initialize stack with root node $C_1$
F2	$S_1$	$S_1$	$C_i$ is a non-leaf node.	Process $C_i$ , then push its direct dependencies $D(C_i)$ onto the stack
F3	$S_1$	$S_2$	$C_i$ is a leaf node.	Process $C_i$ , then set backtrack point $B_j$ to $\text{parent}(C_i)$
F4	$S_2$	$S_1$	$B_j$ has an unprocessed sibling.	Process the next sibling of $B_j$ , push it onto the stack
F5	$S_2$	$S_3$	$B_j$ has no unprocessed siblings.	Initiate validation for the subtree rooted at $B_j$
F6	$S_3$	$S_2$	Stack is not empty.	Continue backtracking to the parent of $B_j$
F7	$S_3$	T	Stack is empty.	Perform final validation and terminate

#### 6. State Machine Diagram

The state machine model for DFD, reflecting transitions DF1–DF7 from Table 12, is shown in Figure 5. The corresponding MermaidJS source code is available in Appendix A.3.2.



**Figure 5.** State machine model of DFD illustrating transitions DF1–DF7.

## 7. CSP Formal Verification Results and Guarantees for DFD

This section confirms that the CSPM model (See Appendix 3.4) of the DFD pipeline satisfies the formal properties verified using the FDR model checker. The verification demonstrates that the concrete DFD implementation adheres to behavioral constraints, stack-based traversal, and liveness requirements expressed in the DFD specification.

The results below show that DFD’s depth-first traversal mechanism—specifically its pre-order node handling, child stack management, and ordered completion—is formally correct (see Table 13).

**Table 13.** Summary of verification results.

Property	CSP Assertion	FDR Result	Engineering Significance
Core Safety	DFD :[deadlock free [F]]	✓ Passed	Ensures no blocking states occur during subtree processing or backtracking
Core Liveness	DFD :[divergence free]	✓ Passed	Confirms absence of $\tau$ -cycles or infinite descent during traversal
Determinism	DFD :[deterministic [F]]	✓ Passed	Guarantees predictable recursion and unambiguous subtree completion
Local Processing Safety	DequeueThenProcess [T=DFD_Core]	✓ Passed	Ensures each dequeued node is immediately processed (DF2 & DF3)
Non-Leaf Descent Logic	NonLeafPushesChildren [T=DFD_Core]	✓ Passed	Enforces DF2: non-leaf nodes must push their children before continuing descent
Leaf/Backtrack Initiation	LeafToBacktrack [T=DFD_Core]	✓ Passed	Enforces DF3: processing a leaf correctly triggers parent-level backtracking
Validation Control Flow	ValidationSequence [T=DFD_Core]	✓ Passed	Ensures validation transitions lead only to backtracking or termination (DF5–DF7)
Termination Reachability	TerminationAllowed [T=DFD_Core]	✓ Passed	Confirms the system can always reach the final successful state

### Interpretation & Contributions

#### Depth-first execution guarantees

Assertions `DequeueThenProcess`, `NonLeafPushesChildren`, and `LeafToBacktrack` formally verify DFD’s pre-order, stack-based traversal:

- Nodes are processed as soon as they are dequeued (DF2–DF3).
  - Non-leaf nodes correctly push their children before descent.
  - Leaf processing reliably initiates the backtracking sequence.
- These behaviors confirm correctness of the S1 (Vertical Processing) state and DF2/DF3 rules.

#### Subtree completion and termination guarantees

Assertions ValidationSequence and TerminationAllowed verify:

- The system cannot stall in backtracking or validation cycles (DF5–DF7).
  - All hierarchical paths are completed before termination.
  - Final termination is guaranteed once traversal is exhausted.
- Together, these ensure proper state flow through S2/S3 and eventual termination.

#### Practical significance

Collectively, the results show that DFD:

- Supports correct recursive descent through hierarchical structures using deterministic stack operations
  - Ensures subtree completion before parent-level progression
  - Avoids deadlocks, livelocks, and nondeterministic backtracking
8. LTL Properties

To ensure correctness and termination of the DFD workflow, we define its global properties using Linear Temporal Logic (LTL), as shown in Table 14.

**Table 14.** LTL properties of DFD ensuring correctness and termination.

Property	Formal Specification	Description
Single Path Completion	$\square \forall P = (C_0, \dots, C^L) \in G: (\text{processed}(C^L) \Rightarrow \forall C_j \in P, \text{processed}(C_j))$	A path is processed completely before moving to siblings (Rules DF2, DF3).
Subtree Validation Completeness	$\square (\text{validated}(B_j) \Rightarrow \forall C_k \in \text{Subtree}(B_j), \text{validated}(C_k))$	A subtree is only validated after all nodes within it are processed (Rules DF5, DF6).
Liveness (No Starvation)	$\forall v \in V, \diamond \text{processed}(v)$	Every node is eventually processed (Rules DF4, DF6).
Termination Guarantee	$\square (\text{start}(\text{DFD}) \Rightarrow \diamond \text{terminate}(\text{DFD}))$	The process eventually terminates for any finite tree (Rule DF7).

9. Advantages

The benefits of applying DFD are summarized in Table 15.

**Table 15.** Advantages of DFD in dependency-aware systems.

Property	Advantage
Early Validation	Foundational logic (e.g., country $\rightarrow$ state $\rightarrow$ city) is validated early.
Modular Testing	Bugs are isolated within narrow vertical paths.
Incremental Scaling	New nodes or branches can be integrated without restructuring validated paths.

The full formal specification for DFD is provided in Appendix A.3.

### 3.3.3. Breadth-First Development (BFD)

Breadth-First Development (BFD) organizes software construction around horizontal progression across architectural levels. The methodology ensures that all nodes at a given depth are processed and validated before advancing to subsequent levels, thereby enforcing layered correctness and predictable advancement. This approach is conceptually derived from the Breadth-First Search (BFS) graph traversal algorithm [63,64].

#### 1. Definition and Formalization

**Definition:** Breadth-First Development (BFD) is a hierarchical methodology that processes all nodes at level  $k$  before descending to level  $k+1$ . This guarantees uniform development across parallel branches of the system and enforces synchronization within each architectural layer, a strategy that aligns with architectural design principles [65].

**Node Semantics:** Each  $N_k$  represents a set of semantic units (e.g., modules, tasks, or components) located at architectural depth  $k$  in the dependency graph.

**Formal Parameters:** The structural elements of BFD are summarized in Table 16. In this model, edges are directional, with  $v \rightarrow u$  indicating that node  $v$  must be completed before node  $u$  can begin. Here,  $D(v)$  refers to the set of direct successors (children) of  $v$ .

**Table 16.** Formal parameters for the BFD model

Symbol	Description
$Q$	Global queue tracking nodes to process
$N_k$	Set of nodes at level $k$
$L$	Maximum depth level of the tree
$D(v)$	Set of direct successors to node $v$ , i.e., $\{u   (v,u) \in E\}$

## 2. Key Characteristics

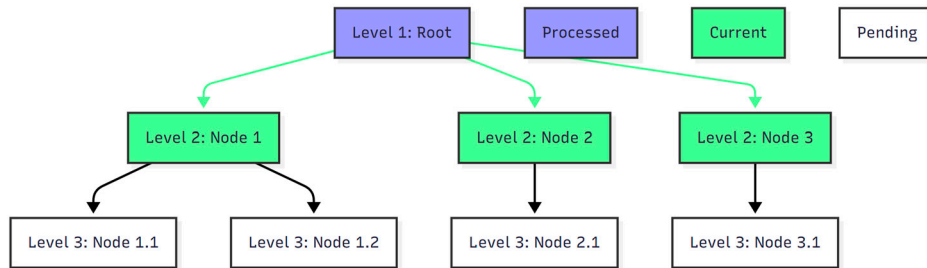
The structural and operational characteristics of BFD are listed in Table 17.

**Table 17.** Key characteristics of BFD.

Characteristic	Description
Horizontal Progression	All nodes at a given level must be processed before the algorithm proceeds to the next level.
Layered Advancement	Advancement from level $k$ to $k+1$ occurs only after all nodes at level $k$ are processed and validated.
Level Synchronization	Maintains level integrity, ensuring consistency across parallel node implementations within the same level.

## 3. Workflow Representation

Figure 6 shows the conceptual flow of an eight-node, three-level BFD model, emphasizing horizontal traversal at each level. The MermaidJS source code is provided in Appendix A.4.1.



**Figure 6.** Structural workflow of BFD illustrating horizontal processing across each level

## 4. State Descriptions

The states of the BFD process model are defined in Table 18.

**Table 18.** State definitions in the BFD process model.

State ID	Phase	Description
$S_0$	Initialization	Load graph and initialize level queues
$S_1$	Level Processing	Process nodes at level $k$
$S_2$	Validation	Validate all nodes at level $k$
$T$	Termination	Final state after all levels are completed

## 5. Unified State Transition Table

The formal transition rules governing the BFD workflow are defined in Table 19.

**Table 19.** Formal state transitions and workflow operations in BFD.

ule ID	Source State	Target State	Condition	Operational Step
F1	$S_0$	$S_1$	Graph loaded.	Initialize queue Q with root
F2	$S_1$	$S_1$	$Q \neq \emptyset \wedge (\exists c \in N_k: \neg \text{processed}(c))$	Process next node in current level
F3	$S_1$	$S_2$	$\forall c \in N_k: \text{processed}(c)$	Validate level k
F4	$S_2$	$S_1$	$k < L$	Advance to level k+1
F5	$S_2$	T	$k = L$	Terminate

## 6. State Machine Diagram

Figure 7 depicts the BFD state machine model, corresponding to the transitions in Table 19. The corresponding MermaidJS source code is available in Appendix A.4.2.

## 7. CSP Formal Verification Results and Guarantees for BFD

This section confirms that the CSPM model (see Appendix A.4.4) of the BFD pipeline satisfies the formal properties verified using the FDR model checker. The verification demonstrates that the concrete BFD implementation adheres to behavioral constraints, liveness requirements, and robustness goals expressed in the BFD specification.

The results below demonstrate that BFD's breadth-first traversal mechanism—particularly its safe handling of level queues, node processing, and level validation—is formally correct (see Table 20).

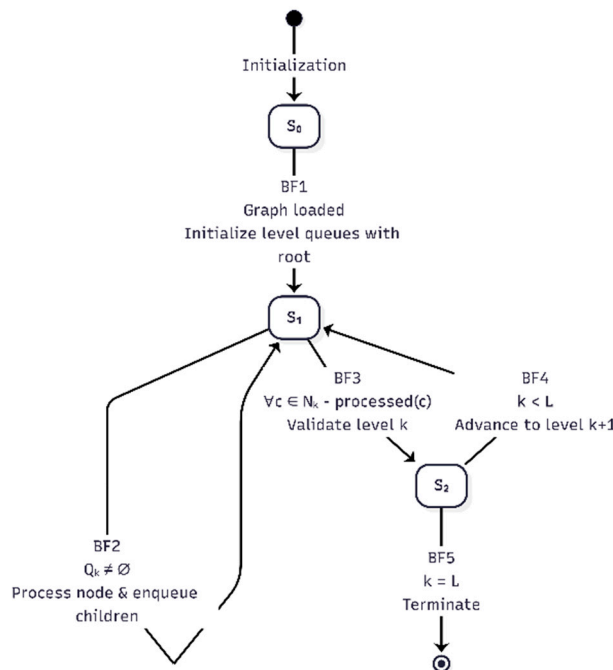


Figure 7. State machine model of BFD showing transitions BF1–BF5.

Table 20. Summary of verification results.

Property	CSP Assertion	FDR Result	Engineering Significance
Core Safety	BFD :[deadlock free [F]]	✓ Passed	Guarantees liveness across node and level processing (no terminal blocking states)
Core Liveness	BFD :[divergence free]	✓ Passed	Confirms absence of livelock and infinite internal loops ( $\tau$ -cycles)

Determinism	BFD :[deterministic [F]]	✓ Passed	Ensures that queue and node processing decisions are uniquely defined for predictable execution
Safety: Dequeue Implies Process	DequeueImpliesProcess [T=BFD_Core]	✓ Passed	Confirms that each dequeued node is immediately processed, preserving workflow correctness (BF2)
Level Validation Before Advancement	ValidateBeforeAdvance [T=BFD_Core]	✓ Passed	Ensures that all nodes at level k are validated before moving to level k+1 (BF3 & BF4)
Post-Validation Behavior	AfterValidation [T=BFD_Core]	✓ Passed	Guarantees that after level validation, the process either advances or terminates (BF4 & BF5), ensuring progress.
Successful Termination	terminate_successfully_actual -> SKIP [T=CanReachTerminate]	✓ Passed	Demonstrates that BFD completes all levels and nodes successfully (BF5)
Termination at End	TerminationAtEnd [T=BFD_Core]	✓ Passed	Confirms that termination occurs only after all processing and validation steps are complete

### Interpretation & Contributions

#### Breadth-first execution guarantees

Assertions DequeueImpliesProcess and ValidateBeforeAdvance formally verify BFD's breadth-first execution semantics:

- Each node in the current level queue is dequeued and processed before moving to the next node.
- Level advancement occurs only after all nodes in the current level are validated.

Together, these ensure that breadth-first traversal respects hierarchical dependencies (BF1–BF4) and prevents premature progression to higher levels.

#### Termination guarantees

Assertions CanReachTerminate and TerminationAtEnd confirm that:

- BFD can always successfully reach the termination state terminate\_successfully\_actual.
- All nodes and levels are fully processed, ensuring liveness and preventing livelock (BF5).

#### Practical significance

Collectively, the results show that BFD:

- Supports safe, level-by-level processing of hierarchical structures
- Guarantees full completion and validation of each level before moving to the next
- Prevents deadlocks or livelocks while ensuring predictable, deterministic behavior
- Ensures internal consistency and milestone integrity through explicit assertions on processing order, validation, and termination

### 8. LTL Properties

To ensure layered correctness and termination, we define the global properties of BFD using Linear Temporal Logic (LTL), as shown in Table 21. Note that processed ( $N_k$ ) is a shorthand for  $\forall c \in N_k: \text{processed}(c)$ .

**Table 21.** LTL properties of BFD ensuring layered correctness and termination.

Property	Formal Specification	Description
Layer Completion	$\square \forall k \leq L: (\text{processed}(N_k) \Rightarrow \neg \exists C_j \in N_k: \neg \text{processed}(C_j))$	All nodes in a level are processed before proceeding (Rules BF2, BF3).
Order Preservation	$\square \forall k < L: (\text{validated}(N_k) \Rightarrow \diamond \text{processed}(N_{k+1}))$	Level k+1 is entered only after all nodes at level k are validated (Rules BF3, BF4).
Termination Guarantee	$\square (\text{start}(\text{BFD}) \Rightarrow \diamond \text{terminate}(\text{BFD}))$	Process reaches completion (Rules BF4, BF5).
Liveness (No Starvation)	$\square \forall v \in V, \diamond \text{processed}(v)$	Every node in the graph is eventually processed.

### 9. Advantages

The benefits of applying BFD are summarized in Table 22.

**Table 22.** Advantages of BFD in dependency-aware systems.

Property	Advantage
----------	-----------

Consistency	Uniform implementation across layers (e.g., all Level 1 nodes completed before Level 2)
Parallelization	Nodes at the same level can be processed concurrently
Predictability	Clear level-based rules simplify debugging (errors are localized to a single level)

The full formal specification for BFD is provided in Appendix A.4.

### 3.3.4. Cyclic Directed Development (CDD)

Cyclic Directed Development (CDD) is a software development methodology that incorporates controlled feedback loops into the development process. Unlike linear or strictly acyclic models, CDD enables revisiting previously developed nodes based on validation or stakeholder feedback. This capability ensures adaptability while imposing formal constraints to avoid infinite regress. CDD formalizes patterns seen in Agile workflows [66], acting as a foundational model for hybrid and iterative development methods. Its behavior is formally specified via a state machine and CSP process algebra (see Appendix A.5).

#### 1. Definition and Formalization

**Definition:** Cyclic Directed Development (CDD) permits iterative refinement of a development graph by enabling controlled feedback loops, subject to formal convergence guarantees.

**Node Semantics:** Each node represents a semantic unit (e.g., module, component, or feature) within a directed graph that may contain cycles, representing iterative refinement points.

**Formal Parameters:** The key parameters of CDD are summarized in Table 23.

**Table 23.** Formal parameters for the CDD model

Symbol	Description
$G = (V, E)$	Directed graph (possibly cyclic) with nodes $V$ and edges $E$ , representing development flow and dependencies
$I_k$	Incremental delivery milestone $k$ , representing a validated subset of the system
$F_k$	Feedback trigger mechanism (e.g., validation failure, stakeholder input) associated with milestone $k$
$R_{\max}$	Maximum allowed refinements per node to ensure convergence

#### 2. Key Characteristics

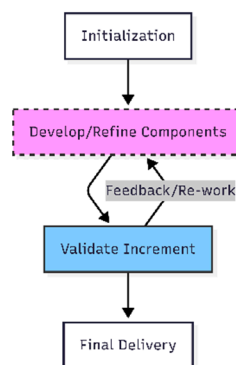
The fundamental characteristics of CDD are outlined in Table 24.

**Table 24.** Key characteristics of CDD supporting iterative and incremental development

Characteristic	Description
Controlled Feedback Loops	Feedback is allowed only when externally triggered and is bounded to prevent infinite iteration.
Incremental Delivery	Components are delivered in validated increments to support continuous integration and testing.

#### 3. Workflow Representation

Figure 8 illustrates the CDD workflow pattern, highlighting the integration of feedback loops within the development cycle to facilitate iterative refinement. The corresponding MermaidJS source code is provided in Appendix A.5.1.



**Figure 8.** CDD workflow model integrating feedback cycles and bounded iteration

#### 4. State Descriptions

The states of the CDD process model are defined in Table 25.

**Table 25.** State definitions in the CDD process model.

State ID	Phase	Description
$S_0$	Initialization	Load graph and initialize dependencies
$S_1$	Node Processing	Develop components under the current milestone
$S_2$	Refinement	Iterate based on validation failure or stakeholder feedback
$S_3$	Validation	Evaluate milestone $I_k$ for completeness and correctness
T	Termination	Final increment successfully validated and delivered

#### 5. Unified State Transition Table

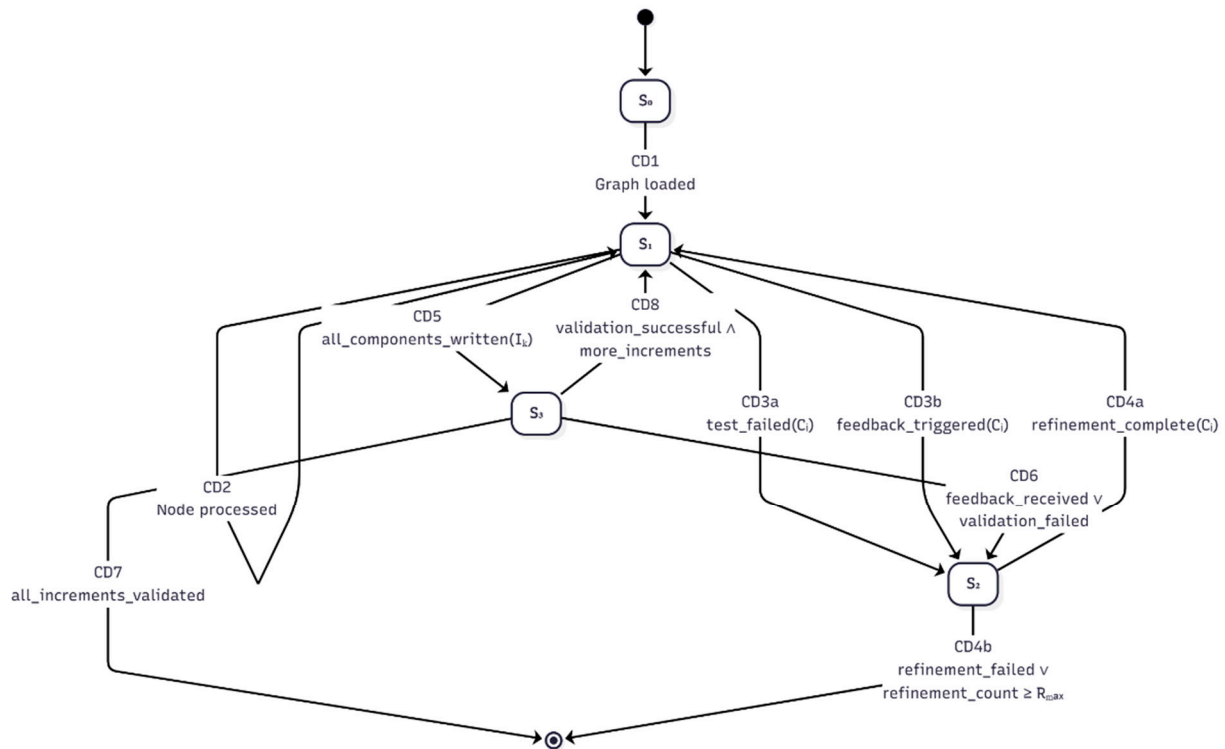
The transitions between different states in the CDD process are captured in Table 26. Function definitions and descriptions can be found in Tables A.1.5 and A.5.1.

**Table 26.** Formal state transitions and workflow operations in CDD.

Rule ID	Source State	Target State	Condition	Operational Step
D1	$S_0$	$S_1$	Graph loaded	Initialize development graph
D2	$S_1$	$S_1$	Node processed	Continue node development
D3a	$S_1$	$S_2$	test_failed( $C_i$ )	Rework after failure
D3b	$S_1$	$S_2$	feedback_triggered( $C_i$ )	Apply bounded feedback loop
D4a	$S_2$	$S_1$	refinement_complete( $C_i$ )	Resume development on node
D4b	$S_2$	T	refinement_failed( $C_i$ ) $\vee$ refinement_count( $C_i$ ) $\geq R_{\max}$	Terminate with error
D5	$S_1$	$S_3$	all_components_written( $I_k$ )	Validate increment
D6	$S_3$	$S_2$	feedback_received( $I_k$ ) $\vee$ validation_failed( $I_k$ )	Revision required
D7	$S_3$	T	all_increments_validated	Finalize delivery
D8	$S_3$	$S_1$	validation_successful( $I_k$ ) $\wedge$ ( $k$ $< L$ )	Advance to milestone $I_{k+1}$

#### 6. State Machine Diagram

The state machine for CDD, illustrating the cyclic transitions for refinement and validation, is depicted in Figure 9. The corresponding MermaidJS source code is available in Appendix A.5.2.



**Figure 9.** State machine diagram of CDD showing cyclic transitions and bounded iteration.

## 7. CSP Formal Verification Results and Refinement Guarantees for CDD

This section confirms that the CSPM model (see Appendix A.5.4) of the CDD pipeline satisfies the formal properties verified using the FDR model checker. The verification demonstrates that the concrete implementation adheres to the behavioral constraints, liveness requirements, and robustness goals expressed in the CDD specification.

The results below demonstrate that CDD’s enhanced architecture—particularly its safe handling of concurrent component dependencies and its guarantee of bounded, terminating refinement cycles—is formally correct (see Table 27).

**Table 27.** Summary of verification results.

Property	CSP Assertion	FDR Result	Engineering Significance
Core Safety	CDD :[deadlock free]	✓ Passed	Guarantees liveness throughout the deployment lifecycle (no terminal blocking states)
Core Liveness	CDD :[divergence free]	✓ Passed	Confirms absence of livelock and infinite internal loops.
Protocol Compliance (Trace)	ProtocolChecker [T=CDDProtocolView]	✓ Passed	Observable deployment traces conform to the defined protocol
Protocol Compliance (Liveness)	CDDProtocolView :[divergence free]	✓ Passed	Livelock-free protocol abstraction
Safety: Initial Guard	NoEarlyTermination [T=CDD]	✓ Passed	Prevents termination before mandatory initialization (load_graph, initialize_dependencies)
Dependency Respect (Contribution N4)	DependencySpec_N4 [T=CDD]	✓ Passed	Proves N4 cannot execute before both N2 and N3 complete
Dependency Respect (Contribution N5)	DependencySpec_N5 [T=CDD]	✓ Passed	Proves N5 cannot execute before N4 completes

Robustness: Bounded Refinement (Deadlock)	CDD_Hostile :[deadlock free]	✓ Passed	Liveness retention and error-termination reachability under adversarial failure
Robustness: Bounded Refinement (Divergence)	CDD_Hostile :[divergence free]	✓ Passed	Shows the system does not livelock under persistent failures; termination is guaranteed
Internal Consistency	ConditionalConsistency [T= STOP]	✓ Passed	Ensures mutually exclusive conditional events do not conflict

### Interpretation & Contributions

#### Dependency-aware safety

Assertions `DependencySpec_N4 [T= CDD]` and `DependencySpec_N5 [T= CDD]` formally verify CDD's concurrency and scheduling guarantees:

- **N4 dependency:** N4 cannot start until both N2 and N3 are complete.
- **N5 dependency:** N5 cannot start until N4 is complete.

Together, these ensure that parallel processing flexibility does not violate critical sequential dependencies.

#### Bounding guarantee under adversary

The hostile-environment check (`CDD_Hostile :[...]`) composes CDD with `HostileEnv_Refinement`, an environment that persistently supplies `validation_failed_actual` and `refinement_failed_actual`. Passing the deadlock and divergence checks confirms the model enforces the refinement bound:

- After  $R_{\max} = 3$  failed refinements, the process issues the error termination event `terminate_with_error_actual` and does not deadlock or livelock.

#### Practical significance

Collectively, the results show that CDD:

- Supports safe, concurrent processing under explicit dependencies
- Provides a provable defense against infinite refinement cycles by bounding retries and enforcing termination in worst-case conditions
- Ensures internal consistency and milestone completion integrity through both guards and dependency assertions

### 8. LTL Properties

The global properties of CDD, defined below using Linear Temporal Logic (LTL), ensure bounded iterative refinement and guarantee termination (see Table 28). Note that `validated(Ik)` implies that all components in  $I_k$  are validated, and `refine(Cj)` denotes the act of reprocessing and revalidating the node  $C_j$ .

**Table 28.** LTL properties of CDD enabling bounded iterative refinement.

Property	Formal Specification	Description
Cycle Integrity	$\square(\text{processed}(C_j) \Rightarrow \diamond \text{refine}(C_j)) \wedge \square(\text{refinement\_count}(C_j) \leq R_{\max})$	Bounded feedback loops are permitted (CD3a/CD3b).
Incremental Soundness	$\square(\diamond \text{finalize}(I_k) \Rightarrow \forall C \in I_k, \text{validated}(C))$	All components in a milestone must be validated before release (CD5, CD7).
Bounded Refinement	$\square \forall v \in V: (\text{refinement\_count}(v) \leq R_{\max})$	The number of refinements for any node is strictly bounded by $R_{\max}$ .
Termination Guarantee	$\square(\text{start}(\text{CDD}) \Rightarrow \diamond T)$	The process eventually reaches successful termination.

### 9. Advantages

The benefits of adopting the CDD methodology are summarized in Table 29.

**Table 29.** Advantages of CDD in dependency-aware systems.

Property	Advantage
Adaptability	Supports bounded iteration in response to validation results or stakeholder feedback
Risk Reduction	Enables early defect detection through milestone-based validation

---

Agile Compliance Aligns with sprint-style incremental delivery while maintaining formal convergence guarantees

---

The full formal specification for CDD is provided in Appendix A.5.

### 3.4. Hybrid Methodologies

Traditional methodologies struggle to reconcile the dual imperatives of modern software development—adaptability and architectural rigor. While Waterfall provides the latter but lacks the former [67], pure Agile emphasizes the former but often lacks the latter at scale [68]. In systems with deep hierarchical dependencies, this dichotomy often leads to coordination bottlenecks and technical debt [69].

These limitations are mirrored in our basic graph-based models. While Depth-First Development (DFD), Breadth-First Development (BFD), and Cyclic Directed Development (CDD) each offer unique structural strengths, they exhibit critical weaknesses in isolation:

- DFD and BFD lack mechanisms for iterative adaptability.
- CDD accommodates iteration but sacrifices hierarchical scaffolding.

To resolve these structural and operational trade-offs, we introduce hybrid methodologies that unify vertical depth, horizontal coordination, and structured refinement. This approach parallels hybrid models in implementation science, which blend clinical effectiveness testing with implementation strategies to accelerate real-world adoption [70]. Similarly, the methodologies proposed here instantiate a dual optimization pattern: simultaneously addressing functional correctness and process efficiency.

We define two primary hybrid strategies:

- **Primary Depth-First Development (PDFD):** An adaptive, vertical progression model optimized for recursive, dependency-heavy systems requiring early risk resolution. It integrates depth-first traversal with bounded parallelism ( $K_i$ ) and cyclic refinement ( $R_{max}$ ) to manage local complexity while securing critical paths.
- **Primary Breadth-First Development (PBFD):** A scalable, horizontal progression model optimized for large-scale systems where architectural stability is paramount. It utilizes pattern-driven modularity (e.g., Three-Level Encapsulation) to establish architectural scaffolds before engaging in selective depth-oriented refinement.

By embedding verification directly into workflow semantics, these hybrids elevate methodology design into a reproducible engineering discipline that balances vertical recursion with horizontal scalability.

#### 3.4.1. Primary Depth-First Development (PDFD)

This section introduces the Primary Depth-First Development (PDFD) methodology, which serves as the foundational control model for hierarchical system development. PDFD formalizes depth-first progression, bounded parallelism, and iterative refinement. It aligns with established software architecture paradigms [65] and supports formal verification through state-space exploration [71].

##### 1. Foundational Concepts and Definitions

###### Definition

PDFD operates over a hierarchical structure of  $L$  levels ( $L \geq 1$ ), where nodes at each level  $i$  are collectively denoted as level( $i$ ). Each node  $n$  maintains a processing state  $P(n) \in \{0, 1, 2\}$ , with  $P(n) = 2$  indicating finalized status.

In the reference implementation, nodes represent discrete business data entities (e.g., continent, country, state), with directed edges capturing hierarchical relationships.

###### Core Paradigms

The methodology synthesizes three core paradigms:

- **Depth-First Development (DFD):** Enables vertical progression through the hierarchy, adapted from graph traversal theory [62] for systematic elaboration of dependencies

- **Breadth-First Development (BFD):** Constrains parallelism via threshold parameter  $K_i$ , enforcing bounded work-in-progress limits that manage cognitive load [66,72,73]
- **Cyclic Directed Development (CDD):** Enables iterative, validation-driven refinement with bounded limit  $R_{max}$ , providing corrective feedback without infinite loops [74]

#### Progression Control

Progression from level  $i$  to level  $i+1$  is permitted only after at least  $K_i$  nodes at level  $i$  reach finalized state ( $P(n) = 2$ ). This completion-driven constraint acts as a synchronization threshold. Unlike traditional Work-In-Progress (WIP) upper bounds,  $K_i$  ensures that a meaningful batch of work is validated before the system permits vertical descent. This prevents premature context switching and maintains flow efficiency.

#### Refinement Mechanism

When validation fails at level  $i$ , the function  $trace\_origin(i)$  identifies the earliest affected level  $J_i$ , triggering refinement across the range  $[J_i, i]$ . This mechanism allows previously finalized nodes to be revisited and reprocessed if validation errors trace to earlier stages.

To ensure termination and architectural consistency, the number of refinements per level is strictly bounded by  $R_{max}$ . While node status may be temporarily reset during active refinement, the process is designed to restore finalized status upon successful re-validation.

#### Finalization Process

Upon reaching terminal or blocked paths, PDFD invokes a structured finalization mechanism. This combines bottom-up subtree verification with top-down passes to complete all unprocessed nodes, ensuring global integrity.

#### Implementation Note

To operationalize bounded parallelism, the PDFD MVP utilizes the Breadth-First-by-Two (BF-by-Two) strategy. This policy sets  $K_i = 2$ , processing sibling nodes in pairs (e.g., one checked feature with one unchecked feature). This balances cognitive load while ensuring systematic feature coverage during hierarchical traversal.

#### Theoretical Grounding

PDFD's state machine formalization follows established workflow verification patterns [75], while its refinement semantics extend formal refinement theory for state-based systems [76]. The approach parallels constraint-graph traversal [72] and incorporates quality control practices from iterative development [74].

#### Formal Parameters

Table 30 lists the minimal and expressive set of control variables.

**Table 30.** Control parameters used in PDFD for regulating progression, refinement, and termination.

Symbol	Description
$K_i$	<b>Progression Threshold:</b> The minimum number of nodes (representing features or components) at level $i$ that must reach a finalized state ( $P(n)=2$ ) before development can progress to level $i+1$ . This threshold acts as a configurable Work-In-Progress (WIP) limit, which can be set statically based on team capacity or adjusted dynamically in real-time based on evolving system constraints and priorities [66]. It enforces structured synchronization points, preventing uncontrolled parallelism and managing complexity
$J_i$	<b>Start of refinement:</b> Earliest level impacted by failures at $i$ , where $J_i = trace\_origin(i)$ .
$L$	Maximum depth (leaf level) of the hierarchical tree.
$R_i$	<b>Refinement range:</b> The number of levels to reprocess, calculated as $R_i = i - J_i + 1$ (bounded by $L$ ).
$R_{max}$	<b>Iteration limit:</b> Maximum refinement attempts per level. Predefined to ensure termination.

**Note:** Parameters  $J_i$  and  $R_i$  define the refinement scope  $[J_i, i]$  of length  $R_i = i - J_i + 1$ , which determines the levels reprocessed during refinement cycles.  $R_i = \min(i - J_i + 1, i)$  rule ensures dependent levels are revisited while respecting hierarchy boundaries. This is conceptually similar to the state-space exploration in model checkers like SPIN, which must also employ efficient traversal and pruning to verify correctness [71], though PDFD

introduces hierarchy-aware rollback semantics not present in SPIN. The PDFD-specific refinement logic itself extends concepts from formal refinement theory applied to state-based systems and process algebras [76].

## 2. Key Characteristics

Table 31 outlines the key conceptual characteristics that guide PDFD's hybrid execution model.

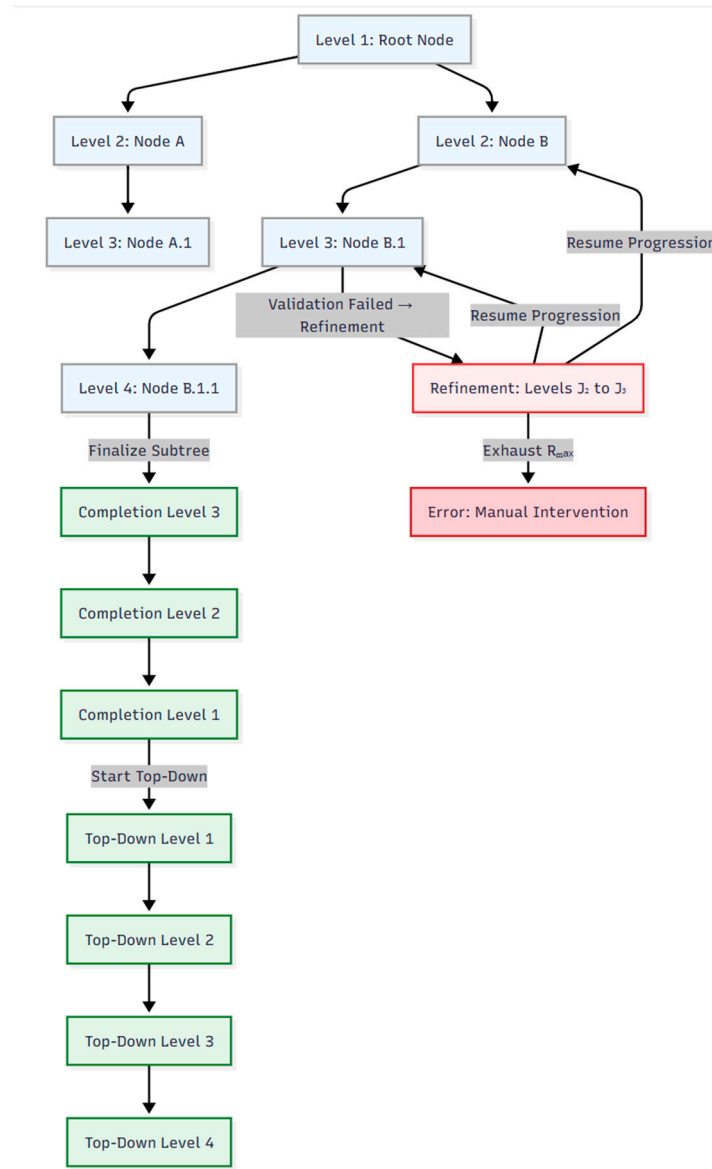
**Table 31.** Conceptual characteristics of PDFD governing its hybrid traversal, concurrency control, and iterative validation.

Characteristic	Description	Theoretical Basis / Inspiration
Vertical Progression	Processing descends level-by-level in a depth-first manner, leveraging DFD principles for focused development paths.	Depth-First Search (Graph Theory), DFD
Controlled Concurrency	Progression to deeper levels depends on meeting a per-level feature threshold $K_i$ of finalized nodes, integrating a controlled breadth-first-like synchronization derived from BFD.	Bounded Parallelism, WIP Limits (Lean/Agile), BFD
Iterative Refinement	The methodology reprocesses and validates levels $[J_i, i]$ to resolve failures, then resumes progression from $J_i$ , directly incorporating CDD's feedback mechanisms.	Iterative Development, Feedback Loops (Spiral Model, Agile) [74], dependency-directed backtracking [77], CDD
Targeted Refinement	Limits rework to $R_{\max}$ attempts per level, balancing precision and scope in iterative cycles.	Bounded Iteration (CDD)
Bottom-Up Finalization	Subtree completion of validated nodes is performed in a bottom-up manner, ensuring localized integrity. It allows backtracking to refinement if unprocessed nodes fail validation and earlier levels have attempts remaining.	Bottom-Up Validation
Top-Down Completion	Finalizes and inherently validates any remaining unprocessed nodes from root to leaves after bottom-up closure, ensuring comprehensive system-wide consistency. Like Bottom-Up Finalization, backtracking to bounded refinement is allowed.	Top-Down Validation
Termination Guarantee	Guarantees process termination once all required conditions are satisfied, considering bounded refinements and finite tree structures.	Formal Methods

## 3. Workflow Representation

Figure 10 illustrates the conceptual flow of a six-node, four-level PDFD model. The diagram visually separates three phases:

- Depth-oriented progression through successive levels
- Iterative refinement cycles via backward jumps
- Completion sweep through bottom-up and top-down finalization



**Figure 10.** Conceptual workflow diagram of PDFD illustrating depth-first progression, iterative refinement, and structured completion phases.

The corresponding source code is available in Appendix A.6.1. Figure A.11.1. of Appendix A.11 is an instance of the PDFD structural workflow in a PDFD MVP.

#### 4. State Descriptions

Table 32 details the various states involved in the PDFD process. Note that in PDFD, validation is an integral part of the Bottom-Up Completion and Top-Down Completion states, reflecting a continuous verification approach rather than a discrete, separate validation phase as in its foundational methodologies. Table A.11.1. of Appendix A.11 is an instance of the PDFD state description in a PDFD MVP.

**Table 32.** State definitions in PDFD capturing progression, refinement, and validation phases.

State ID	Phase	Description
$S_0$	Initialization	Load tree and initialize features
$S_1(i)$	Current Level	Processes selected nodes in level $i$
$S_1(i+1)$	Next Level (Children)	Represents the state of actively processing level $i+1$ , which is derived from children of nodes in level $i$
$S_1(j)$	Refinement Level	Reprocess level $j$ (where $j \leq i$ ) due to failure propagated from a later level $i$

$S_2(i)$	Level Validation	Validate processed nodes in level $i$
$S_2(j)$	Refinement Validation	Validates reprocessed nodes in level $j$ during refinement
$S_3(i)$	Bottom-Up Process	Initiate bottom-up subtree completion for the subtrees rooted at finalized nodes ( $P(n)=2$ ) in level $i$
$S_4(i)$	Completion Level	Finalize unprocessed nodes in level $i$ during the top-down pass
$S_5$	Error	Terminates due to unresolved validation failures after exhausting $R_{\max}$
T	Termination	All nodes processed and finalized

### 5. Unified State Transition Table

Table 33 captures the transitions between different states in the PDFD process. Definitions for predicates and functions used in the table are provided in Tables A.1.5 and A.6.1. Table A.11.2 of Appendix A.11 is an instance of the PDFD state transition table in a PDFD MVP.

**Table 33.** State transition table for PDFD showing rules, triggering conditions, and operational steps.

Rule ID	Source State	Target State	Condition	Operational Step
	$S_0$	$S_1(i)$	$i = 1$	Begin root-level processing
D1	$S_1(i)$	$S_2(i)$	$\text{processing\_complete}(i) \wedge \exists n \in \text{level}(i): \neg \text{validated}(n)$	Validate current level's nodes
D2	$S_2(i)$	$S_1(j)$	$j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max}^{(1)}$	Backtrack to level $j$ and begin refinement if validation fails at level $i$
D2a	$S_2(i)$	$S_1(i+1)$	$\sum_{n \in \text{level}(i)} [P(n)=2] \geq K_i$	Advance to next level after processing batch
D2b	$S_1(j)$	$S_2(j)$	$\text{processing\_complete}(j) \wedge \exists n \in \text{level}(j): \neg \text{validated}(n)$	Validate level $j$ again after refinement ( <i>explicit validation path</i> ) <sup>(2)</sup>
D3	$S_2(j)$	$S_1(j+1)$	$\forall n \in \text{level}(j): \text{validated}(n) \text{ and } j < i$	Resume processing at next level within refinement scope after successful validation
D3a	$S_2(j)$	$S_2(i)$	$\forall n \in \text{level}(j): \text{validated}(n) \text{ and } j=i$	Refinement validation complete; return to original current level for forward pass continuation
D3b	$S_2(j)$	$S_1(j)$	$\exists n \in \text{level}(j): \neg \text{validated}(n) \wedge \text{refinement\_attempts}(j) < R_{\max}$	Retry refinement processing at level $j$
D3c	$S_2(i)$	$S_3(i)$	$i=L \vee \text{level}(i+1) = \emptyset^{(3)}$	Transition to bottom-up process (prematurely or at leaf)
D4	$S_3(i)$	$S_3(i-1)$	$\forall n \in \text{level}(i): \text{validated}(n) \wedge \text{all\_descendants\_validated}(n)$	All unprocessed nodes in the subtree of the processed nodes at level $i$ have been processed and validated; move to level $i-1$
D4a	$S_3(i)$	$S_1(j)$	$\text{processing\_complete}(j) \wedge \exists n \in \text{level}(i): \neg \text{validated}(n) \wedge j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max}$	Backtrack from bottom-up phase to refinement processing
D4b	$S_3(2)$	$S_4(1)$	$i=2 \text{ in bottom up}$	Transition to top-down finalization
D5	$S_4(i)$	$S_4(i+1)$	$\forall n \in \text{level}(i): \text{validated}(n)$	All nodes at level $i$ validated; move to level $i+1$
D6	$S_4(i)$	$S_1(j)$	$\exists n \in \text{level}(i): \neg \text{validated}(n) \wedge j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max}$	Backtrack from completion phase to refinement processing
D6a	$S_4(i)$	$S_5$	$\exists n \in \text{level}(i): \neg \text{validated}(n) \wedge \text{refinement\_attempts}(\text{trace\_origin}(i)) \geq R_{\max}$	Terminate due to unvalidated nodes with no refinement options
D6b				

D7	$S_4(L)$	T	$\forall i \in [1, L], \forall n \in \text{level}(i):$ validated(n)	All nodes validated
D8	$S_1(j)$	$S_5$	$\text{refinement\_attempts}(j)$ $\geq R_{\max}^{(4)}$	Terminate due to refinement cycle exhaustion

**Notes:** (1).  $\text{refinement\_attempts}(j)$  tracks attempts for level  $j$ .  $j = J_i = \text{trace\_origin}(i), R_i = i - j + 1$ . Refinement parameters ( $R_{\max}, J_i, R_i$ ) follow PDFD's level-based logic. (2). Explicit validation again ensures corrections in parallel-processed level are synchronized before progression. Revalidation may include correcting incomplete descendants if needed.  $\text{descendants}(n)$  are implicitly revalidated only if  $P(n)=2$  or analogous. (3). Exceptional finalization if level  $i$  is empty prematurely ( $i < L$ ). Example: If  $\text{level}(i) = \{n_1, n_2\}$  and  $\text{children}(n_1) = \text{children}(n_2) = \emptyset$ , then  $\text{level}(i+1) = \emptyset$ , triggering PD4. This also handles the natural transition to bottom-up when  $i=L$  as  $\text{level}(i+1)$  will be empty. (4). This rule (PD8) triggers termination when a specific level  $j$  (selected for refinement) exhausts its  $R_{\max}$  refinement attempts, specifically after its  $\text{refinement\_attempts}$  counter has been incremented.

## 6. State Machine Diagram

The transitions between different states in the PDFD process, emphasizing the integration of depth-first progression, controlled concurrency, and iterative refinement, are depicted in Figure 11. This state machine diagram illustrates the transitions between different states in the PDFD process. The corresponding source code is available in Appendix A.6.2. Figure A.11.3 of Appendix A.11 is an instance of the PDFD state machine diagram in a PDFD MVP.

**Note:** The state machine diagram uses  $S1\_i$  notation for technical rendering reasons, where  $S1\_i$  corresponds to  $S_1(i)$  in the formal specification. This notation mapping applies to all parameterized states ( $S1\_i \equiv S_1(i), S2\_i \equiv S_2(i)$ , etc.).

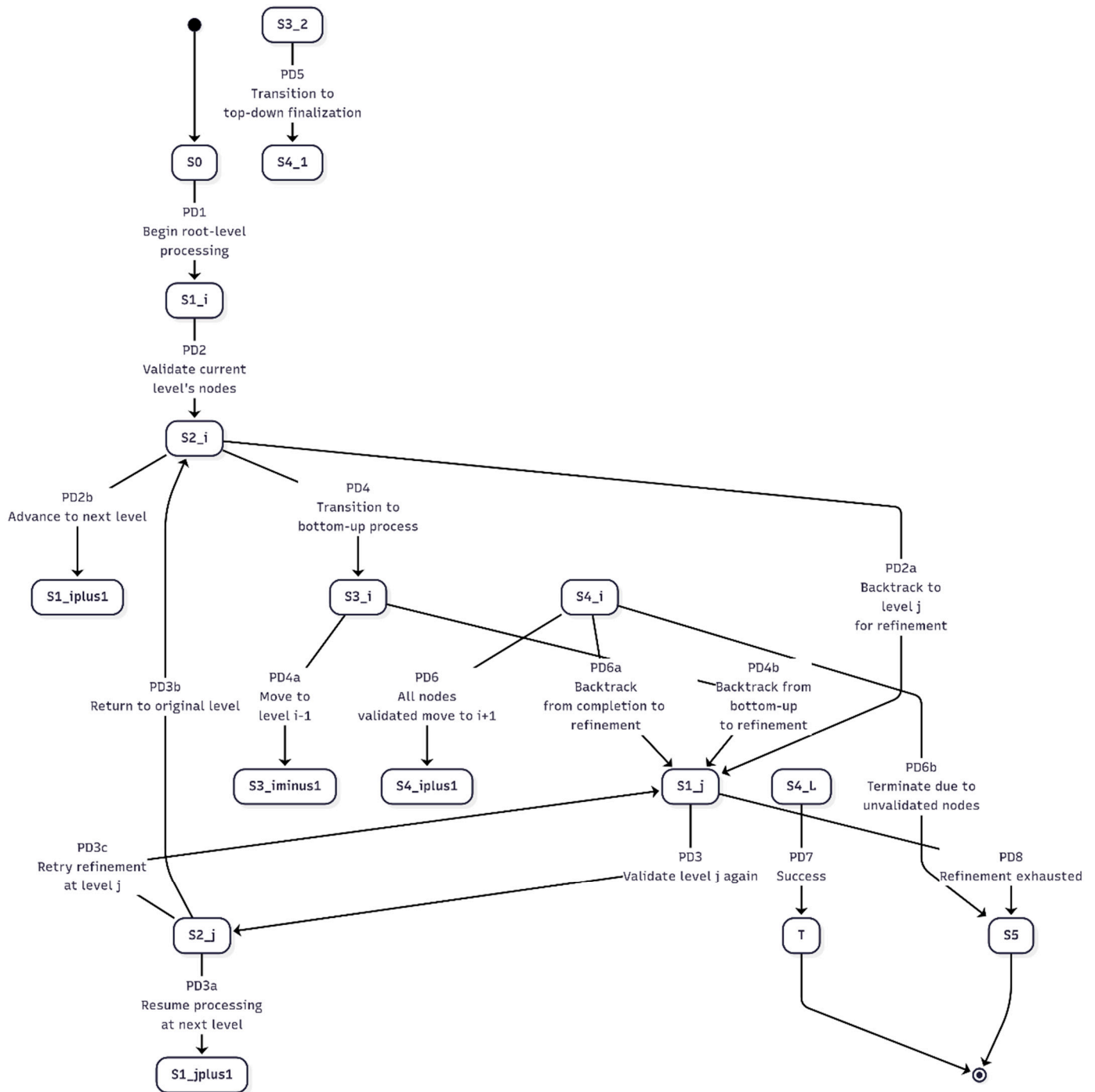
## 7. CSP Formal Verification Results and Refinement Guarantees

This section confirms that the CSPM model of the PDFD methodology (see Appendix A.6.4) satisfies all targeted formal properties verified using the FDR 4.2.7 model checker. The verification demonstrates that the implementation adheres to the structural integrity constraints, safety conditions, and bounding guarantees defined in the PDFD specification.

The results confirm that PDFD's architecture—especially its deterministic processing logic, structured conditional handling, and bounded refinement cycles—meets all correctness objectives (see Table 34).

**Table 34.** Summary of verification results.

Property	CSP Assertion	FDR Result	Engineering Significance
Core Safety	System :[deadlock free], System :[livelock free]	✓ Passed	Ensures progress by eliminating blocking and non-productive cyclic states
Core Liveness	System :[divergence free]	✓ Passed	Confirms absence of infinite internal loops, supporting guaranteed termination
Structural Integrity	System :[deterministic [F]]	✓ Passed	Establishes that behavior is fully determined by environment conditions
Protocol Robustness	SystemProtocolView :[divergence free]	✓ Passed	Confirms that abstracted conditional events do not introduce livelock
General Consistency	ConditionConsistency [T= STOP]	✓ Passed	Validates that the composite conditional environment is non-contradictory
Mutual Exclusivity (5 checks)	ConditionConsistency_Thresh oldMet [T= STOP], etc.	✓ Passed	Confirms that all five core PD decision pairs are logically disjoint and sound



**Figure 11.** State machine of PDFD detailing formal transitions across progression, refinement, and finalization states.

## Interpretation & Contributions

### Deterministic Flow

The assertion System  $:[\text{deterministic [F]}]$  confirms that the next state is strictly determined by the current state and environmental inputs (e.g., threshold conditions, refinement availability). This rules out ambiguous execution paths and ensures predictable refinement behavior.

### Bounding Guarantee via Liveness

The combination of divergence checks and the  $R_{\max}$  constraint proves the process cannot enter unbounded refinement:

- No infinite refinement loops occur.

- On exceeding  $R_{\max}$ , the system transitions to `terminate_error`, enforcing bounded failure handling.

#### Practical significance

These results collectively show that PDFD:

- Ensures termination by always reaching either T (success) or safely halting at S5 (error)
- Provides consistency through six validated conditional soundness checks
- Guarantees predictability via globally deterministic control flow

#### 8. LTL Properties

The LTL properties underpinning PDFD are presented in Table 35.

**Measure Argument:** The termination and liveness proofs rely on a lexicographic measure  $M = (k_1, k_2, k_3, k_4)$  where:

- $k_1$ : Count of unfinalized nodes
- $k_2$ : Remaining refinement attempts across levels
- $k_3$ : Phase ordinal ( $S_0 = 4, S_1 = 3, S_2 = 2, S_3 = 1, S_4 = 0$ )
- $k_4$ : Intra-phase progress measure

Every non-terminal transition decreases  $M$  in lexicographic order.

**Table 35.** LTL properties of PDFD ensuring soundness, termination, completeness, and structural consistency.

Property	Formal Specification	Description & Justification
Total Correctness	$\Box(\text{start} \Rightarrow ((T \wedge \text{Structural Invariants}) \vee S_5))$	<b>Theorem A.8.8:</b> The methodology always terminates (T or $S_5$ ) and, upon successful termination (T), guarantees that all nodes are validated and all structural invariants are satisfied.
Termination	$\Box(\text{start} \Rightarrow \Diamond(T \vee S_5))$	<b>Lemma A.8.4:</b> The algorithm always terminates, either in success (all nodes finalized, T) or bounded failure (refinement exhausted, $S_5$ ).
Bounded Refinement	$\forall k \in [1, L], \Box(\text{refinement\_attempts}(k) \leq R_{\max})$	<b>Lemma A.8.2:</b> The number of refinement attempts for any level $k$ is strictly bounded by the constant $R_{\max}$ .
Refinement Convergence	$\Box \forall j: (\text{refining}(j) \Rightarrow \Diamond(\neg \text{refining}(j) \vee \text{refinement\_attempts}(j) = R_{\max}))$	<b>Lemmas A.8.2. &amp; A.8.3:</b> Each refinement cycle either resolves the issue and exits refinement, or exhausts its attempt bound, ensuring refinement doesn't stall indefinitely within the bounded attempts.
Finalization Monotonicity	$\Box((\bigcirc k_1 \leq k_1) \vee (\bigcirc k_1 > k_1 \wedge \bigcirc k_2 < k_2))$	<b>Lemma A.8.3:</b> The global count of unfinalized nodes ( $k_1$ ) is non-increasing. A strict increase in $k_1$ (reset) is strictly compensated by a decrease in $k_2$ (remaining refinement attempts), ensuring lexicographic progress.
Finalization Permanence	$\forall n \in G: \Box((P(n)=2 \wedge \neg \exists j: (\text{refining}(j) \wedge n \in \text{affected\_nodes}(j))) \Rightarrow \bigcirc(P(n)=2))$	<b>Corollary A.8.3.1:</b> A finalized node's status is permanent except when an active, guarded refinement backtrack resets it; such resets are bounded and compensated by a strict decrease in $k_2$ (remaining refinement attempts).
Finalization Invariant	$\forall n: \Box(P(n)=2 \Rightarrow \forall d \in \text{descendants}(n) \cap \text{processed\_subtree}(n), P(d)=2)$	<b>Lemma A.8.5:</b> A node is not finalized unless all nodes in its processed subtree are also finalized. Enforced by guards in PD4a, PD6, PD7.
Refinement Locality	$\Box \forall i, j: ((\text{state} = S_2(i) \wedge \bigcirc \text{state} = S_1(j)) \vee (\text{state} = S_3(i) \wedge \bigcirc \text{state} = S_1(j)) \vee (\text{state} = S_4(i) \wedge \bigcirc \text{state} = S_1(j))) \Rightarrow (j \leq i \wedge j = \text{trace\_origin}(i))$	<b>Lemma A.8.5:</b> All backtracking transitions target a valid anchor level $j$ within the current progression frontier, and $j$ is the origin of the current trace.
Progression Condition	$\Box \forall i: ((S_2(i) \wedge (\sum \{n \in \text{level}(i)\} [P(n)=2] \geq K_i)) \Rightarrow \bigcirc(S_1(i+1)))$	<b>Rule PD2b (Table A.8.2):</b> The system advances to the next level's Initialization phase ( $S_1$ ) when enough nodes ( $K_i$ ) at the current level are finalized.

Guarded Progression Invariant	$\square((\text{state} = S_2(i) \wedge \sum_{n \in \text{level}(i)} \{ \text{eligible}(n) \geq K_i \} \Rightarrow \bigcirc(S_1(i+1) \wedge \text{selected\_subtree} \subseteq \text{trace}(i)))$	<b>Rule PD2b (Table A.8.2):</b> Progression to the next level is guarded by eligibility criteria and trace constraints, ensuring bounded advancement.
Bottom-Up Finalization	$\square \forall i: ((S_2(i) \wedge (i = L \vee \text{level}(i+1) = \emptyset)) \Rightarrow \bigcirc(S_3(i)))$	<b>Rule PD4 (Table A.8.2):</b> Finalization initiation is triggered upon reaching a leaf node or an empty level, ensuring the transition from progression to completion.
Top-Down Finalization	$\square \forall i: ((S_4(i) \wedge (\forall n \in \text{level}(i): P(n)=2)) \Rightarrow \bigcirc S_4(i+1) \vee \bigcirc T \vee \bigcirc S_5)$	<b>Rule PD6 (Table A.8.2):</b> The top-down completion phase progresses to the next level once the current level is fully finalized (or the process terminates).
Global Consistency	$\square(T \Rightarrow (\forall n \in G, P(n)=2))$	<b>Rule PD7 (Table A.8.2):</b> Successful termination implies all nodes in the graph are finalized.
Vertical Closure (Forward Guarantee)	$\square((P(n)=2 \wedge \text{children}(n) \neq \emptyset) \Rightarrow \diamond \forall d \in \text{children}(n): P(d) \in \{1,2\} \vee T \vee S_5)$	<b>Implied by PD4/PD6 (Table A.8.2):</b> If a parent is finalized, its children are guaranteed to be addressed in the process flow (either by forward progression or completion), barring system termination.
Soundness	$T \Rightarrow (\forall n \in G: \text{consistent}(n) \wedge \text{dependencies\_satisfied}(n))$	<b>Theorem A.8.8:</b> Successful termination implies all nodes are internally consistent and satisfy their architectural dependencies, ensuring the final system is semantically correct.
Unified Progress	$\square((\neg T \wedge \neg S_5) \Rightarrow \exists \text{enabled\_transition})$	<b>Lemma A.8.7:</b> From any non-terminal state, at least one transition rule is enabled, ensuring the system never deadlocks.
Liveness (Progress)	$\square((\neg T \wedge \neg S_5) \Rightarrow \bigcirc(M <_{\text{lex}} M))$	<b>Lemma A.8.7:</b> From any non-terminal state, an enabled transition exists, which decreases the lexicographic measure $M$ , guaranteeing forward movement and preventing deadlock.
Well-Foundedness	$M = (k_1, k_2, k_3, k_4)$ where $k_1 \in [0,  V ]$ , $k_2 \in [0, L \cdot R_{\max}]$ , $k_3 \in \{0,1,2,3,4\}$ , $k_4 \in [0, \text{max\_batch\_size}]$	<b>Lemma A.8.4:</b> Each component of the lexicographic measure $M$ is bounded and ranges over a well-ordered set, ensuring no infinite decreasing sequences exist.

## 9. Advantages

The benefits of adopting the PDFD methodology are summarized in Table 36.

**Table 36.** Summary of design advantages offered by PDFD across validation, scalability, and completeness dimensions.

Property	Advantage
Early Validation	Depth-first traversal enables early detection of structural and behavioral issues in the hierarchy.
Controlled Concurrency	Parameter $K_i$ regulates concurrent workload distribution in real time.
Targeted Refinement	Parameter $R_{\max}$ bounds rework iterations per level, balancing precision and efficiency.
Completeness Guarantee	Combined bottom-up and top-down closure ensures that all components are fully processed.
Scalable Design	Dynamic parameters adapt traversal behavior to diverse tree structures.
Hierarchical Closure	Systematic traversal guarantees complete coverage from root to leaves.

The full formal specification for PDFD is provided in Appendix A.6.

### 3.4.2. Primary Breadth-First Development (Pbfd)

This section presents Primary Breadth-First Development (Pbfd), a hybrid methodology for complex hierarchical system development. Pbfd combines pattern-driven breadth-first progression with selective depth-first traversal and robust cyclic refinement mechanics. It incorporates certain foundational concepts established in PDFD (Section 3.4.1) while introducing pattern-based modularity for managing architectural complexity.

#### 1. Definition and Pattern Encapsulation

PBFD operates over a hierarchical structure of  $L$  levels ( $L \geq 1$ ), where nodes at each level  $i$  are collectively denoted as level( $i$ ) [58]. Each node  $n$  maintains a processing state  $P(n) \in \{0, 1, 2\}$ , with  $P(n) = 2$  indicating finalized status.

To operationalize pattern-based modularity, PBFD employs hierarchical encapsulation mechanisms, realized in this study as Three-Level Encapsulation (TLE). TLE is a structural schema that encapsulates exactly three hierarchical levels into a single processing unit.

Each node is a constituent component of a TLE pattern instance, and can serve as the anchor for a subsequent instance. This anchoring creates a continuous chain of dependency, allowing the methodology to enforce local consistency while traversing the global hierarchy.

**Example:** Consider a geographic hierarchy (Continent  $\rightarrow$  Country  $\rightarrow$  State  $\rightarrow$  County  $\rightarrow$  City):

- **Instance 1 (Continent-anchored):** Continent  $\rightarrow$  Country  $\rightarrow$  State
- **Instance 2 (Country-anchored):** Country  $\rightarrow$  State  $\rightarrow$  County
- **Instance 3 (State-anchored):** State  $\rightarrow$  County  $\rightarrow$  City

#### Core Paradigms

The methodology synthesizes three core paradigms:

- **Breadth-First Development (BFD):** PBFD's primary progression is breadth-first, facilitating sequential, level-by-level processing of the layered directed acyclic graph. Nodes within the same level share structural characteristics defined by discrete structural signatures (e.g., bitmask encoding), enabling efficient pattern-driven initial development and horizontal batch processing. Because BFD processes nodes level-by-level, a single pattern implementation is reused across all nodes sharing the same signature (e.g., bitmask-defined level sets, shared data schemas, or common processing logic).
- **Depth-First Development (DFD):** DFD complements the breadth-first structure by enabling selective vertical traversal. Within TLE structure, DFD is operationalized through selective promotion of parent nodes to grandparent positions. This allows the system to refine specific hierarchical paths (critical subtrees) without processing all branches uniformly.
- **Cyclic Directed Development (CDD):** CDD governs validation-driven refinement by introducing bounded iterative cycles. This permits systematic re-entry into development based on feedback, continuing until predefined resolution criteria or refinement limits are met [78].

#### Pattern-Driven Progression

- **Selection and Advancement:** At level  $i$ , specific patterns (denoted  $\text{Pattern}_i$ , a subset of nodes at level  $i$ ; see Table A.1.4) are selected and processed based on dependency structure or criticality [65,79]. Advancement to level  $i+1$  is permitted only when all nodes within  $\text{Pattern}_i$  reach finalized status ( $P(n) = 2$ ), enabling the derivation of  $\text{Pattern}_{i+1}$  from the children of those finalized nodes.
- **Selective Refinement:** Pattern progression to  $\text{Pattern}_{i+1}$  is governed by selective advancement via function  $\text{select\_critical\_children}(\text{Pattern}_i)$  (Table A.1.5). This mechanism concentrates refinement along critical paths while preserving completeness guarantees through the  $S_4$  completion phase (Table 39). This modularity follows principles of minimizing coupling and maximizing cohesion [80].
- **Implementation Optimization:** To handle the complexity of overlapping patterns, the PBFD MVP implementation utilizes TLE with bitmask encoding (Section 4), which support  $O(1)$  updates and minimize data-access coupling [53,55].

#### Refinement Mechanism

- **Validation-driven refinement:** Upon validation fails at level  $i$ , the function  $\text{trace\_origin}(i)$  identifies the earliest affected level  $J_i$ . This triggers reprocessing across the range  $[J_i, i]$ . This backtracking capability allows previously finalized nodes to be revisited when validation errors originate from earlier levels, ensuring systemic coherence and architectural integrity across the hierarchy [82].
- **Bounded refinement:** CDD enforces the per-level limit  $R_{\max}$  and iteration tracking indices—adhere to the formal model introduced in PDFD (Section 3.4.1), enforcing termination consistent

with lifecycle principles [83]. The PBFD MVP implementation demonstrates this with  $R_{\max} = 50$  (Appendix A.14).

#### Completion Phase

- **Top-down finalization:** Upon reaching the leaf level, PBFD initiates a top-down completion phase [81]. Remaining unprocessed patterns are finalized sequentially from level 1 through level L. This ensures comprehensive system completion while preserving the architectural consistency established during pattern-driven progression.

#### Theoretical Grounding

PBFD's pattern-driven approach aligns with established software architecture paradigms [65] and extends the formal control mechanisms of PDFD to support modular, incremental development of complex hierarchical systems. The selective depth-first elaboration balances breadth-first architectural visibility with targeted vertical refinement, optimizing for both cognitive manageability and architectural coherence.

#### Formal Parameters

The key parameters of PBFD are summarized in Table 37.

**Table 37.** Control parameters used in PBFD: Key parameters guiding progression, validation, and refinement across hierarchical levels.

Symbol	Description
L	Maximum depth (leaf level) of the hierarchical tree
$J_i$	<b>Start of refinement:</b> Earliest level impacted by failures in $\text{Pattern}_i$ (at level i), computed via $\text{trace\_origin}(i)$ (see PDFD, Section 3.4.2)
$R_i$	<b>Refinement range:</b> Number of levels ( $R_i = i - J_i + 1$ ) to reprocess, spanning patterns from level $J_i$ to i, bounded by L
$R_{\max}$	<b>Iteration limit:</b> Maximum refinement attempts per level ( $\text{Pattern}_i$ ), matching PDFD's per-level refinement cap (Section 3.4.2)
$\text{Pattern}_i$	<b>A formal model:</b> A cohesive, feature/function-grouped subset of nodes (data, logic, UI artifacts) at hierarchical level i, encapsulating a distinct unit of business logic [79,80,84]; $\text{Pattern}_{i+1}$ is a selected subset of $\cup_{n \in \text{Pattern}_i} \text{children}(n)$ , chosen based on critical path, dependencies, and development priorities
$r_j$	Current refinement attempt index for $\text{Pattern}_j$

**Note:**  $R_{\max}$  specifies the maximum number of collective attempts allowed for all patterns within a given level, rather than for individual patterns.

## 2. Key Characteristics

PBFD's structural and functional behavior is summarized in Table 38.

**Table 38.** Key Characteristics of PBFD: Summary of pattern-driven traversal, depth transition, and completion behavior.

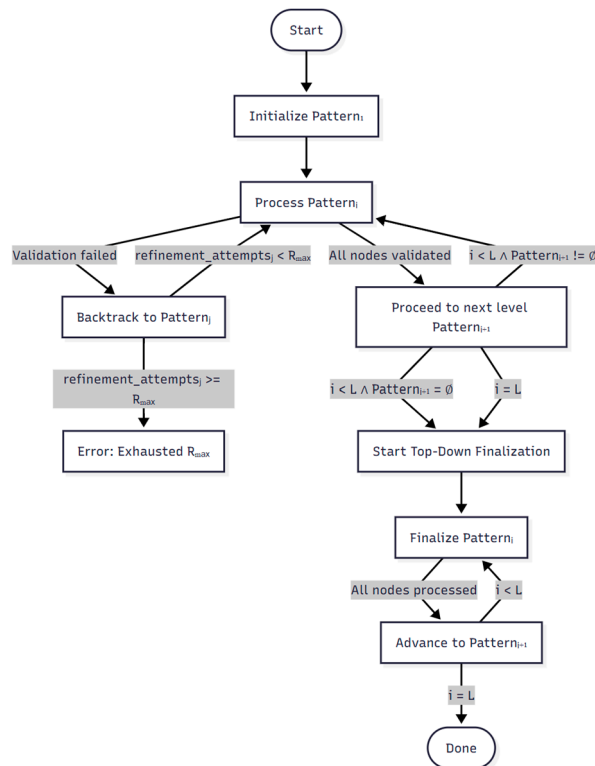
Characteristic	Description	Theoretical Basis / Inspiration
Pattern-Driven Traversal	Nodes are grouped into patterns and processed level-by-level, with selective advancement to critical child nodes at each step, and may be optimized for $O(1)$ data-access efficiency using techniques like bitmask encoding.	Breadth-First Search (BFD), Architectural Patterns [79,84,85]
Depth Transition	Children of current pattern nodes are promoted as the next pattern ( $\text{Pattern}_{i+1}$ )	Dependency Tracing [65], DFD Principles
Pattern-Based Refinement	On validation failure, PBFD rewinds to prior levels ( $\text{Pattern}_i$ ) to correct impacted nodes. Example: Reprocessing level 1's "data access" pattern due to a failure in level 2's "security" pattern.	Iterative Development, Feedback Loops (CDD) [78], Software Evolution [86]

Parallelism	Nodes within a pattern are processed concurrently. Advancement to the next state occurs only after all processed nodes within the pattern are successfully validated.	Scalable Parallelism, Horizontal Concurrency
Top-Down Finalization	Finalization iterates from the root (level 1) to the leaf level (L), ensuring all dependencies are resolved and complete processing from root to leaves is achieved.	Top-Down Validation, Structured Design [81]
Termination Guarantee	Process termination is guaranteed once all required conditions are satisfied, considering bounded refinements and finite tree structures.	Formal Methods, Well-Founded Measures [61], Model Checking (CSP/SPIN) [45,71,87]

Patterns such as “security” or “logging” may be compactly represented as bitmasks, enabling parallel resolution or traversal via techniques like Three-Level Encapsulation (TLE) [53,55] (see Section 4).

### 3. Workflow Representation

Figure 12 illustrates the full PBFDF workflow, including horizontal pattern processing, depth-based transitions, validation-triggered refinement loops, and the finalization phase. Figure A.14.1. in Appendix 14 is an example of data driven PBFDF workflow where the development node is the row data. The corresponding source code is available in Appendix A.7.1.



**Figure 12.** PBFDF Structural Workflow: Hierarchical traversal, refinement feedback loops, and finalization path.

**Description:** The diagram presents a tree-like hierarchy of nodes partitioned into level-wise patterns. Each  $Pattern_i$  is processed horizontally before deriving the next level’s pattern from the children. Nodes failing validation generate feedback that rewinds execution to a prior  $Pattern_j$ , triggering refinement. After reaching the leaf level, unprocessed nodes across all levels are finalized via top-down traversal.

### 4. State Descriptions

PBFDF’s behavior is formally captured via a set of states, described in Table 39. Table A.14.1 of Appendix A.14 is an instance of the PBFDF state description in a PBFDF MVP.

**Table 39.** State definitions for PBFDF: Operational phases during pattern processing, validation, refinement, and completion.

State ID	Phase	Description
$S_0$	Initialization	Load tree and initialize patterns
$S_1(i)$	Current Pattern	Processes nodes in $Pattern_i$
$S_1(i+1)$	Next Pattern (Children)	Represents the state of actively processing $Pattern_{i+1}$ , which is derived from children of $Pattern_i$
$S_1(j)$	Refinement Level	Reprocess $Pattern_j$ due to failure propagated from a later level
$S_2(i)$	Pattern Validation	Validate processed nodes in $Pattern_i$
$S_2(j)$	Refinement Validation	Validate reprocessed nodes in $Pattern_j$ during refinement
$S_3(i)$	Depth-Oriented Resolution	Depth-Oriented Resolution (Normal Context) - Load required data and resolve node implementation before descending
$S_3(j)$	Refinement Depth-Oriented Resolution	Refinement Depth Resolution - Load required data and resolve node implementation for $Pattern_j$ during refinement before descending or returning to the original context
$S_4(i)$	Completion Level	Finalize unprocessed nodes in $Pattern_i$ during the top-down pass
$S_5$	Error	Terminates due to unresolved validation failures after exhausting $R_{max}$
T	Termination	All patterns processed and finalized

### 5. Unified State Transition Table

Table 40 defines the unified transition logic for PBF, mapping each workflow rule to a formal condition and state transition. Note that while the state machine diagrams use simplified labels for readability, the transition conditions in this table remain the formal, detailed specifications. Definitions for predicates and functions used in the table are provided in Tables A.1.5 and A.7.1. Table A.14.2 of Appendix A.14 is an instance of the PBF state transition table in a PBF MVP.

**Table 40.** Unified PBF state transition logic: Workflow rules mapped to conditions and operational state progressions.

Rule ID	Source State	Target State	Condition	Operational Step
B1	$S_0$	$S_1(i)$	$i = 1$	Begin pattern processing at root level
B2	$S_1(i)$	$S_2(i)$	$\exists n \in Pattern_i; \neg validated(n)$	Validate current pattern nodes
B2a	$S_1(i)$	$S_3(i)$	$\forall n \in Pattern_i; validated(n)$	Current pattern processing successful; proceed to depth resolution
B3	$S_2(i)$	$S_1(j)$	$(\exists n \in Pattern_i; \neg validated(n)) \wedge j = trace\_origin(i) \wedge refinement\_attempts(j) < R_{max}$	Backtrack to level j and begin refinement
B3a	$S_1(j)$	$S_2(j)$	$\exists n \in Pattern_j; \neg validated(n)$	Validate $Pattern_j$ again after refinement ( <i>explicit validation path</i> ) <sup>(1)</sup>
B3a1	$S_2(j)$	$S_3(j)$	$\forall n \in Pattern_j; validated(n)$	Resume depth resolution after refinement
B3a2	$S_2(j)$	$S_1(j)$	$\exists n \in Pattern_j; \neg validated(n) \wedge refinement\_attempts(j) < R_{max}$	Retry refinement processing at level j
B3a3	$S_2(j)$	$S_5$	$\exists n \in Pattern_j; \neg validated(n) \wedge refinement\_attempts(j) \geq R_{max}$	Terminate due to unresolved validation failures after exhausted refinement attempts
B3b	$S_1(j)$	$S_3(j)$	$\forall n \in Pattern_j; validated(n)$	Refinement validated; proceed to resolve depth of the finalized nodes ( $P(n)=2$ ) in level j
B3c	$S_2(i)$	$S_5$	$(\exists n \in Pattern_i; \neg validated(n)) \wedge (trace\_origin(i) \text{ undefined} \vee refinement\_attempts(trace\_origin(i)) \geq R_{max})$	Terminate due to $Pattern_i$ has unvalidated nodes but refinement is impossible

B4	$S_2(i)$	$S_3(i)$	$\forall n \in \text{Pattern}_i; \text{validated}(n)$	Proceed to resolve depth and prepare next
B4a	$S_3(i)$	$S_1(i+1)$	$i < L \wedge \text{Pattern}_{i+1} \neq \emptyset$	$\text{Pattern}_{i+1} :=$ select_critical_children( $\text{Pattern}_i$ ); Recurse to level $i+1$ for processing
B4b	$S_3(i)$	$S_4(1)$	$i=L \vee \text{Pattern}_{i+1} = \emptyset$	Transition to top-down finalization (prematurely or at leaf)
B5	$S_3(j)$	$S_1(j+1)$	$j < i$	Resume pattern processing at next level within refinement scope
B6	$S_3(j)$	$S_3(i)$	$j=i$	Refinement range complete; return to original current level for forward pass continuation
B7	$S_4(i)$	$S_4(i+1)$	$\forall n \in \text{Pattern}_i; \text{processed}(n)$	All nodes at level $i$ finalized; move to level $i+1$
B7a	$S_4(i)$	$S_1(j)$	$\exists n \in \text{Pattern}_i; \neg \text{processed}(n) \wedge j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max}$	Backtrack from completion phase to refinement processing
B7b	$S_4(i)$	$S_5$	$\exists n \in \text{Pattern}_i; \neg \text{processed}(n) \wedge \neg (j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max})$	Terminate due to unprocessed nodes with no refinement options
B8	$S_4(L)$	T	$\forall i \in [1, L], \forall n \in \text{Pattern}_i; \text{validated}(n)$	All nodes completed
B9	$S_1(j)$	$S_5$	$\text{refinement\_attempts}(j) \geq R_{\max}$	Terminate due to refinement cycle exhaustion

**Note:** (1). Explicit validation again (PB3a) ensures corrections in parallel-processed patterns are synchronized before progression. Applies to both initial refinement entry (PB3) and retries (PB3a2).

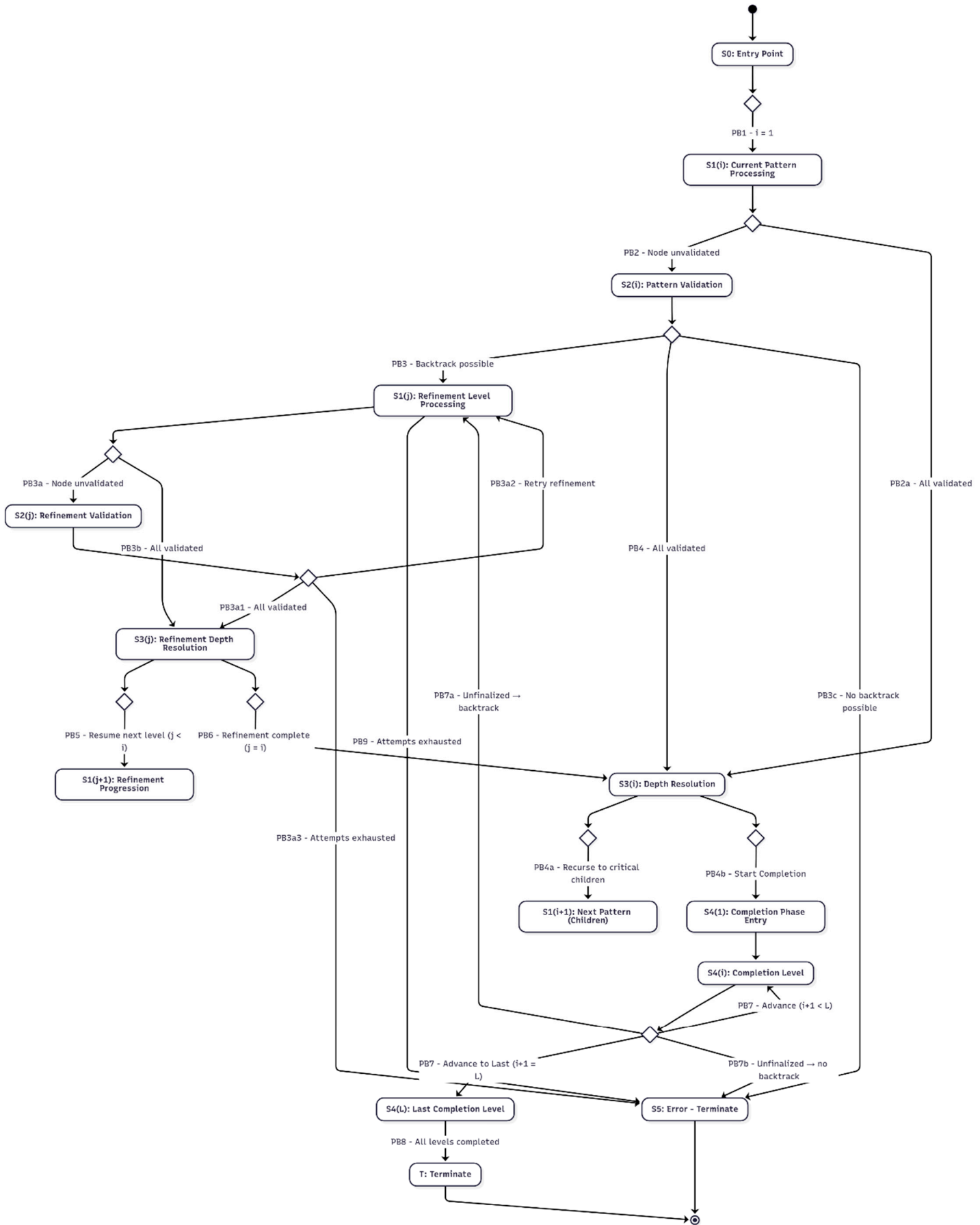
## 6. State Machine Diagram

Figure 13 presents the Pbfd state machine, representing the operational semantics of the methodology, including pattern transitions, validation and refinement feedback, depth resolution, and top-down completion. This diagram provides a visual representation of the workflow described in Table 40. The corresponding source code is available in Appendix A.7.2. Figure A.14.2 of Appendix A.14 is an instance of the Pbfd state machine diagram in a Pbfd MVP.

**Description:** The diagram shows transitions from initialization ( $S_0$ ) into pattern processing states  $S_1(i)$ , where patterns are validated ( $S_2$ ) and resolved ( $S_3$ ) before producing the next pattern. Validation errors may initiate a return to prior pattern levels for refinement ( $S_1(j)$ ). Upon reaching the final level, the workflow transitions to  $S_4(i)$  for top-down finalization, terminating at T when all nodes are processed. Validation failures that exceed  $R_{\max}$  refinement cycles transition to an error state ( $S_5$ ), halting automated execution.

## 7. CSP Formal Verification Results and Refinement Guarantees

This section confirms that the CSPM model (see Appendix A.7.4) of Pbfd satisfies all formal refinement properties when verified using the FDR model checker. The verification (see Table 41) ensures the concrete implementation adheres strictly to the behavioral constraints, liveness properties, and robustness required by the Pbfd specification, especially against an adversarial environment.



**Figure 13.** PBF state machine: Formal transition diagram covering initialization, pattern processing, refinement, and top-down finalization.

**Table 41.** Formal Verification Results for PBF D Model.

Property	CSP Assertion	FDR Result	Engineering Significance
Core Safety	System: [deadlock free]	✓ Passed	Prevents premature halts
Core Liveness	System: [divergence free]; SystemSync: [divergence free]	✓ Passed	Eliminates infinite internal cycles
Initialization Safety	S0: [deadlock free]; S1_InitialProcess(L1): [deadlock free]	✓ Passed	Confirms PB1 startup behavior from Table 40
Hostile Robustness	HostileSystem: [deadlock free]; HostileSystemSync: [deadlock free]	✓ Passed	Ensures correctness under non-cooperative inputs
Conditional Consistency	LegalCondEnv [T = NoContradictions]	✓ Passed	Verifies mutual exclusivity across all decision predicates
State-Level Safety	26 assertions	✓ Passed	All operational and terminal states (S0–S5, T) verified across all level combinations

### Interpretation & Contributions

#### Exhaustive State Coverage

The 26 state-level assertions span every defined state in Table 39, including:

- Initialization (S0, S1 at each level L1, L2, L3)
- Validation (S2\_ValidationInitial and S2\_ValidationRefinement for all valid (j,i) combinations)
- Depth progression (S3\_DepthProgression and S3\_RefinementDepthResolution for all valid (j,i) combinations)
- Completion (S4 at all levels L1, L2, L3)
- Terminal states (S5 for error, T for success)

Each state was proven both deadlock-free and divergence-free for all legal trace origins and conditional environments.

#### Termination via $R_{\max}$

The liveness checks confirm that no refinement loop can continue indefinitely. Transition rules PB3a3, PB7b, and PB9 from Table 40 enforce the bound on refinement attempts, ensuring the process always terminates at either T (success) or S5 (error).

#### Robustness Against Adversarial Conditions

Both hostile-environment assertions passed, confirming that PBF D's logic remains safe even when environmental conditions resolve in the least favorable (but legal) way.

This validates that the state machine correctly handles all possible condition combinations.

#### Implementation Fidelity

All nine transition rules (PB1–PB9) from Table 40 execute as specified, with correct handling of per-level refinement, condition evaluation, and propagation through child nodes.

#### Practical significance

The verification results confirm that PBF D delivers production-grade reliability through the following guarantees:

- **Guaranteed Termination:** The process always reaches either T (success) or S5 (controlled failure), eliminating the risk of system hangs.
- **Bounded Recovery:** Infinite refinement cycles are prevented via enforcement of the  $R_{\max}$  threshold, ensuring resource-bounded execution.
- **Fault Tolerance:** The model maintains correctness under adversarial inputs, supporting deployment in mission-critical environments.

Together, these guarantees ensure that a PBF D implementation cannot hang, enter an inconsistent conditional state, or exceed its refinement budget—regardless of input environment or traversal depth.

#### 8. LTL Properties

PBF D's correctness is grounded in the properties defined in Table 42.

**Measure Argument:** The termination and liveness proofs rely on a lexicographic measure  $M = (k_1, k_2, k_3, k_4)$  where:

- $k_1$ : Count of unfinalized nodes ( $k_1 = |\{n \in G \mid P(n) \neq 2\}|$ )
  - $k_2$ : Remaining refinement attempts across levels (decreases during refinement attempts)
  - $k_3$ : Phase ordinal (Initialization  $S_0=4$ , Progression  $S_1=3$ , Validation  $S_2=2$ , Resolution  $S_3=1$ , Completion  $S_4=0$ ) (decreases during forward phase transition)
  - $k_4$ : Intra-phase progress measure (e.g., progress within  $S_1, S_3$ , or  $S_4$  steps)
- Every non-terminal transition ensures a strict lexicographic decrease in  $M$ , as proven in Lemma A.8.7.

**Table 42.** PBFDF LTL Properties: Correctness guarantees, refinement bounds, and termination invariants.

Property	Formal Specification	Description & Justification
Total Correctness	$\Box(\text{start} \Rightarrow ((T \wedge \text{Structural Invariants}) \vee S_5))$	<b>Theorem A.8.8:</b> The methodology always terminates (T or $S_5$ ), and, upon successful termination (T), guarantees that all nodes are validated and all structural invariants are satisfied.
Termination	$\Box(\text{start} \Rightarrow \Diamond(T \vee S_5))$	<b>Lemma A.8.4:</b> Always, if the system starts, it eventually reaches the successful Termination (T) or bounded Error ( $S_5$ ) state [61].
Well-Foundedness	$M = (k_1, k_2, k_3, k_4)$ where $k_1 \in [0,  V ]$ , $k_2 \in [0, L \cdot R_{\max}]$ , $k_3 \in \{0,1,2,3,4\}$ , $k_4 \in [0, \text{max\_batch\_size}]$	<b>Lemma A.8.4:</b> Each component of the lexicographic measure $M$ is bounded and ranges over a well-ordered set, ensuring no infinite decreasing sequences exist.
Bounded Refinement	$\forall k \in [1, L]$ , $\Box(\text{refinement\_attempts}(k) \leq R_{\max})$	<b>Lemma A.8.2:</b> The number of refinement attempts for any level ( $k$ ) is strictly bounded by the constant $R_{\max}$ (e.g. $R_{\max} = 50$ ) [65,78]. A practical limit, such as $R_{\max} = 50$ , is used in the PBFDF MVP implementation (Appendix A.14).
Refinement Convergence	$\Box \forall j: (\text{refining}(j) \Rightarrow \Diamond(\neg \text{refining}(j) \vee \text{refinement\_attempts}(j) = R_{\max}))$	<b>Lemmas A.8.2 &amp; A.8.3:</b> Each refinement cycle eventually resolves the issue or exhausts its attempt bound, ensuring refinement is not indefinitely stalled [78].
Finalization Monotonicity	$\Box((\bigcirc k_1 \leq k_1) \vee (\bigcirc k_1 > k_1 \wedge \bigcirc k_2 < k_2))$	<b>Lemma A.8.3:</b> The global count of unfinalized nodes ( $k_1$ ) is non-increasing. It strictly decreases during commit transitions (PB4a, PB7) and can only increase during a guarded, bounded refinement reset that is compensated by a strict decrease in $k_2$ .
Finalization Permanence	$\forall n \in G: \Box((P(n)=2 \wedge \neg \exists j: (\text{refining}(j) \wedge n \in \text{affected\_nodes}(j))) \Rightarrow \bigcirc(P(n)=2))$	<b>Corollary A.8.3.1:</b> A finalized node's status is permanent unless actively reset by a guarded, bounded refinement backtrack.
Pattern Processing Order	$\Box \forall i: ((S_3(i) \wedge (i < L \wedge \text{Pattern}_{i+1} \neq \emptyset)) \Rightarrow \bigcirc(S_1(i+1)))$	<b>Lemma A.8.6 (Level-wise Ordering Invariant):</b> Progression to the next level's pattern ( $\text{Pattern}_{i+1}$ ) only occurs after the current pattern ( $\text{Pattern}_i$ ) is fully resolved.
Top-Down Finalization Order	$\Box \forall i: ((S_4(i) \wedge (\forall n \in \text{Pattern}_i: \text{processed}(n))) \Rightarrow \bigcirc S_4(i+1) \vee \bigcirc T \vee \bigcirc S_5)$	<b>Lemma A.8.6 (Top-down Finalization Invariant):</b> The completion phase strictly finalizes levels in sequence from root to leaf. [81].
Refinement Scope	$\Box \forall i, j: (\text{backtrack}(i, j) \Rightarrow (j = \text{trace\_origin}(i) \wedge j \leq i))$	<b>Lemma A.8.6 (Refinement Locality Invariant):</b> Backtracking always targets the calculated trace origin within the current progression frontier $i, j \leq i$ .
Vertical Closure	$\Box((P(n)=2 \wedge \text{children}(n) \neq \emptyset) \Rightarrow \Diamond(\forall c \in \text{children}(n): P(c) \in \{1,2\} \vee T \vee S_5))$	<b>Implied by Lemma A.8.6 invariants:</b> If a parent is finalized, its children are guaranteed to be addressed in the process flow, barring system termination.
Global Consistency	$T \Rightarrow (\forall n \in G, P(n)=2)$	<b>Rule PB8 (Table A.8.3):</b> Successful termination (T) guarantees that every single node in the system is finalized [88].
Soundness	$T \Rightarrow (\forall n \in G: \text{consistent}(n) \wedge \text{dependencies\_satisfied}(n))$	<b>Theorem A.8.8:</b> Successful termination implies all nodes are internally consistent and satisfy their architectural dependencies. [88]

Liveness (Progress)	$\square((\neg T \wedge \neg S_5) \Rightarrow \bigcirc(M <_{\text{lex}} M))$	<b>Lemma A.8.7:</b> From any non-terminal state, an enabled transition exists that strictly decreases the lexicographic measure $M$ , guaranteeing forward movement and preventing deadlock. [61]
Selective Progression Invariant	$\square((\text{state} = S_3(i) \wedge i < L \wedge \text{Pattern}_{i+1} \neq \emptyset) \Rightarrow \bigcirc(\text{state} = S_1(i+1) \wedge \text{Pattern}_{i+1} = \text{select\_critical\_child}(\text{ren}(\text{Pattern}_i))))$	<b>Rule PB4a (Table A.8.3):</b> Progression is guarded by the selection of the next pattern, ensuring only critical nodes are considered for the next processing cycle.
Completion Phase Invariant	$\square(\text{state} = S_4(i) \Rightarrow (\diamond \text{state} = S_4(i+1) \vee \diamond T \vee \diamond S_5))$	<b>Rule PB7 (Table A.8.3):</b> The sequential progression $S_4(1) \rightarrow S_4(2) \rightarrow \dots \rightarrow S_4(L)$ ensures that finalization is strictly top-down for global completeness.

## 9. Advantages

PBFD offers several advantages, as summarized in Table 43.

**Table 43.** PBFD Advantages: Design benefits from hybrid traversal, modular patterning, and bounded refinement.

Property	Advantage
Hybrid Flexibility	Combines the strengths of breadth-first (BFD), depth-first (DFD), and cyclic refinement (CDD) models
Pattern-Centric Traversal	Promotes modular grouping and processing of nodes by feature, layer, or function [89]
Scalable Parallelism	Enables concurrent processing within a pattern (horizontal parallelism)
Controlled Refinement	Supports bounded iteration (via $R_{\max}$ ) to avoid infinite rework loops
Predictable Finalization	Ensures all nodes are finalized through structured top-down traversal
Fine-Grained Dependency Recovery	Allow precise backtracking to affected pattern levels through validation-triggered refinements.
Termination Guarantee	Strong guarantees of convergence and termination, even with partial failures

### Cross-Paradigm References:

- PDFD refinement mechanics (Section 3.4.1) apply to PBFD's  $J_i$ ,  $R_i$ , and  $R_{\max}$  parameters.
  - `trace_origin(i)` follows the PDFD specification (Appendix A.1, Table A.1.5). For details on `trace_origin`, see PDFD's dependency-tracing logic in Section 3.4.1.
- The full formal specification for PBFD is provided in Appendix A.7.

### 3.5. Methodological Synergy and Graph Theory in Practice

The methodologies detailed in this section (DAD, DFD, BFD, CDD, PDFD, and PBFD) each address specific development challenges by applying structured traversal and refinement principles:

- **Directional Rigor:** Methodologies like DAD enforce strict hierarchies to prevent cycles, while DFD/BFD prioritize vertical/horizontal progression for early validation.
- **Iterative Resilience:** CDD enables controlled iterative refinement through structured feedback loops, essential for managing complexity and evolving requirements.
- **Hybrid Efficiency:** PDFD and PBFD apply hybrid traversal strategies, balancing depth-first and breadth-first techniques, and integrating CDD's iterative refinement to meet different scalability and modularity requirements.

By formally mapping these workflows to graph theory, developers can systematically optimize systems for modularity, scalability, and resilience.

These methodologies are not mutually exclusive; rather, they are often strategically blended to balance rigor with adaptability [58,86,90]. This hybridization (e.g., PDFD and PBFD) allows teams to

combine structured workflows with iterative refinement and parallel development. In practice, teams may adapt methods (e.g., using strict DAD for core logic and CDD for UI refinement) to fit specific project needs.

This interplay empowers developers to maintain architectural discipline [80] while adapting to evolving requirements, feedback cycles, and performance constraints—demonstrating the versatility of graph theory [59,88] in modern software engineering.

## 4. Bitmask Encoding and Three-Level Encapsulation

### Overview

Traditional relational models struggle with hierarchical data complexity, often requiring deep joins that inflate storage requirements and degrade performance—a fundamental limitation documented in database literature [54,91] and evidenced by empirical audits in fields like biodiversity informatics [92].

This section introduces a hierarchical encoding framework that addresses these limitations through two integrated techniques:

#### Section 4.1 - Bitmask-Based Encoding (Foundation)

- Compact representation of child node selections
- Each child corresponds to a single bit in an integer
- Enables  $O(1)$  set operations (union, intersection, membership testing)
- Analogous to bitmap-index encoding in relational systems [91]

#### Section 4.2 - Three-Level Encapsulation (Framework)

- Hierarchical pattern organizing data into Grandparent-Parent-Children levels
- Applies bitmask encoding at the Children level
- Enables  $O(1)$  relationship queries without joins
- Combines relational structure with bitmask efficiency

**Relationship:** TLE builds upon bitmask encoding—while Section 4.1 establishes how bitmasks efficiently encode child selections within a parent, Section 4.2 extends this into a complete hierarchical architecture where:

- Grandparent = Table (root context)
- Parent = Columns (intermediate entities)
- Children = Bitmask-encoded values (using Section 4.1 technique)

Both techniques leverage bitwise operations on fixed-width machine words, which execute in  $O(1)$  time for bounded hierarchies [62]. This integrated approach underpinned the 11.7× storage reduction and 7–8× faster query performance observed in our large-scale deployment (Section 5). While demonstrated here within PBFDD, these techniques offer general utility for hierarchical data systems across domains.

The architecture described in this section was implemented in the PBFDD Minimum Viable Product (MVP), with detailed empirical evaluation in Appendix A.14.

### 4.1. Bitmask-Based Pattern Encoding

#### 4.1.1. Motivation and Encoding Mechanism

##### The Problem

In pattern-driven development, particularly PBFDD, each node in a hierarchy may be associated with functional patterns (e.g., "high-density areas," "priority regions," specific geographic selections) that guide traversal, transformation, or validation. Traditional flag-based approaches using per-node Boolean properties incur  $O(N \cdot D)$  predicate evaluation costs across deep hierarchies [91,93].

##### The Solution

Bitmask encoding provides a compact representation where each specific child node corresponds to a single bit in an integer—a technique directly analogous to bitmap-index encoding

in relational systems [91]. A set bit indicates the corresponding child node is active for processing in the current traversal context.

#### Key characteristics:

- $O(1)$  operations for  $n \leq w$  (where  $w$  is machine word size, typically 64 bits)
- $O(\lceil n/w \rceil)$  operations for  $n > w$  (multi-word bitmasks with minimal constant factor)
- Other lifecycle states (e.g., 'processed,' 'validated,' 'finalized') tracked using separate auxiliary bitmask fields

The composition of a pattern—defining a functional classification or unit of business logic—is represented as a bitmask indicating the presence or absence of constituent child nodes. This enables constant-time operations to check, update, or combine selections across parent nodes, providing an efficient mechanism for tracking selected or processed nodes at each hierarchical level.

### 4.1.2. Structure and Operations

#### Bit Assignment

Each child node under a common parent is assigned a specific bit position within a bitmask, enabling rapid bitwise operations for querying, updating, or merging selections [94]. Table 44 illustrates this encoding for geographic nodes.

**Table 44.** Example bitmask assignments for geographic nodes, illustrating the encoding of node selections for PBF traversal and pattern matching.

Node Name	Level	Bit Index	Binary Mask	Decimal Mask (Per Level)
North America	3	0	0b00001	1
Asia	3	4	0b10000	16
United States	4	0	0b00001	1
Canada	4	1	0b00010	2
Mexico	4	2	0b00100	4

Example: If a parent node representing continents has "North America" and "Asia" selected, its combined bitmask is 0b10001 (decimal 17: 1 + 16).

#### Core Operations

Table 45 summarizes key bitwise operations for managing node selections within a parent's bitmask.

**Table 45.** Key bitwise operations for managing node selections and pattern states within parent node bitmasks.

Operation	Symbol	Example	Description
OR		parent_bitmask  = US_mask	Set a child node's bit (ensures selection while preserving prior selections)
AND	&	parent_bitmask & Canada_mask != 0	Check if a specific child node is selected in the parent's bitmask
XOR	^	parent_bitmask ^= Mexico_mask	Toggle the selection status of a child node
NOT	~	parent_bitmask &= ~Europe_mask	Clear a child node's bit (deselected the child)

This representation allows node selection status to be queried and modified in single-cycle operation, enabling efficient pattern-driven control flow.

### 4.1.3. Application in PBF

#### Node Selection and Tracking

In PBF, children nodes are assigned fixed bit positions as defined by their hierarchy. Bitmasks serve multiple purposes:

**Node Selection:** A parent's bitmask indicates which of its children nodes are selected or active for processing.

**Selection tracking:**

- Check if a child node is selected:  $\text{parent\_bitmask} \& \text{child\_node\_mask} \neq 0$
  - Mark a child node as processed/selected:  $\text{parent\_bitmask} \mid= \text{child\_node\_mask}$
- Bitmasks are attached to each relevant parent node during traversal and updated dynamically.

For example:

- A child node is “active” (selected) if its corresponding bit is set in the node's bitmask.
- Once processing for a child node is finalized, additional bits can be toggled to record completion status.

### Integration into the PBFDF Lifecycle

Bitmask fields support PBFDF traversal logic at each stage:

- **Pattern matching:** Select relevant groups of nodes at each level based on their bitmask representation
- **Validation and refinement:** Encoded selection status to avoid redundant node checks
- **Finalization:** Ensures complete coverage for all required node selections before progressing downward or exiting
- **State machine control:** Enables conditional transitions (e.g., transition from  $S_3$  to  $S_4$  only if all required children within a pattern are selected in the relevant parent's bitmask)

#### 4.1.4. Performance Characteristics

##### Storage and Computational Efficiency

Table 46 compares bitmask encoding against traditional row-based approaches.

**Table 46.** Comparative analysis of in-memory storage, query, and update efficiency between traditional row-based node selection methods and bitmask-based encoding.

Feature	Traditional (Row-based)	Bitmask-based
Storage	$O(n \text{ rows})$	$O(1)$ for $n \leq 64$ children; $O(\lceil n/w \rceil)$ with minimal factor for $n > 64$
Query	Recursive join ( $O(n)$ )	Bitwise check ( $O(1)$ )
Update	Row insert/delete ( $O(n)$ )	Bitwise OR/AND ( $O(1)$ )
Integration	SQL joins	Native bitwise ops in SQL & C-style languages, parallelizable

**Note:** Performance metrics reflect in-memory computational complexity for node selection and bitmask manipulation. End-to-end query performance depends on additional factors including I/O latency, network overhead, and database buffer management. Empirical query performance comparisons accounting for these factors are presented in Table 54.

##### Key Advantages:

- **Compact representation:** Up to  $w$  distinct children nodes can be encoded in a single  $w$ -bit word (e.g.,  $w = 64$ ), assigning each node a unique bit position— enabling simultaneous updates and queries via single-cycle bitwise operations [95].
- **Atomic updates:** Selection flags within a parent's bitmask can be updated using atomic bitwise operations if concurrency is involved.
- **Pattern combination:** Bitwise OR or AND across multiple parent nodes supports group operations (e.g., finding all parent nodes that share a common set of selected children).
- **Composable filtering:** Parent nodes can be filtered based on complex combinations of child node selections via simple bitwise comparisons.

#### 4.2. Three-Level Encapsulation (TLE)

Three-Level Encapsulation (TLE) builds upon the bitmask encoding technique introduced in Section 4.1, applying it to a three-level hierarchical structure.

While Section 4.1 demonstrated how bitmasks efficiently encode child node selections within a single parent, TLE extends this concept into a complete hierarchical pattern where:

- **Grandparent level:** Table (root context)
- **Parent level:** Columns (intermediate entities)

- **Children level:** Bitmask-encoded cell values (using the technique from Section 4.1)  
This architectural pattern enables constant-time hierarchical queries by combining relational structure (tables and columns) with bitmask-based child encoding.

#### 4.2.1. Pattern Definition and Core Concepts

##### Pattern Definition

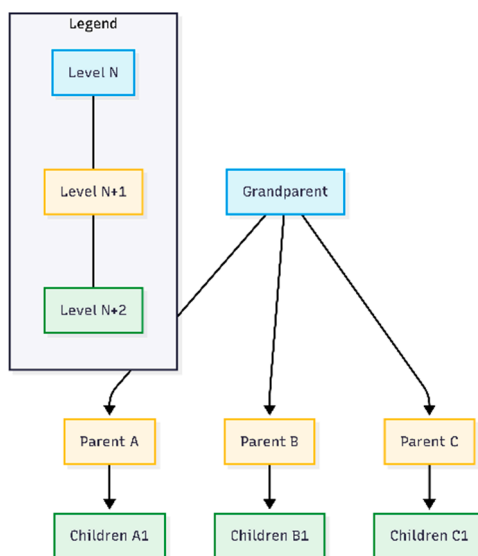
Three-Level Encapsulation (TLE) is a hierarchical encoding pattern designed to overcome the deep join and storage bottlenecks of traditional relational models [54,91]. TLE achieves constant-time ( $O(1)$ ) access to hierarchical relationships by structuring data into three levels of containment and encoding relationships as bitmasks rather than foreign keys.

##### Relational Mapping

Table 47 maps TLE's logical structure to its relational implementation. Figure 14 illustrates an abstract TLE unit, with corresponding source code provided in Appendix A.9.1.

**Table 47.** Three-Level Encapsulation (TLE) hierarchy mapping from logical concepts to relational implementation, showing how bitmask encoding (Section 4.1) is applied at the Children level.

Hierarchy Level	Logical TLE Component	Relational Implementation	Example Value
Level N	Grandparent	Table Name	dbo.[United States]
Level N+1	Parent	Column Name	[Maryland], [California], [Virginia]
Level N+2	Children	Cell Value (Bitmask)	5 (Binary 0b101 for counties in [Maryland]: Allegany, Baltimore)



**Figure 14.** Structural diagram of the Three-Level Encapsulation (TLE) model, showing the grandparent-parent-children mapping.

##### Recursive Extension

TLE supports arbitrary hierarchy depth through recursive application:

Entities that serve as "parents" at level N become "grandparents" at level N+1. For example:

- **Level 1:** [North American] (table) → [United State] (column) → States (bitmask)
- **Level 2:** [United States] (table) → Maryland (column) → Counties (bitmask)
- **Level 3:** Maryland (table) → [Allegany County] (column) → Cities (bitmask)

Each level maintains the same three-tier structure (table → columns → bitmasks), enabling scalable traversal without query complexity growth. This recursive pattern is detailed in Table A.14.4.

##### Implementation Variants

While storage-paradigm-agnostic (Potentially adaptable to key-value, document, or graph databases), TLE admits flexible relational implementations:

- **Canonical pattern (MVP):** One table per grandparent entity, maximizing modularity and independent evolution
- **Consolidated pattern (Enterprise):** Multiple grandparent entities combined into wide tables, optimizing for query performance and reduced I/O overhead

Both preserve TLE's core semantics while adapting to different operational requirements.

#### Bitmask Semantics

The bitmask stored for a parent node uses the encoding technique detailed in Section 4.1. As established there, each bit represents the state of a specific child node, enabling  $O(1)$  operations. In the TLE context, the bitmask stored for a parent node is a compact integer where each bit represents the state of a specific child node. For example, if the column Maryland has a bitmask with decimal value 5 (binary 0b101) representing its counties, the bits decode as follows:

- Bit 0 (LSB) = 1 → Allegany County is active
- Bit 1 = 0 → Anne Arundel County is inactive
- Bit 2 = 1 → Baltimore County is active

Because each county corresponds to a fixed bit position, determining whether a county is active requires only a constant-time bitwise operation:

$$(\text{Maryland} \& (1 \ll \text{county\_bit\_position})) \neq 0$$

A non-zero result indicates that the corresponding county is active for that record in the current traversal context.

#### 4.2.2. Hybrid Architecture and Implementation

##### Architecture Components

The enterprise deployment implements TLE using a hybrid data model that maintains both normalized source data and performance-optimized TLE tables. This architecture balances data integrity with query efficiency—a strategy aligned with evolving best practices for complex data workloads [54].

- **Source hierarchy table:** Maintains normalized parent-child relationships using traditional foreign keys. This serves as the authoritative data source and ensures referential integrity.
- **Derived TLE table:** A denormalized, bitmask-encoded representation materialized from the source table. Structured according to Table 47's mapping, this provides  $O(1)$  hierarchical access without joins.

A detailed implementation of this hybrid architecture is provided in the PBFD MVP (Appendix A.14), including schema definitions and materialization logic.

##### Operational Workflow

The TLE pattern efficiently manages hierarchical data processing through its core operations: LOAD, READ, WRITE, and COMMIT. The compact bitmask representation enables atomic updates and consistent traversal of hierarchical relationships.

For example, in an interactive web application with a relational backend, this general workflow can be instantiated as follows: User selections on a previous page act as the input, prompting the system to LOAD the grandparent table and READ the bitmask cell values from its columns to retrieve a batch of corresponding parent and children nodes for processing and display on the current page. For each parent node, a bitmask encodes the selections of its children. As illustrated in Figure A.17.2 (Appendix A.17), the parent node of "North America" initially has "Canada" and "United States" selected. Upon user submission, the WRITE operation updates this bitmask to reflect the latest selections ("Canada" and "Mexico"), and the COMMIT operation persists the changes back to the grandparent table.

##### Core Operations

The fundamental operations on a TLE structure are:

- **LOAD(Grandparent):** Load the TLE-encoded data for a given grandparent context
- **READ(Parent, Child):** Check the state (selected/active) of a specific Child within a Parent's bitmask
- **WRITE(Parent, Child, State):** Set or clear the state of a specific Child within a Parent's bitmask
- **COMMIT(Grandparent):** Persist the updated TLE-encoded data for the grandparent context

These operations can be composed into workflows suitable for various contexts (interactive web apps, batch data pipelines, streaming services, etc.).

While this denormalized, bitmask-based representation resembles NoSQL's document-oriented storage, the Three-Level Encapsulation (TLE) model is implemented entirely within a relational backend, preserving full ACID guarantees. This hybrid architecture is central to the PBFDP MVP and the enterprise deployment: it achieves the scalability and traversal efficiency characteristic of NoSQL systems while maintaining the integrity and transactional reliability of relational databases.

#### Performance Characteristics

The TLE table's single-row, fixed-width representation of three-level subtrees eliminates multi-table joins and enables constant-time relationship queries. This structural compression—where an entire subtree maps to one table row with bitmask columns—directly produces the empirical performance gains reported in Section 5, where TLE-based queries consistently outperformed normalized designs.

#### Key advantages:

- **Eliminated joins:** Parent-child relationships accessed via bitmask operations within a single row
- **Predictable I/O:** Fixed-width rows enable efficient memory layout and caching
- **Constant-time operations:** Bitwise operations replace recursive traversals

The hybrid architecture allows updates to flow through the normalized source table (preserving ACID properties) while reads leverage the optimized TLE representation (maximizing throughput). Synchronization between source and derived tables can be implemented via triggers, scheduled jobs, or event-driven updates based on consistency requirements.

#### 4.2.3. Formal Specification and Verification

##### Abstract State Descriptions

The lifecycle for processing a hierarchical TLE data unit can be formally described by the abstract states outlined in Table 48.

**Table 48.** Abstract state definitions for the TLE hierarchical data processing lifecycle.

State	Phase	Abstract Description
S <sub>0</sub>	Idle	The TLE structure is at rest; no active unit of work.
S <sub>1</sub>	Data Loaded	A TLE data unit (e.g., a grandparent row) has been loaded into a processing context.
S <sub>2</sub>	Hierarchy Resolved	The grandparent and parent levels have been identified and validated.
S <sub>3</sub>	Children Evaluated	Child node states have been read and logically processed (e.g., filtered, validated).
S <sub>4</sub>	Children Updated	Child node states have been modified via bitmask writes.
S <sub>5</sub>	Changes Committed	All modifications to the TLE structure are persisted to the grandparent entity.
S <sub>6</sub>	Workflow Finalized	The unit of work is complete; the system is ready for the next task (via transition TLE10 to S <sub>0</sub> in the CSP model to ensure system liveness).

##### Unified State Transitions

Transitions between these abstract states are governed by TLE operations and business-logic conditions, detailed in Table 49. Definitions of all functions and variables referenced in this section are provided in Table A.9.1.

**Table 49.** Formal state transition rules for the abstract TLE processing model, defining the lifecycle of hierarchical data operations and ensuring reproducibility of PBFD's traversal logic.

Rule ID	From State	To State	Transition Condition/Trigger	Core TLE Operation/Action
TLE1	[*]	S <sub>0</sub>	System Start	-
TLE2	S <sub>0</sub>	S <sub>1</sub>	initiate_workflow(Grandparent)	LOAD(Grandparent)
TLE3	S <sub>1</sub>	S <sub>2</sub>	resolve_hierarchy()	(Internal resolution)
TLE4	S <sub>2</sub>	S <sub>3</sub>	evaluate_children()	Iterative READ(Parent, Child)
TLE5	S <sub>3</sub>	S <sub>4</sub>	update_required $\wedge$ apply_update()	WRITE(Parent, Child, State)
TLE6	S <sub>3</sub>	S <sub>5</sub>	$\neg$ update_required	-
TLE7	S <sub>4</sub>	S <sub>5</sub>	persist_changes()	COMMIT(Grandparent)
TLE8	S <sub>5</sub>	S <sub>0</sub>	has_next_unit()	-
TLE9	S <sub>5</sub>	S <sub>6</sub>	$\neg$ has_next_unit()	-
TLE10	S <sub>6</sub>	S <sub>0</sub>	Workflow Complete	finalize_process()
TLE11	S <sub>0</sub>	S <sub>6</sub>	$\neg$ has_unprocessed_unit()	-

Conditions such as `update_required` represent atomic composite operations within the state machine. In the CSP specification (Appendix A.9), the  $S_6 \rightarrow S_0$  recursion (Rule TLE10) formally captures the readiness of the TLE engine for continuous, multi-unit processing.

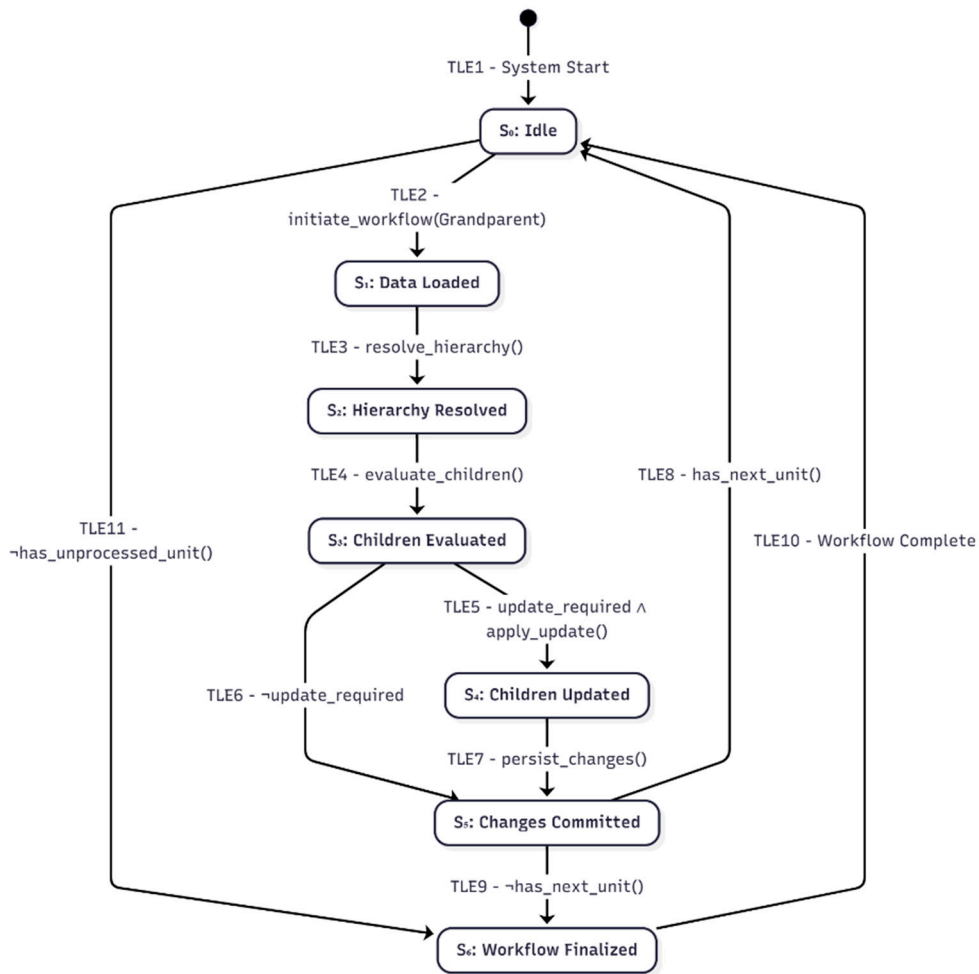
Figure 15 illustrates the state transitions from Table 49. Its source code is in Appendix A.9.2. This model represents the generalized lifecycle. Domain-specific implementations will provide the logic for the transition conditions.

#### Formal Verification and Refinement Guarantees for TLE

This section reports verification results using FDR 4.2.7. The analysis confirms conformance to the abstract model, correctness of parameterized state transitions, and safety of the event-driven execution workflow. The verification demonstrates that the TLE model preserves structural soundness, maintains isolation of per-unit processing, and supports continuous execution without deadlock or divergence (see Table 50).

**Table 50.** Formal Verification Summary for TLE.

Property	CSP Assertion	FDR Result	Engineering Significance
Core System Safety	TLE_Process : [deadlock free], TLE_Process [T = TLE_Abstract_Process], TLE_Process [F = TLE_Abstract_Process], TLE_Process [FD = TLE_Abstract_Process]	✓ Passed (4)	Confirms conformance to the abstract model and absence of halting executions; guarantees full behavioral refinement
State-Level Reliability	TLE_S0, TLE_S1.u1-u3, ..., TLE_S6.u1-u3 (Implementation) TLE_Abstract_S0, TLE_Abstract_S1.u1-u3, ..., TLE_Abstract_S6.u1-u3 (Abstract)	✓ Passed (38)	Ensures deadlock freedom for all operational states across all unit parameters; validates unit-specific determinism
Liveness Guarantees	TLE_Process : [divergence free], TLE_Abstract_Process : [divergence free]	✓ Passed (2)	Confirms absence of infinite internal activity; guarantees workflow continuity
Composition & Robustness	TLE_TwoUnits : [deadlock free], TLE_Abstract_TwoUnits : [deadlock free], TLE_Hostile_System : [deadlock free], TLE_HostileEnv : [deadlock free], TLE_Process : [deterministic [F]]	✓ Passed (5)	Validates safe concurrent execution, robustness under adversarial inputs, and internal determinism of the TLE workflow



**Figure 15.** Abstract state machine diagram for TLE processing, showing transitions between phases of hierarchical data operations.

## Interpretation and Technical Contributions

### State-Space Coverage

The verification covers all 49 assertions across the parameterized TLE state space. The 38 state-level checks reflect:

$$38 = 2 \times [(1 \text{ non-parameterized state } S_0) + (6 \text{ parameterized states} \times 3 \text{ units})]$$

Broken down:

- Implementation specification:  $S_0$  (1) +  $S_1$ – $S_6$  across  $u_1, u_2, u_3$  (18) = 19 assertions
- Abstract specification: Abstract\_ $S_0$  (1) + Abstract\_ $S_1$ – $S_6$  across  $u_1, u_2, u_3$  (18) = 19 assertions
- Total:  $19 + 19 = 38$

### Unit-Specific Determinism

Execution for  $S_1(u)$  through  $S_6(u)$  is verified separately for  $u_1, u_2$ , and  $u_3$ . Parameterized channels ensure events advance only the corresponding state instance, preventing interference across concurrent units.

### Recurrence Guarantee

State  $S_6(u)$  transitions to  $S_0$  via `finalize_process.u`, ensuring continued operation over unbounded streams of TLE units.

### Failures–Divergences Refinement

Passing the FD refinement confirms alignment between `TLE_Process` and `TLE_Abstract_Process`, ensuring that all observable behaviors and refusal sets match their formal specification.

### Hostile-Environment Robustness

Deadlock-freedom under adversarial or out-of-order event injection demonstrates that external disturbances cannot force the system into unschedulable states.

#### Practical Significance

The verification establishes the following guarantees:

- **Isolation:** Parameterized state and channel definitions maintain separation between concurrent units.
- **Robustness:** The system remains safe under adversarial scheduling or unexpected event ordering.
- **Event-Driven Correctness:** Synchronization via parameterized channels mirrors the intended event-driven semantics.
- **Continuous Operation:** The  $S_6 \rightarrow S_0$  recurrence supports unbounded execution without termination or deadlock.

The TLE model has been formally verified for correctness, consistency, and termination, with grounded proofs establishing liveness and the absence of deadlocks and livelocks (full details in Appendix A.9.6).

#### 4.2.4. Performance Characteristics and Complexity Analysis

##### Computational Complexity

The computational characteristics of TLE are derived from its bitmask-based representation and direct-memory semantics. These characteristics determine the operational complexity of core actions such as storage, lookup, update, and batch traversal.

Table 51 summarizes the complexity guarantees formally proven in Appendix A.10 (Theorems A.10.1–A.10.4). These results quantify the performance behavior of TLE under varying hierarchical distributions. The core notation appears in Table A.1.8. of Appendix A.1.

**Table 51.** Computational characteristics of the Three-Level Encapsulation (TLE) model, with complexity guarantees from Theorems A.10.1–A.10.4.

Characteristic	Operation /Complexity	Explanation
Storage Efficiency	Storage ratio: $S_{TLE} / S_{traditional} = \hat{C} / (\hat{c} \cdot k)$	Encodes child-relationship sets in bitmasks instead of foreign key rows. $\hat{C}$ = average bitmask size; $\hat{c}$ = average children per parent; $k$ = metadata overhead per relational child record. For sparse hierarchies where $\hat{C} \ll \hat{c} \cdot k$ , TLE yields substantial storage reduction.
Query Complexity	$O(1)$ ( $n \leq w$ ), $O(\lceil n/w \rceil)$ otherwise	Bitmask lookup enables constant-time child existence checks when the hierarchy fits within a standard word size.
Update Cost	$O(1)$ ( $n \leq w$ ), $O(\lceil n/w \rceil)$ otherwise	Updates (adding/removing child association) are performed via bitwise OR / AND / XOR instead of relational inserts/deletes.
Batch Parent Traversal	$O(P_{total})$	A linear scan over all parent entities eliminates index lookups, since parent-child presence is determined from the mask.
Denormalization Cost	$O(1)$ amortized	There are no join tables, as relationships are encoded directly in each parent row.

TLE compresses hierarchical relationships into word-sized (or compactly encoded) bitmasks and performs direct bitwise computation without joins or secondary index scans. This yields constant-time operations when the hierarchy fits within a machine word and logarithmic scaling otherwise. These performance characteristics explain the empirical gains demonstrated in Section 5.

##### Formal Properties

The TLE model also exhibits properties beyond performance—specifically, properties related to semantics, correctness, and behavioral guarantees. These are summarized in Table 52 and supported by formal proofs in Appendix 10 and FDR model checking in Appendix 9.

**Table 52.** Formal properties of Three-Level Encapsulation (TLE) model.

Property	Description	Formal Basis
Storage Efficiency	Replaces $O(m)$ foreign key storage with $O(\sum C_i)$ bitmask storage, yielding an asymptotic reduction of $O(1/k)$ . Sparse hierarchies amplify the reduction factor	Theorem A.10.1
Query Complexity	$O(1)$ lookup of child-membership status when $n \leq w$ (word size) using bitwise tests; $O(\lfloor n/w \rfloor)$ for larger hierarchies	Theorem A.10.2
Update Complexity	$O(1)$ bitwise update on the mask; does not require relational mutations	Theorem A.10.3
Batch Processing	Direct sequential scan through bitmasks enables parent-level batch traversal in $O(P_{total})$	Theorem A.10.4
Semantic Expressiveness	Maintains explicit root $\rightarrow$ parent $\rightarrow$ child semantics; masks encode relationship cardinality constraints	Section 4.2 (Figs. 14–15), [96]
Behavioral Correctness	Verified deadlock-free lifecycle based on TLE state machine	FDR4 Proof (Appendix A.9)
Empirical Evidence	Demonstrated significant storage savings and faster query execution at MVP and enterprise deployment scale	Section 5

Unlike Table 51, which addresses computational cost, Table 52 synthesizes TLE’s ontological, behavioral, and correctness guarantees—demonstrating that TLE is not only efficient, but also semantically precise, verification-ready, and ACID compliant.

#### 4.3. Summary of Advantages

The key techniques and their advantages are consolidated in Table 53.

**Table 53.** Summary of hierarchical encoding techniques and their benefits, highlighting their role in enabling PBFd’s scalability, maintainability, and empirical performance gains (Section 5).

Technique	Purpose	Role in Architecture	Benefits
Bitmask Encoding (4.1)	Efficient node selection and state tracking	Foundation: Encodes set membership at $O(1)$ complexity	Compact storage, constant-time operations, parallelizable
Three-Level Encapsulation (4.2)	Structured hierarchical data management	Framework: Applies bitmask encoding to Grandparent-Parent-Children structure	Eliminates joins, $O(1)$ relationship queries, scalable design

**Note:** TLE builds upon bitmask encoding, using it at the Children level to encode parent-child relationships within a three-tier relational structure. This layered architecture enables both the storage compactness of bitmasks and the structural efficiency of hierarchical organization.

These encoding strategies underpin the scalability and maintainability demonstrated in PBFd’s empirical deployments. The compactness of bitmask encoding and the join elimination of TLE were direct contributors to the substantial reductions in development effort, execution latency, and storage requirements detailed in Section 5.

Source code and the full formal specification for the described TLE operations are provided in Appendix A.9, ensuring reproducibility and facilitating integration into other hierarchical data systems.

## 5. Evaluation of PBFd and PDFd: From Controlled MVPs to Production Deployment

We evaluated the Primary Breadth-First Development (PBFd) and Primary Depth-First Development (PDFd) methodologies through a multi-method empirical strategy. This approach encompassed both the implementation of open-source Minimum Viable Products (MVPs) to validate the core architectural principles and a longitudinal case study of a production PBFd deployment to measure large-scale performance [97].

This evaluation advances Evidence-Based Software Engineering (EBSE) [98] by providing reproducible artifacts and empirical data. The MVP implementations ground the formal state transitions and methodological workflows in practical systems, extending the vision of improvement-oriented software environments [99].

### **Evidence from MVP Implementations**

The PDFD MVP (Appendix A.11) was essential for validating Hybrid Depth-First Progression (BF-by-Two) and demonstrated early conflict detection across sibling nodes—such as UI state inconsistencies between “Asia” and “North America”—that cannot be detected as early in pure depth-first strategy. It further operationalized bounded refinement ( $R_{\max} = 60$ , chosen empirically) and iterative schema adaptation in response to mid-development changes. This was conducted as a controlled experiment, designed to test bounded refinement and sibling-node conflict detection under reproducible conditions.

The PBFD MVP (Appendix A.14) served as a concrete instantiation of the Three-Level Encapsulation (TLE) architecture and bitmask encoding, providing a reproducible artifact that validated the core mechanisms enabling high performance. It demonstrated the replacement of four to five join traversals with direct one-hop access and confirmed the feasibility of constant-time ( $O(1)$ ) bitmask updates under controlled conditions (See Table A.14.7). This was conducted as a controlled experiment, validating constant-time bitmask updates and one-hop access in a reproducible test harness.

All MVP components—including schema generators, migration scripts, test harnesses, and sample datasets—are publicly available in the artifact repository [28,29], enabling third-party validation and replication under real-world conditions.

### **From Architectural Validation to Production Performance**

The architectural patterns validated in the PBFD MVP—specifically TLE and bitmask-based subtree encoding—were directly deployed in the enterprise system. The production implementation subsequently recorded dramatic performance results, achieving 7–8× faster query execution and an 11.7× reduction in storage requirements compared to normalized relational designs. Development timelines were reduced by 20×, and zero post-release defects were recorded over eight years of continuous operation—outcomes attributable to the structured, constraint-driven application of PBFD.

### **Focus of This Section**

While both methodologies were rigorously evaluated through their MVP implementations, this section emphasizes the longitudinal PBFD enterprise deployment. This case was selected for its scale, ecological validity, and availability of long-term operational data, enabling a comprehensive assessment of methodology impact on development effort, runtime performance, and storage efficiency in a real-world setting. All findings presented are derived from anonymized operational metrics and reproducible performance benchmarks collected over multiple release cycles over a span of eight years.

#### *5.1. Problem Context*

A client required a claim form application to capture detailed incident reports, a domain characterized by high structural complexity [100]. The project faced three core challenges under an aggressive three-week delivery constraint:

- **Complex data requirements:** The system was designed to support the structured capture of incident locations, timelines, multi-tiered classification codes, and detailed employment data, including union affiliations, employment status, and employer information.
- **Deep hierarchical dependencies:** The form structure includes up to eight levels of conditionally dependent elements, which are formally modeled as an  $n$ -ary tree. This depth leads to a combinatorial explosion of possible states, making traditional row-based storage and retrieval inefficient [91].

- Performance and Delivery Demands:** The system required real-time validation and responsive user interaction under production load, with complete feature delivery within three weeks—a timeline incompatible with conventional iterative development approaches.

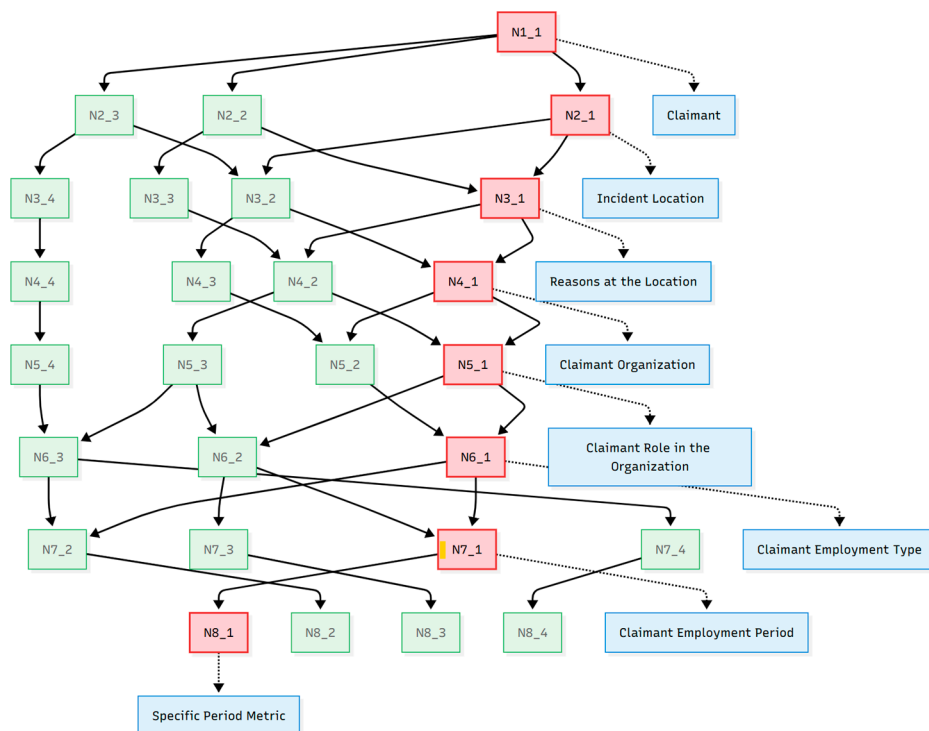
Traditional relational approaches, reliant on normalized schemas and volumetric join operations, exhibited high latency and fragile scalability when maintaining consistency across these hierarchical layers [54], making them unsuitable for both the technical complexity and the compressed delivery schedule.

### 5.2. Solution: Adoption of Pbfd Methodology

To address these challenges, we adopted the Pbfd methodology, leveraging its level-wise processing strategy and bitmask-based hierarchical encoding to achieve constant-time ( $O(1)$ ) operations on hierarchical relationships [101]. The development process followed the structural workflow illustrated in Figure 12 and was guided by four key design principles:

#### Hierarchical modeling

The business logic was formally structured as an 8-level n-ary tree (Figure 16; Mermaid source code in Appendix A.19), providing a graph-based representation that enabled systematic decomposition of the domain's hierarchical structure. This n-ary model allows Pbfd's bitmask encoding to capture complex parent-child relationships while maintaining ( $O(1)$ ) query performance through ancestral path encoding.



**Figure 16.** Eight-level n-ary business hierarchy for claimant management. The highlighted path (red nodes) traces the primary analytical chain from Claimant to Specific Metric. Green nodes represent alternative branches—for example, multiple incident locations at Level 2 (N2\_1, N2\_2, N2\_3) enable different analytical pathways.

#### Bitmask-based representation

Each user selection was stored as a compressed bitmask encoding aligned to its hierarchical level, applying the mechanism detailed in Section 4.1. This enabled efficient storage, traversal, and bitwise set operations (union, intersection, difference) on hierarchical selections [102].

#### Database Optimization via Consolidated TLE Schema

The production deployment adapted the Three-Level Encapsulation (TLE) principles from Section 4.2 into a consolidated, high-performance schema. While the canonical TLE pattern uses one table per grandparent node to maximize theoretical extensibility, the production implementation collapses all nodes into two shared tables, trading structural flexibility for query performance and development simplicity.

#### Consolidation Approach

- **Hierarchy flattening:** The 8-level hierarchy (Figure 16) was flattened by representing grandparent entities as columns within a single table, rather than as separate tables in the canonical TLE design. This creates a recursive column promotion pattern:
  - Parent columns at level N contain bitmask values encoding their children
  - These parent columns are promoted to grandparent columns at level N+1
  - Each column-bitmask pair preserves the parent→child relationship within a unified table structure

For example, a “United States” column (grandparent) is associated with state-level parent columns, which in turn store county-level bitmasks as children. At the next level, state columns are promoted to grandparent roles for their respective county hierarchies. This recursive promotion continues through level L-3 (where L is the total hierarchy depth), stopping two levels before the bottom to ensure sufficient depth for TLE encoding.

- **Preserved semantics:** The core TLE logic remains unchanged—for any parent value, a bitmask column encodes its selected children. Parent-child relationship semantics and bitwise operations are identical to canonical TLE; only the physical storage model differs.
- **Performance outcome:** This consolidation reduced the transactional schema to two tables, minimizing I/O overhead and join complexity while guaranteeing production-scale performance [54].

This adaptation demonstrates TLE’s flexibility: its core bitmask-based encoding supports both canonical multi-table schemas and consolidated wide-table designs, enabling performance-tuned deployments without sacrificing semantic integrity.

#### UI integration

Dynamic user interfaces directly interpreted bitmask-encoded data to render hierarchical form structures, ensuring consistency between the data model and presentation layer.

#### 5.3. Implementation Outcomes

The adoption of Pbfd yielded significant improvements across key engineering metrics. Table 54 summarizes the results while detailed methods and evidence are in the appendices. To support methodological transparency and traceability, Table 55 expands on the study types listed in Table 54 by detailing their design dimensions and evaluation structure.

**Table 54.** Empirical results from a Pbfd enterprise deployment, demonstrating improvements in development speed, runtime performance, and storage efficiency over traditional relational and OmniScript-based implementations.

Aspect	Pbfd Outcome	Reference & Notes
Development Speed	At least 9× faster than equivalent relational development and 20× faster than OmniScript; full-stack system delivered in 1 FTE-month	Appendix A.20 — longitudinal observational study [103,104]
Runtime Performance	7.64× faster (P50), 8.54× faster (P95); P5 equal to baseline (identical latency floor); sustained across 8 years	Appendix A.21 — quasi-experimental runtime comparison under identical infrastructure [105,106]
Storage Efficiency	11.7× less reserved space, 85.7× smaller index size, 113.5× better page utilization; eliminated junction tables	Appendix A.22 — controlled schema-level evaluation comparing Pbfd vs. normalized designs [105,107]

System Stability	Zero critical defects, deadlocks, or regressions across 8 years	Internal monitoring; Longitudinal observational study [97]
Onboarding Efficiency	Junior developer delivered a production feature in one week	Internal engineering metrics — qualitative observational evidence [107]

**Notes:** Study types follow Evidence-Based Software Engineering (EBSE) guidelines [97,105,107], distinguishing observational, quasi-experimental, and controlled design-science evaluations.

**Table 55.** Experimental Designs Dimensions in Pbfd Evaluation.

Design Dimension	Development Speed	Runtime Performance	Storage Efficiency
Unit of Comparison	Implementation methodology (Pbfd vs. relational vs. OmniScript)	Different UI endpoints within the same deployed application	Different schema designs (TLE vs. normalized) within the same database
Evaluation Focus	Effort and time required to implement equivalent functionality	Request latency and execution speed	Reserved space, index size, and page utilization
Controlled Variables	Shared enterprise context, functional requirements, audit logging	Same hardware and application context; workload varies by page logic	Same DBMS, hardware, and data volume
Independent Variable	Development methodology and platform	Page-level logic and rendering paths	Schema structure (TLE vs. normalized joins)
Study Type	Longitudinal observational case study	Quasi-experimental comparison	Controlled schema-level experiment

The findings from Table 54 confirm that Pbfd reduces development effort, improves runtime responsiveness, and optimizes storage for hierarchical workloads—translating its theoretical advantages into sustained production impact.

To clarify the methodological basis for each evaluation, Table 55 summarizes the experimental design dimensions and study types applied in the Pbfd assessments.

#### 5.4. Technical Observations

Analysis of the production deployment yielded the following observations:

- **Rapid Development and Onboarding:** Pbfd enabled one developer to deliver a production system in a single month. Compared to traditional methods ( $\geq 9\times$  faster) and low-code tools ( $\geq 20\times$  faster), this is supported by Appendix A.20's analysis. The graph-driven structure also fostered rapid onboarding, aligning with evidence on the role of coherent mental models in comprehension [108].
- **Compact Storage and Schema Simplification:** Encoding relationships into fixed-width bitmask fields reduced schema complexity from 13 tables (6 factor and 7 junction tables) to 2, while achieving  $11.7\times$  overall storage reduction and  $85.7\times$  index reduction (Appendix A.22).
- **Optimized Write and Query Performance:** Bitwise  $O(1)$  updates replaced traditional  $O(n)$  multi-row operations. This explains the  $7\text{--}8\times$  page-load improvement and lower tail latency (Appendix A.21), mitigating known bottlenecks in hierarchical queries [91].
- **Production-Stable Hybrid Semantics:** Pbfd illustrates a hybrid relational–NoSQL design through TLE: SQL Server is used to achieve document-like modeling within a relational system. Eight years of production stability demonstrate that Pbfd balances hierarchical flexibility with ACID integrity [109].

#### 5.5. Limitations and Threats to Validity

While promising, the results must be qualified by the following threats [97]:

- **Single-case Generalizability:** Findings from one enterprise case, offering strong ecological validity but limited statistical generalization

- **Construct Validity – Developer Expertise:** While all implementations were led by expert developers, expertise levels and domain familiarity vary across individuals. The PBFd vs. relational comparison involves the same expert (PBFd's inventor) leading both, introducing additional confounds from learning effects and problem familiarity. Detailed analysis in Appendix A.20.5
- **Construct Validity – Baseline Heterogeneity:** Heterogeneous systems for baseline comparisons, providing ecological realism and potentially underestimating PBFd's performance advantage (see Appendices A.21.6, A.22.4)
- **Temporal and Maturation Threats:** Data spanning 2016–2024, introducing potential history and maturation effects mitigated by the longitudinal design  
These threats are explicitly addressed in the appendices. Broader replication studies are discussed as future work in Section 7.

## 6. PDFD and PBFd Comparative Analysis

This section evaluates the proposed Primary Depth-First Development (PDFD) and Primary Breadth-First Development (PBFd) methodologies in comparison to traditional Full-Stack Software Development (FSSD) approaches and modern database paradigms, with additional focus on hierarchical encoding techniques specific to PBFd. The comparative analysis is grounded empirically in Section 5 and Appendices A.11–A.22, including the detailed MVP comparisons in Appendix A.18, ensuring rigor and reproducibility.

### 6.1. Traditional FSSD: Situational Advantages and Trade-Offs

While PBFd and PDFD excel in complex hierarchical systems, traditional Full-Software Systems Development (FSSD) approaches may still be preferred in specific, less intricate scenarios. These traditional approaches align with established agile practices that emphasize iterative development and responsiveness to change [110]. Table 56 summarizes these situations and their associated trade-offs, providing a contextual comparison against established practices.

**Table 56.** Situational trade-offs: Traditional FSSD versus PDFD and PBFd across selected project scenarios

Scenario	Traditional FSSD Advantage	Trade-off with PDFD	Trade-off with PBFd
Small-Scale Projects	Minimal setup and tooling overhead consistent with lightweight processes [111]	Vertical slicing overhead unnecessary for trivial systems	Hierarchical encoding and TLE architecture add unnecessary complexity.
Rapid Prototyping	Drag-and-drop tools quick iteration enabled	Slower initial visibility due to vertical rigor	Architecture-first planning delays visible prototypes.
Non-Hierarchical Systems	Works well for simple CRUD apps and dashboards	Hierarchy modeling unnecessary	Hierarchical encoding (TLE, bitmasks) provides no benefit.
Legacy Integration	Compatible with existing monolithic, relational systems	Requires refactoring into vertical feature slices with explicit dependencies	Legacy schemas must be restructured into TLE's three-level hierarchical architecture.
Team Familiarity	Common practice with extensive tooling support [112]	Requires learning feature-first structuring and validation workflows	A solid understanding of TLE, bitmask encoding, and level-wise progression is required.

### 6.2. Methodological Comparison: FSSD vs PDFD vs PBFd

This section provides a side-by-side comparison of the three methodologies across core software engineering dimensions, including their alignment with contemporary practices like Agile and DevOps. The comparison framework follows established software engineering analysis methods that

evaluate methodologies across multiple architectural and process dimensions [65]. Table 57 summarizes this methodological comparison of traditional FSSD, PDFD, and PBFd.

**Table 57.** Methodological comparison of traditional FSSD, PDFD, and PBFd

Criterion	Traditional FSSD	PDFD	PBFd
Method	Iterative feature development with flexible layering [110]	Complete vertical feature slices (UI→Logic→DB) with early integration	Systematic layer-by-layer development with pattern-driven refinement
Focus	Flexible layer transitions; sprint-based iteration	Depth-first traversal per feature slice with bounded refinement ( $R_{max}$ )	Breadth-first level traversal with selective depth-first pattern elaboration and bounded refinement ( $R_{max}$ )
Progression Model	Partial features across layers; integration deferred	Fully functional end-to-end feature slice	Complete architectural skeleton with interface definitions across all layers
Early Deliverable	Late-stage integration and architectural risks [65]	Feature-level integration risks identified and resolved early	Interface contracts and architectural inconsistencies identified early
Risk Visibility	Sprint-based parallelism with cross-functional teams	Controlled parallel feature development via $K_i$ threshold (WIP limit per level)	Parallel layer development after interface stabilization
Concurrency	Emergent architecture evolving through iterative refinement	Explicit dependency structure via directed acyclic graph (DAG) with feature-level adaptation	Strong upfront hierarchical design with DAG-enforced dependencies and TLE-encoded structure
Architectural Discipline	Variable integration timelines; architecture emerges over time	High predictability for vertical slice completion and feature delivery	High predictability for architectural coverage and systematic layer completion
Predictability	Simple consumer applications, low-risk web/mobile projects	Enterprise applications requiring early end-to-end validation; safety-critical systems	Platform systems, distributed architectures, and deeply nested hierarchical data models
Ideal Use Cases			

**Note:** All three approaches can incorporate Agile sprint cycles and DevOps practices. PDFD and PBFd add formal structure (DAG, state machines, bounded refinement) while maintaining iterative development principles.

### 6.3. PBFd vs. Conventional Relational Models (Including PDFD)

This section analyzes the architectural behavior of PBFd, which introduces Three-Level Encapsulation (TLE) and bitmask-based hierarchy encoding within a relational database.

While both PBFd and conventional approaches (including PDFD's graph-oriented model and traditional normalized schemas) employ relational databases as their backend storage layer, they differ fundamentally in schema design and query execution patterns.

PDFD employs directed-graph feature isolation using conventional foreign-key relationships, whereas PBFd encodes hierarchical ancestry through TLE, enabling constant-time hierarchy resolution.

The performance advantages of specialized encoding techniques over traditional relational joins are well-documented in software architecture and database literature [53,111].

Table 58 summarizes the key architectural distinctions, and Section 5.3 presents the corresponding empirical performance results.

**Table 58.** Architectural characteristics of PBFd (TLE schema) versus conventional relational schema designs

Aspect	Conventional Relational Schema	PBFd with TLE Schema
--------	--------------------------------	----------------------

Hierarchy Representation	Foreign-key relationships; graph edges stored as references across tables	Bitmask encoding; child membership compressed into integer fields within parent columns
Hierarchy Resolution	Recursive queries or multi-hop joins ( $O(m \log n)$ for $m$ relationships with B-tree indexes)	Bitwise operations on encoded paths ( $O(1)$ per parent-child query)
Query Pattern	Multi-table joins traversing foreign keys	Single-table queries using bitwise predicates on bitmask columns
Scalability Approach	Functional or domain-based partitioning	Horizontal partitioning at grandparent level with independent TLE table instances
Relationship Storage Overhead	Foreign-key columns with supporting indexes ( $k$ bits per relationship)	Compact bitmask fields (1 bit per child node)
Update Operations	Multi-row INSERT/UPDATE/DELETE across related tables	Single-row bitwise updates within grandparent table cells

**Note:** TLE consolidates three hierarchical levels (Grandparent-Parent-Children) into a single table structure, eliminating inter-table joins while preserving relational ACID guarantees. Complexity comparisons assume bounded hierarchies where  $n \leq w$  (word size).

#### 6.4. Comparison with Modern Database Paradigms

Table 59 presents a comparative analysis of Pbfd and Pdfd relative to modern database paradigms, emphasizing how these methodologies address specific limitations through structured workflow and encoding techniques. These comparisons are grounded in both theoretical insights and empirical observations drawn from Section 5.

**Table 59.** Comparative analysis of Pbfd and Pdfd relative to modern database paradigms.

Approach	Strengths	Weaknesses	How Pbfd/Pdfd Address These
Relational	ACID compliance, mature tooling, strong consistency guarantees	Recursive joins required for hierarchies ( $O(n \log n)$ ); poor native hierarchy support	<b>TLE architecture:</b> Eliminates recursive joins via bitmask-encoded parent-child relationships, achieving $O(1)$ hierarchy queries while preserving ACID guarantees
Graph (Neo4j)	Natural hierarchy traversal and relationship queries [113]	High storage overhead for edge metadata; lacks formal schema discipline	<b>PDFD/Pbfd structure:</b> Enforces formal DAG-based schema with explicit dependency management; <b>TLE encoding:</b> Reduces edge storage via compact bitmask representation
Document Stores (MongoDB)	Schema flexibility; embedded document hierarchies	No formal hierarchy guarantees; inconsistent nested structure	<b>PDFD/Pbfd methodology:</b> Provides formal hierarchical validation and state machine guarantees; <b>TLE pattern:</b> Enforces consistent three-level structure with verified state transitions
XML Databases	Native tree queries via XPath/XQuery [114]	Slow updates due to DOM manipulation; poor horizontal scalability	<b>TLE implementation:</b> Single-row atomic updates via bitwise operations; <b>Pbfd partitioning:</b> Horizontal scaling through grandparent-level table distribution
Columnar Stores (Cassandra)	High-performance batch reads; excellent write throughput [52]	Weak transaction guarantees; limited join support	<b>Hybrid TLE architecture:</b> Combines relational ACID guarantees with columnar-style fixed-width encoding; achieves transactional safety with efficient batch processing

**Note:** Pbfd and Pdfd are development methodologies that can leverage various database backends. TLE (Three-Level Encapsulation) is the specific encoding pattern that enables efficient hierarchical operations when implemented over relational systems, combining the structural benefits of specialized databases with relational ACID guarantees.

#### 6.5. Comparison to Traditional Bitmap Indexing

While PBFDD leverages bitmask encoding, its application differs significantly from traditional bitmap indexing techniques, as outlined in Table 60. Traditional bitmap indexing is primarily optimized for low-cardinality columns in data warehouse environments [115], whereas PBFDD's approach is designed specifically for hierarchical data relationships.

**Table 60.** Comparison of PBFDD's bitmask encoding and traditional bitmap indexing for hierarchical data.

Aspect	Traditional Bitmap Indexing	PBFDD Bitmask Encoding
Primary Purpose	Query optimization for filtering low-cardinality columns [115]	Hierarchical relationship representation and traversal
Granularity	One bitmap per distinct attribute value across all rows	One bit per child node within each parent's bitmask
Hierarchy Awareness	None; operates on flat attribute values only	Native support for multi-level hierarchies via Three-Level Encapsulation (TLE)
Storage	Separate bitmap for each distinct value (external index structure)	Bitmasks embedded within parent rows (one bitmask column per parent type)
Query Pattern	Accelerates WHERE clauses on indexed columns via bitmap operations	Enables O(1) parent-child membership queries via bitwise tests
Use Case	Data warehouse filtering on low-cardinality dimensions	Hierarchical data compaction and constant-time relationship traversal

#### 6.6. Comparison to Multi-Column or Multi-Row

PBFDD's bitmask encoding per parent offers advantages over traditional multi-column or multi-row approaches for representing hierarchical selections, as detailed in Table 61. The storage efficiency benefits align with principles from column-oriented database systems that optimize for specific query patterns [53].

**Table 61.** Comparison of PBFDD bitmask encoding with multi-column and multi-row relational approaches for hierarchical data representation.

Aspect	Multiple Columns	Multiple Rows	PBFDD Bitmask Encoding
Storage Footprint	High: separate column for each child node (e.g., $n$ columns for $n$ children)	High: one row per selected child, requiring foreign keys and indexes	Compact: single integer field per parent (1 bit per child; $n \leq 64$ fits in 64-bit word)
Query Complexity	$O(n)$ column scans to check all children	$O(n)$ joins or subqueries to aggregate selections	$O(1)$ bitwise tests for membership checks (for $n \leq w$ )
Update Operations	$O(n)$ column updates for batch changes	$O(n)$ INSERT/DELETE operations for relationship changes	$O(1)$ bitwise operations (OR, AND, XOR) for atomic updates
Scalability	Schema changes required to add new children (DDL operations)	Join complexity increases with relationship count	Bounded by word size $w$ (typically 64); extensible to $O(\lceil n/w \rceil)$ for $n > w$ via multi-word encoding
Schema Flexibility	Rigid: requires DDL for each new child	Flexible: new relationships via INSERT	Semi-flexible: bounded by bitmask capacity; requires column type upgrade for $n > w$

**Note:** Complexity assumes bounded hierarchies where  $n \leq w$  (word size, typically 64 bits). For  $n > w$ , PBFDD bitmask operations scale to  $O(\lceil n/w \rceil)$  with minimal constant factor overhead.

#### 6.7. Key Takeaways: Advancing FSSD with Directed Graph-Based Methodologies

PDFD and PBFDD apply directed graph structuring to Full-Stack Software Development (FSSD), providing clear management of complex, non-linear dependencies and hierarchies. This represents a shift from traditional emergent architecture toward more intentional, structured approaches to software design [65]. While PDFD focuses on depth-first, feature-oriented development, PBFDD

applies pattern-based, level-wise progression to support modularity and scalability in layered systems.

The following key takeaways summarize the comparative benefits and positioning of PDFD and PBFD:

- **Methodological Fit:** PBFD excels in layered or dependency-driven domains (e.g., claims processing, product taxonomies), while PDFD suits feature-centric, quick end-to-end testing needs consistent with the iterative, feature-focused delivery principles of Extreme Programming [110].
- **Complexity Management:** Both reduce maintenance burdens by decoupling dependencies and enforcing structure, addressing common software evolution challenges [111].
- **Adoption Potential:** Their conceptual clarity facilitates onboarding and modular scaling, supporting integration into low-code and DSL-based workflows.
- **Scalability:** Empirical results confirm stability at large user scales, affirming their suitability for evolving, long-lived systems.

Together, PBFD and PDFD advance FSSD by combining rigor, modularity, and performance in managing deeply structured data.

### 6.8. Limitations of PDFD and PBFD

Despite their advantages, both methods introduce specific challenges that align with known adoption barriers for structured methodologies [112]:

- **Learning Curve:** Understanding bitmasks (PBFD) or state transitions and directed graph slicing (PDFD) can be nontrivial for teams used to traditional relational models.
- **Tooling and Middleware:** PBFD may require custom middleware to support cross-shard aggregation of TLE-encoded bitmasks. Both PBFD and PDFD rely on dependency- or hierarchy-aware tooling to manage their underlying traversal graphs (e.g., DAG slicing in PDFD and TLE-based parent-child graph navigation in PBFD).
- **Model Rigidity:** PDFD assumes well-isolated features; PBFD assumes a relatively stable hierarchy—both may be challenged in dynamic, unstructured domains (e.g., social graphs).
- **Initial Overhead:** Upfront modeling and pattern definition require more investment than ad hoc FSSD approaches, consistent with the trade-offs of plan-driven methodologies [111].

In summary, PBFD and PDFD effectively bridge critical gaps in the management of complex hierarchical data by offering a unique combination of performance, scalability, and storage efficiency as demonstrated in our empirical evaluation. Table 62 encapsulates the key benefits of these two approaches.

**Table 62.** Comparative synthesis of PDFD and PBFD benefits across development velocity, runtime scalability, rigor, and architectural clarity

Benefit	PDFD	PBFD
Development Velocity	Enables early completion of fully functional vertical feature slices	Accelerates development via pattern-driven modularity and level-wise batch processing
Scalability	Supports independent scaling of modular feature slices	Supports horizontal partitioning at the TLE grandparent level, enabling distributed processing [53]
Rigor and Quality	Enforces formal state transitions with bounded refinement cycles ( $R_{\max}$ ) ensuring termination	Combines pattern-level validation with bounded refinement cycles ( $R_{\max}$ ), ensuring both horizontal coverage and vertical correctness
Architectural Clarity	Enforces explicit feature boundaries and dependency structures via directed acyclic graphs	Enforces layered hierarchical design via directed graphs and Three-Level Encapsulation (TLE), aligning with architectural modularity principles [65]

**Note:** Both methodologies share core guarantees (bounded refinement, formal verification, DAG-based structure) but differ in traversal strategy: PDFD prioritizes depth-first feature completion while PBFD emphasizes breadth-first pattern coverage with selective depth-first elaboration.

## 7. Discussion

This section interprets the study's findings, contextualizes their implications, outlines limitations, and proposes directions for future research.

### 7.1. Significance of the Study

This work addresses a critical gap in formalizing and rigorously engineering data-driven Full-Stack Software Development (FSSD) workflows. Its significance lies in providing a unified formal and practical framework that introduces novel capabilities for complex, scalable, and reliable FSSD systems.

Theoretically, we advance FSSD by applying graph-theoretic constructs (e.g., directed graph-based workflows in PDFD) and state machine models (e.g., Three-Level Encapsulation in PBFD). This formalization offers a rigorous, provably correct foundation for FSSD, enabling deterministic control over traversal, validation, and refinement—a capability largely absent in traditional approaches. Formal verification using CSP and LTL [45,46,116,117] further establishes guarantees on correctness, termination, and safety properties.

Methodologically, PBFD and PDFD define novel graph-based methodologies operationalizing this framework. They offer systematic, predictable strategies that mitigate risks of emergent development. The bitmask-based TLE fundamentally transforms hierarchical data management, achieving  $O(1)$  ancestor-descendant lookups and substantial storage and index reductions compared to multi-join traversals, while maintaining full architectural compatibility with relational systems. This approach aligns with established database design principles that emphasize efficient data organization and access as a cornerstone of system performance [54,118].

Empirically, the study provides compelling validation through open-source MVPs and an eight-year enterprise deployment. We demonstrate a substantial reduction in development effort ( $\geq 20\times$  faster than commercial alternatives) and significant performance improvements (7–8 $\times$  faster queries, 11.7 $\times$  storage reduction).

Practically, these outcomes substantiate our theoretical underpinnings and establish new benchmarks for highly scalable, reliable, and maintainable full-stack systems. The exceptional long-term system stability (zero critical defects supporting 100K+ users) and its efficacy in legacy modernization underscore its real-world impact.

In summary, this study unifies theoretical, methodological, and practical contributions to FSSD, linking formal models, engineering procedures, and empirical validation in a single coherent framework.

### 7.2. Mechanisms Underpinning PBFD and PDFD Efficiency

Our case study analysis (Section 5; Appendices A.11 and A.14) identifies three principal design factors that influence the development and operational performance of PDFD and PBFD:

1. **Graph-Based Abstraction for Business Logic:** Modeling business processes as directed graphs (Figures 3 and 16) profoundly reduced cognitive load and streamlined development, leading to over 20 $\times$  speedup compared to conventional tools (Table 54, Appendix A.20) [119].
2. **Context Consistency in Sequential Development:** Disciplined sequential development across refinement layers minimized context switching and cross-module regressions (Appendices A.11 & A.14), improving modular testability and reducing verification cycles [120].
3. **Encoded Data Optimization:** The combination of Three-Level Encapsulation (TLE) and bitmask techniques (Section 4) yielded substantial space savings (11.7 $\times$  compression; Appendix A.22) and dramatically improved lookup speed ( $O(1)$  complexity, Table 61). The efficiency gains from such

encoding are a well-understood principle in database systems, where optimized data structures are critical for high-performance query execution [53,55]. The use of bitmask techniques in PBFDF aligns with established indexing strategies such as bitmap indexes, which are widely used in data warehouses to accelerate query processing over low-cardinality columns [54].

### 7.3. Early Adoption Challenges for PBFDF

Initial PBFDF adoption faced resistance from database teams due to its unconventional structure (e.g., absence of junction tables) and limited early documentation. These barriers were gradually overcome through targeted onboarding and live demonstrations. This experience underscores that integrating formal methodologies into enterprise workflows is not solely a technical challenge—it is also an educational one, requiring accessible reference guides, intuitive tooling, and sustained developer engagement [41,121].

### 7.4. Adapting TLE to Non-Relational Database Systems

While TLE and bitmask-based hierarchical encoding are implemented and validated on relational platforms in our MVP and enterprise deployment, the underlying conceptual principles may be adaptable to other storage paradigms. However, the specific performance guarantees ( $O(1)$  operations,  $11.7\times$  storage reduction) demonstrated in Section 5 are tied to the relational implementation and require empirical validation in other contexts.

Graph databases (e.g., Neo4j, Amazon Neptune) natively support hierarchical traversal [113], potentially making TLE's encoding layer unnecessary. Document stores (e.g., MongoDB) offer flexible schemas [90] but lack columnar structure. Key-value stores may enable optimizations beyond relational word-size constraints. This direction aligns with trends toward polyglot persistence and application-specific data modeling [118].

Table 63 outlines preliminary conceptual mappings for cross-paradigm investigation. These mappings are speculative and require prototyping and benchmarking to determine whether TLE's benefits transfer to these paradigms.

**Table 63.** Preliminary mappings of TLE concepts for cross-paradigm investigation (speculative; requires empirical validation)

Data Model	Proposed TLE Mapping	Key Research Question
Document Database (MongoDB)	Collection $\rightarrow$ Document $\rightarrow$ Nested bitmask fields	Do MongoDB's bitwise operators ( <code>\$bitsAllSet</code> ) provide query advantages over array-based flags, or do index scan costs outweigh storage benefits in row-oriented BSON?
Key-Value Store (Redis)	Key namespace prefix $\rightarrow$ Structured keys $\rightarrow$ Bitmask values	Why does user $\rightarrow$ bitmask fail for cohort queries, and how does permission $\rightarrow$ bitmap achieve $O(1)$ filtering with BITOP operations?
Graph Database (Neo4j)	Node labels $\rightarrow$ Node instances $\rightarrow$ Properties with bitmasks	When do bitmask properties undermine index-free adjacency, and how do native edges preserve traversal performance?

Formalizing these mappings and conducting comparative benchmarking across paradigms represent essential future research directions. Such studies would establish the generality of TLE's design principles, identify paradigm-specific performance trade-offs, and provide evidence-based guidance for practitioners selecting optimal platforms for hierarchical data processing at scale [90,113]. Until such empirical work is completed, TLE's benefits remain proven only in relational systems.

### 7.5. Relational Constraints and Design Trade-Offs in PBFDF Deployments

PBFDF's relational implementation favors structural determinism over schema flexibility. Its Three-Level Encapsulation (TLE) replaces conventional junction tables with bitmask-encoded relationship fields, enabling constant-time hierarchy resolution within a compact, fixed schema. By

removing multi-table joins and recursive queries, PBFD transforms relational traversal from  $O(n)$  joins to  $O(1)$  bitwise evaluations, yielding predictable and efficient execution paths.

This optimization introduces deliberate constraints. Because hierarchical relationships are encoded rather than dynamically modeled, schema evolution requires controlled restructuring, limiting runtime flexibility. Likewise, PBFD delegates integrity management and relationship validation to application-level logic, minimizing reliance on stored procedures or foreign-key constraints.

Despite these restrictions, PBFD remains fully compatible with native SQL query planners and indexing mechanisms. Its deterministic schema structure supports cost-based optimization and stable execution plans, aligning with the principle that physical design must directly support the logical data model and workload characteristics to achieve efficiency [54,118].

### 7.6. Study Limitations

This study is constrained by a limited number of in-depth case implementations. Comprehensive quantitative comparisons between PBFD/PDFD and traditional FSSD (e.g., latency, throughput) remain underexplored. Future work must prioritize systematic, controlled benchmarking under varied operating conditions—including workload diversity, concurrency levels, and schema complexity—for broader generalization [122,123].

### 7.7. Unexpected Benefits

Beyond primary objectives, post-deployment feedback revealed unanticipated benefits. PBFD's clear separation of OLTP and OLAP workflows significantly improved operational clarity, streamlined data pipeline management, and enhanced reporting flexibility. This successful separation of concerns resonates with established database design practices for managing complex, high-throughput systems [54,118]. These advantages were particularly pronounced in large-scale claims processing, enabling cleaner architectural segregation and improved system resilience.

### 7.8. Additional Future Research Directions

Additional future research can further extend PBFD and PDFD's impact and applicability:

- **Domain Generalization:** Extend methodologies to other contexts (e.g., ETL, BI, rules engines) by mapping abstract nodes to domain primitives and refining traversal semantics
- **Distributed and Modular Systems:** Investigate utility in microservice and edge computing, focusing on runtime synchronization, orchestration, and modular validation
- **Tooling and Developer Ecosystem:** Develop companion tooling (e.g., IDE plugins, visualizers) to translate abstract process models into accessible engineering workflows
- **Rigorous Empirical Validation:** Conduct controlled comparative studies against conventional methods across performance, scalability, maintainability, and defect density. Future empirical work could build upon the comprehensive frameworks for evaluating database system performance as laid out in standard texts [54,118]

This study positions PBFD and PDFD as formally grounded, empirically validated alternatives for FSSD. Despite initial adoption barriers and relational trade-offs, they demonstrate robust performance, maintainability, and efficiency in production. By generalizing these algorithms, enhancing developer tooling, and expanding empirical validation, future research can establish PBFD and PDFD as foundational paradigms for scalable, formally grounded software engineering.

## 8. Conclusion

This paper introduces Primary Breadth-First Development (PBFD) and Primary Depth-First Development (PDFD)—formally grounded methodologies that address Full-Stack Software Development's persistent challenges in dependency management, hierarchical data efficiency, and cross-layer coordination. Built upon four foundational models (Directed Acyclic Development,

Depth-First Development, Breadth-First Development, and Cyclic Directed Development), these approaches integrate graph traversal strategies, state machine workflow models, and bitmask-encoded data structures to provide rigorous foundations for hierarchical system development.

**Theoretical Contributions.** PBFD and PDFD extend classical graph traversal with hybrid strategies offering provable termination under bounded refinement ( $R_{\max}$ ) and formal guarantees including deadlock freedom, dependency preservation, and finalization invariance. These properties are validated through Communicating Sequential Processes (CSP) and Linear Temporal Logic (LTL) specifications, with verification via FDR4 model checking. The Three-Level Encapsulation (TLE) pattern enables  $O(1)$  hierarchical operations through bitmask encoding, with complexity bounds proven in Theorems A.10.1–A.10.4. and operational correctness verified through CSP failures-divergences refinement.

**Empirical Validation.** An eight-year production deployment of PBFD demonstrates exceptional reliability (zero critical failures) with substantial performance gains: over  $20\times$  faster development cycles,  $7\text{--}8\times$  faster query execution, and  $11.7\times$  storage reduction. These results, established through longitudinal observational studies, quasi-experimental runtime comparisons, and controlled schema-level experiments, confirm that formally verified, graph-based development can deliver measurable improvements in enterprise systems. Publicly available Minimum Viable Products ensure reproducibility and practical accessibility.

**Broader Impact.** This work demonstrates that formal methods can enhance rather than hinder industrial software practice. PBFD and PDFD provide a practical pathway for modernizing hierarchical enterprise systems with provable correctness while achieving significant performance improvements. The successful eight-year deployment establishes that verification-driven development and industrial pragmatism are not opposing forces but complementary approaches to building reliable, scalable systems.

**Future Directions.** Key research avenues include cross-paradigm generalization (NoSQL, graph databases), automated tooling for pattern-driven development, and expanded empirical evaluation across diverse enterprise contexts. By advancing the rigor, efficiency, and scalability of complex system development, PBFD and PDFD lay groundwork for broader adoption of formally grounded methodologies in industrial software engineering.

**Acknowledgments:** The author gratefully acknowledges the support of IBM managers Jen Kostenko, Ricardo Zavaleta Cruz, and Anton Cwu for facilitating the publication process and for reviewing and authorizing the inclusion of the enterprise deployment case study materials used in this work. Portions of this manuscript benefited from AI-assisted editing tools used solely to improve clarity, consistency, formatting, and code debugging. All conceptual contributions—including research design, ideas, interpretations, analyses, and conclusions—are entirely the author's own.

**Data Availability Statement:** All non-proprietary data supporting the findings of this study are openly available. MVP implementations, formal specifications (CSP/CSPM models), validation datasets, and supplementary materials are available at <https://github.com/IBM-Consulting-Formal-Methods>. Additional detailed results, transition tables, and validation outcomes are provided in the manuscript appendices. The raw enterprise deployment data from the eight-year IBM case study is proprietary and cannot be publicly released due to client confidentiality agreements; the experimental environment, aggregated performance metrics, and a representative high-level technical architecture are included in the manuscript.

**Author Contributions:** Conceptualization, D.L.; methodology, D.L.; software, D.L.; validation, D.L.; investigation, D.L.; writing—original draft preparation, D.L.; writing—review and editing, D.L.; visualization, D.L. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Conflicts of Interest:** The author is an employee of IBM Consulting and declares inventorship of PBFDF and PDFDF.

## Appendix A

### Appendix A.1. Formal Notation and Semantic Symbols

This appendix defines the logical and algebraic notations used throughout the formal models of Directed Acyclic Development (DAD), Breadth-First Development (BFD), Depth-First Development (DFD), Cyclic Directed Development (CDD), Primary Depth-First Development (PDFD), and Primary Breadth-First Development (PBFDF).

**Table A.1.1.** Logical and Temporal Operators

Symbol	Meaning
$\Box\varphi$	Always $\varphi$ (globally true) – “Globally” in LTL
$\bigcirc\varphi$	Next state $\varphi$ – $\varphi$ will be true in the very next state
$\diamond\varphi$	Eventually $\varphi$ – $\varphi$ will be true at some future time
$\varphi \Rightarrow \psi$	Implication – if $\varphi$ holds, then $\psi$ must also hold
$\neg\varphi$	Negation – $\varphi$ does not hold
$\varphi \wedge \psi$	Conjunction – both $\varphi$ and $\psi$ hold
$\varphi \vee \psi$	Disjunction – at least one of $\varphi$ or $\psi$ holds
$<_{\text{lex}}$	Lexicographical comparison. The operator evaluates if the tuple on the left is strictly less than the tuple on the right. Comparison proceeds from left to right, element by element.

**Table A.1.2.** Quantifiers and Set-Based Expressions

Expression	Meaning
$\forall x \in X$	Universal quantifier: for all $x$ in set $X$
$\exists x \in X$	Existential quantifier: there exists $x$ in set $X$
$\nexists$	There does not exist (e.g., no cycles, no path)
$X \subseteq Y$	Set inclusion: $X$ is a subset of $Y$
$X \setminus Y$	Set difference: elements in $X$ but not in $Y$

**Table A.1.3.** Process State Notation

Notation	Meaning
$P(n) = 0$	Node $n$ is unprocessed
$P(n) = 1$	Node $n$ is in progress
$P(n) = 2$	Node $n$ is fully processed and validated
processed( $n$ )	$P(n)=1$ or $P(n)=2$
validated( $n$ )	$P(n) = 2$
finalized( $n$ )	$P(n) = 2$ . Used interchangeably with validated( $n$ )

This table defines fundamental concepts from graph theory and universal mathematical properties used throughout the methodologies.

**Table A.1.4.** General / Mathematical Definitions

Term	Definition / Description
$G=(V,E)$	A Directed Acyclic Graph (DAG) with vertex set $V$ and edge set $E$
children( $v$ )	The set of direct successor nodes to node $v$ in the graph or tree
$D(v)$	Direct dependencies of node $v$ : the set of nodes $u$ such that there is a directed edge from $u$ to $v$ (i.e., $\{u \mid (u,v) \in E\}$ )
$Tr$	Rooted, finite, acyclic tree structure with nodes $V$ and edges $E$
$C_i$	The current node being processed in the traversal

$B_j$	A backtrack point (a node on the current path with unvisited siblings)
$Q$	Global queue tracking nodes to process
$N_k$	Set of nodes at level $k$
$I_k$	Incremental delivery milestone $k$ , representing a validated subset of the system
$F_k$	Feedback trigger mechanism (e.g., validation failure, stakeholder input) associated with milestone $k$
$\text{depth}(v)$	The length of the longest path from a root node to node $v$
$\text{ancestors}(v)$	The set of all nodes from which node $v$ is reachable in the graph (i.e., $\{u \in V \mid \text{there exists a path from } u \text{ to } v\}$ )
$\text{descendants}(v)$	The set of all nodes reachable from node $v$ in the graph (i.e., $\{u \in V \mid \text{there exists a path from } v \text{ to } u\}$ )
$\text{level}(k)$	The set of all nodes at a specific depth $k$ in a tree or layered graph (i.e., $\{v \in V \mid \text{depth}(v)=k\}$ )
$\text{Path}(v)$	A directed path from a root node to node $v$
$\text{state}(B_j)$	A function mapping node $B_j$ to its processing state
$\text{Subtree}(B_j)$	All descendants of node $B_j$
$\text{invalid}(s)$	True if state $s$ violates the state machine constraints or invariant conditions
$\text{ReachableStates}$	The set of all states reachable from the initial state through legal transitions
$\text{follows\_rules}(t)$	True if the transition $t$ complies with the transition rules
$\text{consistent}(n, a, d)$	True if node $n$ is consistent with its ancestor $a$ and descendant $d$ in terms of structure/data
$\text{valid\_state}(s)$	A state is considered valid if and only if it is not $\text{invalid}(s)$
$\text{succ}(L)$	Returns the successor level to $L$
$\text{pred}(L)$	Returns the predecessor level to $L$
$\text{Next}(\text{level})$	Returns the logically next level from the current level (e.g., level + 1), capped at the maximum depth $L$ . Used for sequential level progression
$\text{Pattern}_i$	A formal model: a cohesive, feature/function-grouped subset of nodes (comprising data, logic, and UI artifacts) at hierarchical level $i$ , encapsulating a distinct unit of business logic or system functionality (See Section 3.4.2 for detailed discussion)
$\text{roots}(G)$	The set of root nodes in graph $G$ : $\{v \in V \mid \neg \exists u: (u,v) \in E\}$
$\text{leaves}(G)$	The set of leaf nodes in graph $G$ : $\{v \in V \mid \neg \exists u: (v,u) \in E\}$
$L$	The maximum depth of the graph/tree hierarchy: $\max\{\text{depth}(v) \mid v \in V\}$
$[P]$	Iverson bracket: $[P] = 1$ if predicate $P$ is true, 0 otherwise
$\text{bitmask}$	Binary representation of child relationships under a parent, supporting constant-time access

This table serves as a central reference, defining the fundamental predicates, functions, and constants utilized in the formal specifications and particularly in the transition conditions across all methodologies.

**Table A.1.5.** Core Definitions for Formal Methodologies: Predicates, Functions, and Constants

Term	Type	Description	Methodologies
$\text{processed}(n)$	Predicate	Evaluates to True if node $n$ has undergone its core processing or development action	DAD, DFD, BFD, CDD
$R_{\max}$	Constant	The maximum number of refinement attempts allowed for any specific level or pattern before an error state is triggered	PDFD, PBFD
$J_i$	Constant	Start of refinement: Earliest level impacted by failures at $i$ , where $J_i = \text{trace\_origin}(i)$	PDFD, PBFD
$R_i$	Constant	Refinement range: The number of levels to reprocess, calculated as $R_i = i - J_i + 1$ (bounded by $L$ )	PDFD, PBFD

$K_i$	Constant	Progression Threshold: Minimum finalized nodes ( $P(n)=2$ ) at level $i$ required before advancing to $i+1$ . Acts as a configurable WIP limit enforcing structured synchronization points	PDFD, PBFD
$r_j$	Constant	Current refinement attempt index for Pattern $_j$	PDFD
Reset( $n$ )	Predicate	Evaluates to True if node $n$ 's processing status or validation state is reverted, requiring re-evaluation or re-processing.	PDFD, PBFD
refinement_attempts( $j$ )	Counter	Tracks the number of refinement attempts for a specific level/pattern $j$ . Resets when a new refinement cycle begins	PDFD, PBFD
trace_origin( $i$ )	Function	Determines the root cause level $J_i$ (or pattern $J_i$ ) based on a validation failure detected at level $i$	PDFD, PBFD
trace( $i$ )	Function	The path or sequence of levels leading to level $i$ , used to constrain progression and ensure bounded advancement	PDFD
selected_subtree	Set	The subset of nodes selected for processing within a level or pattern, constrained by trace and eligibility criteria	PDFD
max_batch_size	Constant	The maximum number of nodes that can be processed in a single batch within a level	PDFD
validated( $n$ )	Predicate	Evaluates to True if node $n$ has successfully passed all its associated validation criteria	DFD, BFD, CDD, PDFD, PBFD
critical( $n$ )	Predicate	True if node $n$ requires vertical processing (children must be processed)	PBFD
start( $i$ )	Pseudocode	Initial state transition (idle $\rightarrow$ active)	DAD, DFD, BFD, CDD
terminate( $i$ )	Pseudocode	Terminal state (all nodes processed)	DAD, DFD, BFD
refine( $c$ )	Function	A node that needs iterative improvement.	CDD
finalize( $i$ )	Function	Finalizes a single node	CDD
processing_complete( $i$ )	Predicate	Evaluates to True when processing at level $i$ is complete	PDFD
refining( $j$ )	Predicate	True when the system is executing a refinement cycle targeting level $j$ (state = $S_1(j) \wedge$ refinement_attempts( $j$ ) > 0)	PDFD, PBFD
affected_nodes( $j$ )	Function	Returns the set of nodes $\{n \in G \mid \exists k \in [j, L]: n \in \text{level}(k)\}$ that may be reset during refinement at level $j$	PDFD, PBFD
consistent( $n$ )	Predicate	True if node $n$ satisfies all internal consistency constraints and validation criteria specific to its domain	PDFD, PBFD
dependencies_satisfied( $n$ )	Predicate	True if node $n$ satisfies all architectural dependencies and interface contracts with related nodes	PDFD, PBFD
all_descendants_validated( $n$ )	Predicate	True if all descendant nodes of $n$ have been validated	PDFD, PBFD
processed_subtree( $n$ )	Function	Returns the set of nodes selected for processing in the subtree of $n$	PDFD, PBFD
dequeue( $v$ )	Predicate	True when node $v$ is dequeued for processing	DAD
process( $v$ )	Function	Initiates core processing for node $v$	DAD
select_critical_children(Pattern $_i$ )	Function	Returns a subset of $\cup_{n \in \text{Pattern}_i} \text{children}(n)$ selected based on critical path analysis, dependency ordering, and resource constraints. Ensures architectural coherence while allowing efficient progression, with remaining nodes handled in $S_4$ completion phase	PBFD
$k_1$ (unfinalized_nodes)	Function	Returns the count of nodes with $P(n) \neq 2$	PDFD, PBFD
$k_2$ (remaining_attempts)	Function	Returns $\sum_{j \in \text{ActiveLevels}} (R_{\max} - \text{refinement\_attempts}(j))$	PDFD, PBFD
$k_3$ (phase_ordinal)	Function	Maps state phases to ordinals: $S_0 = 4, S_1=3, S_2=2, S_3=1, S_4=0$	PDFD, PBFD

$k_4$ (intra_phase_progres s)	Function	Tracks progress within the current phase	PDFD, PBFD
M enabled_transition(s)	Function Predicat e	Lexicographic measure $M = (k_1, k_2, k_3, k_4)$ True if at least one transition is enabled in state s	PDFD, PBFD PDFD
eligible(n)	Predicat e	True if node n meets all local validation and architectural criteria, allowing it to be part of the set considered for the $K_i$ threshold in $S_2$ progression. (Implies validated(n) and consistent(n))	PDFD
Structural Invariants	Set/Term	The set of all fundamental structural properties required for correct termination, including: Global Consistency, Descendant Finalization Invariant, and dependencies_satisfied for all nodes	PDFD, PBFD
test_failed( $C_i$ )	Predicat e	True if testing of node $C_i$ fails	CDD
feedback_triggered( $C_i$ )	Predicat e	True if feedback is triggered for node $C_i$	CDD
refinement_complete( $C_i$ )	Predicat e	True if refinement of node $C_i$ is complete	CDD
refinement_failed( $C_i$ )	Predicat e	True if refinement of node $C_i$ fails	CDD
refinement_count( $C_i$ )	Counter	Tracks the number of refinements for node $C_i$	CDD
all_components_writ ten( $I_k$ )	Predicat e	True if all components in milestone $I_k$ are written	CDD
feedback_received( $I_k$ )	Predicat e	True if feedback is received for milestone $I_k$	CDD
validation_failed( $I_k$ )	Predicat e	True if validation of milestone $I_k$ fails	CDD
all_increments_valid ated	Predicat e	True if all increments are validated	CDD
validation_successfu l( $I_k$ )	Predicat e	True if validation of milestone $I_k$ is successful	CDD
initiate_workflow(Gr andparent)	Function / Operatio n	Starts the TLE workflow for a given grandparent unit (loads context, registers processing unit)	TLE
LOAD(Grandparent)	Operatio n	Atomic load of grandparent data and metadata into TLE context	TLE
resolve_hierarchy()	Function / Operatio n	Internal resolution that computes parent/child relationships and prepares traversal order	TLE
evaluate_children(Pa rent)	Predicat e / Operatio n	Iteratively evaluates each child of Parent for processing eligibility (reads child state, bitmask tests)	TLE
READ(Parent, Child)	Operatio n	Read access to Parent and Child data (used during evaluate_children)	TLE
update_required(Par ent, Child)	Predicat e	True iff a child/parent pair requires an update (e.g., bitmask change or state change)	TLE
apply_update(Parent , Child, State)	Operatio n	Apply the computed update to Parent/Child in-memory state (pre-commit)	TLE

persist_changes()	Operation	Flush pending updates to durable storage (pre-commit stage)	TLE
WRITE(Parent, Child, State)	Operation	Durable write of Parent/Child state (used when persisting updates)	TLE
COMMIT(Grandparent)	Operation	Commit the grandparent-level changes (atomic commit of bitmask / selection)	TLE
has_next_unit()	Predicate	True if there is another TLE processing unit (grandparent) to process in the workload	TLE
has_unprocessed_unit()	Predicate	True if there exists at least one grandparent unit not yet processed	TLE
finalize_process()	Operation	Finalize the overall TLE workflow (cleanup, release resources, produce summary)	TLE

Table A.1.6. State Machine Identifiers (Used in Tables and Diagrams)

State ID	Global Label	Description	Methodologies Using This State
S <sub>0</sub>	Initialization	The initial state, involving loading foundational structures (e.g., DAGs, trees, or graphs) and initializing necessary parameters, queues, or dependency structures	All (DAD, DFD, BFD, CDD, PDFD, PBFD, TLE)
S <sub>1</sub>	Active Processing	Represents the core development or processing phase where active work is performed on nodes, levels, or components (e.g., enqueueing, pushing, resolving patterns)	DAD, DFD, BFD, CDD
S <sub>1</sub> (i)	Current Pattern/Level	Indicates active processing of nodes within Pattern <sub>i</sub> or level i	PDFD, PBFD
S <sub>1</sub> (i+1)	Next Level/Pattern Progression	Processing of Pattern <sub>i+1</sub> or level i+1, typically derived from children of Pattern <sub>i</sub> or level i	PDFD, PBFD
S <sub>1</sub> (j)	Refinement Level	Reprocessing Pattern <sub>j</sub> or level j due to a validation failure detected in a later stage	PDFD, PBFD
S <sub>1</sub> (TLE)	Parent Batch Loaded	Indicates the parent node batch has been loaded and is ready for context-aware evaluation	TLE
S <sub>2</sub>	General Validation / Dependency Check/Refinement	A non-parameterized validation phase. Examples include verifying dependency completeness (DAD), backtracking to a parent node (DFD), validating an entire level (BFD), or refining nodes and levels (CDD)	DAD, DFD, BFD, CDD
S <sub>2</sub> (i)	Pattern/Level Validation	Validates the processed nodes within Pattern <sub>i</sub> or level i	PDFD, PBFD
S <sub>2</sub> (j)	Refinement Validation	Validates the reprocessed nodes in Pattern <sub>j</sub> or level j during an active refinement cycle	PDFD, PBFD
S <sub>2</sub> (TLE)	Context Established	Resolves grandparent-level context to support child node resolution and bitmask evaluation	TLE
S <sub>3</sub>	Graph Extension / Validation	General adaptation including node/edge addition and iterative design validation	DAD, DFD, CDD
S <sub>3</sub> (i)	Depth-Oriented Process / Resolution	Bottom-up subtree validation and subtree resolution before descent	PDFD, PBFD
S <sub>3</sub> (j)	Refinement Depth-Oriented Resolution	Refinement Depth Resolution - Load required data and resolve node implementation for Pattern <sub>j</sub> during refinement before descending or returning to the original context	PBFD
S <sub>3</sub> (TLE)	Ancestor Data Prepared	Loads ancestor-level metadata to support bitmask-based child node resolution	TLE

S <sub>4</sub>	Completion Phase	A top-down traversal phase used to finalize unprocessed nodes or patterns, ensuring full coverage and correctness prior to termination	PDFD, PBF
S <sub>4</sub> (i)	Level / Pattern Completion Phase	Completes all unprocessed nodes within Pattern <sub>i</sub> or level i during top-down finalization	PDFD, PBF
S <sub>4</sub> (TLE)	Children Evaluated	Child Node Evaluation via Bitmask Logic – Determines structural inclusion or filtering	TLE
S <sub>5</sub>	Error / Failure Termination	Triggered when validation or refinement fails irrecoverably, or R <sub>max</sub> (maximum refinement attempts) is exceeded	PDFD, PBF
S <sub>5</sub> (TLE)	Bitmask Committed	Ancestor-Level Bitmask Update – Writes finalized selection to ancestor or top-level structure	TLE
S <sub>6</sub> (TLE)	Traversal Finalized	Indicates that the traversal is complete and no further node evaluation remains for the current resolution pass.	TLE
T	Termination	The successful conclusion of all phases: all nodes, patterns, and components are validated and finalized. Applies to both flat and hierarchical methods, including hybrid workflows (PBF, PDF).	All (DAD, DFD, BFD, CDD, PDF, PBF, TLE)

This notation glossary corresponds to the CSPM models verified under FDR 4.2.7 (full specifications hosted in the project's GitHub repository).

**Table A.1.7.** Core CSP Operators Used in DAD, DFD, BFD, CDD, PBF, PDF, and TLE Formal Specifications.

Symbol	Meaning
->	Action Prefix / Event Sequencing: Defines sequential event occurrences where event a occurs then process P executes (Example: a -> P)
[]	External Choice: Allows environment selection between processes where either A or B can occur based on external input (Example: (event1 -> P1) [] (event2 -> P2))
;	Process Sequencing: Ensures process P completes (reaches SKIP) before process Q begins (Example: P ; Q)
SKIP	Successful Termination: Represents successful completion of an event or process
?	Input Parameter: Receives input from the environment for parameterized events (Example: ?node)
!	Output Parameter: Sends output to the environment for parameterized events (Example: !result)
[] x:S @ P	Indexed External Choice: Enables non-deterministic selection where the environment chooses any element from set S to initiate process P (Example: [] c:NodeID @ process_c)
STOP	Deadlock / Halt: Represents a blocked state where no events are possible
?x / !x	Channel Input / Output: Receives values via ?x or sends values via !x
if ... then ... else ...	Conditional Branching: Enables guard-based process selection
let ... within ...	Local Variable Assignment: Defines local variables for intermediate computation
RUN(A)	Infinite Acceptance: Accepts any event from alphabet A indefinitely
[T= P]	Trace Refinement: Verifies that process behavior conforms to specification P
\	Hiding: Makes specified events internal and unobservable
[  X  ]	Synchronized Parallel Composition: Executes two processes in parallel with required synchronization on events in set X while allowing independent execution of events outside X
~	Internal Non-deterministic Choice: Enables system-internal selection among multiple options without environment influence
	Interleaving / Independent Parallel: Executes processes independently without event synchronization

This table defines the core notation for the bitmask-based hierarchical data model.

**Table A.1.8.** Three-Level Encapsulation (TLE) Notation

Symbol	Meaning
--------	---------

$n$	Number of root entities (grandparent units)
$n_{max}$	Maximum number of children for any parent entity
$c\_id$	Identifier of a specific child within a parent bitmask; used for bitwise indexing
$P_i$	Variable number of parent entities for grandparent unit $i$
$P_{total}$	Total number of parent entities across all grandparents
$T_{query}$	Time complexity of a single lookup query (Theorem A.10.2)
$T_{update}$	Time complexity of a single update operation (Theorem A.10.3)
$T_{batch}$	Total time complexity of processing all relationships (Theorem A.10.4)
$C_j$	Variable bitmask size in bits for a parent entity $j$ (e.g., 8, 16, 32, 64, or $\text{varchar}(n)$ )
$k$	Bit length of a traditional foreign key used in the baseline relational representation
$m$	Total number of child relationships in the hierarchy
$\hat{c}$	The average number of children per parent across all parent entities
$\hat{C}$	The average bitmask size (in bits) across all parent entities
$w$	Machine word size used for bitmask storage (e.g., 64 for BIGINT)
$S_{TLE}$	Total storage size (in bits) required by the TLE model
$S_{traditional}$	Total storage size (in bits) required by the traditional foreign key representation
Grandparent	Root-level entity that encapsulates multiple parent entities and their hierarchical context
Parent	Intermediate entity that manages child relationships through bitmask-based selection
Child	Leaf-level entity evaluated for inclusion/exclusion via parent's bitmask logic

### Appendix A.2. DAD Mermaid Code, Algorithm, and Process Algebra

Appendix A.2 provides the formal specification for the Directed Acyclic Development (DAD) methodology, covering its Mermaid diagrams, pseudocode, and CSP model.

#### Appendix A.2.1. Structural Workflow Mermaid Code

##### graph TD

```
N1[Node1 Root]-->|Dependency|N2[Node2]; N1-->|Dependency|N3[Node3]
N2-->|Dependency|N4[Node4]; N3-->|Dependency|N4
N4-->|Dependency|N5[Node5]
```

```
legend["DAD Principles:<br>- Acyclicity<br>- Hierarchy<br>- Scalability"];
legendCore[Core]:::core; legendExtended[Extended]:::extended
```

```
classDef core fill:#E1F5FE,stroke:#039BE5;
classDef extended fill:#F0F4C3,stroke:#AFB42B;
classDef legend fill:#FFFFFF,stroke:#BDBDBD;
class N1,N2,N3,N4 core; class N5 extended; class legend legend
```

#### Appendix A.2.2. State Machine Mermaid Code

##### stateDiagram-v2

```
direction TB
[*] --> S0: DA1 - Load DAG
S0 --> S1: DAG Validated
S1 --> S2: DA2 - Validate Dependencies
S2 --> S1: DA3 - Dependencies Satisfied
S2 --> S3: DA4 - Missing Dependencies
S3 --> S1: DA5 - Extension Complete
S1 --> T: DA6 - All Nodes Processed
T --> [*]
```

## Appendix A.2.3. Algorithm (Pseudo Code)

**Algorithm DAD**

Procedure DAD( $G$ : DAG,  $v_1$ : Node)

Input:  $G$ , a Directed Acyclic Graph;  $v_1$ , its root node

Output: Fully processed DAG with validated dependencies

// State  $S_0$ : Initialization (Table 4)

// Transition DA1:  $S_0 \rightarrow S_1$  (Table 5)

1. LoadDAG( $G$ )

2. queue  $Q \leftarrow [v_1]$

// State  $S_1$ : Node Processing (Table 4) - Main DAD loop

3. While  $Q$  is not empty:

3a.  $v \leftarrow \text{Dequeue}(Q)$

3b. Process( $v$ )

// Transition DA2:  $S_1 \rightarrow S_2$  (Table 5) - Initiate dependency check

3c. ValidateDependencies( $D(v)$ )

// State  $S_2$ : Dependency Check (Table 4) - Logic for transitions from  $S_2$

// Transition DA3:  $S_2 \rightarrow S_1$  (Table 5) - All dependencies resolved

3d. If all\_u\_in\_Dv\_are\_processed( $v$ ): // Check if all direct dependencies of  $v$  are processed

3e. Enqueue(children( $v$ )) // Process children of  $v$  for next iteration

// Transition DA4:  $S_2 \rightarrow S_3$  (Table 5) - Missing dependencies detected

3f. Else: // If there are missing dependencies

// State  $S_3$ : Graph Extension (Table 4) - Extend DAG with missing node

3g. ExtendGraph( $v_{\text{new}}$ ) // Add new node  $v_{\text{new}}$  to resolve dependency

// Transition DA5:  $S_3 \rightarrow S_1$  (Table 5) - Extension complete

3h. Enqueue( $v_{\text{new}}$ ) // Enqueue new node  $v_{\text{new}}$  for future processing

// Transition DA6:  $S_1 \rightarrow T$  (Table 5) - Final validation and termination

4. FinalValidation() // Perform final validation and conclude workflow

// State  $T$ : Termination (Table 4)

// Algorithm ends here.

// --- Helper Functions (Detailed implementation omitted for conciseness)

// These functions operate on the graph  $G$  and implicitly manage a 'processed' set.

function all\_u\_in\_Dv\_are\_processed( $v$ ):

// Checks if all direct dependencies of node  $v$  are marked as processed.

function ExtendGraph( $v_{\text{new}}$ ):

// Adds a new node  $v_{\text{new}}$  and its necessary edges to the DAG,

// ensuring acyclicity is preserved.

function FinalValidation():

// Performs any final checks before termination, e.g.,

// ensuring all necessary nodes have been processed.

End Procedure

#### Appendix A.2.4. CSP Implementation and Formal Verification

The complete CSP model (CSPM syntax, FDR 4.2.7 compatible) implementing all operations from Algorithm A.2.3 and state transitions from Table 4 and Table 5 is available in our supplementary repository.

Verification Status: All 10 formal properties verified (deadlock-free, divergence-free, deterministic, correct sequencing for DA2-DA6).

Repository Access:

- GitHub: [https://github.com/IBM-Consulting-Formal-Methods/CDD\\_CSP](https://github.com/IBM-Consulting-Formal-Methods/CDD_CSP) (commit: 03b972d)

The model includes all processes (S0-S3) and events documented in Tables A.2.1-A.2.2. See repository README for verification instructions.

#### Appendix A.2.5. DAD (Directed Acyclic Development) Methodology Tables

The DAD methodology's formal specification is detailed through unified tables linking pseudocode and CSP models. Table A.2.1 defines terms and operations, while Table A.2.2 maps core CSP states and transitions directly to pseudocode lines and events.

**Table A.2.1.** DAD Methodology - Unified Definitions (Pseudocode + CSP)

Pseudocode Term	Type	Description	Pseudocode Lines	CSP Mapping
<b>Initialization</b>				
LoadDAG(G)	Function	Initializes the DAD process by loading the Directed Acyclic Graph structure G	1	load_dag_actual!g_initial
queue Q ← [v <sub>1</sub> ]	Function	Initializes the processing queue Q with the root node v <sub>1</sub>	2	initialize_queue_actual!v1_root
<b>Node Processing Loop</b>				
Q is not empty	Condition	True if the processing queue Q has no nodes (loop termination condition)	3	queue_not_empty
v ← Dequeue(Q)	Function	Removes and returns a node v from the front of the processing queue Q	3a	dequeue_actual!node
Process(v)	Function	Perform core processing action for node v	3b	process_actual!node
<b>Dependency Validation</b>				
ValidateDependencies(D(v))	Function	Verify completeness of v's dependencies	3c	validate_dependencies_actual!node
all_u_in_Dv_are_processed(v)	Condition	True if all direct dependencies of v are processed	3d	all_dependencies_processed!node
Enqueue(children(v))	Function	Add children of v to the queue for next iteration	3e	generate_children_actual!node / enqueue_nodes_actual!children(node)
<b>Graph Extension (Missing Dependencies)</b>				
Else (missing dependency)	Control	Handles unresolved dependencies	3f	missing_dependency!node
ExtendGraph(v_new)	Function	Add new node v_new and its necessary edges to the DAG to resolve dependency	3g	extend_graph_actual!node!v_new_param
Enqueue(v_new)	Function	Enqueue new node v_new for future processing	3h	enqueue_nodes_actual!{v_new}
<b>Termination</b>				

FinalValidation()	Function	Perform final validation and conclude workflow	4	perform_final_validation_actual
-------------------	----------	--	---	---------------------------------

Table A.2.2. DAD Methodology - CSP Process Algebra Core (States + Transitions)

CSP Process	Key Transitions	Pseudocode Lines	CSP Events
S0 (Initialization)	DA1: $\rightarrow S1$ (Load DAG & Init Queue)	1-2	load_dag_actual!g_initial, initialize_queue_actual!v1_root
S1 (Node Processing)	DA2: $\rightarrow S2$ ValidateOutcome(v) (Dequeue & Process)	3a-3c	queue_not_empty, dequeue_actual!node, process_actual!node, validate_dependencies_actual!node
S2ValidateOutcome(v)	DA3: $\rightarrow S1$ (Dependencies Processed)	3d-3e	all_nodes_processed, perform_final_validation_actual, all_dependencies_processed!node, generate_children_actual!node, enqueue_nodes_actual!(children(node))
S3ExtendCompletion(v_new)	DA4: $\rightarrow S3$ ExtendCompletion(v_new) (Missing Dependency)	3f-3g	missing_dependency!node, extend_graph_actual!node!v_new_program
S3ExtendCompletion(v_new)	DA5: $\rightarrow S1$ (Enqueue New Node)	3h	enqueue_nodes_actual!{v_new}
T_SUCCESS (Successful Termination)	N/A	N/A	terminate_successfully_actual
T_ERROR (Error Termination)	N/A	N/A	terminate_with_error_actual

#### Appendix A.2.6. Formal Verification Details for DAD Model and Guarantees

All verification checks were performed using FDR 4.2.7 with standard configuration:

- Compression: default behavioral reduction (e.g., diamond elimination, sbisim).
- Search order: Breadth-first exploration (default, ensures shortest counterexample discovery).
- The model state space was fully explored. Verification confirms tractability and correctness for all ten critical assertions.

##### Assertions 1–10

- Core safety and liveness (Assertions 1–3): Confirm predictable, non-blocking dependency-first traversal.
- Local processing and dependency control (Assertions 4–8): Enforce strict adherence to DA2–DA3 sequencing.
- Validation and termination (Assertions 9–10): Guarantee that traversal, final validation, and termination complete correctly.

#### Appendix A.3. DFD Mermaid Code, Algorithm, and Process Algebra

Appendix A.3 provides the formal specification for the Depth-First Development (DFD) methodology, covering its Mermaid diagrams, pseudocode, and CSP model.

##### Appendix A.3.1. Structural Workflow Mermaid Code

###### graph TD

```
%% Tree Structure
```

```

C1((C1)) --> C2_1((C21))
C1 --> C2_2((C22))
C1 --> C2_3((C23))
C2_1 --> C3_1((C31))
C2_2 --> C3_2((C32))
C2_3 --> C3_3((C33))
%% C3_3 and C3_4 are siblings of C2_3
C2_3 --> C3_4((C34))

%% Traversal Path with Backtracking and Sibling Processing
C1 -->|"1: Process C1"| C2_1
C2_1 -->|"2: Process C21"| C3_1
C3_1 -->|"3: Backtrack to C21"| C2_1
%% All children of C2_1 processed, backtrack
C2_1 -->|"4: Backtrack to C1"| C1
%% Go to next sibling of C2_1
C1 -->|"5: Process C22"| C2_2
C2_2 -->|"6: Process C32"| C3_2
C3_2 -->|"7: Backtrack to C22"| C2_2
C2_2 -->|"8: Backtrack to C1"| C1
C1 -->|"9: Process C23"| C2_3
C2_3 -->|"10: Process C33"| C3_3
C3_3 -->|"11: Backtrack to C23"| C2_3
%% Go to next sibling of C3_3 (under C2_3)
C2_3 -->|"12: Process C34"| C3_4
C3_4 -->|"13: Backtrack to C23"| C2_3
C2_3 -->|"14: Backtrack to C1"| C1
%% explicit termination node
C1 -->|"15: All nodes processed"| T((Terminate))

%% Legend with more distinct colors
subgraph Legend
  note[Superscripts like 1, 2, 3 indicate ordering of sibling nodes]
  L2[" "]::legendNode
  L2_text[Processed]
  L3[" "]::currentNode
  L3_text[Current]
  L4[" "]::pendingNode
  L4_text[Pending]
end

%% Connect legend elements
L2 --- L2_text
L3 --- L3_text
L4 --- L4_text

%% Styling with more distinct colors
classDef legendNode fill:#6495ED,stroke:#000,stroke-width:2px
classDef currentNode fill:#32CD32,stroke:#000,stroke-width:2px
classDef pendingNode fill:#FFF,stroke:#000,stroke-width:2px
classDef legendBox fill:#f9f9f9,stroke:#ccc,stroke-dasharray: 5 5

```

```

%% Color classes for tree nodes (adjust as needed for the visual representation of current
state)
class C1 legendNode
class C2_1,C3_1 currentNode
class C2_2,C2_3,C3_2,C3_3,C3_4 pendingNode
class Legend legendBox

%% Style text nodes to be transparent
classDef textNode fill:transparent,stroke:transparent
class L2_text,L3_text,L4_text,note textNode

```

### Appendix A.3.2. State Machine Mermaid Code

```

stateDiagram-v2
    direction TB
    [*] --> S0: Initialize
    S0 --> S1: DF1 - Load Tree & Init Stack

    S1 --> S1: DF2 - Process Child
    S1 --> S2: DF3 - Set Backtrack Point

    S2 --> S1: DF4 - Unprocessed Sibling
    S2 --> S3: DF5 - Validate Subtree

    S3 --> S2: DF6 - Backtrack
    S3 --> T: DF7 - Terminate

    T --> [*]

```

### Appendix A.3.3. Algorithm (Pseudo Code)

#### Algorithm DFD

Procedure DFD(T: Tree)

Input: T, a hierarchical tree with root node  $C_1$

Output: Validated and completed node set

```

// State S0: Initialization (Table 11)
// Transition DF1: S0 → S1 (Table 12)
1. LoadProject(T)           // Initialize project and tree structure
2. stack ← [C1]           // LIFO stack for Depth-First Search, initialized with root
3. Processed ← ∅           // Set to track processed nodes for validation and preventing re-
processing

// State S1: Vertical Processing (Table 11) - Main DFD loop
4. while stack is not empty:
    4a. C ← pop(stack)       // Dequeue the current node Ci for processing
    4b. Process(C)           // Perform core processing action for node Ci
    4c. Add C to Processed   // Mark node as processed

// Transition DF2: S1 → S1 (Table 12) - Move to child if non-leaf
// Transition DF3: S1 → S2 (Table 12) - Set backtrack point if leaf
4d. if C is a non-leaf:

```

```

// Push children for deeper traversal; next iteration processes a child
4e. push(reverse(children(C)), stack)
4f. else: // C is a leaf node
// State S2: Backtracking (Table 11) - Initiate backtracking from leaf
4g. Bj ← parent(C) // Set backtrack point to the parent of the processed leaf

// Loop represents returning to ancestor nodes for alternatives within S2
4h. while Bj is not null:
// Transition DF4: S2 → S1 (Table 12) - Process next sibling if it exists
4i. if has_unprocessed_sibling(Bj):
4j. push(get_unprocessed_sibling(Bj), stack) // Enqueue sibling
4k. break // Stop backtracking, return to S1 to process sibling

// Transition DF5: S2 → S3 (Table 12) - No alternatives, validate subtree
4l. else: // No alternative siblings at Bj
// Transition S2 → S3: DF5 - ValidateSubtree()
4m. ValidateSubtree(Bj) // Perform validation for the subtree rooted at Bj

// State S3: Validation (Table 11) - Decide next step after validation
// Transition DF7: S3 → T (Table 12) - Terminate if all nodes processed
4n. if stack is empty and no_more_backtrack_points_above(Bj): // Check if overall
traversal is complete
4o. Terminate() // Final termination
4p. return // Exit algorithm

// Transition DF6: S3 → S2 (Table 12) - More backtracking needed
4q. else: // Subtree validated, continue backtracking to next ancestor
4r. Bj ← parent(Bj) // Move to the next higher backtrack level

// Final termination if the main loop completes (all nodes processed)
5. Terminate()

// --- Helper Functions (Detailed implementation omitted for conciseness)

function has_unprocessed_sibling(node):
// Checks if 'node' has unprocessed siblings under its parent
// Requires access to 'Processed' set.

function get_unprocessed_sibling(node):
// Retrieves an unprocessed sibling of 'node'

function ValidateSubtree(node):
// Validates the subtree rooted at 'node'.
// Requires checking status of all nodes in subtree against validation criteria.

function no_more_backtrack_points_above(node):
// Returns true if there are no remaining ancestors or nodes on stack to process,
// indicating the overall traversal is not yet complete.
End Procedure

```

#### Appendix A.3.4. CSP Implementation and Formal Verification

The complete CSP model (CSPM syntax, FDR 4.2.7 compatible) implementing all operations from Algorithm A.3.3 and state transitions from Table 11 and Table 12 is available in our supplementary repository.

**Verification Status:** All 8 formal properties verified (deadlock-free, divergence-free, deterministic, correct sequencing for DF2-DF7)

**Repository Access:**

- GitHub: [https://github.com/IBM-Consulting-Formal-Methods/DFD\\_CSP](https://github.com/IBM-Consulting-Formal-Methods/DFD_CSP) (commit: b421b32)  
The model includes all processes (S0-S3, PushChildren) and events documented in Tables A.3.1-A.3.2. See repository README for verification instructions.

#### Appendix A.3.5. DFD (Depth-First Development) Methodology Tables

The DFD methodology's formal specification is further detailed through Table A.3.1, which provides a unified set of definitions for both the pseudocode and CSP models. Table A.3.2 then outlines the core CSP process algebra, detailing the state transitions and key events that correspond to the pseudocode.

**Table A.3.1.** DFD Methodology - Unified Definitions (Pseudocode + CSP)

Pseudocode Term	Type	Description	Pseudocode Lines	CSP Mapping
<b>Initialization</b>				
LoadProject(T)	Function	Initializes tree structure	1	load_tree_actual!t_initial
stack $\leftarrow$ [C <sub>1</sub> ]	Function	Initializes DFS stack	2	initialize_stack_actual!c_root
<b>Node Processing Loop</b>				
stack is not empty	Condition	Loop continuation	4	stack_not_empty!c
stack is empty	Condition	Termination check	4	stack_is_empty
C $\leftarrow$ pop(stack)	Function	Pops node from stack	4a	dequeue_actual!c
Process(C)	Function	Core processing	4b	dequeue_actual!c
Add C to Processed	Operation	Mark node as processed	4c	Tracked in processed set parameter
<b>Non-Leaf Processing</b>				
C is a non-leaf	Condition	Node has children	4d	is_non_leaf!c
push(reverse(children(C)), stack)	Function	Push children for DFS traversal	4e	process_child_actual!c $\rightarrow$ push_children_actual!c $\rightarrow$ PushChildren process
<b>Leaf Processing &amp; Backtracking</b>				
C is a leaf	Condition	Node is leaf	4f	is_leaf!c
B <sub>j</sub> $\leftarrow$ parent(C)	Function	Set backtrack point to parent	4g	set_backtrack_point_actual!parent(c)
B <sub>j</sub> is not null	Condition	Backtracking loop continuation	4h	Implicit in S2/S3 recursion
has_unprocessed_sibling(B <sub>j</sub> )	Condition	Check for unprocessed siblings	4i	has_unprocessed_sibling!b <sub>j</sub>
push(get_unprocessed_sibling(B <sub>j</sub> ), stack)	Function	Push sibling to stack	4j	get_unprocessed_sibling_actual! b <sub>j</sub> $\rightarrow$ push_sibling_actual!sibling
no alternative siblings at B <sub>j</sub>	Condition	No unprocessed siblings remain	4l	no_unprocessed_sibling!b <sub>j</sub>
ValidateSubtree(B <sub>j</sub> )	Function	Subtree validation	4m	validate_subtree_actual.B <sub>j</sub>
<b>Termination Checks</b>				

stack is empty and no_more_backtrack_p oints_above(B <sub>j</sub> )	Condition	Final termination check	4n	no_more_backtrack_points_above!b_j
Terminate()	Function	Final termination	4o, 5	terminate_successfully_actual
B <sub>j</sub> ← parent(B <sub>j</sub> )	Function	Backtrack upward to parent	4r	backtrack_to_actual!b_j!parent(b_j)

Table A.3.2. DFD Methodology - CSP Process Algebra Core (States + Transitions)

CSP Process	Key Transitions	Pseudocode Lines	CSP Events
S0 (Initialization)	DF1: →S1 (Load tree & initialize stack)	1-2	load_tree_actual!t_initial, initialize_stack_actual!c_root
S1 (Vertical Processing)	DF7: →T (Stack empty termination)	4,5	stack_is_empty, terminate_successfully_actual
	DF2: →S1 (Non-leaf processing)	4a-4e	stack_not_empty!c, dequeue_actual!c, process_actual!c, is_non_leaf!c, process_child_actual!c, push_children_actual!c, PushChildren process (iterates over children)
	DF3: →S2 (Leaf processing)	4a-4g	stack_not_empty!c, dequeue_actual!c, process_actual!c, is_leaf!c, set_backtrack_point_actual!parent(c)
S2(B <sub>j</sub> ) (Backtracking)	DF4: →S1 (Process unprocessed sibling)	4h-4j	has_unprocessed_sibling!b_j, get_unprocessed_sibling_actual!b_j, push_sibling_actual!sibling
	DF5: →S3 (No siblings, validate subtree)	4h, 4l-4m	no_unprocessed_sibling!b_j, validate_subtree_actual!b_j
S3(B <sub>j</sub> ) (Validation)	DF7: →T (Terminate at root)	4n-4o	no_more_backtrack_points_above.B <sub>j</sub> , terminate_successfully_actual
	DF6: →S2 (Continue backtracking upward)	4q-4r	subtree_validated.B <sub>j</sub> , backtrack_to_actual.parent(B <sub>j</sub> )
T (Termination)	Final state	5	terminate_successfully_actual

#### Appendix A.3.6. Formal Verification Details for DFD Model and Guarantees

All verification checks were performed using FDR 4.2.7 with standard configuration:

- **Compression:** default behavioral reduction (e.g., diamond elimination, sbisim)
- **Search order:** Breadth-first exploration (default, ensures shortest counterexample discovery)

The model state space was fully explored. Verification confirms tractability and correctness for all eight critical assertions.

##### Assertions 1–8

- Core safety and liveness (Assertions 1–3): Confirm predictable, non-blocking traversal
- Local processing and control flow (Assertions 4–6, 8): Enforce strict adherence to stack-based sequencing (DF2→DF3)
- Validation and termination (Assertion 7): Guarantee that traversal and validation complete before halting

#### Appendix A.4. BFD Mermaid Code, Algorithm, and Process Algebra

Appendix A.4 provides the formal specification for the Breadth-First Development (BFD) methodology, covering its Mermaid diagrams, pseudocode, and CSP model.

## Appendix A.4.1. Structural Workflow Mermaid Code

```

graph TD
  A[Level 1: Root] --> B[Level 2: Node 1]
  A --> C[Level 2: Node 2]
  A --> D[Level 2: Node 3]
  B --> E[Level 3: Node 1.1]
  B --> F[Level 3: Node 1.2]
  C --> G[Level 3: Node 2.1]
  D --> H[Level 3: Node 3.1]

  %% Legend components
  legendProcessed[Processed]:::processed
  legendCurrent[Current]:::current
  legendPending[Pending]:::pending

  %% Traversal Order
  classDef processed fill:#99f,stroke:#333
  classDef current fill:#9f9,stroke:#333
  classDef pending fill:#fff,stroke:#333

  %% Apply styling to nodes
  class A processed
  class B,C,D current
  class E,F,G,H pending

  %% Style edges
  linkStyle 0,1,2 stroke:#9f9,stroke-width:2px

```

## Appendix A.4.2. State Machine Mermaid Code

```

stateDiagram-v2
  [*] --> S0 : Initialization
  S0 --> S1 : BF1<br>Graph loaded<br>Initialize level queues with root
  S1 --> S1 : BF2<br>Qk ≠ ∅<br>Process node & enqueue children
  S1 --> S2 : BF3<br>∀c ∈ Nk - processed(c)<br>Validate level k
  S2 --> S1 : BF4<br>k < L<br>Advance to level k+1
  S2 --> [*] : BF5<br>k = L<br>Terminate

```

## Appendix A.4.3. Algorithm (Pseudo Code)

**Algorithm BFD**

Procedure BFD(T: Tree)

Input: T, a hierarchical tree with root node  $C_1$ 

Output: Level-synchronized implementation

// State  $S_0$ : Initialization (Table 18)// Transition BF1:  $S_0 \rightarrow S_1$  (Table 19)

1. LoadProject(T) // Initialize project and tree structure
2. level\_queues  $\leftarrow$   $[[C_1]]$  // Initialize list of level queues
3. k  $\leftarrow$  0 // Initialize current level index
4. Processed  $\leftarrow$   $\emptyset$  // Set to track processed nodes

```

// State S1: Level Processing (Table 18) - Main BFD loop
5. while k < len(level_queues):
    6. Qk ← level_queues[k] // Get queue for current level k
    7. while Qk is not empty:
        // Transition BF2: S1 → S1 (Table 19) - Process nodes at level k
        7a. C ← Dequeue(Qk)
        7b. Process(C) // Core processing action
        7c. Add C to Processed

        // Enqueue children for next level
        7d. for each child in children(C):
            7e. if len(level_queues) ≤ k+1:
                7f. level_queues.append(new_queue())
            7g. enqueue(child, level_queues[k+1])

// Transition BF3: S1 → S2 (Table 19) - Current level fully processed
8. ValidateLevel(k) // Validate all nodes at level k

// State S2: Validation (Table 18) - Decide next step after validation
9. if k+1 < len(level_queues):
    // Transition BF4: S2 → S1 (Table 19) - Advance to next level
    9a. k ← k + 1
10. else:
    // Transition BF5: S2 → T (Table 19) - All levels processed
    10a. Terminate()
    10b. return

// --- Helper Functions ---
function ValidateLevel(k):
    // Validates all nodes at level k
End Procedure

```

#### Appendix A.4.4. CSP Implementation and Formal Verification

The complete CSP model (CSPM syntax, FDR 4.2.7 compatible) implementing all operations from Algorithm A.4.3 and state transitions from Table 18 and Table 19 is available in our supplementary repository.

**Verification Status:** All formal properties verified (deadlock-free, divergence-free, deterministic, correct sequencing for BF1-BF5 transitions, and behavioral specifications including DequeueImpliesProcess, ValidateBeforeAdvance, and TerminationAtEnd)

**Repository Access:**

- GitHub: [https://github.com/IBM-Consulting-Formal-Methods/BFD\\_CSP](https://github.com/IBM-Consulting-Formal-Methods/BFD_CSP) (commit: 2dd71de)  
The model includes all processes (S0, S1, S2, T, EnqueueChildSeq) and events documented in Tables A.4.1-A.4.2. See repository README for verification instructions and complete FDR 4.2.7 assertion results.

#### Appendix A.4.5. BFD (Breadth-First Development) Methodology Tables

The BFD methodology's formal specification is further detailed through Table A.4.1, which provides a unified set of definitions for both the pseudocode and CSP models. Table A.4.2 then outlines the core CSP process algebra, detailing the state transitions and key events that correspond to the pseudocode.

Table A.4.1. BFD Methodology - Unified Definitions (Pseudocode + CSP)

Pseudocode Term	Type	Description	Pseudocode Lines	CSP Mapping
<b>Initialization</b>				
LoadProject(T)	Function	Initializes tree structure	1	load_tree_actual!t_initial
level_queues ← [[C <sub>1</sub> ]]	Function	Initializes level queue structure	2	initialize_level_queues_actual!c_root
k ← 0	Variable	Current level index	3	(tracked implicitly in S1 parameter lv)
<b>Level Processing</b>				
k < len(level_queues)	Condition	Check whether more levels remain	5	get_level_queue_actual!k
Q <sub>k</sub> is not empty	Condition	Nodes available at current level k	7	level_queue_not_empty!k
Q <sub>k</sub> is empty	Condition	Current level finished — trigger validation	7	level_queue_empty!k
<b>Node Operations</b>				
C ← Dequeue(Q <sub>k</sub> )	Function	Dequeues node from level k	7a	dequeue_actual!k!C
Process(C)	Function	Perform core processing action for node C	7b	process_actual!C
Add C to Processed	Operation	Mark node C as processed for validation/ordering	7c	tracked in processed parameter of S1/S2
for each child in children(C) → enqueue(child, level_queues[k+1])	Function	Add C's children to next level queue (create next queue if needed)	7d–7g	append_new_queue_actual!(k+1) (if needed) then enqueue_child_actual!(k+1)!child for each child
<b>Validation &amp; Level Transition</b>				
ValidateLevel(k)	Function	Validate all nodes at level k; enter S2 (Validation)	8	validate_level_actual!k → (S2 entry) → level_validated!k
k ← k + 1	Operation	Advance to next level after successful validation	9a	level_validated!k → advance_level_actual!k
<b>Termination</b>				
k + 1 < len(level_queues)	Condition	Check for next level existence (Advance case)	9	level_validated!k → advance_level_actual!k
k + 1 ≥ len(level_queues) / no_more_levels	Condition	No further levels — final termination case	10	level_validated!k → no_more_levels!k
Terminate()	Function	Final termination of the algorithm	10a, 10b	terminate_successfully_actual

Table A.4.2. BFD Methodology - CSP Process Algebra Core (States + Transitions)

CSP Process	Key Transitions	Pseudo-code Lines	CSP Events
S0	BF1: →S1	1-4	load_tree_actual!t_initial, initialize_level_queues_actual!c_root
S1(k)	BF2: →S1 (process node)	7a-7g	get_level_queue_actual!k, level_queue_not_empty!k, dequeue_actual!k!C, process_actual!C, [ap-

			pend_new_queue_actual!(k+1)]?, enqueue_child_actual!(k+1)!child* — * means repeated per child; ? means conditional append if next level not present
	BF3: →S2 (Enter valida- tion)	7, 8	get_level_queue_actual!k, level_queue_empty!k, vali- date_level_actual!k (enters S2; validation result is emitted from S2 as level_validated!k)
S2(k)	BF4: →S1 (advance level)	9, 9a	level_validated!k, advance_level_actual!k — then continue at S1(k+1)
	BF5: →T (terminate)	10, 10a	level_validated!k, no_more_levels!k, termi- nate_successfully_actual
T	—	final	terminate_successfully_actual

#### Appendix A.4.6. Formal Verification Details for BFD Model and Guarantees

All verification checks were performed using FDR 4.2.7 with standard configuration:

- **Compression:** Default behavioral reduction (e.g., diamond elimination, sbisim)
- **Search order:** Breadth-first state exploration

The model state space—tracking six nodes across four levels—was exhaustively explored. Verification confirms tractability and correctness for all eight critical assertions.

##### Assertions 1–8

- Core safety and liveness (Assertions 1–2) guarantee no deadlocks or livelocks.
- Determinism (Assertion 3) ensures unique execution paths for any given state.
- Dequeue implies process and level validation (Assertions 4–5) ensure correct breadth-first hierarchical processing.
- Post-validation behavior and termination correctness (Assertions 6–8) guarantee that BFD completes all levels and nodes.

##### Notes on methodology

The breadth-first model assumes no external adversarial interference. Correctness under this model implies correctness under any operational scenario.

Passing all FDR assertions demonstrates that BFD’s traversal and level-handling logic is sound, bounded, and deterministic.

#### Appendix A.5. CDD Mermaid Code, Algorithm, and Process Algebra

Appendix A.5 provides the formal specification for the Cyclic Directed Development (CDD) methodology, covering its Mermaid diagrams, pseudocode, and CSP model.

##### Appendix A.5.1. Structural Workflow Mermaid Code

###### graph TD

```
A[Initialization] --> B[Develop/Refine Components]
B --> C[Validate Increment]
C -->|Feedback/Re-work| B
C --> D[Final Delivery]
```

```
style B fill:#f9f,stroke:#333,stroke-width:2px,stroke-dasharray:5 5
style C fill:#9cf,stroke:#333,stroke-width:2px
```

##### Appendix A.5.2. State Machine Mermaid Code

###### stateDiagram-v2

```
[*] --> S0
S0 --> S1: CD1<br>Graph loaded
S1 --> S1: CD2<br>Node processed
```

$S_1 \rightarrow S_2$ : CD3a  
test\_failed( $C_i$ )  
 $S_1 \rightarrow S_2$ : CD3b  
feedback\_triggered( $C_i$ )  
 $S_2 \rightarrow S_1$ : CD4a  
refinement\_complete( $C_i$ )  
 $S_1 \rightarrow S_3$ : CD5  
all\_components\_written( $I_k$ )  
 $S_3 \rightarrow S_2$ : CD6  
feedback\_received  $\vee$  validation\_failed  
 $S_3 \rightarrow [*]$ : CD7  
all\_increments\_validated  
 $S_2 \rightarrow [*]$ : CD4b  
refinement\_failed  $\vee$  refinement\_count  $\geq M$   
 $S_3 \rightarrow S_1$ : CD8  
validation\_successful  $\wedge$  more\_increments

### Appendix A.5.3. Algorithm (Pseudo Code)

#### Algorithm CDD

//Refer to Table 25 and Table 26 for the transition rules

Procedure CDD( $G$ : Graph,  $R_{max}$ : Integer,  $L$ : Integer)

Input:  $G$  — A directed project graph

Input:  $R_{max}$  — Maximum allowed refinements per component

Input:  $L$  — Total number of milestones

Output: Successfully deployed system, or error

// State  $S_0$ : Initialization

1. LoadGraph( $G$ )
2. InitializeDependencies( $G$ )
3. current\_milestone  $\leftarrow 1$
4. refinement\_counts  $\leftarrow$  empty\_map()
5. SystemState  $\leftarrow S_1$

// Main Loop

6. while SystemState  $\neq T$ :

    // State  $S_1$ : Node Processing

    6a. if SystemState =  $S_1$ :

        6b. if all\_components\_written(current\_milestone) then

            // Transition CD5:  $S_1 \rightarrow S_3$

            6c. SystemState  $\leftarrow S_3$

        6d. else:

            // Transition CD2:  $S_1 \rightarrow S_1$

            6e.  $C \leftarrow$  SelectAndProcessNode(current\_milestone)

            6f. Process( $C$ )

            6g. Mark  $C$  as processed

            // Transition CD3a, CD3b:  $S_1 \rightarrow S_2$

            6h. if test\_failed( $C$ ) or feedback\_triggered( $C$ ) then

                6i. ComponentToRefine  $\leftarrow C$

                6j. SystemState  $\leftarrow S_2$

    // State  $S_2$ : Refinement

    6k. else if SystemState =  $S_2$ :

        6l. if refinement\_counts[ComponentToRefine]  $\geq R_{max}$  then

            // Transition CD4b:  $S_2 \rightarrow T$

            6m. TerminateWithError(ComponentToRefine)

        6n. else:

            6o. refinement\_counts[ComponentToRefine] += 1

            6p. RefineComponent(ComponentToRefine)

            6q. if refinement\_successful(ComponentToRefine) then

                // Transition CD4a:  $S_2 \rightarrow S_1$

```

6r. SystemState  $\leftarrow S_1$ 
6s. else:
    // Transition CD4b:  $S_2 \rightarrow T$ 
    6t. TerminateWithError(ComponentToRefine)
// State  $S_3$ : Validation
6u. else if SystemState =  $S_3$ :
    6v. ValidateIncrement(current_milestone)
    6w. if validation_failed or feedback_received then
        // Transition CD6:  $S_3 \rightarrow S_2$ 
        6x. ComponentToRefine  $\leftarrow$  IdentifyFlaw()
        6y. SystemState  $\leftarrow S_2$ 
    6z. else:
        6aa. if current_milestone < L then
            // Transition CD8:  $S_3 \rightarrow S_1$ 
            6ab. current_milestone += 1
            6ac. SystemState  $\leftarrow S_1$ 
        6ad. else:
            // Transition CD7:  $S_3 \rightarrow T$ 
            6ae. TerminateSuccess()

```

Procedure TerminateSuccess()

7. SystemState  $\leftarrow T$

End Procedure

Procedure TerminateWithError(C: NodeID)

8. SystemState  $\leftarrow T$

End Procedure

End Procedure

#### Appendix A.5.4. CSP Implementation and Formal Verification

The complete CSP model (CSPM syntax, FDR 4.2.7 compatible) implementing all operations from Algorithm A.5.3 and state transitions from Table 25 and Table 26 is available in our supplementary repository.

**Verification Status:** All formal properties verified (deadlock-free, divergence-free, deterministic, correct sequencing for CD1-CD8 transitions, dependency respect verification for N4 and N5, bounded refinement with Rmax enforcement, and hostile environment verification for worst-case refinement scenarios)

**Repository Access:**

- GitHub: [https://github.com/IBM-Consulting-Formal-Methods/CDD\\_CSP](https://github.com/IBM-Consulting-Formal-Methods/CDD_CSP) (commit: 03b972d)

The model includes all processes ( $S_0, S_1, S_2, S_3$ ) and events documented in Tables A.5.1-A.5.2, featuring actual dependency graph modeling with parallel processing capabilities and bounded refinement loops. See repository README for verification instructions and complete FDR 4.2.7 assertion results including dependency compliance proofs and refinement bound verification.

#### Appendix A.5.5. CDD (Cyclic Directed Development) Methodology Tables

The CDD methodology's formal specification is further detailed through Table A.5.1, which provides a unified set of definitions for both the pseudocode and CSP models. Table A.5.2 then outlines the core CSP process algebra, detailing the state transitions and key events that correspond to the pseudocode.

**Table A.5.1.** CDD Methodology - Unified Definitions (Pseudocode + CSP)

Pseudocode Term	Type	Description	Pseudocode Lines	CSP Mapping
<b>Initialization</b>				
LoadGraph(G)	Function	Loads project graph	1	load_graph_actual!Graph
InitializeDependencies()	Function	Initializes dependencies	2	initialize_dependencies_actual
current_milestone $\leftarrow 1$	Variable	Set initial milestone	3	(Implied in S1(M1) parameter)
<b>Internal State</b>				
refinement_counts	Variable	Tracks refinement attempts (parameter attempts in S2)	4, 6o	(Abstracted as attempts parameter in S2)
<b>Component Processing</b>				
SelectAndProcessNode()	Function	Node processing action	6e-6f	process_node_actual!NodeID
test_failed(C)	Condition	Test failure $\rightarrow$ S2 (CD3a)	6h	test_failed_actual!NodeID
feedback_triggered(C)	Condition	Feedback detected $\rightarrow$ S2 (CD3b)	6h	feedback_triggered_actual!NodeID
all_components_written(k)	Condition	Milestone complete check	6b	all_components_written_actual!MilestoneID
<b>Refinement</b>				
RefineComponent(C)	Function	Initiates refinement attempt	6p	refine_component_actual!NodeID $\rightarrow$ refinement_confirmed_actual!NodeID
refinement_successful(C)	Condition	Refinement successful	6q	refinement_complete_actual!NodeID
refinement_failed(C)	Condition	Refinement failed $\rightarrow$ check Rmax	6s	refinement_failed_actual!NodeID
<b>Validation</b>				
ValidateIncrement(k)	Function	Validates milestone increment k	6v	validate_increment_actual!MilestoneID
validation_failed	Condition	Validation failed $\rightarrow$ S2 (CD6)	6w	validation_failed_actual!MilestoneID
feedback_received	Condition	Feedback received after validation $\rightarrow$ S2 (CD6)	6w	feedback_received_actual!MilestoneID
IdentifyFlaw()	Function	Identifies flawed component	6x	identify_flaw_actual?NodeID
<b>Termination</b>				
current_milestone $< L$	Condition	Advance to next milestone check	6aa	milestone_lt(k, L_max) (Implied in S3 logic)
current_milestone $\neq 1$	Variable Assignment	Increments milestone counter	6ab	advance_milestone_actual!NextMilestone(k)
FinalDeployment()	Function	Final deployment	6ae	final_deployment_actual
TerminateSuccess()	Function	Successful termination	7, 6ae	final_development_actual $\rightarrow$ terminate_successfully_actual
TerminateWithError()	Function	Error termination (Rmax exceeded)	8, 6m, 6t	terminate_with_error_actual!NodeID

Table A.5.2. CDD Methodology - CSP Process Algebra Core (States + Transitions)

CSP Process	Key Transitions	Pseudocode Lines	CSP Events
S0	CD1: $\rightarrow$ S1 (Load & init)	1-5	load_graph_actual!Graph, initialize_dependencies_actual

S1(k, n1..n5)	CD2: →S1 (Process success)	6e-6g	process_node_actual!C → mark_completed → S1 self-loop
	CD3a: →S2 (Test failure)	6h-6j	process_node_actual!C → test_failed_actual!C → S2(C, k, n1..n5, 0)
	CD3b: →S2 (Feedback)	6h-6j	process_node_actual!C → feedback_triggered_actual!C → S2(C, k, n1..n5, 0)
	CD5: →S3 (Milestone complete)	6b-6c	all_components_written_actual!k → validate_increment_actual!k → S3(k, n1..n5)
S2(c, k, n1..n5, attempts)	CD4a: →S1 (Refinement success)	6p-6r	refine_component_actual!c → refinement_confirmed_actual!c → refinement_complete_actual!c → S1(k, n1..n5)
	CD4b: → S0 (Error termination with S0 instead of T for FDR liveness verification)	6m, 6t	refine_component_actual!c → refinement_confirmed_actual!c → refinement_failed_actual!c → [Rmax check] → terminate_with_error_actual!c → S0
S3(k, n1..n5)	CD6: →S2 (Validation failure)	6w-6y	(validation_failed_actual!k → identify_flaw_actual?c → mark_not_completed) □ (feedback_received_actual!k → identify_flaw_actual?c → mark_not_completed) → S2(c, k, n1..n5, 0)
	CD8: →S1 (Advance milestone)	6z-6ac	milestone_lt(k, L_max) → advance_milestone_actual!Next_Milestone(k) → S1(Next_Milestone(k), NotCompleted, ...)
	CD7: → 0 (Final success)	6ad-6ae	¬ milestone_lt(k, L_max) → final_development_actual → terminate_successfully_actual → S0
T	Termination	final	Not explicitly used as a final state; replaced by → S0 for liveness verification.

#### Appendix A.5.6. Formal Verification Details for CDD Model and Guarantees

All verification checks were performed using FDR 4.2.7 with standard configuration:

- **Compression:** Default behavioral reduction (e.g., diamond elimination, sbisim)
- **Search order:** Breadth-first state exploration

The model state space—tracking five nodes across three milestones plus the refinement counter—was exhaustively explored. The cumulative verification demonstrates tractability for all 10 assertions.

#### Dependency respect verification (Assertions 6 & 7)

- **N4 (Assertion 6):** Verified that N4 cannot execute until both N2 and N3 complete. Trace refinement confirms all observable behaviors respect this dependency.
- **N5 (Assertion 7):** Verified that N5 cannot execute until N4 completes. Trace refinement confirms strict sequential enforcement.

#### Refinement bound verification (Assertions 8 & 9)

- Using the Hostile Environment technique, the system is exposed to persistent refinement failures:
  - Always triggers validation\_failed\_actual
  - Always triggers refinement\_failed\_actual
- Passing deadlock and divergence checks confirms:
  - Maximum  $R_{\max}$  attempts are enforced.
  - System terminates with terminate\_with\_error\_actual.
  - Infinite refinement loops are prevented.

#### Other assertions (1–5, 10)

- Core safety and liveness (Assertions 1–2) guarantee no deadlocks or livelocks.
- Protocol compliance (Assertions 3–4) ensures deployment sequences conform to the expected events.
- Initial guard (Assertion 5) prevents premature shutdown before initialization.
- Internal consistency (Assertion 10) ensures mutually exclusive event sequences cannot occur.

#### Notes on methodology

The hostile environment represents a conservative worst-case adversary. Correctness under this scenario implies correctness under any weaker, more benign conditions. This approach avoids the need for complex failures-refinement encodings while still providing strong, provable guarantees for bounded retries and safe dependency-respecting execution.

#### Appendix A.6. PDFD Mermaid Code, Algorithm, and Process Algebra

Appendix A.6 provides the formal specification for the Primary Depth-First Development (PDFD) methodology, covering its Mermaid diagrams, pseudocode, and CSP model.

##### Appendix A.6.1. Structural Workflow Mermaid Code

```
graph TD
  %% Vertical Progression (Depth-First)
  L1[Level 1: Root Node] --> L2a[Level 2: Node A]
  L1 --> L2b[Level 2: Node B]
  L2a --> L3a[Level 3: Node A.1]
  L2b --> L3b[Level 3: Node B.1]
  L3b --> L4a[Level 4: Node B.1.1]

  %% Refinement Phase (Bounded by Rmax)
  L3b -->|Validation Failed → Refinement| RF[Refinement: Levels J2 to J3]
  RF -->|Resume Progression| L2b
  RF -->|Resume Progression| L3b
  RF -->|Exhaust Rmax| E[Error: Manual Intervention]

  %% Bottom-Up Finalization (Levels L to 1)
  L4a -->|Finalize Subtree| C3[Completion Level 3]
  C3 --> C2[Completion Level 2]
  C2 --> C1[Completion Level 1]

  %% Top-Down Finalization (Levels 1 to L)
  C1 -->|Start Top-Down| T1[Top-Down Level 1]
  T1 --> T2[Top-Down Level 2]
  T2 --> T3[Top-Down Level 3]
  T3 --> T4[Top-Down Level 4]

  %% Styling
  classDef level fill:#F0F8FF,stroke:#999
  classDef refine fill:#FFE8E8,stroke:#D32F2F
  classDef complete fill:#E8F5E9,stroke:#2E7D32,stroke-width:2px
  classDef error fill:#FFCDD2,stroke:#B71C1C

  class L1 level
  class L2a level
  class L2b level
  class L3a level
  class L3b level
  class L4a level
  class RF refine
  class C1 complete
  class C2 complete
  class C3 complete
```

```

class T1 complete
class T2 complete
class T3 complete
class T4 complete
class E error

```

#### Appendix A.6.2. State Machine Mermaid Code

##### stateDiagram-v2

```

[*] --> S0
S0 --> S1_i : PD1<br>Begin root-level<br>processing

S1_i --> S2_i : PD2<br>Validate current<br>level's nodes
S1_j --> S5 : PD8<br>Refinement exhausted

S2_i --> S1_j : PD2a<br>Backtrack to<br>level j<br>for refinement
S2_i --> S1_iplus1 : PD2b<br>Advance to next level
S2_i --> S3_i : PD4<br>Transition to<br>bottom-up process

S1_j --> S2_j : PD3<br>Validate level j again
S2_j --> S1_jplus1 : PD3a<br>Resume processing<br>at next level
S2_j --> S2_i : PD3b<br>Return to original level
S2_j --> S1_j : PD3c<br>Retry refinement<br>at level j

S3_i --> S3_iminus1 : PD4a<br>Move to<br>level i-1
S3_i --> S1_j : PD4b<br>Backtrack from<br>bottom-up<br>to refinement

S3_2 --> S4_1 : PD5<br>Transition to<br>top-down finalization

S4_i --> S4_iplus1 : PD6<br>All nodes<br>validated move to i+1
S4_i --> S1_j : PD6a<br>Backtrack<br>from completion to refinement
S4_i --> S5 : PD6b<br>Terminate due to<br>unvalidated nodes

S4_L --> T : PD7<br>Success

S5 --> [*]
T --> [*]

```

#### Appendix A.6.3. Algorithm (Pseudo Code)

##### Algorithm PDFD

```

//Refer to Table 32 and Table 33 for the transition rules
procedure PDFD_Validation(T, L, R_MAX):
1. // S0: Initialization (PD1)
2. Load Tree T, set L (levels), set R_MAX.
3. Initialize refinement_attempts[1..L] = 0.
4.
5. // PD1: Transition S0 -> S1(1)
6. call S1_InitialProcess(L1)
7.
8. // S1_InitialProcess(i): Current Level Processing (PD2 entry)
9. procedure S1_InitialProcess(i):
10. // PD8: Check for immediate R_MAX exhaustion

```

```

11.   if refinement_attempts[i] >= R_MAX then call S5 // Error
12.
13.   // PD2: Process nodes
14.   Process_Level(i)
15.
16.   // PD2: Transition S1(i) -> S2(i) Validation (Implicit)
17.   call S2_LevelValidation(i)
18.
19. // S1_RefinementProcess(j, i_orig): Refinement Level Processing (PD3 entry)
20. procedure S1_RefinementProcess(j, i_orig):
21.   // PD8: Check for immediate R_MAX exhaustion
22.   if refinement_attempts[j] >= R_MAX then call S5 // Error
23.
24.   // PD3: Process nodes
25.   Process_Level(j)
26.
27.   // PD3: Transition S1(j) -> S2(j) Validation (Implicit)
28.   call S2_RefinementValidation(j, i_orig)
29.
30. // S2_LevelValidation(i): Validation Decision Point (PD2, PD4)
31. procedure S2_LevelValidation(i):
32.   is_threshold_met = Validate_Level(i)
33.
34.   if is_threshold_met:
35.     // PD2b: Threshold met -> Advance to next level
36.     if (i = L) OR (level(i+1) = empty) OR (has_no_children(i)):
37.       // PD4: Go Bottom-Up Completion
38.       call S3_BottomUpCompletion(i)
39.     else:
40.       call S1_InitialProcess(Next(i))
41.   else:
42.     // PD2a / PD4: Threshold NOT met
43.     // PD2a: Attempt Refinement at some j
44.     j = Find_Refinement-Origin(i, L)
45.     if j is not null and refinement_attempts[j] < R_MAX:
46.       refinement_attempts[j] += 1
47.       call S1_RefinementProcess(j, i)
48.     else:
49.       // PD8: Refinement exhausted globally (fallback error)
50.       call S5 // Error
51.
52. // S2_RefinementValidation(j, i_orig): Refinement Validation (PD3)
53. procedure S2_RefinementValidation(j, i_orig):
54.   is_threshold_met = Validate_Level(j)
55.
56.   if is_threshold_met:
57.     // PD3a/PD3b: Refinement successful at j
58.
59.     if j < i_orig:
60.       // PD3a: Continue refinement deeper
61.       call S1_RefinementProcess(Next(j), i_orig)

```

```

62.     else:
63.         // PD3b: Resume original validation context
64.         call S2_LevelValidation(i_orig)
65.     else:
66.         // PD3c: Refinement at j failed
67.         j_new = Find_New_Refinement_Origin(j, i_orig)
68.         if j_new is not null and refinement_attempts[j_new] < R_MAX:
69.             refinement_attempts[j_new] += 1
70.             call S1_RefinementProcess(j_new, i_orig)
71.         else:
72.             // PD8: Refinement exhausted
73.             call S5 // Error
74.
75. // S3_BottomUpCompletion(i): Bottom-Up Pass (PD4, PD5)
76. procedure S3_BottomUpCompletion(i):
77.     Finalize_Subtrees(i)
78.     is_validated = Check_All_Descendants_Validated(i)
79.
80.     if is_validated:
81.         if i != L1:
82.             // PD4a: Move up to parent level
83.             call S3_BottomUpCompletion(Prev(i))
84.         else:
85.             // PD5: Reached root -> Start Top-Down Pass
86.             call S4_TopDownCompletion(L1)
87.     else:
88.         // PD4b: Some descendants failed validation -> Refinement needed
89.         j = Find_Refinement_Origin(i, L)
90.         if j is not null and refinement_attempts[j] < R_MAX:
91.             refinement_attempts[j] += 1
92.             call S1_RefinementProcess(j, i)
93.         else:
94.             // PD8: Refinement exhausted
95.             call S5 // Error
96.
97. // S4_TopDownCompletion(i): Top-Down Pass (PD6, PD7)
98. procedure S4_TopDownCompletion(i):
99.     Finalize_Unprocessed_Nodes(i)
100.    is_validated = Check_All_Descendants_Validated(i)
101.
102.    if is_validated:
103.        if i != L5:
104.            // PD6: Move to next level down
105.            call S4_TopDownCompletion(Next(i))
106.        else:
107.            // PD7: Reached end of levels -> Success
108.            call T // Success
109.    else:
110.        // PD6a / PD6b: Validation failed
111.        if Trace_Origin_Exists(i):
112.            // PD6a: Refinement trace exists -> Refinement needed

```

```

113.         j = Find_Refinement-Origin(i, L)
114.         if j is not null and refinement_attempts[j] < R_MAX:
115.             refinement_attempts[j] += 1
116.             call S1_RefinementProcess(j, i)
117.         else:
118.             // PD8: Refinement exhausted
119.             call S5 // Error
120.     else:
121.         // PD6b: No trace origin exists -> Error
122.         call S5 // Error
123.
124. // T: Success Termination
125. procedure T:
126.     // Implementation to signal SUCCESS
127.
128. // S5: Error Termination
129. procedure S5:
130.     // Implementation to signal ERROR

```

#### Appendix A.6.4. CSP Implementation and Formal Verification

The complete CSP model (CSPM syntax, FDR 4.2.7 compatible) implementing all operations of the Primary Depth First Development (PDFD) methodology from Algorithm A.6.3. and state transitions from Table 32 and Table 33 —including its recursive structure, state transitions, conditional decision logic, and Rmax bounding mechanism—is available in our supplementary repository.

Verification Status: All 11 core formal properties verified successfully: deadlock-free, livelock-free, divergence-free, deterministic (System :[deterministic [F]]), protocol safety (SystemProtocolView :[divergence free]), and six consistency checks guaranteeing mutually exclusive conditional handling (see Appendix A.6.6)

##### Repository Access:

- GitHub: [https://github.com/IBM-Consulting-Formal-Methods/PDFD\\_CSP](https://github.com/IBM-Consulting-Formal-Methods/PDFD_CSP) (commit: b5107ac)  
The model includes the main system process (System), the conditional environment (CondEnv), and all necessary supporting processes for state and counter management. It features a fully deterministic flow that is guaranteed to be bounded by the Rmax refinement limit, ensuring safe termination in all worst-case scenarios.

See the repository README for verification instructions and complete FDR 4.2.7 assertion results, including the proofs of Determinism and Conditional Soundness.

#### Appendix A.6.5. PDFD (Primary Depth-First Development) Methodology Tables

The PDFD methodology's formal specification is further detailed through Table A.6.1, which provides a unified set of definitions for both the pseudocode and CSP models. Table A.6.2 then outlines the core CSP process algebra, detailing the state transitions and key events that correspond to the pseudocode.

Table A.6.1. PDFD Methodology - Unified Definitions (Pseudocode + CSP)

Pseudocode Term	Type	Description	Pseudocode Lines	CSP Mapping
<b>Initialization</b>				
Load T, initialize	Procedure	Initializes tree T and refinement attempt counters to zero.	1-3	(Implicit)

call S1_InitialProcess(L1 )	Call	Starts the process at the initial level L1.	6	PD1: process_level!L1 → S1_InitialProcess(L1)
<b>S<sub>1</sub>: Level Processing</b>				
Process_Level(i)	Procedure	Performs the core processing for the given level i or j.	14, 25	process_level!i
if refinement_attempts [i] ≥ R_MAX	Condition	Checks if refinement attempts for the current level are exhausted.	11, 22	PD8: cond_refinement_exh austed?i → S5
<b>S<sub>2</sub>: Validation</b>				
is_threshold_met = Validate_Level(i)	Function	Performs the level validation check.	32, 54	validate_level!i
if is_threshold_met	Condition	Threshold met (PD2b) or refinement success (PD3a/3b).	34, 56	cond_threshold_met? i
call S1_InitialProcess(Ne xt(i))	State Transition	Advances to process the next level.	40	PD2b: S1_InitialProcess(Nex t(i))
if j < i_orig	Condition	Successful refinement continues deeper.	59	PD3a: cond_j_lt_i.j.i_orig
else: call S2_LevelValidation( i_orig)	State Transition	Successful refinement resumes validation context.	63-64	PD3b: cond_j_eq_i.j.i_orig → S2_LevelValidation(i_ orig) (CSP uses S2_LevelValidation which includes S3 call)
<b>Refinement / Bottom-Up Logic</b>				
if (i=L) OR ... (has_no_children(i))	Condition	Checks if Bottom-Up is mandatory or an option (PD4).	36	cond_has_no_childre n?i
j = Find_Refinement_O rigin(i, L)	Function	Identifies the root cause level j for refinement backtracking.	44, 67, 89, 113	cond_refinement_ava ilable?j (Non- deterministic choice)
refinement_attempts [j] += 1	Action	Increments refinement attempt counter for level j.	46, 69, 91, 115	increment_attempts!j
call S1_RefinementProce ss(j, i_orig)	State Transition	Transitions to the Level Processing state for refinement.	47, 70, 92, 116	S1_RefinementProces s(j, i_orig)
<b>S<sub>3</sub>: Bottom-Up Completion</b>				
Finalize_Subtrees(i)	Procedure	Processes and validates subtrees at the current level.	77	finalize_subtrees!i
if is_validated	Condition	Checks if all nodes in a subtree are successfully validated.	80	cond_all_descendants _validated?i
if i != L1: call S3_BottomUpComple tion(Prev(i))	State Transition	Continues bottom-up to the previous level (PD4a).	81-83	S3_BottomUpComple tion(Prev(i))
else: call S4_TopDownComple tion(L1)	State Transition	Transitions to the Top-Down Completion state (PD5).	84-86	S4_TopDownComple tion(L1)
<b>S<sub>4</sub>: Top-Down Completion</b>				
Finalize_Unprocesse d_Nodes(i)	Procedure	Finalizes and validates any remaining unprocessed nodes.	99	finalize_unprocessed! i

if $i \neq L5$ : call $S4\_TopDownCompl$ $etion(Next(i))$	State Transition	Continues top-down to the next level (PD6).	103-105	$S4\_TopDownComple$ $tion(Next(i))$
else: call T	State Transition	Transitions to the successful termination state (PD7).	106-108	T
if $Trace\_Origin\_Exists$ $(i)$	Condition	Checks if refinement is possible after failure (PD6a).	111	$cond\_trace\_origin\_exi$ $sts?i$
else: call S5	State Transition	Transitions to the terminal error state (PD6b).	121-122	$cond\_trace\_origin\_no$ $t\_exists?i \rightarrow S5$
<b>Final Outcome</b>				
call T	Terminatio n	The system terminates successfully.	125-126	$terminate\_success \rightarrow$ T
call S5	Terminatio n	The system terminates with an error.	129-130	$terminate\_error \rightarrow S5$

Table A.6.2. PDFD Methodology - CSP Process Algebra Core (States + Transitions)

CSP Process	Key Transitions	Pseudocode Lines	CSP Events (Simplified)
$S_0$	PD1: Initial start	1–6	$process\_level!L1 \rightarrow S1\_InitialProcess(L1)$
$S1\_InitialProcess(i)$	PD2: Core sequence start PD8: Exhaustion check	9–14 11	$process\_level!i \rightarrow S2\_LevelValidation(i)$ $cond\_refinement\_exhausted?i \rightarrow S5$
$S1\_RefinementPro$ $cess(j, i\_orig)$	PD3: Core sequence start PD8: Exhaustion check	20–25 22	$process\_level!j \rightarrow$ $S2\_RefinementValidation(j, i\_orig)$ $cond\_refinement\_exhausted?j \rightarrow S5$
$S2\_RefinementVali$ $ation(j, i\_orig)$	PD3 (Entry) PD3a/PD3b: Refinement success PD3c: Refinement failure	53–54 56–64 66–73	$validate\_level!j \rightarrow \dots$ $cond\_threshold\_met?j \rightarrow$ $S3\_RefinementResolution(\dots)$ $cond\_threshold\_not\_met?j \rightarrow$ (refinement choice)
$S3\_RefinementRes$ $olution(j, i\_orig)$	PD3a: Continue deep refinement PD3b: Resume validation context	58–61 62–64	$cond\_j\_lt\_i.j.i\_orig \rightarrow$ $S1\_RefinementProcess$ $cond\_j\_lt\_i.j.i\_orig \rightarrow$ $S1\_RefinementProcess(Next(j), i\_orig)$
$S2\_LevelValidatio$ $n(i)$	PD2b: Advance level PD4: Go bottom-up (mandatory) PD2a: Refine (failure path)	39–40 48–50 44–47	$cond\_threshold\_met?i \rightarrow$ $S1\_InitialProcess(Next(i))$ $cond\_has\_no\_children?i \rightarrow$ $S3\_BottomUpCompletion(i)$ $cond\_refinement\_available?j \rightarrow$ $increment\_attempts!j \rightarrow$ $S1\_RefinementProcess(j, i)$
$S3\_BottomUpCom$ $pletion(i)$	PD4a: Move up PD5: Start top-down PD4b: Refine (failure)	80–83 84–86 88–95	$finalize\_subtrees!i \rightarrow$ $cond\_all\_descendants\_validated?i \rightarrow$ $S3\_BottomUpCompletion(Prev(i))$ $finalize\_subtrees!i \rightarrow$ $cond\_all\_descendants\_validated?i \rightarrow$ $S4\_TopDownCompletion(L1)$ $cond\_not\_all\_descendants\_validated?i \rightarrow$ $SimpleRefinementHandler(i)$
$S4\_TopDownCom$ $pletion(i)$	PD6: Move down	102–105	$finalize\_unprocessed!i \rightarrow$ $cond\_all\_descendants\_validated?i \rightarrow$ $S4\_TopDownCompletion(Next(i))$

	PD7: Success	106–108	finalize_unprocessed!i → cond_all_descendants_validated?i → T
	PD6a: Refine (failure)	110–119	cond_not_all_descendants_validated?i → cond_trace_origin_exists?i → SimpleRefinementHandler(i)
	PD6b: Error	120–122	cond_not_all_descendants_validated?i → cond_trace_origin_not_exists?i → S5
S <sub>5</sub> / T	Termination	125–130	terminate_error → S5 / terminate_success → T

### Appendix A.6.6. Formal Verification Details for PDFD Model and Guarantees

All verifications were performed in FDR 4.2.7 using default behavioral reduction (e.g., sbisim, diamond elimination) and breadth-first exploration.

#### Scope

The model tracks:

- Five core levels (L1–L5)
- Core and refinement transitions
- The refinement attempt counter

All 11 assertions completed exhaustively within this state space.

#### 1. Structural Integrity (1 Assertion)

##### Determinism

System :[deterministic [F]] confirms the system's progression is fully driven by conditional events offered by CondEnv, with no implicit nondeterminism.

#### 2. Consistency and Soundness (6 Assertions)

Mutual Exclusivity All conditional decision pairs (cond\_X) were proven disjoint.

Example: ConditionConsistency\_ThresholdMet [T= STOP] guarantees cond\_threshold\_met and cond\_threshold\_not\_met cannot both be enabled.

This validates the soundness of the transition rules at every decision point.

#### 3. Liveness and Bounded Termination (4 Assertions)

##### Deadlock-, Livelock-, and Divergence-Free

These checks confirm that termination is always reached safely and that bounded refinement is enforced without hidden cycles.

##### Protocol View Confirmation

SystemProtocolView :[divergence free] confirms that correctness is preserved even when conditional events are abstracted.

### Appendix A.7. PBFD Mermaid Code, Algorithm, and Process Algebra

Appendix A.7 provides the formal specification for the Primary Breadth-First Development (PBFD) methodology, covering its Mermaid diagrams, pseudocode, and CSP model.

#### Appendix A.7.1. Structural Workflow Mermaid Code

##### flowchart TD

```
A0([Start]) --> A1[Initialize Pattern1]
```

```
A1 --> A2[Process Patterni]
```

```
%% Proceed if all nodes are validated
```

```
A2 -->|All nodes validated| A3[Proceed to next level Patterni+1]
```

```
A2 -->|Validation failed| A4[Backtrack to Patternj]
```

```
%% j is determined by trace_origin(i)
```

$A4 \rightarrow |refinement\_attempts_j < R_{max} | A2$   
 $A4 \rightarrow |refinement\_attempts_j \geq R_{max} | A5[Error: Exhausted R_{max}]$

$A3 \rightarrow |i < L \wedge Pattern_{i+1} \neq \emptyset | A2$   
 $A3 \rightarrow |i < L \wedge Pattern_{i+1} = \emptyset | A6[Start Top-Down Finalization]$   
 $A3 \rightarrow |i = L | A6$

$A6 \rightarrow A7[Finalize Pattern_i]$

$A7 \rightarrow |All\ nodes\ processed | A8[Advance\ to\ Pattern_{i+1}]$   
 $A8 \rightarrow |i < L | A7$   
 $A8 \rightarrow |i = L | A9([Done])$

## Appendix A.7.2. State Machine Mermaid Code

### stateDiagram-v2

```

%% ----- Initialization Phase -----
state "S0: Entry Point" as S0_init

%% ----- Progression Phase -----
state "S1(i): Current Pattern Processing" as S1_i
state "S1(i+1): Next Pattern (Children)" as S1_i_plus_1
state "S2(i): Pattern Validation" as S2_i
state "S3(i): Depth Resolution" as S3_i

%% ----- Refinement Phase -----
state "S1(j): Refinement Level Processing" as S1_j
state "S1(j+1): Refinement Progression" as S1_j_plus_1
state "S2(j): Refinement Validation" as S2_j
state "S3(j): Refinement Depth Resolution" as S3_j

%% ----- Completion Phase -----
state "S4(1): Completion Phase Entry" as S4_1_entry
state "S4(i): Completion Level" as S4_i
state "S4(L): Last Completion Level" as S4_L

%% ----- Terminal States -----
state "S5: Error - Terminate" as S5_error
state "T: Terminate" as T_success

%% ----- Choice Pseudostates -----
state PB1_ch <<choice>>
state PB2_ch <<choice>>
state PB3_ch <<choice>>
state PB3a_ch <<choice>>
state PB3a_post_ch <<choice>>
state PB4a_ch <<choice>>
state PB4b_ch <<choice>>
state PB5_ch <<choice>>
state PB6_ch <<choice>>
state PB7_ch <<choice>>

```

%% \_\_\_\_\_ Initial Flow \_\_\_\_\_

[\*] --> S0\_init  
 S0\_init --> PB1\_ch  
 PB1\_ch --> S1\_i : PB1 - i = 1

%% \_\_\_\_\_ Pattern Progression \_\_\_\_\_

S1\_i --> PB2\_ch  
 PB2\_ch --> S2\_i : PB2 - Node unvalidated  
 PB2\_ch --> S3\_i : PB2a - All validated

%% \_\_\_\_\_ Pattern Validation (S2\_i) \_\_\_\_\_

S2\_i --> PB3\_ch  
 PB3\_ch --> S1\_j : PB3 - Backtrack possible  
 PB3\_ch --> S3\_i : PB4 - All validated  
 PB3\_ch --> S5\_error : PB3c - No backtrack possible

%% \_\_\_\_\_ Refinement Handling (S1\_j to S3\_j) \_\_\_\_\_

S1\_j --> PB3a\_ch  
 PB3a\_ch --> S2\_j : PB3a - Node unvalidated  
 PB3a\_ch --> S3\_j : PB3b - All validated  
 S1\_j --> S5\_error : PB9 - Attempts exhausted

S2\_j --> PB3a\_post\_ch  
 PB3a\_post\_ch --> S3\_j : PB3a1 - All validated  
 PB3a\_post\_ch --> S1\_j : PB3a2 - Retry refinement  
 PB3a\_post\_ch --> S5\_error : PB3a3 - Attempts exhausted

%% \_\_\_\_\_ Post-Refinement Actions (S3\_j) \_\_\_\_\_

S3\_j --> PB5\_ch  
 PB5\_ch --> S1\_j\_plus\_1 : PB5 - Resume next level ( $j < i$ )

S3\_j --> PB6\_ch  
 PB6\_ch --> S3\_i : PB6 - Refinement complete ( $j = i$ )

%% \_\_\_\_\_ Descent or Completion Decision (S3\_i) \_\_\_\_\_

S3\_i --> PB4a\_ch  
 PB4a\_ch --> S1\_i\_plus\_1 : PB4a - Recurse to critical children

S3\_i --> PB4b\_ch  
 PB4b\_ch --> S4\_1\_entry : PB4b - Start Completion

%% \_\_\_\_\_ Completion Phase \_\_\_\_\_

S4\_1\_entry --> S4\_i  
 S4\_i --> PB7\_ch  
 PB7\_ch --> S4\_i : PB7 - Advance ( $i+1 < L$ )  
 PB7\_ch --> S4\_L : PB7 - Advance to Last ( $i+1 = L$ )  
 PB7\_ch --> S1\_j : PB7a - Unfinalized → backtrack  
 PB7\_ch --> S5\_error : PB7b - Unfinalized → no backtrack

S4\_L --> T\_success : PB8 - All levels completed

```

%%----- Final Transitions -----
S5_error --> [*]
T_success --> [*]

```

### Appendix A.7.3. Algorithm (Pseudo Code)

#### Algorithm Pbfd

```

//=====
// Structural Helper Functions
//=====

// Table 40, Rule PB3/PB7a: Determines the lowest-level pattern that caused the failure.
Function trace_origin(i: Integer, check_predicate: Function) Returns Integer
  // Find  $j = \min\{k \mid k < i \wedge \text{check\_predicate}(\text{Pattern}_k, \text{Pattern}_i)\}$ 
  // The check_predicate is either 'affected_by' (for PB3) or 'affected_by_unprocessed' (for
PB7a).
  j_list  $\leftarrow \{k \mid k < i \wedge \text{check\_predicate}(\text{Pattern}_k, \text{Pattern}_i)\}$ 
  if j_list is empty then
    return UNDEFINED // Handles PB3c condition: trace_origin undefined
  else
    return min(j_list)
End Function

// Table 40, Rule PB5: Finds the next level to process within the original refinement scope (j to
i_orig).
Function determine_next_refinement_level(j: Integer, i_orig: Integer) Returns Integer
  // In Pbfd, refinement is horizontal advancement after a success at j.
  // The next level is simply j+1, provided j+1 is still within the original scope.
  if  $j + 1 \leq i\_orig$  then
    return j + 1
  else
    // This case should be caught by the PB6 condition ( $j = i\_orig$ ) but included for safety.
    return UNDEFINED
End Function

//=====
// Critical Children Selection Procedure
//=====
Function select_critical_children(available_children: Set[Node], level: Integer)
  // Selection criteria based on architectural criticality
  critical_children  $\leftarrow \emptyset$ 

  for each child in available_children do
    if is_on_critical_path(child)  $\vee$ 
      has_high_fanout(child)  $\vee$ 
      is_foundational_component(child, level) then
      critical_children  $\leftarrow$  critical_children  $\cup$  {child}
    end if
  end for
end for

```

```

    return critical_children
End Function

// =====
// Consolidated Refinement Handler
// Covers Table 40: Rules PB3/PB3c and PB7a/PB7b
// =====
Function HandlePBFDFailureRefinement(
    current_failed_level: Integer,
    R_MAX: Integer,
    find_j_predicate: Function
) Returns State

// Table 40, Rule PB3/PB7a: Find root cause level (using trace_origin)
1: j ← trace_origin(current_failed_level, find_j_predicate)

// Table 40, Rule PB3/PB7a: Check refinement possibility (j defined AND attempts < R_MAX)
2: if j is defined and refinement_attempts[j] < R_MAX then
3:   refinement_attempts[j]++
4:   Return S1_RefinementProcess(j, current_failed_level) // → S1(j) via PB3/PB7a

// Table 40, Rule PB3c/PB7b: Termination (j undefined OR attempts exhausted)
5: else
6:   Return S5 // → S5 via PB3c/PB7b
End Function

// =====
// Main PBFDF Algorithm
// =====
Procedure PBFDF(T: Tree, L: Integer, R_MAX: Integer)
Input: Tree T (L levels), R_max
Output: Processed tree or error

// Table 39: S0 Initialization
1: Load T, initialize refinement_attempts[1..L] = 0
2: i ← 1, currentState ← S1_InitialProcess(i) // Table 40, Rule PB1: → S1(1)

3: while currentState ∉ {T, S5} do
4:   case currentState of

        // Table 39: S1(i) Main Pattern Processing
5:     S1_InitialProcess(i):
6:       Process Patterni
7:       if ∃n ∈ Patterni: ¬validated(n) then // Rule PB2: → S2(i)
8:         currentState ← S2_ValidationInitial(i)
9:       else if ∀n ∈ Patterni: validated(n) then // Rule PB2a: → S3(i)
10:        currentState ← S3_DepthProgression(i)

        // Table 39: S2(i) Initial Pattern Validation
11:      S2_ValidationInitial(i):
12:        Validate Patterni // Rule PB4 Action

```

```

13:     if  $\forall n \in \text{Pattern}_i$ : validated(n) then // Rule PB4:  $\rightarrow S3(i)$ 
14:     currentState  $\leftarrow$  S3_DepthProgression(i)
15: else if  $\exists n \in \text{Pattern}_i$ :  $\neg$ validated(n) then // Rule PB3/PB3c: Refinement or Termination
16:     currentState  $\leftarrow$  HandlePBFDFailureRefinement(i, R_MAX, affected_by)

// Table 39: S1(j) Refinement Processing
17: S1_RefinementProcess(j, i_orig):
18:     if refinement_attempts[j]  $\geq$  R_max then // Rule PB9:  $\rightarrow S5$ 
19:     currentState  $\leftarrow$  S5
20:     else
21:     Process Patternj
22:     if  $\exists n \in \text{Pattern}_j$ :  $\neg$ validated(n) then // Rule PB3a:  $\rightarrow S2(j)$ 
23:     currentState  $\leftarrow$  S2_ValidationRefinement(j, i_orig)
24:     else if  $\forall n \in \text{Pattern}_j$ : validated(n) then // Rule PB3b:  $\rightarrow S3(j)$ 
25:     currentState  $\leftarrow$  S3_RefinementDepthResolution(j, i_orig)

// Table 39: S2(j) Refinement Validation
26: S2_ValidationRefinement(j, i_orig):
27:     if  $\forall n \in \text{Pattern}_j$ : validated(n) then // Rule PB3a1:  $\rightarrow S3(j)$ 
28:     currentState  $\leftarrow$  S3_RefinementDepthResolution(j, i_orig)
29:     else if  $\exists n \in \text{Pattern}_j$ :  $\neg$ validated(n) and refinement_attempts[j] < R_max then // PB3a2
30:     refinement_attempts[j]++
31:     currentState  $\leftarrow$  S1_RefinementProcess(j, i_orig) //  $\rightarrow S1(j)$ 
32:     else if  $\exists n \in \text{Pattern}_j$ :  $\neg$ validated(n) and refinement_attempts[j]  $\geq$  R_max then // PB3a3
33:     currentState  $\leftarrow$  S5 //  $\rightarrow S5$ 

// Table 39: S3(i) Depth-Oriented Resolution
34: S3_DepthProgression(i):
35:     //Implement Pattern Derivation (Table 40, Rule PB4a action); Select critical children
for next pattern (not all children)
36:     Patterni+1  $\leftarrow$   $\emptyset$ 
37:     available_children  $\leftarrow$  {c  $\in$  V |  $\exists n \in \text{Pattern}_i$ : (n,c)  $\in$  E}
38:     Patterni+1  $\leftarrow$  select_critical_children(available_children, i)

39:     if i < L and Patterni+1  $\neq$   $\emptyset$  then // Rule PB4a:  $\rightarrow S1(i+1)$ 
40:     i  $\leftarrow$  i+1, currentState  $\leftarrow$  S1_InitialProcess(i)
41:     else if i = L or Patterni+1 =  $\emptyset$  then // Rule PB4b:  $\rightarrow S4(1)$ 
42:     i  $\leftarrow$  1, currentState  $\leftarrow$  S4(i)

// Table 39: S3(j) Refinement Depth Resolution
43: S3_RefinementDepthResolution(j, i_orig):
44:     if j < i_orig then // Rule PB5:  $\rightarrow S1(j+1)$ 
45:     next_level  $\leftarrow$  determine_next_refinement_level(j, i_orig) //Get next level
46:     currentState  $\leftarrow$  S1_RefinementProcess(next_level, i_orig)
47:     else if j = i_orig then // Rule PB6:  $\rightarrow S3(i_orig)$ 
48:     currentState  $\leftarrow$  S3_DepthProgression(i_orig)

// Table 39: S4(i) Completion Phase
49: S4(i):
50:     Finalize Patterni
51:     if  $\forall n \in \text{Pattern}_i$ : processed(n) then

```

```

52:         if i < L then // Rule PB7: → S4(i+1)
53:             i ← i+1, currentState ← S4(i)
54:         else if i = L then // Rule PB8: → T
55:             currentState ← T
56:         else if ∃ n ∈ Pattern; ¬ processed(n) then
57:             currentState ← HandlePBFDFailureRefinement(i, R_MAX,
affected_by_unprocessed) // PB7a/PB7b

58:     end case
59: end while

// Final Termination (Table 40)
60: if currentState = S5 then Terminate with error
61: else if currentState = T then Terminate successfully
End Procedure

```

#### Appendix A.7.4. CSP Implementation and Formal Verification

The complete CSP model (CSPM syntax, FDR 4.2.7 compatible) implementing all operations of the Primary Breadth-First Development (Pbfd) methodology from Algorithm A.7.3. and state transitions from Table 39 and Table 40 —including its breadth-first with S3\_DepthProgression logic, state transitions, conditional decision predicates, and R\_max bounding mechanism—is available in our supplementary repository.

##### **Verification Status:**

All 33 core formal properties verified successfully:

**Core Safety & Liveness:** Deadlock-free and divergence-free under both normal and hostile conditions

**State-Level Safety:** Successful verification of 26 state-level assertions, covering every operational and terminal state (S0–S5, T) across all level combinations (L1, L2, L3) in both normal and refinement contexts

**Conditional Soundness:** Verified mutual exclusivity of validation conditions, ensuring no contradictory conditional states

**Hostile Environment Robustness:** Deadlock-free operation under adversarial conditional environments

**Bounding Guarantee:** Verified R\_max enforcement, ensuring termination even in failure scenarios.

The model includes the main system process (Pbfd → System), the conditional environment (LegalCondEnv), the hostile conditional environment (HostileEnv), and all necessary supporting processes for state management. The flow is guaranteed to be bounded by the R\_max refinement limit, ensuring safe termination in all worst-case scenarios.

##### **Repository Access:**

- GitHub: [https://github.com/IBM-Consulting-Formal-Methods/Pbfd\\_CSP](https://github.com/IBM-Consulting-Formal-Methods/Pbfd_CSP) (commit: ea1a3bc)  
See the repository README for verification instructions and complete FDR 4.2.7 assertion results detailing all 33 passing assertions.

#### Appendix A.7.5. Pbfd (Primary Breadth-First Development) Methodology Tables

The Pbfd methodology's formal specification is further detailed through Table A.7.1, which provides a unified set of definitions for both the pseudocode and CSP models. Table A.7.2 then outlines the core CSP process algebra, detailing the state transitions and key events that correspond to the pseudocode.

**Table A.7.1.** Pbfd Methodology - Unified Definitions (Pseudocode + CSP)

Pseudocode Term	Type	Description	Pseudocode Lines	CSP Mapping
<b>Initialization</b>				
Load T	System Function	Initializes tree structure and pattern hierarchy	PBFD: 1	load_tree_actual
initialize_refinement_attempts	System Function	Sets all level refinement counters to 0	PBFD: 1	initialize_refinement_attempts_actual
currentState ← S1_InitialProcess	State Transition	Begins main pattern processing (PB1)	PBFD: 2	S1_InitialProcess(L1)
<b>Pattern Processing</b>				
Process Pattern <sub>i</sub>	Pattern Function	Executes core pattern processing (PB2)	PBFD: 6	process_pattern_actual.i
Validate Pattern <sub>i</sub>	Validation Action	Performs pattern validation (PB4 Action)	PBFD: 12, 27	validate_pattern_actual.i
$\exists n \in \text{Pattern}_i: \neg \text{validated}(n)$	Validation Condition	Pattern validation failed (PB2)	PBFD: 7, 22, 29, 32	cond_not_all_validated?i
$\forall n \in \text{Pattern}_i: \text{validated}(n)$	Validation Condition	Pattern validation succeeded (PB2a, PB4)	PBFD: 9, 13, 24, 27	cond_all_validated?i
<b>Refinement Control</b>				
Find j	Trace Function	Identifies minimal root cause level j (PB3/PB7a)	HandlePBFDFailureRefinement: 1	(Implicit in TryTraceOrigin using cond_trace_origin)
affected_by_unprocessed	Trace Function	Finds patterns affecting unprocessed nodes	PBFD: 57	(Implicit in TryTraceOrigin_Completion)
refinement_attempts[j]++	Counter Operation	Increments refinement attempts for level j (PB3/PB3a2/PB7a)	HandlePBFDFailureRefinement: 3, PBFD: 30	increment_refinement_attempts_actual.j
refinement_attempts[j] ≥ R <sub>max</sub>	Limit Check	True when refinement attempts for level j ≥ R <sub>max</sub> (PB3c/PB3a3/PB7b/PB9)	HandlePBFDFailureRefinement: 5 (else branch), PBFD: 18, 32	cond_ref_attempts_ge_Rmax?j
refinement_attempts[j] < R <sub>max</sub>	Limit Check	True when refinement attempts for level j < R <sub>max</sub> (PB3/PB3a2/PB7a)	HandlePBFDFailureRefinement: 2, PBFD: 29	cond_ref_attempts_lt_Rmax?j
HandlePBFDFailureRefinement	Procedure	Handles PB3/PB3c/PB7a/PB7b logic	PBFD: 16, 57	TryTraceOrigin_Initial/Completion
<b>Critical Children Selection</b>				
available_children (Pattern <sub>i</sub> )	Function	Returns set of direct child nodes: $\{c \in V \mid \exists n \in \text{Pattern}_i: (n,c) \in E\}$	PBFD: 37	(Implied by resolve_depth_actual)
is_on_critical_path(c)	Predicate	True if node c lies on critical path from roots to leaves	select_critical_children	(Not directly mapped, external logic)
has_high_fanout(c)	Predicate	True if node c has ≥3 dependents	select_critical_children	(Not directly mapped, external logic)
is_foundational_component(c, level)	Predicate	True if node c provides foundational services for its level	select_critical_children	(Not directly mapped, external logic)
select_critical_children(available_children, level)	Procedure	Selects architecturally critical nodes for Pattern <sub>i+1</sub>	PBFD: 38	select_critical_children_actual.i
<b>Depth Processing</b>				

$Pattern_{i+1} \neq \emptyset$	Existence Check	True when next level has no pattern entries (PB4b)	PBFD: 39	<code>cond_pattern_next_none_mpty.i</code>
$i < L$	Boundary Check	True when not at max level (PB4a/PB7)	PBFD: 39, 52	<code>cond_i_lt_L?i</code>
$i = L$	Boundary Check	True at max level (PB4b/PB8)	PBFD: 41, 54	<code>cond_i_eq_L?i</code>
$Pattern_{i+1} = \emptyset$	Existence Check	True when next level has patterns (PB4b)	PBFD: 41	<code>cond_pattern_next_empty?i</code>
<b>Completion Phase</b>				
Finalize $Pattern_i$	Completion Function	Processes remaining nodes (PB7/PB8)	PBFD: 50	<code>finalize_pattern_actual.i</code>
$processed(n)$	State Predicate	True when node $n$ is fully processed ( $P(n)=1 \vee P(n)=2$ )	Implied by PBFD: 51, 56	(Implied by <code>cond_all_processed</code> )
$\exists n \in Pattern_i: \neg processed(n)$	Validation Condition	Pattern has unprocessed nodes (PB7a/PB7b)	PBFD: 56	<code>cond_not_all_processed?i</code>
$\forall n \in Pattern_i: processed(n)$	Validation Condition	All nodes processed (PB7/PB8)	PBFD: 51	<code>cond_all_processed?i</code>
<b>Termination</b>				
S5	Error State	Terminal state for all error conditions (PB3c/PB3a3/PB7b/PB9)	PBFD: 60	<code>terminate_failure_actual → S5</code>
T	Success State	Terminal state for successful completion (PB8)	PBFD: 61	<code>terminate_success_actual → T</code>

Table A.7.2. PBFD Methodology - CSP Process Algebra Core (States + Transitions)

CSP Process	Key Transitions (PB Ref.)	Pseudocode Lines	CSP Events (Simplified)
S0	PB1: $\rightarrow$ S1_InitialProcess(L1)	PBFD: 1-2	<code>load_tree_actual → initialize_refinement_attempts_actual → S1_InitialProcess(L1)</code>
S1_InitialProcess(i)	PB2: $False \rightarrow S2$ ; PB2a: $True \rightarrow S3$	PBFD: 6-10	<code>process_pattern_actual.i → (cond_not_all_validated?i → S2_ValidationInitial(i) [] cond_all_validated?i → S3_DepthProgression(i))</code>
S2_ValidationInitial(i)	PB4: $True \rightarrow S3$ ; PB3/PB3c: $False \rightarrow TryTraceOrigin$	PBFD: 12-16	<code>validate_pattern_actual.i → (cond_all_validated?i → S3_DepthProgression(i) [] cond_not_all_validated?i → TryTraceOrigin_Initial(i))</code>
S1_RefinementProcess(j, i_orig)	PB9: $attempts \geq Rmax \rightarrow S5$ ; PB3a: $attempts < Rmax \rightarrow S2$	PBFD: 18-25	<code>(cond_ref_attempts_ge_Rmax?j → S5) [] cond_ref_attempts_lt_Rmax?j → process_refinement_pattern_actual.j → ...</code>
S2_ValidationRefinement(j, i_orig)	PB4a: $i < L, Pattern_{i+1} \neq \emptyset \rightarrow S1(i+1)$ ; PB4b: $i = L \vee Pattern_{i+1} = \emptyset \rightarrow S4(1)$	PBFD: 27-33	<code>validate_refinement_pattern_actual.j → (cond_all_validated?j → S3_RefinementDepthResolution(j, i_orig) [] cond_not_all_validated?j → ...)</code>
S3_DepthProgression(i)	PB5: $j < i\_orig \rightarrow S1(Next(j))$ ; PB6: $j = i\_orig \rightarrow S3(i\_orig)$	PBFD: 37-42	<code>resolve_depth_actual.i → select_critical_children_actual.i → (cond_pattern_next_nonempty?i ∧ cond_i_lt_L?i → S1_InitialProcess(i+1) [] ... → S4(L1))</code>
S3_RefinementDepthResolution(j, i_orig)	PB5: $j < i\_orig \rightarrow S1(Next(j))$ ; PB6: $j = i\_orig \rightarrow S3(i\_orig)$	PBFD: 44-48	<code>resolve_refinement_depth_actual.j → (if LessThan(j, i_orig) then S1_RefinementProcess(Next(j), i_orig) else S3_DepthProgression(i_orig))</code>
S4(i)	PB7: $i < L, processed \rightarrow S4(i+1)$ ; PB8: $i = L, processed \rightarrow T$ ; PB7a/PB7b:	PBFD: 50-57	<code>finalize_pattern_actual.i → (cond_all_processed?i → (cond_i_lt_L?i → S4(i+1) [] cond_i_eq_L?i → T) [] cond_not_all_processed?i → TryTraceOrigin_Completion(i))</code>

	→processed → TryTraceOrigin		
S5	N/A (Terminal Failure State)	PBFD: 60	terminate_failure_actual → S5
T	N/A (Terminal Success State)	PBFD: 61	terminate_success_actual → T

#### Appendix A.7.6. Formal Verification Details for PBFD model and Refinement Guarantees

All results were obtained in FDR 4.2.7 using breadth-first state exploration and default behavioral reductions (e.g., sbisim, diamond elimination).

##### Scope and Configuration

- **Three depth levels:** L1, L2, L3. The verification guarantees correctness up to this depth.
- **State set:** S0 through S5 and T
- **Full transition set:** PB1–PB9 from Table 40
- **Bounded refinement:**  $R_{\max} = 5$
- **Complete conditional environment:** Both legal and hostile variants

##### Assertion Breakdown

See Table A.7.3. for the details.

**Table A.7.3.** Assertion Breakdown (Total: 33)

Category	Count	Coverage
Core Safety/Liveness	5	System deadlock/divergence freedom plus initialization safety
State-Level Safety	26	All operational and terminal states across all level combinations
Conditional Soundness	1	Mutual exclusivity of conditional predicates
Hostile Environment	2	Adversarial robustness under non-cooperative inputs
Total	33	Complete verification

##### State-Space Characteristics

The bounded refinement ( $R_{\max} = 5$ ) and limited levels (L1–L3) ensure a finite, tractable model. All checks completed successfully, confirming:

- Bounded progression through at most 3 levels
- Bounded refinement with at most  $R_{\max} = 5$  attempts per level
- Guaranteed termination at either T (success) or S5 (error)

##### Performance

Most checks complete in under one second. Hostile-environment checks may take 5–30 seconds due to nondeterministic conditional choices and larger state space exploration, but always pass consistently.

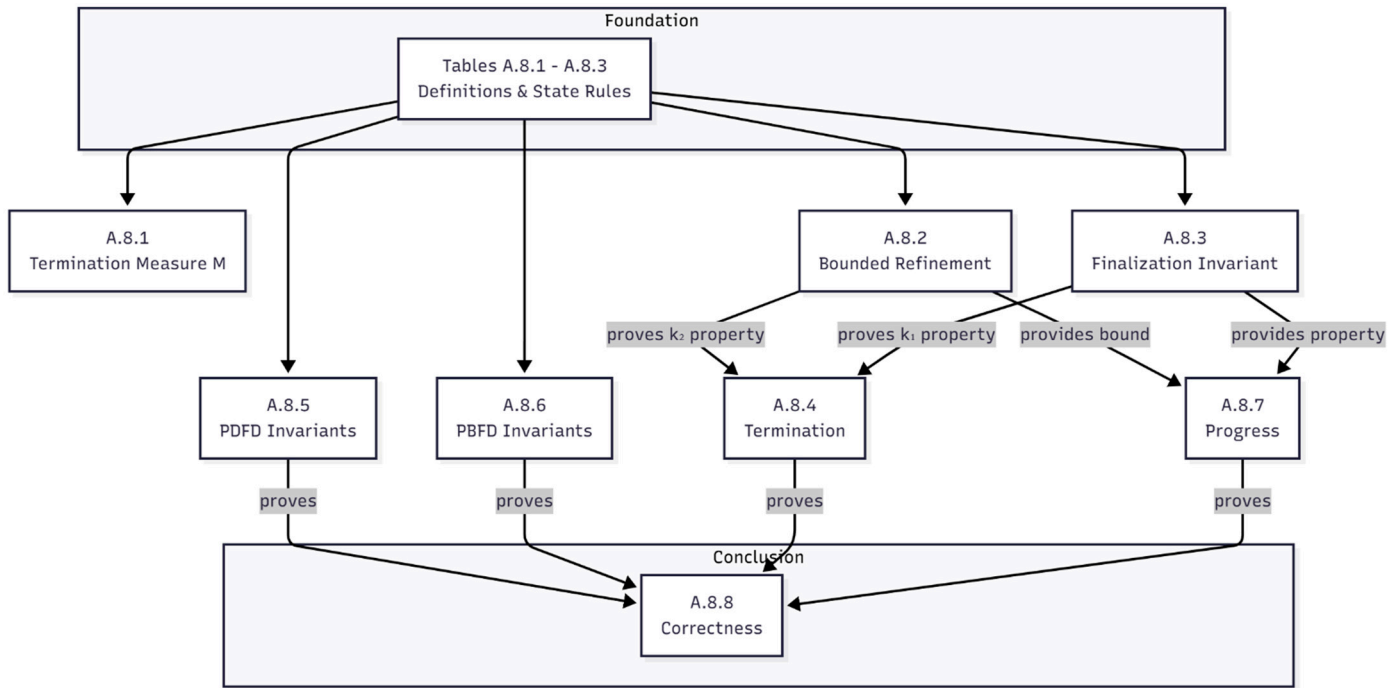
##### Reproducibility

To reproduce results:

- Load pbfd\_model.csp in FDR 4.2.7
- Run all 33 assertions
- Expected outcome: all checks pass with no warnings or counterexamples

#### Appendix A.8. Formal Proofs

This section provides detailed proofs for PBFD/PDFD's core properties (termination and correctness). The proofs are built on the state transition rules defined in Subsection A.8.1 and the lexicographic measure  $M$ . The logical dependencies between the lemmas are shown in Figure A.8.1. The mermaid code for Figure A.8.1. is in A.8.9.



**Figure A.8.1. (Dependency Graph):** Lemmas A.8.2 and A.8.3 depend directly on the state rules; Lemmas A.8.4–A.8.7 build on those; Theorem A.8.8 depends on A.8.4–A.8.7.

#### Appendix A.8.1. Termination Measure and State Transition Analysis

This subsection defines the lexicographic measure and state transition rules that form the basis of the termination argument. The subsequent lemmas prove the critical properties that ensure this measure is well-founded.

##### Definitions for Termination Proofs

**Table A.8.1.** Definitions and Invariants for Termination Proofs

Term / Invariant Name	Type	Formal Definition / Condition
processing_complete(i)	Predicate	All nodes $n$ in level(i) have been processed by the current phase's validation logic.
descendants_validated(n)	Predicate	All nodes in the processed subtree rooted at $n$ have been permanently finalized ( $P(n) = 2$ ).
nrl(j)	Function	The Next Refinement Level function, returning the lowest level $k < j$ that still requires validation.
$K_i$	Constant	A fixed batch size threshold for level $i$ , used to trigger a batch commit in transition PD2b.
Descendant Finalization Invariant	Invariant	A node $n$ is finalized only if all its processed descendants are finalized.
Refinement Locality Invariant	Invariant	Any backtrack targets $j = \text{trace\_origin}(i)$ and the refinement scope is contiguous.
Level-wise Ordering Invariant	Invariant	New patterns at level $i+1$ are produced only after Pattern <sub><math>i</math></sub> is validated. (Ensured by PB4a guard.)
Top-down Finalization Invariant	Invariant	The $S_4$ completion phase proceeds sequentially from level 1 up to $L$ , ensuring no level is skipped. (PB7)
Refinement Locality Invariant (PBF D)	Invariant	Any backtrack targets $j = \text{trace\_origin}(i)$ and the refinement scope is limited to levels $k \in [j, i]$ . (PB3)

##### Lexicographic Measure

Define the tuple

$$M = (k_1, k_2, k_3, k_4)$$

With components:

- $k_1$ : Count of unfinalized nodes —  $k_1 = |\{n \in G \mid P(n) \neq 2\}|$ . (Highest priority.)
- $k_2$ : Remaining refinement attempts across all levels —  $k_2 = \sum_{j \in \text{ActiveLevels}} (R_{\max} - \text{refinement\_attempts}(j))$ . (Finite,  $>0$  in non-terminal states while attempts remain.)
- $k_3 \in \{4, 3, 2, 1, 0\} \rightarrow$  Phase ordinal (map phases to ordinals:  $S_0 = 4, S_1 = 3, S_2 = 2, S_3 = 1, S_4 = 0$ ). A transition to a later phase reduces the numerical value of  $k_3$ )
- $k_4 \in \mathbb{N} \rightarrow$  Intra-phase progress measure (e.g., remaining nodes in a batch or pattern)

We use the lexicographic order on tuples  $(k_1, k_2, k_3, k_4)$ . The termination proof requires that every non-terminal transition causes a strict lexicographic decrease of  $M$ . For each non-terminal transition, we identify the first non-zero component of  $\Delta M$  (from left). The transition guarantees progress if and only if that component is negative. Termination proofs for software systems via lexicographic ranking functions [124–129] support this methodology.

**Notation.** We adopt:  $\text{validated}(n) \Leftrightarrow P(n)=2$ .  $\text{trace\_origin}(i)$  and  $\text{refinement\_attempts}(j)$  are as defined in Sections 3.4.1 and 3.4.2.  $R_{\max} \in \mathbb{N}^+$  is fixed.

#### Relationship of Measure Components to the Rules (Intuitive)

- $k_1$  decreases only on commit/finalization transitions (when nodes are permanently set  $P(n)=2$ ).
- $k_2$  strictly decreases on refinement-entry transitions (each such transition consumes one refinement attempt for a level).
- $k_3, k_4$  measure local progress within phases and provide the necessary descent when  $k_1, k_2$  remain unchanged for short steps. Multiple-component (lexicographic or multi-ranking) proofs remain a mainstream tool in termination analysis [125].

The remainder of this subsection lists the state transitions and their  $\Delta M$  effects, which are used exhaustively in the proofs. The PDFD and Pbfd state transition tables remain unchanged, but  $\Delta M$  annotations are now supported by references [124–129] for lexicographic reasoning and [116,130] for CSP/concurrency reasoning.

Table A.8.2. PDFD State Transition Impacts on  $M$

Rule	Transition	$\Delta M$ ( $\Delta k_1, \Delta k_2, \Delta k_3, \Delta k_4$ )	Key Condition	Type	Progress Justification (first non-zero component)
PD1	$S_0 \rightarrow S_1(1)$	—	$i = 1$ (initial)	Initial	Initialization (not used in lexicographic descent)
PD2	$S_1(i) \rightarrow S_2(i)$	$(0, 0, \downarrow, \downarrow)$	$\text{processing\_complete}(i) \wedge \exists n \in \text{level}(i): \neg \text{validated}(n)$	Non-terminal	$k_3$ decreases ( $S_1 \rightarrow S_2$ ) $\rightarrow$ progress
PD2a	$S_2(i) \rightarrow S_1(j)$	$(0, \downarrow, \uparrow, 0)$	$j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max}$ (backtrack/refinement entry)	Non-terminal	$k_2$ decreases (attempt consumed) $\rightarrow$ progress
PD2b	$S_2(i) \rightarrow S_1(i+1)$	$(\downarrow, 0, \uparrow, 0)$	$\sum_{n \in \text{level}(i)} [P(n)=2] \geq K_i$ (commit/finalize batch)	Non-terminal	$k_1$ decreases (batch commit) $\rightarrow$ progress
PD3	$S_1(j) \rightarrow S_2(j)$	$(0, 0, \downarrow, \downarrow)$	$\text{processing\_complete}(j) \wedge \exists n \in \text{level}(j): \neg \text{validated}(n)$	Non-terminal	$k_3$ decreases ( $S_1 \rightarrow S_2$ ) $\rightarrow$ progress
PD3a	$S_2(j) \rightarrow S_1(\text{nrl}(j), i_{\text{orig}})$	$(0, 0, 0, \downarrow)$	$\forall n \in \text{level}(j): \text{validated}(n) \wedge j < i$ (advance to next refinement level $\text{nrl}(j)$ )	Non-terminal	$k_4$ decreases (intra-phase progress) $\rightarrow$ progress — PD3a treated intra-phase for $M$
PD3b	$S_2(j) \rightarrow S_2(i)$	$(0, 0, 0, \downarrow)$	$\forall n \in \text{level}(j): \text{validated}(n) \wedge j = i$ (resume original validation at level $i$ )	Non-terminal	$k_4$ decreases (intra-phase progress) $\rightarrow$ progress
PD3c	$S_2(j) \rightarrow S_1(j)$	$(0, \downarrow, \uparrow, 0)$	$\text{processing\_complete}(j) \wedge \exists n \in \text{level}(j): \neg \text{validated}(n) \wedge \text{refinement\_attempts}(j) < R_{\max}$ (retry refinement — consumes attempt)	Non-terminal	$k_2$ decreases (attempt consumed) $\rightarrow$ progress

PD4	$S_2(i) \rightarrow S_3(i)$	$(0,0,\downarrow,0)$	$\text{processing\_complete}(i) \wedge (i = L \vee \text{level}(i+1) = \emptyset)$	Non-terminal	$k_3$ decreases ( $S_2 \rightarrow S_3$ ) $\rightarrow$ progress
PD4a	$S_3(i) \rightarrow S_3(i-1)$	$(0,0,0,\downarrow)$	$\forall n \in \text{level}(i): \text{validated}(n) \wedge \text{descendants\_validated}(n)$	Non-terminal	$k_4$ decreases (intra-phase progress) $\rightarrow$ progress
PD4b	$S_3(i) \rightarrow S_1(j)$	$(0,\downarrow,\uparrow,0)$	$\exists n \in \text{level}(i): \neg \text{validated}(n) \wedge j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max}$ (backtrack from bottom-up)	Non-terminal	$k_2$ decreases (attempt consumed) $\rightarrow$ progress
PD5	$S_3(2) \rightarrow S_4(1)$	$(0,0,\downarrow,\downarrow)$	$i = 2$ (bottom-up progress boundary)	Non-terminal	$k_3$ decreases ( $S_3 \rightarrow S_4$ ) $\rightarrow$ progress
PD6	$S_4(i) \rightarrow S_4(i+1)$	$(\downarrow,0,0,0)$	$\forall n \in \text{level}(i): \text{validated}(n)$	Non-terminal	$k_1$ decreases (commit/finalize of level $i$ ).
PD6a	$S_4(i) \rightarrow S_1(j)$	$(0,\downarrow,\uparrow,0)$	$\exists n \in \text{level}(i): \neg \text{validated}(n) \wedge j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max}$ (backtrack from completion)	Non-terminal	$k_2$ decreases (attempt consumed) $\rightarrow$ progress
PD6b	$S_4(i) \rightarrow S_5$	—	$\exists n \in \text{level}(i): \neg \text{validated}(n) \wedge$ (no refinement path remains for $\text{trace\_origin}(i)$ ) (equivalently $\text{refinement\_attempts}(\text{trace\_origin}(i)) \geq R_{\max}$ )	Terminal 1	Terminal (error)
PD7	$S_4(L) \rightarrow T$	—	$\forall i \in [1,L], \forall n \in \text{level}(i): \text{validated}(n)$	Terminal 1	Terminal (complete)
PD8 (generalized)	From $\in \{S_1(j), S_2(j), S_3(j)\} \rightarrow S_5$	—	$\text{refinement\_attempts}(j) \geq R_{\max}$ (no further attempts remain for level $j$ )	Terminal 1	Terminal (exhaustion)

**Note:** For the lexicographic measure  $M$ , PD3a ( $S_2 \rightarrow S_1(\text{nr}(j), i_{\text{orig}})$ ) is treated as intra-phase progress ( $k_3$  unchanged) and the progress for this transition is recorded in  $k_4$ .

For every non-terminal rule in Table A.8.2, the lexicographic measure  $M = (k_1, k_2, k_3, k_4)$  undergoes a strict decrease. This is guaranteed by the following:

- **$k_1$  Strict Decrease:** The finalization transition PD2b and PD6 strictly reduces  $k_1$  (unfinalized nodes), overriding any changes in lower-priority components.
- **$k_2$  Strict Decrease:** The refinement-entry transitions PD2a, PD3c, PD4b, and PD6a strictly reduce  $k_2$  (remaining refinement attempts), ensuring lexicographic progress even when backtracking causes  $k_3$  to increase temporarily.
- **$k_3$  Decrease Role:** Phase-progression transitions (PD2, PD3, PD4, PD5) strictly reduce  $k_3$ , ensuring forward progress. Although  $k_3$  may temporarily increase during backtracking (PD2a, PD2b, PD3c, PD4b, PD6a), the overall lexicographic decrease is maintained by strict reduction of higher-priority components  $k_1$  or  $k_2$ .
- **$k_4$  Strict Decrease:** The intra-phase traversals PD3a, PD3b, and PD4a strictly reduce  $k_4$  (intra-phase progress), providing the necessary descent when all higher-priority components remain unchanged.

Terminal rules PD6b, PD7, and PD8 end the computation, yielding no further measure. Since every non-terminal transition guarantees a strict lexicographic decrease in  $M$ , the measure is well-founded, and the algorithm is guaranteed to terminate.

Table A.8.3. PBF State Transition Impacts on  $M$

Rule	Transition	$\Delta M$ ( $\Delta k_1, \Delta k_2, \Delta k_3,$ $\Delta k_4$ )	Key Condition	Type	Progress Justification
------	------------	---	---------------	------	------------------------

PB1	$S_0 \rightarrow S_1(1)$	—	$i = 1$	Initial	—
PB2	$S_1(i) \rightarrow S_2(i)$	$(0,0,\downarrow,\downarrow)$	$\exists n \in \text{Pattern}_i; \neg \text{validated}(n)$	Non-terminal	$k_3$ decreases $(3 \rightarrow 2)$ .
PB2a	$S_1(i) \rightarrow S_3(i)$	$(0,0,\downarrow,0)$	$\forall n \in \text{Pattern}_i; \text{validated}(n)$	Non-terminal	$k_3$ decreases $(3 \rightarrow 1)$ .
PB3	$S_2(i) \rightarrow S_1(j)$	$(0,\downarrow,\uparrow,0)$	$(\exists n \in \text{Pattern}_i; \neg \text{validated}(n)) \wedge j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max}$ (refinement entry)	Non-terminal	$k_2$ decreases (attempt consumed).
PB3a	$S_1(j) \rightarrow S_2(j)$	$(0,0,\downarrow,\downarrow)$	$\exists n \in \text{Pattern}_j; \neg \text{validated}(n)$	Non-terminal	$k_3$ decreases $(3 \rightarrow 2)$ .
PB3a1	$S_2(j) \rightarrow S_3(j)$	$(0,0,\downarrow,0)$	$\forall n \in \text{Pattern}_j; \text{validated}(n)$	Non-terminal	$k_3$ decreases $(2 \rightarrow 1)$ .
PB3a2	$S_2(j) \rightarrow S_1(j)$	$(0,\downarrow,\uparrow,0)$	$\exists n \in \text{Pattern}_j; \neg \text{validated}(n) \wedge \text{refinement\_attempts}(j) < R_{\max}$ (retry refinement)	Non-terminal	$k_2$ decreases (attempt consumed).
PB3a3	$S_2(j) \rightarrow S_5$	—	$\exists n \in \text{Pattern}_j; \neg \text{validated}(n) \wedge \text{refinement\_attempts}(j) \geq R_{\max}$ (refinement exhausted)	Terminal	—
PB3b	$S_1(j) \rightarrow S_3(j)$	$(0,0,\downarrow,0)$	$\forall n \in \text{Pattern}_j; \text{validated}(n)$	Non-terminal	$k_3$ decreases $(3 \rightarrow 1)$ .
PB3c	$S_2(i) \rightarrow S_5$	—	$(\exists n \in \text{Pattern}_i; \neg \text{validated}(n)) \wedge (\text{trace\_origin}(i) \text{ undefined } \vee \text{refinement\_attempts}(\text{trace\_origin}(i)) \geq R_{\max})$ (no valid trace_origin or attempts exhausted)	Terminal	—
PB4	$S_2(i) \rightarrow S_3(i)$	$(0,0,\downarrow,0)$	$\forall n \in \text{Pattern}_i; \text{validated}(n)$ (refinement validated)	Non-terminal	$k_3$ decreases $(2 \rightarrow 1)$ .
PB4a	$S_3(i) \rightarrow S_1(i+1)$	$(\downarrow,0,\uparrow,0)$	$i < L \wedge \text{Pattern}_{[i+1]} \neq \emptyset$ ((commit/finalize))	Non-terminal	$k_1$ decreases (commit/finalize of $\text{Pattern}_i$ ).
PB4b	$S_3(i) \rightarrow S_4(1)$	$(0,0,\downarrow,0)$	$i = L \vee \text{Pattern}_{[i+1]} = \emptyset$ (enter completion)	Non-terminal	$k_3$ decreases $(1 \rightarrow 0)$ .
PB5	$S_3(j) \rightarrow S_1(j+1)$	$(0,0,0,\downarrow)$	$j < i$ (refinement-range progress)	Non-terminal	$k_4$ decreases (refinement-range progress).
PB6	$S_3(j) \rightarrow S_3(i)$	$(0,0,0,\downarrow)$	$j = i$ (return from refinement)	Non-terminal	$k_4$ decreases (intra-phase progress/return).
PB7	$S_4(i) \rightarrow S_4(i+1)$	$(\downarrow,0,0,0)$	$\forall n \in \text{Pattern}_i; \text{processed}(n)$	Non-terminal	$k_1$ decreases (commit/finalize of $\text{Pattern}_i$ ).
PB7a	$S_4(i) \rightarrow S_1(j)$	$(0,\downarrow,\uparrow,0)$	$\exists n \in \text{Pattern}_i; \neg \text{processed}(n) \wedge j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max}$ (backtrack from completion)	Non-terminal	$k_2$ decreases (attempt consumed).
PB7b	$S_4(i) \rightarrow S_5$	—	$\exists n \in \text{Pattern}_i; \neg \text{processed}(n) \wedge \neg (j = \text{trace\_origin}(i) \wedge \text{refinement\_attempts}(j) < R_{\max})$ (unvalidated nodes and no refinement option)	Terminal	—
PB8	$S_4(L) \rightarrow T$	—	$\forall i \in [1,L], \forall n \in \text{Pattern}_i; \text{validated}(n)$ (all validated)	Terminal	—

PB9	$S_1(j) \rightarrow S_5$	—	refinement_attempts(j) $\geq R_{\max}$ (attempts exhausted)	Terminal	—
-----	--------------------------	---	--	----------	---

**Notes:**

- Transitions that decrement  $k_2$  (remaining refinement attempts) are PB3, PB3a2, and PB7a. Each consumes exactly one attempt.
- $k_1$  (unfinalized nodes) is strictly reduced only by the commit/finalization transitions PB4a (forward pass) and PB7 (completion phase). These dominate all lower-priority changes.
- PB4a is the forward commit step finalizing Pattern<sub>i</sub> before moving to Pattern<sub>i+1</sub>.
- PB5 and PB6 represent intra-refinement navigation and strictly reduce  $k_4$ , not  $k_1$ .

For every non-terminal rule in Table A.8.3, the lexicographic measure

$M = (k_1, k_2, k_3, k_4)$  strictly decreases. This is ensured by:

- **$k_1$  Strict Decrease:** PB4a and PB7 finalize nodes, reducing the highest-priority component.
- **$k_2$  Strict Decrease:** PB3, PB3a2, and PB7a consume refinement attempts and strictly reduce  $k_2$ , ensuring lexicographic progress even when backtracking causes  $k_3$  to increase temporarily.
- **$k_3$  Decrease Role:** The phase-progression transitions PB2, PB2a, PB3a, PB3a1, PB3b, PB4, and PB4b strictly reduce  $k_3$  (phase ordinal), ensuring forward progress through the main execution path. Although  $k_3$  may temporarily increase in commit transition PB4a and refinement/backtracking transitions (PB3, PB3a2, PB7a), the overall lexicographic decrease is guaranteed by the strict reduction of higher-priority components  $k_1$  or  $k_2$ .
- **$k_4$  Strict Decrease:** PB5 and PB6 reduce intra-phase progress when higher-priority components remain unchanged.

Terminal rules PB3a3, PB3c, PB7b, PB8, and PB9 end the computation and do not require measure reduction.

Since every non-terminal transition strictly decreases  $M$  lexicographically, the measure is well-founded and termination is guaranteed.

■

### Appendix A.8.2. Lemma (Bounded Refinement)

**Statement.** For all levels  $k \in [1, L]$ :  $\square(\text{refinement\_attempts}(k) \leq R_{\max})$ . In any non-terminal state, any active refinement target  $j$  satisfies  $\text{refinement\_attempts}(j) < R_{\max}$ . Terminal states  $S_5$  are reached only when an attempt bound is exhausted.

**Proof.**

- **Base Case.** At initial state  $S_0$ :  $\forall k: \text{refinement\_attempts}(k)=0 \leq R_{\max}$ . The statement holds vacuously.
- **Inductive Step.** Assume in state  $S$  the invariant holds. Consider a transition  $S \rightarrow S'$ . Only refinement-entry rules increment  $\text{refinement\_attempts}(j)$ . From Tables A.8.2. - A.8.3. these are explicitly guarded by  $\text{refinement\_attempts}(j) < R_{\max}$  (PD2a, PD3c, PD4b, PD6a for PDFD; PB3, PB3a2, PB7a for PBFd). Hence any increment preserves  $\text{refinement\_attempts}(j) \leq R_{\max}$ . All other rules leave all refinement counters unchanged. Terminal rules (e.g., PD6b, PD8, PB3a3, PB9, PB7b, PB3c) fire only when  $\text{refinement\_attempts}(j) \geq R_{\max}$  for some  $j$ . Terminal transitions (which fire only when  $\text{refinement\_attempts}(j) \geq R_{\max}$ ) do not increment counters, preserving the invariant.
- **Conclusion.** By induction on transitions, the counter is bounded by  $R_{\max}$  at all times. Since at most  $L$  levels can each suffer at most  $R_{\max}$  increments, the total number of refinement attempts is bounded by  $L \cdot R_{\max}$ . Thus  $k_2$  is finite and strictly decreases on each refinement entry until exhaustion.

■

### Appendix A.8.3. Lemma (Finalization Monotonicity)

**Statement.** Once a node  $n$  has been permanently finalized ( $P(n)=2$ ), it remains finalized unless a refinement backtrack explicitly resets it. Resets occur only on refinement-entry rules and are strictly controlled by attempt bounds. Moreover, across execution,  $k_1$  (the count of unfinalized nodes) is monotone non-increasing except when a controlled reset (paired with a decrease in  $k_2$ ) occurs.

**Proof.**

- **Base Case.** Initially no node is finalized ( $P(n) \neq 2$  for all  $n$ ). The statement holds vacuously in the initial state.
- **Finalization Step:** Per Tables A.8.2-A.8.3, the rules that set nodes to finalized (i.e., produce committed  $P(n)=2$ ) are the commit/finalize transitions PDFD: PD2b and PD6; Pbfd: PB4a and PB7). In both algorithms, these transitions strictly reduce  $k_1$ . No other transition creates  $P(n)=2$ .
- **Reset rules.** The only rules that may reset previously finalized nodes to non-finalized ones (i.e., potentially  $\Delta k_1 > 0$ ) are refinement-entry/backtrack rules (PD2a, PD3c, PD4b, PD6a; PB3, PB3a2, PB7a). Each such rule has the guard  $\text{refinement\_attempts}(j) < R_{\max}$  and the operational semantics of attempting correction. On taking such a rule,  $k_2$  strictly decreases (since  $\text{refinement\_attempts}(j)$  is incremented). No non-refinement rule resets finalized nodes.
- **Lexicographic compensation.** Therefore, any transition that reverses finalization (i.e., a reset that potentially increases  $k_1$ ) is guaranteed to be a refinement-entry transition that strictly decreases  $k_2$ . Hence the pair  $(k_1, k_2)$  is lexicographically non-increasing across transitions: a rise in  $k_1$  is strictly compensated by a fall in  $k_2$ .
- **Conclusion.**  $k_1$  is monotone non-increasing unless a bounded, recorded refinement reset occurs; such resets are bounded by Lemma A.8.2. Thus the finalization invariant holds.

■

#### Appendix A.8.4. Lemma (Termination Guarantee)

**Statement.** For any finite tree  $G = (V, E)$  and finite parameters  $L, R_{\max} \in \mathbb{N}^+$ , any execution of PDFD or Pbfd terminates in either:

- Success T: all nodes finalized ( $\forall n \in V: P(n) = 2$ ), or
- Bounded failure  $S_5$ : refinement exhausted for some level ( $\exists j: \text{refinement\_attempts}(j) = R_{\max}$ ).

**Proof.**

- **Well-foundedness.** Each component of  $M = (k_1, k_2, k_3, k_4)$  ranges over a well-founded (finite or well-ordered) set:
  - $0 \leq k_1 \leq |V|$ .
  - $0 \leq k_2 \leq L \cdot R_{\max}$ .
  - $k_3 \in \{0, 1, 2, 3, 4\}$ .
  - $k_4$  bounded by finite batch sizes ( $\leq |V|$ ).
 Thus no infinite strictly decreasing sequence in  $M$  exists.
- **Measure descent on transitions.** From the exhaustive  $\Delta M$  annotations in Tables A.8.2- A.8.3, every non-terminal transition strictly decreases  $M$  in lexicographic order:
  - If a non-terminal transition finalizes nodes, it decreases  $k_1$ .
  - If it is a refinement-entry, it decreases  $k_2$ .
  - Otherwise the phase/intra-phase components  $(k_3, k_4)$  strictly decrease.
- **No infinite execution sequences.** Since  $M$  decreases on every non-terminal step and  $M$  is well-founded, the system cannot execute infinitely many non-terminal moves. Therefore, every execution sequence reaches a terminal state.
- **Terminal classification.** Terminal rules in Tables A.8.2-A.8.3 correspond exactly to either all nodes validated (PD7, PB8) or to a bounded failure from exhausted refinements (PD6b, PD8, PB3a3, PB3c, PB7b, PB9). These cases partition all terminal states. Hence termination leads to either T or  $S_5$ .

■

## Appendix A.8.5. Lemma (Invariant Preservation for PDFD)

*Statement.* Across all reachable states of PDFD, the following invariants hold:

- **Descendant finalization invariant.** A node at level  $i$  is not considered finally complete unless all nodes in its processed subtree are finalized (guards enforced by PD4a/PD6/PD7).
- **Refinement locality.** Backtracks always target  $j = \text{trace\_origin}(i)$  with  $j \leq i$ ; refinement scope is contiguous and anchored.

**Proof.**

- **Base Case.** The initial state  $S_0$  satisfies both invariants vacuously: no nodes are finalized yet, and no refinement operations have been initiated. Therefore, both the descendant finalization invariant and refinement locality invariant hold trivially.
- **Inductive Step.** Assume both invariants hold in state  $S$ . Consider any transition  $S \rightarrow S'$  according to Table A.8.2. We show that  $S'$  preserves both invariants:
  - **Descendant finalization invariant.** Transitions that finalize nodes or advance levels (PD4a, PD6, PD7) are strictly guarded by conditions requiring  $\text{validated}(n)$  or  $\text{descendants\_validated}(n)$  to be true. These guards explicitly enforce that a node is finalized only when its processed descendants are already finalized. All other transitions either do not affect finalization status or are refinement backtracks that temporarily reset nodes (addressed by refinement locality).
  - **Refinement locality invariant.** Backtrack transitions (PD2a, PD3c, PD4b, PD6a) compute the target level  $j$  using the  $\text{trace\_origin}$  function, which by definition satisfies  $j \leq i$ . The guard conditions ensure that refinement scope remains contiguous within the range  $[j, i]$ . Non-backtrack transitions do not modify refinement relationships.
- **Conclusion.** By induction on the transition sequence, both invariants are preserved across all reachable states. The exhaustive nature of the state transitions in Table A.8.2 guarantees that no invariant-violating state is reachable.

■

## Appendix A.8.6. Lemma (Invariant Preservation for PBFD)

*Statement.* Across all reachable states of PBFD:

1. **Level-wise ordering.** Children/pattern at level  $i+1$  are produced only after  $\text{Pattern}_i$  is validated (PB4a).
2. **Top-down finalization in completion.** PB7/PB8 iterate from level 1 upward without skipping.
3. **Refinement locality.** Backtracks always target  $j = \text{trace\_origin}(i)$  with  $j \leq i$ ; refinement scope is contiguous and anchored (PB3).

**Proof.**

- **Base Case.** The initial state  $S_0$  satisfies all three invariants vacuously: no patterns have been processed, no finalization has begun, and no refinement operations have been initiated. Therefore, all invariants hold trivially in the initial state.
- **Inductive Step.** Assume all three invariants hold in state  $S$ . Consider any transition  $S \rightarrow S'$  according to Table A.8.3. We show that  $S'$  preserves all invariants:
  - **Level-wise Ordering Invariant.** The transition PB4a, which advances from  $\text{Pattern}_i$  to  $\text{Pattern}_{i+1}$ , is strictly guarded by the condition that  $\text{Pattern}_i$  is fully validated. This guard ensures that no pattern at level  $i+1$  is produced unless the preceding pattern has been successfully validated. All other transitions either operate within a single level or do not produce new patterns.
  - **Top-down Finalization Invariant.** The completion phase transitions (PB7, PB8) progress sequentially through  $S_4(i) \rightarrow S_4(i+1)$ , with each step guarded by  $\forall n \in \text{Pattern}_i; \text{processed}(n)$ . This ensures that levels are finalized in strict ascending order from 1 to  $L$  without skipping. Backtrack transitions from  $S_4$  (PB7a) do not violate this invariant as they temporarily exit completion mode.

- **Refinement Locality Invariant.** Refinement backtrack transitions (PB3, PB3a2, PB7a) compute the target level  $j$  using the `trace_origin` function, which by definition satisfies  $j \leq i$ . The guard conditions and operational semantics ensure that refinement scope remains contiguous within  $[j, i]$ . Non-refinement transitions do not modify these relationships.
  - **Conclusion.** By induction on the transition sequence, all three invariants are preserved across all reachable states. The exhaustive nature of the state transitions in Table A.8.3. guarantees that no invariant-violating state is reachable.
- 

#### Appendix A.8.7. Lemma (Unified Progress)

*Statement.* From any non-terminal state, there exists an enabled transition whose execution causes a strict lexicographic decrease in  $M$ .

##### Proof.

This is guaranteed by the design of the state machines and measure: By the exhaustive annotation of Tables A.8.2 and A.8.3, for every non-terminal state, at least one transition rule is enabled by its guard condition, and the  $\Delta M$  for that rule shows a strict lexicographic decrease. This is by construction of the state machines. Lemmas A.8.2 and A.8.3 guarantee that decreases in  $k_2$  and  $k_1$  are well-founded and therefore prevent indefinite stuttering in  $k_3, k_4$ .

■

#### Appendix A.8.8. Theorem (Total Correctness)

*Statement.* PDFD and PBFD always terminate and upon termination satisfy their postconditions:

- Terminate in  $T$  (all nodes validated) or  $S_5$  (refinement exhausted).
- Structural invariants (descendant finalization, refinement locality, level ordering) hold at all reachable states.

##### Proof.

Follows directly from Lemmas A.8.2–A.8.7 and the invariant guarantees in A.8.5 and A.8.6:

- **Termination** by Lemma A.8.4.
- **Partial correctness** by Lemmas A.8.5–A.8.6 (invariants). Upon termination in state  $T$ , the postcondition  $\forall n \in V, P(n)=2$  is met directly by the guard of the terminal rule (PD7/PB8). The structural invariants ensure this final state is internally consistent.
- **Progress/no stalling** by Lemma A.8.7.

Therefore both algorithms satisfy total correctness: termination and preservation of required invariants; terminal states meet the declared postconditions.

■

##### Corollaries

- A.8.2.1 (Boundedness). Total number of refinement attempts  $\leq L \cdot R_{\max}$ .
- A.8.3.1 (Finalization Permanence). Once  $P(n)=2$  outside an active refinement rollback, it remains 2; any temporary reset is only through guarded refinement-entry transitions, is bounded by Lemma A.8.2, and is always accompanied by a strict decrease in the  $k_2$  component of the measure  $M$ .
- A.8.4.1 (Temporal completeness). From start, eventually the run reaches either success  $T$  or bounded failure  $S_5$ :  $\square(\text{start} \Rightarrow \diamond(T \vee S_5))$ .

#### Appendix A.8.9. Proof Mermaid Code

##### flowchart TD

```

subgraph Foundation [Foundation]
  A[Tables A.8.1-A.8.3<br>Definitions & State Rules]
end

```

end

```

A --> B[A.8.1<br>Termination Measure M]

A --> C[A.8.2<br>Bounded Refinement]
A --> D[A.8.3<br>Finalization Invariant]
A --> E[A.8.5<br>PDFD Invariants]
A --> F[A.8.6<br>PBFD Invariants]

C -- proves  $k_2$  property --> G[A.8.4<br>Termination]
D -- proves  $k_1$  property --> G

C -- provides bound --> H[A.8.7<br>Progress]
D -- provides property --> H

subgraph Conclusion [Conclusion]
  I[A.8.8<br>Correctness]
end

E -- proves --> I
F -- proves --> I
G -- proves --> I
H -- proves --> I

```

#### Appendix A.9. TLE Mermaid Code, Algorithm, and Process Algebra

Appendix A.9 provides the formal specification for the Three-Level Encapsulation (TLE) technique, covering its Mermaid diagrams, pseudocode, and CSP model.

##### Appendix A.9.1. Structural Workflow Mermaid Code

```

graph TD
  %% Compact Layout for Single Column
  subgraph Legend
    LG1[Level N]
    LG2[Level N+1]
    LG3[Level N+2]

    %% Vertical layout within legend
    LG1 --- LG2
    LG2 --- LG3
  end

  %% Main structure with condensed labels
  G[Grandparent] --> P1[Parent A]
  G --> P2[Parent B]
  G --> P3[Parent C]

  P1 --> B1[Bitmask A1]
  P2 --> B2[Bitmask B1]
  P3 --> B3[Bitmask C1]

  %% Colors
  classDef level1 fill:#E1F5FE,stroke:#039BE5

```

```
classDef level2 fill:#FFF8E1,stroke:#FBC02D
classDef level3 fill:#E8F5E9,stroke:#388E3C
```

```
class G level1
class P1,P2,P3 level2
class B1,B2,B3 level3
class LG1 level1
class LG2 level2
class LG3 level3
```

### Appendix A.9.2. State Machine Mermaid Code

#### stateDiagram-v2

```
state "S0: Idle" as S0
state "S1: Data Loaded" as S1
state "S2: Hierarchy Resolved" as S2
state "S3: Children Evaluated" as S3
state "S4: Children Updated" as S4
state "S5: Changes Committed" as S5
state "S6: Workflow Finalized" as S6

[*] --> S0 : TLE1 - System Start
S0 --> S1 : TLE2 - initiate_workflow(Grandparent)
S0 --> S6 : TLE11 - ¬has_unprocessed_unit()

S1 --> S2 : TLE3 - resolve_hierarchy()
S2 --> S3 : TLE4 - evaluate_children()

S3 --> S4 : TLE5 - update_required ∧ apply_update()
S3 --> S5 : TLE6 - ¬update_required

S4 --> S5 : TLE7 - persist_changes()

S5 --> S0 : TLE8 - has_next_unit()
S5 --> S6 : TLE9 - ¬has_next_unit()

S6 --> S0 : TLE10 - Workflow Complete
```

### Appendix A.9.3. Algorithm (Pseudo Code)

#### Algorithm TLE(Pages)

Procedure TLE\_EventDriven(Units)

Input: Units – list of TLE data units (e.g., grandparent entities) to process

Output: Tree with bitmask-encoded children selections finalized

1: currentState ← S<sub>0</sub> // TLE1: [\*] → S<sub>0</sub>. System Start

2: currentUnit ← NULL

// TLE process runs continuously, reacting to external events

3: while System\_Running do

4:     switch currentState

5:         case S<sub>0</sub>: // Idle (TLE\_S0). Awaiting load or finalization signal.

6:             // TLE2: load(u) → S<sub>1</sub> | TLE11: no\_next\_unit(u) → S<sub>6</sub>

7:             event ← WaitForEvent({load, no\_next\_unit}) // Wait for next unit or end-of-

batch

```

8:         if event.type == load then
9:             currentUnit ← event.Unit // Store the unit parameter (u)
10:            currentState ← S1(currentUnit)
11:            else if event.type == no_next_unit then
12:                currentUnit ← event.Unit // Unit being finalized (passed from
environment)
13:                currentState ← S6(currentUnit)
14:                // Note: Unit parameter is always received from the environment here
(load/no_next_unit)
15:
16:            case S1(u): // Data Loaded (TLE_S1(u)). Awaiting hierarchy resolution.
17:                // TLE3: hierarchy_resolved(u) → S2
18:                event ← WaitForEvent({hierarchy_resolved})
19:                if event.Unit == u then // Check for unit-specific synchronization
20:                    resolve_hierarchy() // TLE3 Action (Internal resolution)
21:                    currentState ← S2(u)
22:
23:            case S2(u): // Hierarchy Resolved (TLE_S2(u)). Awaiting children evaluation.
24:                // TLE4: children_evaluated(u) → S3
25:                event ← WaitForEvent({children_evaluated})
26:                if event.Unit == u then
27:                    child_nodes ← evaluate_children() // TLE4 Action: Iterative READ
28:                    currentState ← S3(u)
29:
30:            case S3(u): // Children Evaluated (TLE_S3(u)). Conditional path: update or skip.
31:                // TLE5: children_updated(u) → S4 | TLE6: skip_update(u) → S5
32:                event ← WaitForEvent({children_updated, skip_update})
33:                if event.Unit == u then
34:                    if event.type == children_updated then // TLE5 (WRITE required)
35:                        apply_update(child_nodes) // TLE5 Action
36:                        currentState ← S4(u)
37:                    else // event.type == skip_update (TLE6)
38:                        currentState ← S5(u)
39:
40:            case S4(u): // Children Updated (TLE_S4(u)). Awaiting changes commit.
41:                // TLE7: changes_committed(u) → S5
42:                event ← WaitForEvent({changes_committed})
43:                if event.Unit == u then
44:                    persist_changes() // TLE7 Action: COMMIT
45:                    currentState ← S5(u)
46:
47:            case S5(u): // Changes Committed (TLE_S5(u)). Signalling readiness or finalization.
48:                // TLE8: has_next_unit → S0 | TLE9: no_next_unit(u) → S6
49:                // The process emits the readiness/finalization signal and transitions
immediately.
50:                if HasNextUnitAvailable() then
51:                    EmitEvent(has_next_unit) // TLE8 Action (Unparameterized signal)
52:                    currentState ← S0 // Loop back to S0 to await new work
53:                else
54:                    EmitEvent(no_next_unit.u) // TLE9 Action (Parameterized signal)
55:                    currentState ← S6(u)

```

```

56:
57:     case S6(u): // Workflow Finalized (TLE_S6(u)). Final action and system reset.
58:         // TLE10: finalize_process(u) → S0
59:         EmitEvent(finalize_process.u) // TLE10 Action
60:         currentState ← S0 // TLE10: Transition back to S0 to await new unit
61:
62:     end switch
63: end while
64: return
End Procedure

```

#### Appendix A.9.4. CSP Implementation and Formal Verification

The complete CSPM model (FDR 4.2.7 compatible) implementing all operations from Algorithm A.9.3 and state transitions from Table 48 and Table 49 is available in our supplementary repository.

**Verification Status:** All 49 formal properties were successfully verified, including deadlock freedom, divergence freedom, deterministic behavior, correct sequencing of TLE1–TLE11 transitions, and behavioral conformance to the abstract specification (TLE\_Abstract\_Process). Unit-specific guarantees such as WaitForEvent(u) synchronization, EmitEvent(u) propagation, and recurrence  $S_6 \rightarrow S_0$  were validated.

##### Repository Access:

GitHub: [https://github.com/IBM-Consulting-Formal-Methods/TLE\\_CSP](https://github.com/IBM-Consulting-Formal-Methods/TLE_CSP) (commit: 7e5b6c3)

The model includes all TLE processes ( $S_0, S_1(u), S_2(u), S_3(u), S_4(u), S_5(u), S_6(u)$ ), event channels, and unit parameterization ( $u_1, u_2, u_3$ ) as documented in Tables A.9.1. - A.9.2. The repository README provides detailed verification instructions and complete FDR 4.2.7 assertion results.

#### Appendix A.9.5. TLE (Three-Level Encapsulation) Technique Tables

The TLE technique's formal specification is further detailed through Table A.9.1, which provides a unified set of definitions for both the pseudocode and CSP models. Table A.9.2. then outlines the core CSP process algebra, detailing the state transitions and key events that correspond to the pseudocode.

**Table A.9.1.** TLE Technique - Unified Definitions (Pseudocode + CSP)

Pseudocode Term	Type	Description	Pseudocode Lines	CSP Mapping
<b>Algorithm &amp; States</b>				
Algorithm TLE(Units)	Meta-Process	Coordinates the tree-leaf encoding pipeline.	Header	TLE_Process(start→TLE_S <sub>0</sub> )
currentState	State Variable	Tracks the current stage of the TLE process.	1, 4, 10, 13, 21, 28, 36, 38, 45, 52, 55, 60	(Implicit in CSP State Processes TLE_S <sub>x</sub> (u))
S <sub>0</sub>	State	Idle. Waiting for input.	5, 52, 60	TLE_S <sub>0</sub>
S <sub>1</sub>	State	Data Loaded. A TLE unit is loaded.	10, 16	TLE_S <sub>1</sub> (u)
S <sub>2</sub>	State	Hierarchy Resolved. Parent levels identified.	21, 23	TLE_S <sub>2</sub> (u)
S <sub>3</sub>	State	Children Evaluated. Child states processed.	28, 30	TLE_S <sub>3</sub> (u)
S <sub>4</sub>	State	Children Updated. Child states modified.	36, 40	TLE_S <sub>4</sub> (u)
S <sub>5</sub>	State	Changes Committed. Modifications persisted.	38, 45, 47	TLE_S <sub>5</sub> (u)

$S_6$	System End State	Workflow Finalized. Process complete.	13, 55, 57	TLE_S6(u)
<b>Functions &amp; Actions</b>				
LOAD(Grandparent)	Core TLE Op	Loads a TLE data unit.	9	load?u:UNIT (Input)
resolve_hierarchy()	Processing Function	Resolves and validates hierarchy.	20	hierarchy_resolved.u (Output)
evaluate_children()	Processing Function	Reads and logically processes children.	27	children_evaluated.u (Output)
apply_update(...)	Core TLE Op	WRITE. Modifies child states.	35	children_updated.u (Output)
persist_changes()	Core TLE Op	COMMIT. Persists changes.	44	changes_committed.u (Output)
finalize_process()	System Function	Completes the TLE algorithm.	59	finalize_process.u (Output)
<b>Conditions</b>				
update_required	Condition	Trigger for WRITE operation.	34	(Implied by children_updated.u choice in TLE_S3)
has_next_unit()	Condition / Signal	Checks if more units exist.	50	has_next_unit (Output, Valueless)
$\exists$ unprocessed unit...	Condition	Checks if more units exist.	7	(Implicit in load?u:UNIT choice in TLE_S0)
<b>CSP-Specific Events</b>				
load	CSP Input	Signals a unit is ready for processing.	7	load?u:UNIT
no_next_unit	CSP I/O	Signals no more units.	7, 11, 48, 54	S0: Input (?u); S5: Output (.u)
skip_update	CSP Output	Signals no update was required, skipping to commit.	32, 37	skip_update.u

Table A.9.2. TLE Technique - CSP Process Algebra Core (States + Transitions)

CSP Process	Key Transitions (TLE Ref.)	Pseudocode Lines	CSP Events (Simplified)
$S_0$ (TLE_S0)	TLE1: Start $\rightarrow S_0$	1	(start $\rightarrow$ TLE_S0) $\rightarrow$ TLE_S0 (via TLE_Process)
	TLE2: load(u) $\rightarrow S_1$	7–10	load?u:UNIT $\rightarrow$ TLE_S1(u)
$S_1$ (u) (TLE_S1(u))	TLE11: no_next_unit(u) $\rightarrow S_6$	7, 11–13	no_next_unit?u:UNIT $\rightarrow$ TLE_S6(u)
$S_2$ (u) (TLE_S2(u))	TLE3: hierarchy_resolved(u) $\rightarrow S_2$	18–21	hierarchy_resolved.u $\rightarrow$ TLE_S2(u)
$S_3$ (u) (TLE_S3(u))	TLE4: children_evaluated(u) $\rightarrow S_3$	25–28	children_evaluated.u $\rightarrow$ TLE_S3(u)
	TLE5: children_updated(u) $\rightarrow S_4$	32, 34–36	children_updated.u $\rightarrow$ TLE_S4(u)
	TLE6: skip_update(u) $\rightarrow S_5$	32, 37–38	skip_update.u $\rightarrow$ TLE_S5(u)
$S_4$ (u) (TLE_S4(u))	TLE7: changes_committed(u) $\rightarrow S_5$	42–45	changes_committed.u $\rightarrow$ TLE_S5(u)
$S_5$ (u) (TLE_S5(u))	TLE8: has_next_unit $\rightarrow S_0$	50–52	has_next_unit $\rightarrow$ TLE_S0
	TLE9: no_next_unit(u) $\rightarrow S_6$	53–55	no_next_unit.u $\rightarrow$ TLE_S6(u)
$S_6$ (u) (TLE_S6(u))	TLE10: finalize_process(u) $\rightarrow S_0$	58–60	finalize_process.u $\rightarrow$ TLE_S0
Top-Level (TLE_Process)	System Start $\rightarrow S_0$	1	start $\rightarrow$ TLE_S0

### Appendix A.9.6. Formal Verification Methodology and Scope

#### Verification Framework

All analyses were conducted using FDR 4.2.7 with standard behavioral reduction (sbisim, diamond elimination) and breadth-first state exploration.

**Table A.9.3.** Coverage of the 49 Verification Assertions

Category	Count	Coverage
Core System Safety	4	Deadlock freedom; behavioral refinement (T, F, FD)
State-Level Reliability	38	Two specifications: $S_0$ (non-param) + $S_1$ – $S_6$ (3 units each)
Liveness Guarantees	2	Divergence checks for TLE_Process and TLE_Abstract_Process
Composition & Robustness	5	Concurrency checks (2), hostile-environment checks (2), determinism (1)
Total	49	Complete verification of safety, liveness, and concurrency

**Assertion Breakdown****Core System Safety (4):**

1. TLE\_Process :[deadlock free]
2. TLE\_Process [T= TLE\_Abstract\_Process]
3. TLE\_Process [F= TLE\_Abstract\_Process]
4. TLE\_Process [FD= TLE\_Abstract\_Process]

**State-Level Reliability (38):**

- Implementation states:  $S_0$  (1) +  $S_1$ – $S_6$  × ( $u_1, u_2, u_3$ ) (18) = 19
- Abstract states: Abstract\_ $S_0$  (1) + Abstract\_ $S_1$ – $S_6$  × ( $u_1, u_2, u_3$ ) (18) = 19

**Liveness Guarantees (2):**

1. TLE\_Process :[divergence free]
2. TLE\_Abstract\_Process :[divergence free]

**Composition & Robustness (5):**

1. TLE\_TwoUnits :[deadlock free] (parallel composition test)
2. TLE\_Abstract\_TwoUnits :[deadlock free] (abstract parallel test)
3. TLE\_Hostile\_System :[deadlock free] (hostile environment robustness)
4. TLE\_HostileEnv :[deadlock free] (hostile environment itself)
5. TLE\_Process :[deterministic [F]] (internal determinism)

**Reproducibility**

All 49 checks can be reproduced by loading the CSP model (tle\_model.csp) in FDR 4.2.7 and executing the assertions. The parameterized unit design ( $u_1, u_2, u_3$ ) enables tractable exploration of both sequential and concurrent scenarios, with all assertions passing consistently.

*Appendix A.10. Proofs of TLE Theorems*

**Notation:** See Table A.1.8 for formal definitions of symbols used in this section.

**Theorem A.10.1. (Storage Complexity).** *The TLE storage ratio compared to traditional foreign key representation is*

$$\frac{S_{TLE}}{S_{traditional}} = \frac{\hat{C}}{\hat{c} \cdot k}$$

where:

- $\hat{C}$  is the average bitmask size (in bits) across all parent entities,
  - $\hat{c}$  is the average number of children per parent,
  - $k$  is the storage size (in bits) required per stored relationship in the traditional representation.
- For sparse hierarchies where  $\hat{C} \ll \hat{c} \cdot k$ , TLE yields substantial storage reduction.*

**Proof.**

In the traditional foreign-key relational schema, each parent→child relationship requires storing a foreign key.

Let:

$$m = \sum_{j=1}^{P_{total}} |children(j)|$$

be the total number of parent→child relationships across the hierarchy.  
Each relationship requires  $k$  bits of storage, so:

$$S_{traditional} = m \cdot k$$

In TLE, each parent stores a bitmask of size  $C_j$  bits. Total TLE storage is the sum of all bitmask sizes:

$$S_{TLE} = \sum_{j=1}^{P_{total}} C_j$$

Define:

$$\hat{c} = \frac{m}{P_{total}} \text{ (average number of children per parent)}$$

$$\hat{C} = \frac{\sum_{j=1}^{P_{total}} C_j}{P_{total}} \text{ (average bitmask size)}$$

Then:

$$S_{TLE} = P_{total} \cdot \hat{C}$$

and the storage ratio becomes:

$$\frac{S_{TLE}}{S_{traditional}} = \frac{P_{total} \cdot \hat{C}}{m \cdot k} = \frac{\hat{C}}{\hat{c} \cdot k}$$

### Interpretation.

If the bitmask size is approximately equal to the average number of children:

$$\hat{C} \approx \hat{c}$$

Then

$$\frac{S_{TLE}}{S_{traditional}} \approx \frac{1}{k}$$

→ TLE yields a  $k$ -fold storage reduction.

For sparse hierarchies where bitmasks are much smaller:

$$\hat{C} \ll \hat{c} \cdot k$$

TLE achieves even greater savings (ratio  $< 1/k$ ).

In practice, TLE minimizes storage when children are sparse and bitmasks remain compact, as confirmed by empirical evaluation in Section 5.

■

**Theorem A.10.2. (Query Complexity).** For hierarchies where the number of children per parent  $n \leq w$  (machine word size, typically 64 bits), TLE enables constant-time  $O(1)$  lookups for child selection status. For  $n > w$  requiring multi-word bitmasks, lookup complexity is  $O(\lceil n/w \rceil)$ .

### Proof.

For  $n \leq w$ , the lookup operation for a specific child  $c$  under parent  $p$  and root (grandparent) entity  $g$  consists of:

1. **Root Access:**  $O(1)$  via direct or indexed lookup on  $g$ .
2. **Bitmask Retrieval:**  $O(1)$  access to the fixed-width integer column for  $p$ .
3. **Bitwise Check:**  $O(1)$  operation:  $(\text{bitmask} \gg c\_id) \& 1$ .

Each step is a constant-time operation. The total time complexity is therefore:

$$T_{query} = O(1) + O(1) + O(1) = O(1).$$

For  $n > w$ , the bitmask requires  $\lceil n/w \rceil$  words (or equivalent variable-width encoding). The bitwise check requires identifying the correct word segment and bit position, yielding  $O(\lceil n/w \rceil)$  complexity. In practice, for hierarchies with bounded branching factors ( $n \leq 64$ ), which is typical in enterprise systems, the operation is constant-time.

■

**Theorem A.10.3. (Update Complexity).** For hierarchies where the number of children per parent  $n \leq w$  (machine word size, typically 64 bits), TLE supports constant-time  $O(1)$  updates to child states. For  $n > w$  requiring multi-word bitmasks, update complexity is  $O(\lceil n/w \rceil)$ .

**Proof.**

For  $n \leq w$ , the update operation for a specific child  $c$  under parent  $p$  and root (grandparent) entity  $g$  consists of:

1. **Root Access:**  $O(1)$  via direct or indexed lookup on  $g$ .
2. **Bitmask Update:** A single, constant-time bitwise operation:

**Set:**  $\text{bitmask} |= (1 \ll c\_id)$

**Clear:**  $\text{bitmask} \&= \sim(1 \ll c\_id)$

**Toggle:**  $\text{bitmask} ^= (1 \ll c\_id)$

3. **Write-back:**  $O(1)$  operation to persist the updated fixed-width field.

Each step is a constant-time operation. The total time complexity is therefore:

$$T_{\text{update}} = O(1) + O(1) + O(1) = O(1).$$

For  $n > w$ , the bitmask update requires identifying and modifying the appropriate word segment, yielding  $O(\lceil n/w \rceil)$  complexity for both the bitwise operation and write-back.

In practice, for hierarchies with bounded branching factors ( $n \leq 64$ ), which is typical in enterprise systems, the operation is constant-time.

■

**Theorem A.10.4. (Batch Processing Complexity).** For hierarchies with bounded branching factor ( $n_{\max} \leq w$ ), processing all relationships in a TLE structure requires  $O(P_{\text{total}})$  time, where  $P_{\text{total}}$  is the total number of parent entities.

**Proof.**

An operation that must process every relationship (e.g., a full data export) must:

1. Iterate over each grandparent entity.
2. For each grandparent, iterate over each of its  $P_i$  parent entities.
3. For each parent entity, process its bitmask.

The bitmask processing cost depends on the number of children  $n$  relative to word size  $w$ :

- $O(1)$  for fixed-width integer fields when  $n \leq w$
- $O(\lceil n/w \rceil)$  for variable-width encodings when  $n > w$

Thus, each parent's bitmask can be processed in  $O(\lceil n_{\max}/w \rceil)$  time, where  $n_{\max}$  is the maximum children per parent across the hierarchy, the total time complexity is :

$$T_{\text{batch}} = \sum_{i=1}^{P_{\text{total}}} O\left(\left\lceil \frac{n_{\max}}{w} \right\rceil\right) = O(P_{\text{total}} * \left\lceil \frac{n_{\max}}{w} \right\rceil)$$

For bounded branching factors ( $n_{\max} \leq w$ , typical in enterprise hierarchies with 64-bit integers), this simplifies to:

$$T_{\text{batch}} = O(P_{\text{total}})$$

### Comparison to Alternative Approaches

Alternative hierarchy traversal methods incur higher computational cost (see Table A.10.1).

**Table A.10.1.** Complexity comparison of hierarchical traversal approaches

Approach	Complexity	Practical Characteristics
TLE( $n_{max} \leq w$ )	$O(P_{total})$	Linear scan, cache-friendly, predictable
B-tree indexed adjacency	$O(P_{total} * \log n)$	Logarithmic overhead per parent lookup
ContinentViewModel	$O(P_{total} * d)$	Depth-dependent; degrades for deep hierarchies

**B-tree indexed adjacency lists:** Each parent lookup requires  $O(\log n)$  time in an  $n$ -node hierarchy. Processing all  $P_{total}$  parents to locate their children requires  $O(P_{total} * \log n)$  for index traversals. For a single parent with  $k$  children, the total cost is  $O(\log n + k)$ :  $O(\log n)$  index search plus  $O(k)$  retrieval time.

**Recursive CTEs:** Evaluating hierarchy materialization requires iterative processing proportional to hierarchy depth  $d$ , yielding  $O(P_{total} * d)$ . While theoretical complexity bounds exist [131], practical performance degrades significantly for deep hierarchies where  $d \gg \log n$ , compared to TLE's flat  $O(P_{total})$  traversal.

### Conclusion

TLE traversal achieves asymptotic optimality for bounded hierarchies:  $O(P_{total})$  matches the theoretical lower bound  $\Omega(P_{total})$  for reading  $P_{total}$  entities. This efficiency, combined with cache-friendly sequential access patterns, enables scalable PBF pattern evaluation over TLE-encoded tables, supporting efficient pattern-driven development workflows.

■

### Discussion

Beyond the complexity advantages established in Theorems A.10.1–A.10.4, the Three-Level Encapsulation (TLE) model offers structural benefits not available in conventional hierarchical encodings. Unlike nested sets [132], which require  $O(n)$  relabeling when modifying tree structure, or standard adjacency lists [133], which depend on recursive traversal or materialized transitive closure to reconstruct hierarchy, TLE enables constant-time bitmask operations while preserving a fully normalized relational schema.

These theoretical bounds are further supported by empirical results (Section 5 and Appendix A.14), confirming that TLE's asymptotic advantages yield measurable performance improvements in PBF batch evaluation and pattern-driven development workflows.

## Appendix A.11. The PDFD MVP

### Appendix A.11.1. Overview of the PDFD MVP

**Purpose:** This section details a working implementation of the Primary Depth-First Development (PDFD) methodology within a real-world application: the "Logging Visited Places" use case (Section 3.3.1, item 10), developed mainly between 12/11/2024 and 12/25/2024 using Microsoft ASP.NET MVC. This MVP serves as a concrete instantiation of the formal PDFD framework, grounded on the PDFD formal model detailed in Section 3.4.1.

**Caveat:** For brevity, this PDFD demonstration is an MVP focusing on core traversal and pattern derivation. While reflecting PDFD's progression criteria (Section 3.4.1, item 5, Table 33), it omits exhaustive processing phases/features of the full methodology. Our formal guarantees (Appendix A.8) apply solely to this complete specification.

**Reproducibility & Research Context:** The repository includes generation/migration scripts, sample datasets, and deployment instructions [28]. These artifacts enable reproducible experiments and controlled comparisons against normalized or graph-based alternatives, supporting the formal empirical evaluation presented in Section 5.

### Appendix A.11.2. Objective

The primary objective of developing this Minimum Viable Product (MVP) was to validate the practical applicability of the PDFD methodology (as defined in Section 3.4.1) to real-world hierarchical workflows, as exemplified by the "Logging Visited Places" use case and its alignment with the business model in Figure 3.

### Appendix A.11.3. Strategy in Practice

The MVP operationalizes the PDFD model (defined in Section 3.4.1) with a real-world dataset. Rather than restating the methodology, we highlight the instantiation of PDFD's key components within this application. Each node corresponds to a business data element (e.g., continent, country, state, or county), with directed edges capturing hierarchical relationships. PDFD MVP directly uses raw business data to drive the development process, enabling traversal, refinement, and validation without intermediate pattern abstraction.

#### 1. Hybrid Depth-First Progression with Controlled Breadth

- **Vertical Execution (DFD-style):** Hierarchical levels (e.g., State → Country → Province) were traversed sequentially, focusing on in-depth development along a primary path.
- **Controlled Breadth (Breadth-First by Two, or BF-by-Two):** At each hierarchical level, two peer nodes (e.g., "Asia" and "North America") are processed in parallel to validate both their combinatorial selection states and the resulting feature-driven workflows. The BF-by-Two approach corresponds to a controlled parallel expansion strategy, conceptually aligned with branch-and-bound techniques used to manage combinatorial state spaces [72].

#### 2. Iterative Refinement via Feedback

- **CDD Cycles:** The cycles were triggered upon the detection of inconsistencies or schema limitations (e.g., missing intermediate tables or key definitions). This prompted a return to previous hierarchical levels for necessary corrections.

#### 3. Application Scalability and Portability

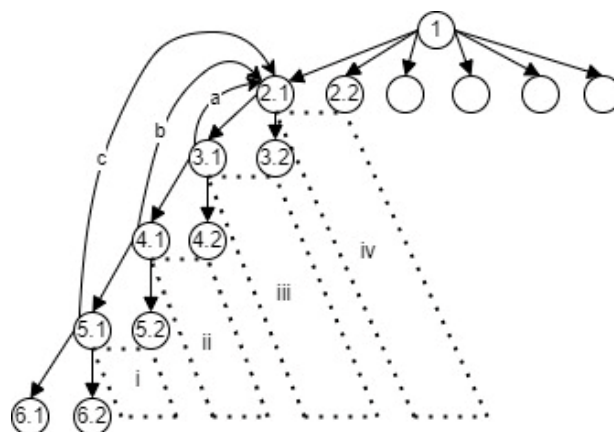
- The solution was designed to be stack-agnostic and modular. Though built in ASP.NET MVC, PDFD's structure maps naturally to other frameworks (e.g., React/Node.js), making the pattern portable and extensible.

### Appendix A.11.4. Workflow and Database Structure

This subsection details the application workflow implementing the PDFD methodology and the underlying relational database schema used in the MVP.

#### Application Workflow

The hierarchical traversal across levels—such as Continent → Country → Province—is illustrated in Figure A.11.1. This workflow exemplifies the BF-by-Two strategy, which selectively deepens the hierarchy by expanding only key nodes at each level. When inconsistencies are detected, the process initiates refinement through a feedback mechanism that incorporates dependency-directed backtracking [77].



**Figure A.11.1** PDFD MVP structural workflow implementing hybrid depth-first progression, BF-by-Two node selection, and feedback-based refinement in a multi-level geographic hierarchy

In the figure:

- Arrows represent dependencies between nodes.
- Dotted areas highlight subsets of the hierarchy that are deferred for population until after initial validation.
- Curved arrows indicate feedback loops that activate the CDD process for iterative refinement.
- Nodes are labeled according to their hierarchical position—e.g., 1 denotes the root node, 2.1 refers to the first node at Level 2, and so on—providing a structured view of the progressive traversal and refinement workflow.

### Relational Schema

The normalized relational schema underpinning the MVP, designed to represent the multi-level hierarchical relationships (e.g., Continent → Country → Province), is depicted in Figure A.11.2. This schema represents a simplified hierarchical relationship for the MVP. In some real-world scenarios, certain relationships might be more complex (e.g., many-to-many) and would require additional linking tables.

### Appendix A.11.5. State Machine Representation

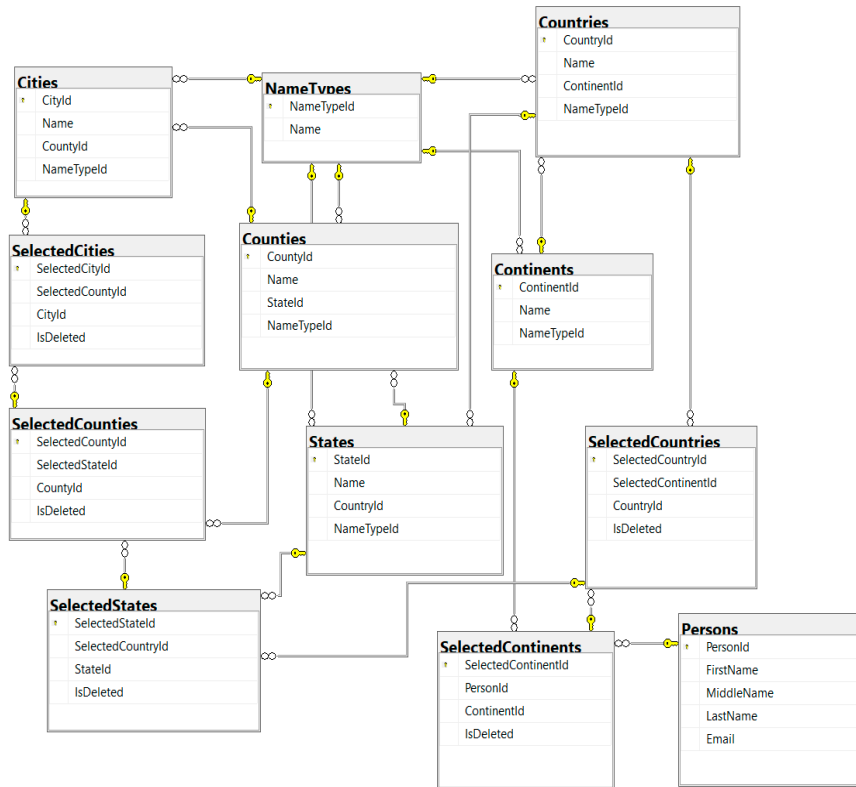
#### 1. Parameters

The behavior of the PDFD application workflow can be formally modeled using a state machine. This state machine is a specific instantiation of the generic mapping in Section 3.4.1. The following steps tailor the generic model for this specific application:

#### **Step 1: Configure Parameters for Fixed Levels**

The MVP fixes parameters from the general model to emulate real-world constraints:

- $L = 6$  (max level)
- $R_{\max} = 60$  (Predefined refinement iterative limit, allowing refinement up to 60 times per level in the MVP while ensuring termination guarantees.)
- For  $i=3,4,5$ ,  $J_i = \text{trace\_origin}(i) = 2$ , indicating that each level traces back to Level 2. This enforces refinement to Level 2 in the MVP, emphasizing critical dependency fixes.



**Figure A.11.2.** Normalized relational database schema used in the PDFD MVP to support progressive development and validation of multi-level geographic data (Continent → Country → State)

- For  $i=3,4,5$ ,  $R_i = \min(i - J_i + 1, i)$  ensures that dependent levels are revisited while respecting hierarchy boundaries. This mirrors the state-space exploration strategy in model checkers like SPIN, which also rely on efficient traversal and pruning to verify correctness [71]. However, PDFD introduces hierarchy-aware semantics absent from SPIN, enabling structured backtracking aligned with layered dependencies.

### Step 2: Customize State Logic to Emulate MVP

**Refinement Scope.** Modify the refinement phase to begin at Level 2 and span  $R_i$  levels:

$$S_3 = \text{refine}([2, 2 + R_i - 1]) \rightarrow S_1(i)$$

Here,  $\text{refine}([2, 2 + R_i - 1])$  denotes a bounded refinement over levels 2 through  $2 + R_i - 1$ , producing the updated state  $S_1(i)$  for node  $i$ .

## 2. States and Transitions

Tables A.11.1-A.11.2 present the states and transitions of the PDFD MVP model. The state machine formalization follows established patterns for workflow verification and conformance checking, as explored in the field of process mining [75]. The PDFD-specific refinement semantics extend concepts from formal refinement theory—particularly those applied to state-based systems and process algebras [76], demonstrating how iterative development can maintain formal correctness guarantees.

Generic mapping and rules in Tables A.11.1-A.11.2 are defined in Tables 33 and 34.

**Table A.11.1.** PDFD MVP application state descriptions and their mappings to generic PDFD state categories and parameter configurations

State ID	Phase	Description	Generic Mapping (State + Parameters)
S1	Process & Validate Level 1	Root node (Node 1)	$S_1(1) \rightarrow S_2(1)$
S2	Process & Validate Level 2	Nodes 2.1 and 2.2	$S_1(2) \rightarrow S_2(2)$

S3	Process & Validate Level 3	Nodes 3.1 and 3.2	$S_1(3) \rightarrow S_2(3)$
S4	Process & Validate Level 4	Nodes 4.1 and 4.2	$S_1(4) \rightarrow S_2(4)$
S5	Process & Validate Level 5	Nodes 5.1 and 5.2	$S_1(5) \rightarrow S_2(5)$
S6	Process & Validate Level 6	Nodes 6.1 and 6.2	$S_1(6) \rightarrow S_2(6)$
S2_R1	Refine Levels 2-3	Reprocess Levels 2-3 due to failure at Level 3	$S_1(j=2) \rightarrow S_2(j=2)$
S2_R2	Refine Levels 2-4	Reprocess Levels 2-4 due to failure at Level 4	$S_1(j=2) \rightarrow S_2(j=2)$
S2_R3	Refine Levels 2-5	Reprocess Levels 2-5 due to failure at Level 5	$S_1(j=2) \rightarrow S_2(j=2)$
S7	Finalize Level 5 Subtree	Finalize subtree under 5.1 and 5.2	$S_3(5)$
S8	Finalize Level 4 Subtree	Finalize subtree under 4.1 and 4.2	$S_3(4)$
S9	Finalize Level 3 Subtree	Finalize subtree under 3.1 and 3.2	$S_3(3)$
S10	Finalize Level 2 Subtree	Finalize subtree under 2.1 and 2.2	$S_3(2)$
S11	Finalize Root Subtree	Finalize root node and ensure completeness	$S_4(1)$
S_ERROR	Terminate on Failure	Refinement limit exceeded or validation failed	$S_5$

**Table A.11.2.** PDFD MVP state transition rules, triggers, and their corresponding formal definitions in the generic PDFD model

Rule ID	From State -> To State	Formal Condition / Trigger	Workflow Step	Generic Rule (PD# + Parameters )
PDFD1	[*] $\rightarrow$ S1	System initialized	Begin root-level processing	PD1
PDFD2	S1 $\rightarrow$ S2	Root validated	Advance to Level 2	PD2b (i=1)
PDFD3	S2 $\rightarrow$ S3	Level 2 validated	Advance to Level 3	PD2b (i=2)
PDFD4	S3 $\rightarrow$ S2_R1	Level 3 validation failed	Backtrack to refine Levels 2-3	PD2a (i=3, j=2)
PDFD5	S2_R1 $\rightarrow$ S3	Levels 2-3 refinement validated	Revalidate Level 3	PD3b (j=2 $\rightarrow$ i=3)
PDFD6	S3 $\rightarrow$ S4	Level 3 validated	Advance to Level 4	PD2b (i=3)
PDFD7	S4 $\rightarrow$ S2_R2	Level 4 validation failed	Backtrack to refine Levels 2-4	PD2a (i=4, j=2)
PDFD8	S2_R2 $\rightarrow$ S4	Levels 2-4 refinement validated	Revalidate Level 4	PD3b (j=2 $\rightarrow$ i=4)
PDFD9	S4 $\rightarrow$ S5	Level 4 validated	Advance to Level 5	PD2b (i=4)
PDFD10	S5 $\rightarrow$ S2_R3	Level 5 validation failed	Backtrack to refine Levels 2-5	PD2a (i=5, j=2)
PDFD11	S2_R3 $\rightarrow$ S5	Levels 2-5 refinement validated	Revalidate Level 5	PD3b (j=2 $\rightarrow$ i=5)
PDFD12	S5 $\rightarrow$ S6	Level 5 validated	Advance to Level 6	PD2b (i=5)
PDFD13	S6 $\rightarrow$ S7	Level 6 validated	Finalize Level 5 subtrees	PD4 (i=6)
PDFD14	S7 $\rightarrow$ S8	Subtree at Level 5 validated	Finalize Level 4 subtrees	PD4a
PDFD15	S8 $\rightarrow$ S9	Subtree at Level 4 validated	Finalize Level 3 subtrees	PD4a
PDFD16	S9 $\rightarrow$ S10	Subtree at Level 3 validated	Finalize Level 2 subtrees	PD4a
PDFD17	S10 $\rightarrow$ S11	Subtree at Level 2 validated	Finalize root node	PD5
PDFD18	S11 $\rightarrow$ [*]	Root finalized	Terminate	PD6 $\rightarrow$ PD7
PDFD19	S2_R1/S2_R2/S2_R3 $\rightarrow$ S_ERROR	Refinement validation failed AND refinement_attempts[2] $\geq$ 60	Terminate	PD3c $\rightarrow$ PD8
PDFD20	S3/S4/S5 $\rightarrow$ S_ERROR	refinement_attempts[2] $\geq$ 60	Terminate	PD8

For simplicity, the level-by-level top-down process in the generic model is compacted and replaced by S11's subtree top-down state, governed by the PDFD18 rules. While the formal state categories ( $S_1$ ,  $S_2$ ,  $S_3$ ,  $S_4$ , and  $S_5$ ) follow the definitions in Section 3.4.1, this particular state machine reflects the actual control flow of the MVP implementation and does not enumerate all possible

scenarios defined by the generic PDFD methodology. The table captures the practical subset of transitions that occurred during execution and validation of the MVP system.

In this MVP, bottom-up subtree finalization ( $S_3(i)$ ) culminates in a top-down global finalization pass ( $S_4(1)$ ), recognizing the root-driven pass as a streamlined final step.

The state machine diagram (see Figures A.11.3) visually depicts the flow, with transitions corresponding to the rules in Table A.11.2. Please refer to Appendix A.12 for the State Machine Mermaid code.

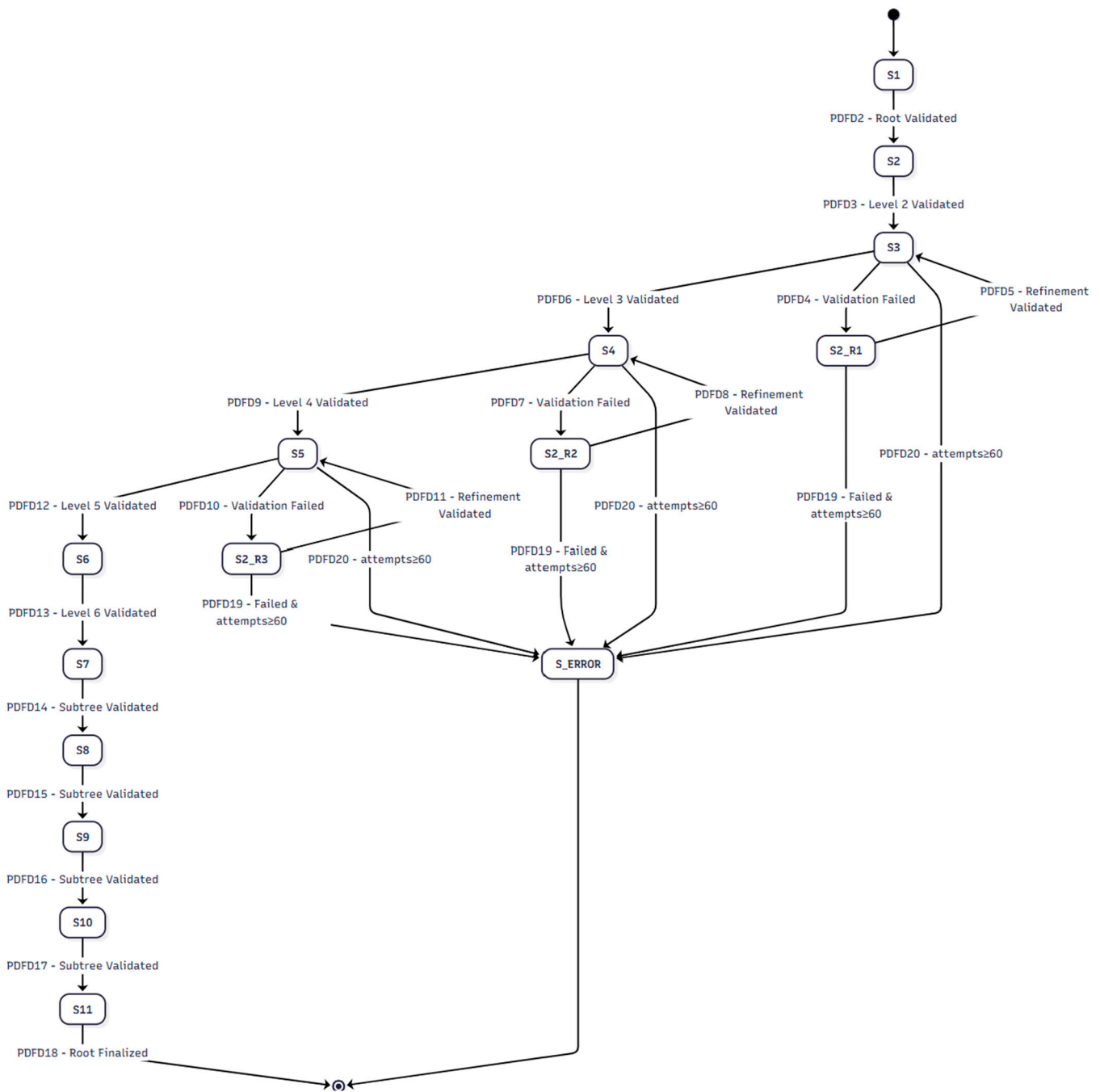
#### Appendix A.11.6. Development Process

For detailed step-by-step implementation traces of the MVP, including screenshots, transaction sequences, and database evolution, refer to Appendix A.13.

#### Appendix A.11.7. Key Technical Highlights

This MVP implementation illustrates the practical strengths of the Primary Depth-First Development (PDFD) methodology through several key technical highlights:

- **Controlled Depth Parallelism (BF-by-Two Adaptation):**
  - **Benefit:** By processing two sibling nodes in parallel at each hierarchical level during the depth-first traversal, the system can expose cross-branch inconsistencies and UI state conflicts early in development, rather than deferring them to integration.
  - **Contrast:** A pure DFD approach may postpone the detection of lateral interactions until deeper refinement phases, whereas a pure BFD approach—by prioritizing horizontal breadth—may introduce significant coordination overhead and delay cross-level dependency validation.
  - **Example:** Simultaneously testing the nodes “Asia” and “North America” at the continent level revealed UI inconsistencies in regional naming conventions (e.g., “state” in the US vs. “province” in China). Early resolution of these discrepancies prevented cascading structural conflicts at deeper country-specific levels of the hierarchy.
- **Iterative Schema Refinement**
  - **Benefit:** The integration of CDD allows for flexible schema evolution during the development process, accommodating necessary mid-development changes such as the introduction of surrogate keys.
  - **Contrast:** Traditional, more rigid development methodologies like Waterfall, with their upfront and inflexible schema design, often hinder the incorporation of necessary updates identified later in the cycle.
  - **Example:** Initially, composite keys (e.g., combining PersonId and ContinentId) were used. However, during backtracking at the continent level, these were refactored to simpler surrogate keys (e.g., SelectedContinentId), significantly simplifying downstream data relationships and query logic.



**Figure A.11.3.** State machine diagram for the PDFD MVP showing progression, refinement, and termination paths mapped to formal rule identifiers

- Hierarchical Backtracking
  - **Benefit:** Backtracking to previously validated hierarchical levels to incorporate new branches enhances the stability and reusability of the developed components by ensuring core paths are solid before extensive horizontal expansion.
  - **Contrast:** Monolithic development methods often require significant rework or even rollback when errors are discovered late in the process, especially after substantial horizontal expansion.
  - **Example:** After thoroughly validating the path USA → Maryland → Howard, PDFD facilitated backtracking to the state level to add branches for Virginia. This allowed for the reuse of existing controllers and views, minimizing redundant development effort.
- Methodological Cohesion

- The PDFD methodology effectively integrates DFD, BFD through the BF-by-Two strategy, and CDD.
- This MVP serves as a practical instantiation of the hybrid approach, demonstrating its ability to maintain the formal properties of the underlying methodologies (as discussed in Section 3.4.1) while offering a pragmatic and adaptable development process for hierarchical systems.

#### Appendix A.12. PDFD MVP State Machine Workflow Mermaid Code

##### Appendix A.12.1. Mermaid Code for Figure A.11.3

###### stateDiagram-v2

direction TB

[\*] --> S1

state S1: Process & Validate Level 1

S1 --> S2: PDFD2 - Root Validated

state S2: Process & Validate Level 2

S2 --> S3: PDFD3 - Level 2 Validated

state S3: Process & Validate Level 3

S3 --> S4: PDFD6 - Level 3 Validated

S3 --> S2\_R1: PDFD4 - Validation Failed

S3 --> S\_ERROR: PDFD20 - attempts≥60

state S2\_R1: Refine Levels 2-3

S2\_R1 --> S3: PDFD5 - Refinement Validated

S2\_R1 --> S\_ERROR: PDFD19 - Failed & attempts≥60

state S4: Process & Validate Level 4

S4 --> S5: PDFD9 - Level 4 Validated

S4 --> S2\_R2: PDFD7 - Validation Failed

S4 --> S\_ERROR: PDFD20 - attempts≥60

state S2\_R2: Refine Levels 2-4

S2\_R2 --> S4: PDFD8 - Refinement Validated

S2\_R2 --> S\_ERROR: PDFD19 - Failed & attempts≥60

state S5: Process & Validate Level 5

S5 --> S6: PDFD12 - Level 5 Validated

S5 --> S2\_R3: PDFD10 - Validation Failed

S5 --> S\_ERROR: PDFD20 - attempts≥60

state S2\_R3: Refine Levels 2-5

S2\_R3 --> S5: PDFD11 - Refinement Validated

S2\_R3 --> S\_ERROR: PDFD19 - Failed & attempts≥60

state S6: Process & Validate Level 6

S6 --> S7: PDFD13 - Level 6 Validated

state S7: Finalize Level 5

S7 --> S8: PDFD14 - Subtree Validated

```

state S8: Finalize Level 4
S8 --> S9: PDFD15 - Subtree Validated
state S9: Finalize Level 3
S9 --> S10: PDFD16 - Subtree Validated
state S10: Finalize Level 2
S10 --> S11: PDFD17 - Subtree Validated
state S11: Finalize Root
S11 --> [*]: PDFD18 - Root Finalized

```

```

state S_ERROR: Terminate on Failure
S_ERROR --> [*]

```

### Appendix A.13. PDFD MVP Development Process

This section details the step-by-step progression of the PDFD MVP's development process; the corresponding source code is provided in [28].

#### Appendix A.13.1. Root Node Level – Visitor

The root node (Node 1 in Figure A.13.1) represents visitor information, serving as the entry point for the application's hierarchical workflow.

Figure A.13.1. PDFD MVP Root Node (Visitor Entry) User Interface

#### Implementation Details

- **Model:** The Person class maps to the Persons database table (Table A.13.1), with PersonId as the primary key.
- **Controller:** The PersonsController processes HTTP requests, binds the Person model to the view, and handles form submissions.
- **View:** ASP.NET Razor syntax is used to render the visitor entry interface (Figure A.13.1).
- **Workflow:** Users input visitor details, which are persisted in SQL Server (Table A.13.1) upon submission. This process, representing Level 1 (S1 in Figure A.11.3), then redirects users to the Continent Level (Level 2) via PDFD2 (Table A.11.2).

Table A.13.1. Sample Data for Person (Root Level) in PDFD MVP Hierarchy

PersonId	First Name	Middle Name	Last Name	Email
1	Test	T	Tester	tester@test.com

#### Appendix A.13.2. Continent Level – Asia and North America

This level handles continent selection and integrates with downstream geographical hierarchies.

##### 1. Implementation Overview

Table A.13.2 outlines the key components, including models, database tables, and core data fields.

**Table A.13.2.** Model, Database Table, and Data Field Summary for PDFD MVP Continent Level

Model	SQL Table	Function	Key Data Fields
Continent	Continents	Reference Data	ContinentId, Name, NameTypeId
SelectedContinent	SelectedContinents	Selection Tracking	SelectedContinentId, PersonId, ContinentId, IsDeleted
ContinentViewModel	N/A	View Model	ContinentId, ContinentName, PersonId, IsSelected

## 2. Source Tables

The PDFD MVP uses the following tables as source data, with some shared across all hierarchy levels:

- Persons (Table A.13.1) – Shared across all levels
- Continents (Table A.13.3)
- NameTypes (Table A.13.4) – Shared across all levels
- SelectedContinents (Table A.13.5)

**Table A.13.3.** Reference Data for Continents in PDFD MVP

ContinentId	Name	NameTypeId
1	Asia	1
2	North America	1

**Table A.13.4.** Reference Data for NameTypes (Hierarchy Levels) in PDFD MVP

NameTypeId	Name
1	Continent
2	Country
3	State
4	County
5	City
6	District
7	Province
11	Region

**Table A.13.5.** Sample Transaction Data for SelectedContinents in PDFD MVP

SelectedContinentId	PersonId	ContinentId	IsDeleted
1	1	1	1
2	1	2	0

## 3. Workflow Logic

**User Interaction**

- Users interact with the continent selection interface (Figure A.13.2), which triggers updates to the SelectedContinents table (Table A.13.5). Upon submission, the system updates Table A.13.5. according to the following rules—also applicable at subsequent hierarchy levels:
  - New selections are added with IsDeleted = 0.
  - Deselections are marked with IsDeleted = 1 (soft delete).
  - Restored selections have IsDeleted reset to 0.
- User selections at the continent level trigger cascaded updates to downstream levels (e.g., countries).

### Select Continents

#	Continent Name	Name Type	Select
1	Asia	Continent	<input type="checkbox"/>
2	North America	Continent	<input checked="" type="checkbox"/>

[Submit](#)

Figure A.13.2. PDFD MVP Continent Selection User Interface

#### State Machine (Figure A.11.3)

- Level 2 (S2) processed.
- Transitions to Level 3 (S3) follow PDFD3 ( $\sum P(n) \geq K_2$ ).

#### Structural Workflow (Figure A.11.1)

- Level 2 with  $K_2 = 2$ :
  - **Node 2.1:** North America (ContinentId = 2)
  - **Node 2.2:** Asia (ContinentId = 1)
- 4. Hierarchical Context

#### Refinement Logic (Figure A.11.3)

- Errors detected at Level 3 (S3) trigger refinement starting at  $J_i=2$  (PDFD4).

#### Appendix A.13.3. Country Level – United States and Canada

This level manages country selection within the continent hierarchy.

##### 1. Implementation Overview

#### CDD Intervention (Figure A.11.3)

- Missing IsSelected field triggered refinement (PDFD4) for Levels 2–3.
- Post-refinement, processing resumed at Level 3 (PDFD5).

#### Models

- Country, SelectedCountry, CountryViewModel (see Table A.13.6)

#### Tables

- Countries Lookup (Table A.13.7), SelectedCountries Transaction Data (Table A.13.8)

Table A.13.6. summarizes the models, corresponding tables, functions, and their roles at the country level.

Table A.13.6. Model, Database Table, and Data Field Summary for PDFD MVP Country Level

Model	SQL Table	Function	Key Data Fields
Country	Countries	Reference Data	CountryId, Name, ContinentId, NameTypeId
SelectedCountry	SelectedCountries	Selection Tracking	SelectedCountryId, SelectedContinentId, CountryId, IsDeleted
CountryViewMode 1	N/A	View Model	CountryId, CountryName, SelectedContinentId, IsSelected

Table A.13.7. Reference Data for Countries in PDFD MVP

CountryId	Name	ContinentId	NameTypeId
1	USA	2	2
2	Canada	2	2

Table A.13.8. Sample Transaction Data for SelectedCountries in PDFD MVP

SelectedCountryId	SelectedContinentId	CountryId	IsDeleted
1	2	1	0
2	2	2	1

## 2. Workflow Logic

**User Interaction**

The CountryController uses the CountryViewModel to populate the interface (Figure A.13.3), where users toggle country selections (e.g., USA, Canada). Changes are persisted to the SelectedCountries table (Table A.13.8) using soft deletion (IsDeleted flag).

## Select Countries

North America

#	Country Name	Name Type	Select
1	USA	Country	<input checked="" type="checkbox"/>
2	Canada	Country	<input type="checkbox"/>

**Figure A.13.3.** PDFD MVP Country Selection User Interface

**Pre-Checked Entries**

Previously selected countries (e.g., USA in Table A.13.8) are pre-checked in the interface, reflecting historical data stored in SelectedCountries.

- State Machine (Figure A.11.3)
  - S3 processing step failed
  - Transitions to S2\_R1
- Structural Workflow (Figure A.11.1)
  - Level 3 with  $K_3 = 2$  (indicating two nodes processed at this level):
    - **Node 3.1:** USA (CountryId = 1)
    - **Node 3.2:** Canada (CountryId = 2)

## Appendix A.13.4. State Level – Maryland and Virginia

This level handles state/province selection within countries, adhering to the hierarchical structure defined in PDFD. It is state S4 in Figure A.11.3. Here, a surrogate key was found to be a better choice for database design, prompting the use of the CDD strategy to refine levels 2-4. Refer to 'Transition from Composite to Surrogate Keys' in item 1 of section A.13.7, curve b in Figure A.11.1, and state S2\_R2 in Figure A.11.3 for more details.

## 1. Implementation Overview

**CDD Intervention (Figure A.11.3)**

- Surrogate key introduction triggered refinement (PDFD7) for Levels 2-4.
- Processing resumed at Level 4 (PDFD8).

**Models**

- State, SelectedState, StateViewModel. (Table A.13.9)

**Tables**

- States Lookup (Table A.13.10), SelectedStates (Table A.13.11)

Table A.13.9. summarizes the models, corresponding tables, functions, and their roles at the state level.

**Table A.13.9.** Model, Database Table, and Data Field Summary for PDFD MVP State Level

Model	SQL Table	Functions	Key Data Fields
State	States	Reference Data	StateId, Name, CountryId, NameTypeId
SelectedState	SelectedStates	Selection Tracking	SelectedStateId, SelectedCountryId, StateId, IsDeleted
StateViewModel	N/A	View Model	StateId, StateName, SelectedCountryId, IsSelected

**Table A.13.10.** Reference Data for States in PDFD MVP

StateId	Name	CountryId	NameTypeId
1	Maryland	1	3
2	Virginia	1	3

**Table A.13.11.** Sample Transaction Data for SelectedStates in PDFD MVP

SelectedStateId	SelectedCountryId	StateId	IsDeleted
1	1	1	0
2	1	2	1

## 2. Workflow Logic

### User Interaction

- The StateController uses the StateViewModel to populate the interface (Figure A.13.4), where users toggle state selections (e.g., Maryland, Virginia). Changes are saved to the SelectedStates table (Table A.13.11) using soft deletion (IsDeleted flag).

**Select States**

#	State Name	Select
USA		
1	Maryland - State	<input checked="" type="checkbox"/>
2	Virginia - State	<input type="checkbox"/>

**Figure A.13.4.** PDFD MVP State Selection User Interface

- Users modify state selections, with pre-checked entries reflecting prior choices stored in SelectedStates.

### State Machine (Figure A.11.3)

- Level 4 processing
- Transitions to S2\_R2 (PDFD7)

### Structural Workflow (Figure A.11.1)

Level 4 with  $K_4 = 2$  (indicating two nodes processed at this level):

- Node 4.1:** Maryland (StateId = 1)
- Node 4.2:** Virginia (StateId = 2)

## Appendix A.13.5. County Level – Howard and Baltimore

This level manages county/district selection within states, corresponding to S5 in Figure A.11.3's 'Processing & Refinement' state. A missing IsDeleted field at this stage triggered the CDD methodology to refine levels 2-5. For details, refer to 'Introduction of the IsDeleted Flag' in A.11.7.1, curve c in Figure A.11.1, and S2\_R3 in Figure A.11.3.

### 1. Implementation Overview

#### CDD Intervention (Figure A.11.3)

- Missing IsDeleted flag triggered refinement (PDFD10) for Levels 2–5.
- Processing resumed at Level 5 (PDFD11).

#### Models

- County, SelectedCounty, CountyViewModel (Table A.13.12)

#### Tables

- Counties Lookup (Table A.13.13), SelectedCounties Transaction Data (Table A.13.14)

**Table A.13.12.** Model, Database Table, and Data Field Summary for PDFD MVP County Level

Model	SQL Table	Function	Key Data Fields
-------	-----------	----------	-----------------

County	Counties	Reference Data	CountyId, Name, StateId, NameTypeId
SelectedCounty	SelectedCounties	Selection Tracking	SelectedCountyId, SelectedStateId, CountyId, IsDeleted
CountyViewModel	N/A	View Model	CountyId, CountyName, SelectedStateId, IsSelected

Table A.13.13. Reference Data for Counties in PDFD MVP

CountyId	Name	StateId	NameTypeId
1	Howard	1	4
2	Baltimore	1	4

Table A.13.14. Sample Transaction Data for SelectedCounties in PDFD MVP

SelectedCountyId	SelectedStateId	CountyId	IsDeleted
1	1	1	0

## 2. Workflow Logic

**User Interaction**

- Users toggle county selections (e.g., Howard, Baltimore) within Maryland via the interface (Figure A.13.5), with updates persisted to SelectedCounties (Table A.13.14).

## Select Counties

#	County Name	Select
Maryland		
1	Howard - County	<input checked="" type="checkbox"/>
2	Baltimore - County	<input type="checkbox"/>

Figure A.13.5. PDFD MVP County Selection User Interface

**State Machine (Figure A.11.3)**

- Level 5 processing
- Transitions to S2\_R3 (PDFD10)

**Structural Workflow (Figure A.11.1)**

Level 5 with  $K_5 = 2$  (indicating two nodes processed at this level):

- **Node 5.1:** Howard County (CountyId = 1)
- **Node 5.2:** Baltimore County (CountyId = 2)

## Appendix A.13.6. City Level – Ellicott City and Columbia

This level handles city selection within counties.

## 1. Implementation Overview

**Models**

- City, SelectedCity, CityViewModel (Table A.13.15)

**Tables**

- Cities Lookup (Table A.13.16), SelectedCities Transaction Data (Table A.13.17)

Table A.13.15. Model, Database Table, and Data Field Summary for PDFD MVP City Level

Model	SQL Table	Function	Key Data Fields
City	Cities	Reference Data	CityId, Name, CountyId, NameTypeId
SelectedCity	SelectedCities	Selection Tracking	SelectedCityId, SelectedCountyId, CityId, IsDeleted
CityViewModel	N/A	View Model	CityId, CityName, SelectedCountyId, IsSelected

Table A.13.16. Reference Data for Cities in PDFD MVP

CityId	Name	CountyId	NameTypeId
--------	------	----------	------------

1	Ellicott City	1	5
2	Columbia	1	5

Table A.13.17. Sample Transaction Data for SelectedCities in PDFD MVP

SelectedCityId	SelectedCountyId	CityId	IsDeleted
1	1	1	0
2	1	2	0

## 2. Workflow Logic

**User Interaction**

- Users finalize city selections (e.g., Ellicott City, Columbia) within Howard County via the interface (Figure A.13.6), with data stored in SelectedCities (Table A.13.17).

**Select Cities**

#	City Name	Name Type	Select
<b>Howard</b>			
1	Ellicott City	City	<input checked="" type="checkbox"/>
2	Columbia	City	<input checked="" type="checkbox"/>

Figure A.13.6. PDFD MVP City Selection User Interface

**State Machine (Figure A.11.3)**

- Level 6 processing.
- Transition to completion phase follows PDFD13.

**Structural Workflow (Figure A.11.1)**

Level 6 with  $K_6 = 2$  (indicating two nodes processed at this level):

- **Node 6.1:** Ellicott City (CityId = 1).
- **Node 6.2:** Columbia (CityId = 2).

## Appendix A.13.7. Intermediate Development with CDD

CDD played a crucial role in refining the PDFD application's architecture, addressing evolving requirements, and resolving unanticipated gaps during implementation. While the final workflow comprises six hierarchical levels (Figure A.11.1), iterative cycles were essential in ensuring structural integrity and scalability throughout the development process.

**Key Iterations and CDD Interventions**

1. Addition of the IsSelected Field
  - **Challenge:** The IsSelected flag—essential for tracking user selections—was omitted during initial continent-level development and identified only at the country level.
  - **CDD Intervention:** A feedback loop (curve a in Figure A.11.1) redirected development back to the continent level to add the IsSelected field, ensuring consistent state management and user selection tracking across all levels.
2. Transition from Composite to Surrogate Keys
  - **Initial Design:** Composite keys (e.g., PersonId + ContinentId for SelectedContinents) were initially used to enforce uniqueness across tables.
  - **Challenge:** As development progressed to deeper levels of the hierarchy (e.g., states, counties), composite keys became cumbersome, complicating foreign key relationships and reducing scalability.
  - **CDD Intervention:** A surrogate key (SelectedContinentId) was introduced at the continent level (curve b in Figure A.11.1), simplifying downstream dependencies and improving scalability.
3. Introduction of the IsDeleted Flag
  - **Challenge:** Soft-deletion functionality, essential for marking deselected entries without losing data, was overlooked initially, risking permanent data loss when users deselected entries.

- **CDD Intervention:** The IsDeleted field was retrofitted into transaction tables (e.g., SelectedContinents) via a feedback loop (represented by curve c in Figure A.11.1), allowing for dynamic updates to selections without data loss.

Table A.13.18. summarizes the key information of these interventions. Refers to Table A.11.1. and Table A.11.2. for the rule id and state transition.

**Table A.13.18.** Summary of CDD Interventions and Their Mapping to PDFD MVP State Transitions

Intervention	Scope Levels	i	R <sub>i</sub>	Depth	Rule ID	State Transition	Figure Reference
Addition of IsSelected	2-3	3	2	2	PDFD4 → PDFD5	S3 → S2_R1 → S3	Curve a (Figure A.11.1)
Transition to Surrogate Keys	2-4	4	3	3	PDFD7 → PDFD8	S4 → S2_R2 → S4	Curve b (Figure A.11.1)
Introduction of IsDeleted	2-5	5	4	4	PDFD10 → PDFD11	S5 → S2_R3 → S5	Curve c (Figure A.11.1)

**Note:** Depth = R<sub>i</sub> = i - j + 1 (j=2 for all refinements).

### Outcomes of CDD Iterations

- **Data Integrity:** Retroactive fixes ensured consistent tracking of user selections and deletions across all levels, preventing data inconsistencies.
- **Scalability:** The introduction of surrogate keys reduced relational complexity, supporting seamless expansion to accommodate deeper hierarchical levels as the system grew.
- **Workflow Cohesion:** Iterative refinements aligned the system with real-world user behavior (e.g., revisiting selections), resulting in a more intuitive user experience.

### Key Takeaways

CDD's cyclical workflow enabled the team to incrementally address gaps, refine dependencies, and adapt to emerging requirements. This iterative approach highlights the methodology's strength in balancing structured development with Agile flexibility, ensuring robust outcomes in complex hierarchical systems.

Formal validation prioritizes CDD because its refinement cycles introduce NP-hard cyclomatic dependencies - the methodology's highest-risk domain requiring termination proofs (R<sub>max</sub>=60). Sequentially processed components are verifiable through conventional techniques, inheriting correctness from CDD's state conformance guarantees.

#### Termination Assurance

- **Per-level refinement limit:** refinement\_attempts[j] ≤ R<sub>max</sub> = 60 (Section A.11.5)
- S\_ERROR enforcement:
  - **PDFD19:** Refinement failure after 60 attempts
  - **PDFD20:** Forward-pass failure after 60 attempts

#### State Machine Conformance

- Development phases map 1:1 to PDFD states (Table A.11.1)
- CDD interventions trigger exact refinement rules (Table A.13.18)

#### Parameter Invariance

- J<sub>i</sub>=2 maintained for all refinements (root-cause level)
- Refinement Scope Consistency:
  - R<sub>i</sub>=2: Levels 2-3 (S2\_R1)
  - R<sub>i</sub>=3: Levels 2-4 (S2\_R2)
  - R<sub>i</sub>=4: Levels 2-5 (S2\_R3)

#### Formal Bounds

- Tree Parameters:
  - Depth: L=6 (Levels 1-6)
  - State Complexity: |Q|=15 states
- Refinement Attempts:

- Level 2: 3 attempts  $\ll R_{\max}=60$
- Level 3: 3 attempts  $\ll 60$
- Level 4: 2 attempts  $\ll 60$
- Level 5: 1 attempts  $\ll 60$
- Transition Complexity:
  - $|\delta|=20$  rules (Table A.11.2)
  - Max depth:  $O(L)=6$

### Appendix A.13.8. The Report Page

The Report Page consolidates and displays hierarchical selections made across all levels (Figure A.11.1), offering a comprehensive view of visited locations.

#### 1. Implementation Overview

Table A.13.19. outlines the components and data flow for generating the report.

**Table A.13.19.** Components and Data Flow for Generating the PDFD MVP Report Page

Type	Name	Role	Key Data Fields
Database View	vw_Report	Data Aggregation	Persons, SelectedContinents, Continents, SelectedCountries, Countries, SelectedStates, States, SelectedCounties, Counties, SelectedCities, Cities, NameTypes
Model	Report	UI Presentation	PersonName, ContinentName, CountryName, StateName, CountyName, CityName

#### 2. Workflow Logic

##### Data Aggregation

The SQL View vw\_Report aggregates data by joining transactional tables (e.g., SelectedContinents, SelectedCountries) with reference tables (e.g., Continents, Countries). It uses the NameTypes table to standardize naming conventions (e.g., "State" vs. "Province").

##### View Model Mapping

The Report ViewModel extracts user-friendly fields (e.g., PersonName, ContinentName) from vw\_Report to render the data for the UI.

Figure A.13.7 presents a visitor's selections in a hierarchical format (e.g., Test Tester → North America → USA → Maryland → Howard → Ellicott City).

### Appendix A.13.9. Backtracking to complete the entire application

This section is not part of the source code referenced in [28], as the PDFD MVP does not fully implement the complete PDFD specification. It is included here to provide a comprehensive explanation of the full specification.

The backtracking process is composed of bottom-up and top-down parts.

#### Report

Person Name	Continent	Country	State	County	City
Test T Tester	North America - Continent	USA - Country	Maryland - State	Howard - County	Ellicott City - City
Test T Tester	North America - Continent	USA - Country	Maryland - State	Howard - County	Columbia - City

**Figure A.13.7.** PDFD MVP Report Page Displaying Hierarchical Visitor Selections

##### Bottom-Up Completion with Local Top-Down Verification

States S7-S10 implement bottom-up completion with integrated local top-down verification:

- Bottom-Up Processing:
  - Finalizes subtrees level-by-level from leaves toward root
  - Handles localized subtree completion
- Local Top-Down Verification:

- Validates parent-child relationships within the current subtree
- Ensures hierarchical integrity from subtree root to leaves
- Example: S7 verifies Maryland→Howard County→Ellicott City

#### Global Top-Down Finalization (S11 Only)

- State S11 performs global top-down finalization:
  - Verifies completeness from root perspective (Person→Continent→Country→...)
  - Ensures cross-subtree consistency
  - Executes final validation pass before termination (PDFD18)

Following the core implementation detailed in Sections A.13.1–A.13.8, PDFD employs iterative backtracking in this section to systematically expand data coverage and validate business scenarios. This approach ensures manageable system updates by progressively populating hierarchical subsets (indicated by dotted areas in Figure A.11.1) and refining the code as needed. This process commences after PDFD13 (transition to State S7, see Figure A.11.3).

- **Phase 1: County-Level Completion** (Subset i in Figure A.11.1 and state S7 in Figure A.11.3)
  - **Objective:** Expand Howard County by adding remaining cities (e.g., Columbia) and populate all cities in Baltimore County
  - **Actions:** Update the Cities table with missing entries (Table A.13.16)
  - **State Machine:** Maps to S7 → S8 (PDFD14) (Table A.11.2)
- **Phase 2: State-Level Expansion** (Subset ii in Figure A.11.1 and state S8 in Figure A.11.3)
  - **Objective:** Implement remaining counties/cities in Maryland and Virginia
  - **Actions:** Populate Counties and Cities tables for Virginia (e.g., Fairfax County, Arlington)
  - **State Machine:** Maps to S8 → S9 (PDFD15) (Table A.11.2)
- **Phase 3: National Scalability** (Subset iii in Figure A.11.1 and state S9 in Figure A.11.3)
  - **Objective:** Scale to all U.S. states and Canadian provinces
  - **Actions:** Populate States, Counties, and Cities tables for the U.S. (e.g., Texas, California) and Canada (e.g., Ontario, Quebec)
  - **State Machine:** Maps to S9 → S10 (PDFD16) (Table A.11.2)
- **Phase 4: Continental Integration** (Subset iv in Figure A.11.1 and state S10 in Figure A.11.3)
  - **Objective:** Integrate North American and Asian datasets
  - **Actions:** Populate Asian countries (e.g., China, Japan) with region-specific hierarchies (e.g., provinces, prefectures)
  - **State Machine:** Maps to S10 → S11 (PDFD17, Transitions to global top-down finalization)
- **Phase 5: Global Coverage (Unpopulated Nodes)** (Figure A.11.1 and S11 in Figure A.11.3)
  - **Objective:** Achieve global completeness by adding remaining continents (e.g., Europe, Africa)
  - **Actions:** Populate Countries, States, Counties, and Cities for all regions
  - **State Machine:** Executes during S11 (global top-down finalization) and terminates via PDFD18

#### Appendix A.14. PBFDP MVP WITH PATTERN-BASED TRAVERSAL AND TLE

##### Appendix A.14.1. Overview of the PBFDP MVP

**Purpose:** This section presents a Minimum Viable Product (MVP) of Primary Breadth-First Development (PBFDP) developed mainly between 12/26/2024 and 01/15/2025. The MVP demonstrates pattern-driven, level-wise traversal combined with Three-Level Encapsulation (TLE) and bitmask encoding for relational optimization. The implementation follows the PBFDP formal model (Section 3.4.2) and the bitmask-based TLE optimizations outlined in Section 4. [53,55]

**Caveat:** For brevity the MVP applies a pragmatic progression rule (advancing after processing a subset of Pattern<sub>n</sub> nodes). Consequently, the full formal guarantees in Appendix A.8 apply to the complete PBFDP methodology (Section 3.4.2, Table 40), not the simplified MVP.

**Reproducibility & Research Context:** The repository includes generation/migration scripts, sample datasets, and deployment instructions [29]. These artifacts enable reproducible experiments and controlled comparisons against normalized or graph-based alternatives, supporting empirical evaluation in Section 5.

#### Appendix A.14.2. Technology Stack and Key Design Decisions

Built from the "Logging Visited Places" use case (Section 3.3.1, item 10), the Pbfd MVP is implemented using Microsoft ASP.NET MVC with SQL Server for backend persistence. Each node is a business-level data item (consistent with the PDFD MVP), but nodes above the final two hierarchical levels (county and city) also serve as Level 1 anchors of TLE instances (see A.14.7).

For example, the raw data "United States" functions both as a business entity and as the grandparent element of a TLE structure that encodes:

- **Level 1:** the country ("United States"), implemented in the MVP as the table name representing the grandparent pattern
- **Level 2:** its constituent states (e.g., Maryland, California), represented as columns within the Level 1 table
- **Level 3:** the counties within each state, encoded as bitmask values stored in the corresponding Level 2 column cells

This dual role enables each upper-level node to embed a fixed three-level hierarchical pattern (Level 1 → Level 2 → Level 3) while remaining a normal record in the application domain. TLE's bitmask-based encoding preserves hierarchical semantics across levels and ensures predictable, constant-time operations for lookup, traversal, and update.

Key design decisions reflect established trade-offs between encoded, columnar-style access patterns and conventional relational semantics:

- **Breadth-First Core:** Level-wise grouping of TLE-anchored nodes reduces multi-join traversal and improves cache locality, inspired by column-store and encoding principles [53,55].
- **Selective Depth Exploration:** After resolving a Level 1 or Level 2 pattern, the MVP performs controlled descent into the corresponding TLE instance to validate cross-level constraints while maintaining early UI feedback.
- **Iterative Refinements (CDD):** Bounded refinement cycles allow schema or pattern adjustments when validations fail. This preserves termination guarantees while supporting correction and incremental evolution of the hierarchy.

#### Appendix A.14.3. Strategy in Practice

Pbfd MVP combines horizontal pattern-based development with depth-first extensions and iterative refinement. The approach maintains flexibility without compromising structure.

##### **Breadth-First Core: Level-Wise Consolidation**

- **Pattern Grouping:** nodes at the same level are processed together using shared templates and validation logic to maximize reuse and reduce development overhead. This reduces repeated join logic and mirrors encoded/columnar techniques for group-oriented queries [53,55,118].
- **Example:** continents such as "North America" and "Asia" are presented as checkboxes in a shared view, enabling batch-processing logic.
- **Efficiency:** server-side Razor views with shared models reduce UI duplication.

##### **Selective Depth-First Exploration**

- **Depth After Pattern:** after a pattern (e.g., continent selection) is validated, the system descends into the children of selected parents only (e.g., countries inside selected continents), enabling earlier detection of cross-level invariants [62].

##### **Iterative Refinement via CDD**

- **Feedback Loops:** mid-development changes (shared components, schema adjustments) were integrated via bounded CDD cycles; failures at deeper levels trigger controlled backtracking and

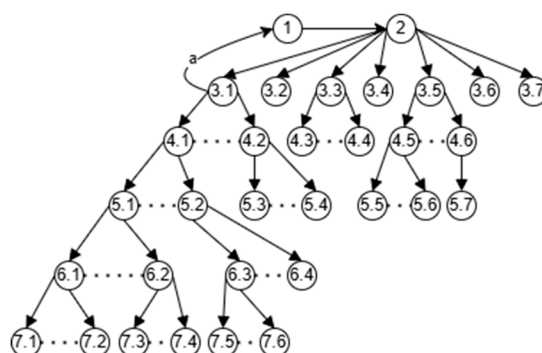
refinement of parent-level patterns. This mirrors dependency-directed backtracking techniques used in knowledge refinement and constraint search [77].

#### MVP Parameters (following Table 37)

- $R_{\max} = 50$  (empirical maximum refinement attempts per level before bounded failure)
- $J_i = \text{trace\_origin}(i)$  (refinement origin tracing)
- $R_i = i - J_i + 1$  (refinement span)

#### Appendix A.14.4. Structural Workflow

Figure A.14.1 illustrates the PBFDP MVP hybrid flow: breadth-first pattern consolidation, selective depth validation, and iterative refinement backtracks (CDD). The figure annotations emphasize TLE units and where bitmask operations provide single-row, constant-time checks for child selection. [53,55,118].



**Figure A.14.1.** Structural workflow of PBFDP MVP illustrating breadth-first progression, selective depth-first traversal, and iterative refinements

The visual conventions used in Figure A.14.1 are defined as follows:

#### Node Conventions

- **Root Node:** Level 1 (ContinentGrandparent)
- **Numbering:** First digit = level, second digit = position (e.g., Node 3.1 = North America)

#### Annotations

- **Arrows:** Progression through hierarchical levels
- **Dotted Lines:** Unselected nodes
- **Curve a:** CDD-driven refinements (Levels 1–3) triggered by Level 3 failures

#### Appendix A.14.5. State Machine Representation

The PBFDP MVP is captured by a specialized state machine (see Tables A.14.1 & A.14.2). Several PBFDP states integrate level processing plus TLE-based resolution for subsequent levels (e.g., `Level_3_Processing_Validating_Resolving` handles levels 3–5 as a single TLE scope). This coalescing reduces protocol overhead and mirrors the encapsulated access patterns characteristic of columnar and encoded storage architectures [53,55].

**Key note:** While the MVP's state transitions preserve the generic PBFDP semantics—progression, refinement, and finalization—they are implemented in a simplified and consolidated form. The MVP employs coarser TLE-scoped states to optimize data transfer volume and improve query efficiency.

Generic mapping and rules in Tables A.14.1-A.14.2 are defined in Tables 39 and 40.

**Table A.14.1.** PBFDP MVP-specific state definitions with corresponding TLE scopes (functioning as dynamic traversal windows) and generic rule mappings

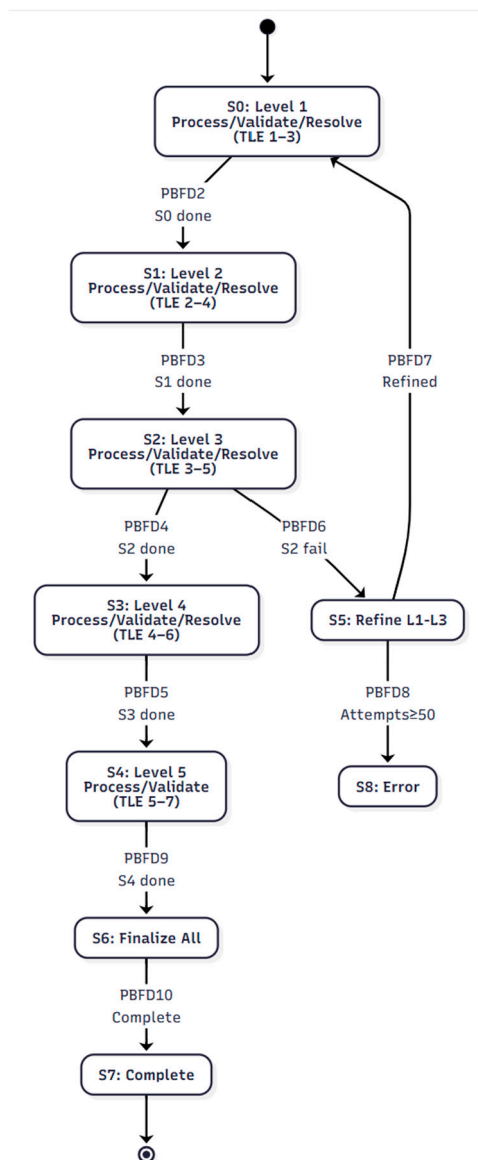
State Id	Label	Phase	Generic Mapping	TLE Scope
----------	-------	-------	-----------------	-----------

S0	Level_1_Processing_Validating_Resolving	Process & Validate Level 1 & resolve Level 2 (TLE Root: ContinentGrandparent)	$S_1(1) \rightarrow S_2(1) \rightarrow S_3(1)$	Levels 1–3
S1	Level_2_Processing_Validating_Resolving	Process & Validate Level 2 & resolve Level 3 (TLE Root: ContinentParent)	$S_1(2) \rightarrow S_2(2) \rightarrow S_3(2)$	Levels 2–4
S2	Level_3_Processing_Validating_Resolving	Process & Validate Level 3 & resolve Level 4 (TLE Root: a continent)	$S_1(3) \rightarrow S_2(3) \rightarrow S_3(3)$	Levels 3–5
S3	Level_4_Processing_Validating_Resolving	Process & Validate Level 4 & resolve Level 5 (TLE Root: a country)	$S_1(4) \rightarrow S_2(4) \rightarrow S_3(4)$	Levels 4–6
S4	Level_5_Processing_Validating	Process & Validate Level 5 (TLE Root: a state)	$S_1(5) \rightarrow S_2(5)$	Levels 5–7
S5	Refine_Level1-3	Refine Levels 1–3 (Level 3 failure)	$S_1(j) \rightarrow S_2(j) \rightarrow S_3(j)$ ( $j=1$ )	Levels 1–3
S6	Finalize_All	Finalize all nodes top-down	$S_4(1) \rightarrow \dots \rightarrow S_4(7)$	Levels 1–7
S7	Complete	Termination state	T	–
S8	Validation_Failure	Terminate due to $R_{\max} = 50$ exhaustion	$S_5$	–

**Table A.14.2.** Unified state transitions for Pbfd MVP, integrating generic rule references and workflow logic

Rule ID	From State	To State	Condition	Generic Rule	Workflow Step
Pbfd1	[*]	S0	Start	PB1	Initialize Level 1 (TLE 1–3)
Pbfd2	S0	S1	Level 1 validated & resolved	PB4a	Proceed to Level 2 (TLE 2–4)
Pbfd3	S1	S2	Level 2 validated & resolved	PB4a	Proceed to Level 3 (TLE 3–5)
Pbfd4	S2	S3	Level 3 validated & resolved	PB4a	Proceed to Level 4 (TLE 4–6)
Pbfd5	S3	S4	Level 4 validated & resolved	PB4a	Proceed to Level 5 (TLE 5–7)
Pbfd6	S2	S5	Level 3 validation failed	PB3	Refine Levels 1-3
Pbfd7	S5	S0	Levels 1-3 reprocessed	PB3a	Resume Level 1 (TLE 1–3)
Pbfd8	S5	S8	$\text{refinement\_attempts} \geq R_{\max}$	PB9	Terminate with error
Pbfd9	S4	S6	Level 5 validated	PB4b	Finalize all levels
Pbfd10	S6	S7	All nodes finalized. Finalization (S6) combines PB7 and PB8, resolving all levels top-down in a single step for efficiency.	PB8	Complete

The state machine representation visually depicts the flow of the Pbfd application, as shown in Figure A.14.2. The transitions between states correspond to the progression and refinement steps of the methodology, with each transition labeled according to the rules defined in Table A.14.2. State S5 (Refine\_Level1-3, Pbfd6) reprocesses Levels 1–3 to resolve inconsistencies before resuming at Level 1. Mermaid code for Figure A.14.2 is provided in Appendix A.15.



**Figure A.14.2.** State machine diagram for PBFDP MVP, showing pattern transitions and completion rules across hierarchical levels

#### Appendix A.14.6. Data Structure and Relationships

The PBFDP MVP relies on a hierarchical, pattern-driven relational schema to represent and traverse location-based data. This structure underpins both the backend logic and the dynamic frontend traversal behavior governed by the TLE Rule (see Section 4.2).

##### 1. Sample Locations Dataset

At the heart of the PBFDP MVP system lies the Locations table (Table A.14.3) — a static reference structure containing all nodes and their hierarchical relationships. This metadata table serves as the input for dynamically generating the grandparent-level tables that form the three-level traversal model.

**Table A.14.3.** Static Locations dataset schema supporting PBFDP pattern traversal and bitmask encoding

Id	Name	Name Type Id	Type	Parent Id	Child Id	Level
0	ContinentGrandparent	null	INT	null	0	1
1	ContinentParent	null	INT	0	0	2
2	North America	1	INT	1	0	3

3	South America	1	INT	1	1	3
9	United States	2	BIGINT	2	0	4
10	Canada	2	INT	2	1	4
14	Brazil	2	INT	3	0	4
38	Virginia	3	VARCHAR(120)	9	11	5
45	Maryland	3	INT	9	18	5
102	Howard County	4	INT	45	12	6
148	Ellicott City	5	INT	102	1	7

### Explanation of Key Fields

- **Id:** Unique identifier for the node
- **Name:** Entity name (e.g., "North America", "Maryland")
- **Name Type Id:** Categorize the entity type (e.g., continent = 1, country = 2). ContinentGrandparent and ContinentParent are structural placeholders for TLE
- **Type:** The SQL data type for the node's bitmask, determined by the maximum number of children:
  - **INT:** Supports up to 32 child selections
  - **BIGINT:** Supports up to 64 child selections
  - **VARCHAR(X):** For >64 children, storing a character-based bitmask representation
- **Parent Id:** References the parent node's Id
- **Child Id:** The node's zero-based position within its parent's bitmask encoding
- **Level:** The node's depth in the hierarchy  
The ChildId enables constant-time bitwise operations for setting, clearing, and testing selection flags, minimizing computational overhead once the target row is accessed [53,55].

### 2. Design Rationale

This static table design supports:

- **Hierarchical Querying:** ParentId define the tree structure.
  - **Pattern Encoding:** ChildId enables bitmask-based grouping within TLE tables.
  - **Dynamic Generation:** Serves as input to recursively generate TLE tables at runtime, adapting bitmask data types as needed for flexibility.
  - **Consistency:** Levels 1–5 follow a consistent schema; Levels 6–7 are embedded as bitmasks within parent levels.
- ### 3. Integration with TLE
- Every TLE-compliant grandparent table derives its structure from the Locations table:
- ParentId defines column-to-row relationships.
  - ChildId defines the bit position in the bitmask.

#### Example:

- "United States" (ChildId = 0) → 0b0001 = bitmask 1
- "Canada" (ChildId = 1) → 0b0010 = bitmask 2

This approach of replacing deep recursive joins with precomputed, encoded tables reduces I/O and aligns with design rationales in columnar storage systems [53,55], though it introduces the operational complexity of dynamic schema generation— a trade-off that aligns with foundational database architecture principles, where encoded storage and performance optimizations often necessitate increased system complexity [134].

### Appendix A.14.7. Three-Level Encapsulation (TLE) Rule

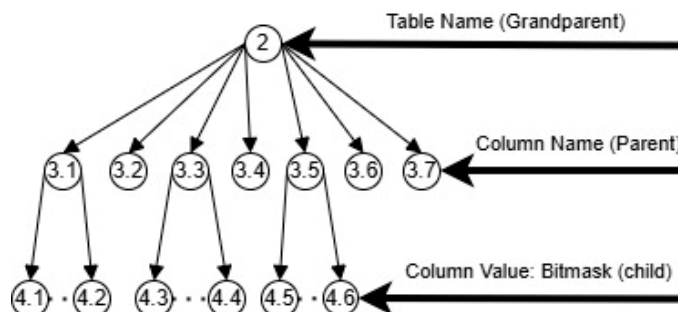
PBFD applies the TLE (Three-Level Encapsulation) rule to model each three-level span in the hierarchy using a single table. This design maps a contiguous span (grandparent→parent columns→child bitmask) into one table, enabling one-hop reads from a root record to its grandchild selections and avoiding multi-join traversal for pattern queries. This

approach is analogous to materialized or denormalized encodings used in high-performance DBMS designs (columnar and encoded stores) [53,55,118].

For optimization purposes, the handling of the final three-level span, encompassing the lowest two hierarchical levels, deviates from the standard dynamic table generation.

#### Example of a TLE Unit

In a regional structure (see Figure A.14.3):



**Figure A.14.3.** Example of a Three-Level Encapsulation (TLE) unit mapping levels 2–4 in the PBF hierarchy

- **Grandparent (Level 2):** ContinentParent (Grandparent, Node 2)
- **Parent (Level 3):** [North America], [South America], etc. (Parent columns, Nodes 3.1 – 3.7)
- **Child (Level 4):** Bitmask for selected countries within each continent (Child state, Nodes 4.1 – 4.6)

#### Grandparent Table Hierarchy

The hierarchy begins at the conceptual ContinentGrandparent (Level 1) and extends downward. The fictitious top-level nodes (ContinentGrandparent, ContinentParent) act as structural sentinels [135]—providing a stable anchor for the TLE encapsulation boundaries. They prevent root-level special cases and allow the TLE pattern to be applied uniformly across all hierarchical segments. Table A.14.4. summarizes the TLE scope for the three-level segments.

**Table A.14.4.** Mapping of hierarchical levels to TLE units in PBF MVP, including node roles and bitmasks

Level	Grandparent Node (Table)	Parent Nodes (Columns)	Child Nodes (Bitmask)	Three-Level Scope
1	ContinentGrandparent	Continentparent	Continent selections (e.g. North America (1))	Levels 1–3
2	Continentparent	e.g. Asia, North America	Country selections (e.g. United States (1))	Levels 2–4
3	Continent	e.g. United States, Canada	State selections (e.g., Maryland (262,144))	Levels 3–5
4	Country	e.g. Virginia, Maryland	County selections (e.g., Howard County (4096))	Levels 4–6
5	State	e.g. Howard County, Baltimore County	City selections (e.g., (Columbia MD + Ellicott City) (3))	Levels 5–7

**Note:** Parenthesized values represent decimal bitmasks.

#### Handling the Lowest Two Hierarchical Levels

As the asymptotic analysis in Appendix A.16 demonstrates, the lowest hierarchical levels in a perfect ternary tree contain approximately 89% of all nodes. To mitigate the potential explosion of dynamic tables, the PBF methodology leverages TLE’s hierarchical encapsulation by embedding Levels 6 (County) and 7 (City) into their grandparent table (State, Level 5):

- **County Level (Level 6):** Represented as dedicated columns within the State table (Level 5)
  - **City Level (Level 7):** Stored as bitmasks within the corresponding County columns
- This embedding minimizes the number of dynamic tables and preserves compact storage.

Table A.14.5. (Dynamic Table Maryland (Level 5)) illustrates this structure, where counties are represented as columns, and city selections are stored as bitmasks within those columns for a specific state.

**Table A.14.5.** Bitmask-encoded dynamic table for Maryland (Level 5), illustrating embedded county/city selections

PersonId	Howard County (bitmask)	.....
1	3	.....

#### Justification

This TLE-based relational design provides several key benefits:

- It encapsulates the grandparent-parent-child hierarchy within a single unit, using bitmasks for O(1) updates and enabling parallel resolution of nodes within a pattern.
- Leveraging the analytical findings from Appendix A.16, it avoids creating hundreds of tables for leaf-level data by embedding their states, thus maintaining modularity and performance despite the exponential node growth in deeper levels.
- Scalability Alignment: By minimizing dynamic table proliferation and maintaining compact storage, this approach supports the horizontal scaling and operational efficiency required in cloud-native environments.

#### Appendix A.14.8. Database Implementation (SQL Server)

The Pbfd MVP backend uses SQL Server and combines static tables with dynamically generated Three-Level Encapsulation (TLE) tables. This design replaces deep recursive joins with compact, schema-on-demand structures optimized via bitmask encoding [90].

#### Dynamic TLE Table Generation

Dynamic tables are derived from the static Locations lookup table through an automated transformation pipeline. Rather than storing each hierarchical level in a fully normalized chain of joins, Pbfd generates three-level encapsulated tables that encode grandparent–parent–child relationships. Bitmask columns encode child selections as binary flags, enabling constant-time set, clear, and test operations within SQL Server.

##### Algorithm: Dynamic TLE Table Generator

##### Let

- N denote the current hierarchical level
- L denote the maximum depth of the hierarchy (in Pbfd MVP, L=7)
- The algorithm iterates from level 1 to L - 2, generating one dynamic table per grandparent node

##### Input:

- Locations metadata (table or JSON)
- Maximum dynamic depth = 5 (up to the State level)

##### Output:

- SQL table per grandparent that follows the TLE rule (level N)
- One column per parent (level N+1)
- One bitmask field encoding child selections (level N+2)

##### Steps:

1. Load the Locations data
2. Group nodes by hierarchical level
3. For each level N from 1 to L-2:

For each node at level N, generate a dynamic table corresponding to that grandparent node, with:

- One column for each parent node at level N+1
  - One bitmask field encoding child selections at level N+2
4. Skip dynamic table creation for the lowest two levels (L-1 and L):
    - These levels are embedded into their grandparent's table as described in Appendix A.14.7, using dedicated columns and bitmask fields

This approach scales to arbitrary depth while maintaining constant-time lookup and update via bitwise operations. It reflects principles seen in schema-on-read and evolution-oriented persistence models [90].

**Example root-level table:**

- ContinentGrandparent (Level 1, Id = 0)
- Serves as the hierarchical entry point and contains bitmask columns for descendant states or subregions

**Operational Safeguards and Deployment**

To prevent schema drift or runtime faults:

- Deterministic CREATE TABLE generation occurs as part of controlled deployment scripts.
- All DDL changes are executed inside transactions to ensure rollback safety.
- Preflight checks validate bitmask width, column compatibility, and backward consistency before applying any schema upgrades.
- Type escalation (e.g., INT → BIGINT → VARCHAR) is handled automatically when child-node cardinality outgrows the existing bitmask type.

These safeguards align with established practices in schema evolution and controlled denormalization within polyglot persistence systems [90].

**Integrated Schema Structure**

The resulting database consists of:

**Static Tables:**

- Persons (core entity table)
- Locations (full hierarchy metadata)
- NameTypes (categorization of nodes: continent, country, etc.)

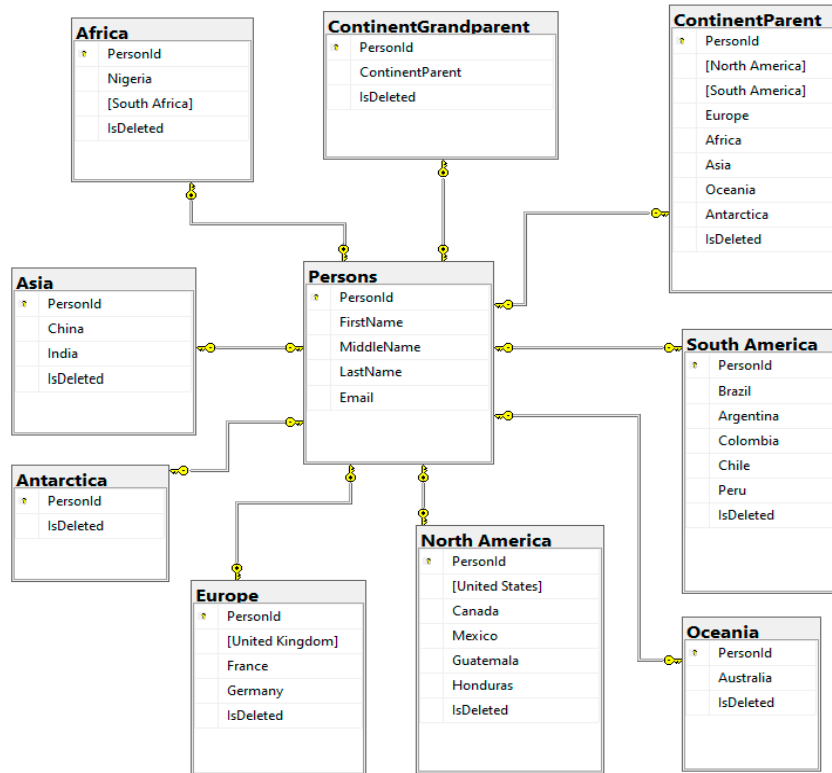
**Dynamic TLE Tables (auto-generated):**

- **Level 1:** ContinentGrandparent
- **Level 2:** ContinentParent
- **Level 3:** one table per continent (e.g., NorthAmerica, Asia, etc.)
- **Level 4:** one table per country (e.g., [United State], Canada, etc.)
- **Level 5:** one table per state (e.g., Alabama, California, etc.)
- Lower levels embedded via bitmask columns rather than additional tables

**Figure A.14.4. illustrates:**

- The Persons table as the static entry point
- Dynamically generated TLE structures for the first three hierarchical levels
- One-hop access paths from Persons
- Clear delineation of bitmask fields and level boundaries within each dynamic table

Clear delineation of hierarchical roles—table name as grandparent, columns as parents, and bitmask fields as children—within each dynamic TLE table.



**Figure A.14.4.** PBFDP MVP database schema integrating static and dynamic TLE-compliant tables with bitmask encoding

Appendix A.14.9. PBFDP Loosely Coupled Table Design Benefits

PBFDP's dynamic Three-Level Encapsulation (TLE) design replaces rigid, deeply joined schemas with a scalable, loosely coupled architecture. This approach preserves the core advantages of relational databases while systematically addressing common performance and operational bottlenecks. The benefits are summarized in the tables below.

**Table A.14.6.** Key relational database benefits preserved in PBFDP MVP's TLE-based design

Feature	Benefit
Normalization [136]	Static tables are highly normalized.
Security [137]	Table-level permissions enforce granular access control (e.g., permitting team-specific access to regional data), a foundational relational security model.
Optimization [55,138]	Each grandparent table can utilize separate indexes and be independently partitioned or sharded, allowing for targeted performance tuning.

**Table A.14.7.** Relational challenges and PBFDP MVP's architectural solutions

Challenge	PBFDP Solution
Multi-Table Joins [139]	Replaces 4–5 join traversals with direct, one-hop access to precomputed grandparent tables, dramatically reducing query complexity.
ORM/Workflow Complexity [140]	Employs a single controller and view model across all hierarchical levels, simplifying the application layer and minimizing code duplication.
Backup/Restore Bottlenecks [141]	Enables modular, table-level operations (e.g., backing up only the "Europe" dataset), which aligns with modern, cloud-native operational practices [90].

The empirical benefits observed in the MVP stem from three key design outcomes: **(a)** a significant reduction in joins per pattern query, **(b)** a compact bitmask representation that lowers I/O for read-heavy paths, and **(c)** a table-level granularity that facilitates independent management. This architectural strategy embodies a practical form of denormalization, trading initial schema

complexity for sustained query and operational efficiency, a trade-off well-documented in literature on schema evolution and polyglot persistence.

#### Appendix A.14.10. Development Process

The Pbfd MVP follows a top-down hierarchical construction guided by the central Locations metadata table and TLE-compliant data models. The process is engineered for reproducibility and for validating the methodology's core claims. The complete, step-by-step implementation details are available for inspection and verification in Appendix A.17.

##### Process Flow (high level)

1. **Frontend – Visitor entry & pattern selection:** The frontend collects visitor data, including each party's initial pattern choices.
2. **Backend – Dynamic generation:** The Locations table is consulted to deterministically generate TLE tables (CREATE TABLE statements).
3. **UI – Shared rendering:** A single Razor view and ViewModel are reused across levels to render pattern options, reducing duplication.
4. **Data update – Bitmask write:** User actions are persisted by updating the bitmask column in the grandparent table (typically a single-row  $O(1)$  operation).

##### Key Methodology Claims (Instantiated in the MVP)

- **Hierarchy-Aware Design:** Logical table boundaries are enforced for each three-level scope via TLE, aligning with structured decomposition principles in hierarchical relational schemas [118].
- **Bitmask Optimization:** Compact selection encoding enables constant-time set, clear, and test operations using native bitwise expressions in SQL Server, reflecting established practices in encoded and columnar data representations [23,53,55,118].
- **Reusable Workflow:** A single MVC controller and ViewModel operate across all hierarchical levels, minimizing ORM complexity and duplication in line with multi-view reuse patterns in enterprise MVC frameworks [142].
- **Bounded Refinement:** Refinement steps are capped at  $R_{\max} = 50$  per level, as defined in Table 42, enforcing loop bounds consistent with formal lifecycle-driven termination strategies [83].
- Exceeding  $R_{\max}$  transitions the workflow to state S8, as specified in Table A.14.2, enforcing bounded iteration and controlled bailout paths consistent with ISO/IEC 12207 lifecycle termination principles [87].

#### Appendix A.14.11. Key Claims Supported and Academic Grounding

This MVP provides empirical evidence supporting the following claims, grounded in established computer science and database literature (See Table A.14.8).

**Table A.14.8.** Key Claims Supported and Academic Grounding

Claim	Academic Grounding
Bitwise/encoded access provides substantial read efficiency for pattern queries.	Grounded in columnar/encoding database literature [23,53,55]
Recursive-CTE/adjacency-list traversal has depth-dependent costs (worse for broad/deep hierarchies).	Grounded in classical database texts on hierarchical representations and relational trade-offs [118]
TLE's dynamic table approach is a practical denormalization strategy that trades schema complexity for query and operational efficiency.	Consistent with schema evolution and polyglot persistence research [90]
Bounded iterative refinement and backtracking map to classical search/backtracking techniques.	Supported by DFS/BFS algorithmic foundations and process-refinement literature [62,77,83]

Formal verification of workflow/state-machine behavior aligns with CSP paradigms, and the MVP inherits its structural and behavioral guarantees from the verified Generic model.

Grounded in process algebra and model checking guidance (CSP) [45,71,87], as applied to the Generic model from which the MVP is derived

By integrating these elements, the Pbfd MVP operationalizes concepts typically treated in isolation—encoded storage, bounded search, hierarchical partitioning, and verification—into a unified and reproducible development methodology.

#### Appendix A.15. Pbfd MVP State Machine Workflow Mermaid Code

Mermaid Code for Figure A.14.2:

##### stateDiagram-v2

direction TB

[\*] --> S0

state "S0: Level 1<br>Process/Validate/Resolve<br>(TLE 1–3)" as S0

state "S1: Level 2<br>Process/Validate/Resolve<br>(TLE 2–4)" as S1

state "S2: Level 3<br>Process/Validate/Resolve<br>(TLE 3–5)" as S2

state "S3: Level 4<br>Process/Validate/Resolve<br>(TLE 4–6)" as S3

state "S4: Level 5<br>Process/Validate<br>(TLE 5–7)" as S4

state "S5: Refine L1-L3" as S5

state "S6: Finalize All" as S6

state "S7: Complete" as S7

state "S8: Error" as S8

S0 --> S1 : Pbfd2<br>S0 done

S1 --> S2 : Pbfd3<br>S1 done

S2 --> S3 : Pbfd4<br>S2 done

S3 --> S4 : Pbfd5<br>S3 done

S2 --> S5 : Pbfd6<br>S2 fail

S5 --> S0 : Pbfd7<br>Refined

S5 --> S8 : Pbfd8<br>Attempts≥50

S4 --> S6 : Pbfd9<br>S4 done

S6 --> S7 : Pbfd10<br>Complete

S7 --> [\*]

#### Appendix A.16. Quantifying Node Reduction in Perfect N-ary Trees

This section quantifies the number of nodes remaining in a perfect n-ary tree after removing all leaves (nodes at the deepest level) and their immediate parent nodes. We assume a perfect n-ary tree of height h, where all levels are fully filled.

##### Key Formula

- Total Nodes (before removal):

$$\sum_{k=0}^h n^k = \frac{n^{(h+1)} - 1}{n - 1}$$

- Nodes removed:
  - Leaves (level h):  $n^h$  nodes
  - Parent level (level h-1):  $n^{(h-1)}$  nodes
- Remaining nodes (after removing leaves and their parents):

$$N_{remaining} = N_{total} - (n^h + n^{(h-1)}) = \frac{n^{(h+1)} - 1}{n - 1} - (n^h + n^{(h-1)})$$

Remaining Nodes (after removing leaves and their parents):

$$P_{remaining} = \left( \frac{N_{remaining}}{N_{total}} \right) \times 100\%$$

**Example:** Ternary Tree ( $n = 3$ ) of Height  $h = 6$

**Step 1:** Compute the Total Nodes

$$N_{total} = \frac{3^{(6+1)} - 1}{3 - 1} = \frac{3^{(7)} - 1}{2} = \frac{2187 - 1}{2} = 1093 \text{ nodes}$$

**Step 2:** Compute the Nodes to Remove

- Leaves (Level 6):  $3^6 = 729$  nodes
- Parent Level (Level 5):  $3^5 = 243$  nodes
- Total Nodes Removed:  $729 + 243 = 972$  nodes

**Step 3:** Compute the Remaining Nodes

$$N_{remaining} = 1093 - 972 = 121 \text{ nodes}$$

**Step 4:** Compute the Remaining Nodes' Percentage

$$P_{remaining} = \frac{121}{1093} \times 100\% \approx 11.07\%$$

**Step 5:** Percentage of Last Two Levels

- Nodes in last two levels:  $729 + 243 = 972$  nodes
- Percentage of last two levels:  $(972 / 1093) \times 100\% \approx 88.93\%$

Thus, after removing the leaves and their parent level, only 121 nodes or approximately 11% remain in the tree. The last two levels (5 and 6) constitute approximately 89% of the total tree (see Table A.16.1).

**Table A.16.1.** Summary for Ternary Tree ( $n = 3$ ,  $h = 6$ )

Metric	Value	Percentage
Total nodes	1,093	100.00%
Level 6 (leaves)	729	66.70%
Level 5 (parents)	243	22.23%
Last two levels combined	972	88.93%
Remaining nodes (Levels 0–4)	121	11.07%

This analysis informs the Pbfd MVP design (Appendix A.14), in which the bottom two hierarchical levels— representing approximately 89% of nodes in a ternary tree— are fully encapsulated within their grandparent table. This prevents excessive table proliferation while representing TLE's performance characteristics.

#### Appendix A.17. Pbfd MVP Development Process

This section details the step-by-step progression of the Pbfd MVP's development process. The corresponding source code is provided in [29].

##### Appendix A.17.1. The Visitor Page

- **Purpose:** Captures initial visitor information (e.g., name, contact details) and persists it to the static Persons table (Table A.13.1)
- **Design:**
  - **Model:** Person (maps to Persons table)
  - **UI:** Person node excluded from Pbfd MVP hierarchy (Figure A.15.1) but serving as root node in Pbfd MVP design (Figure A.11.1)
- **Workflow:** On submission, redirects to the Continent Page to begin hierarchical selections

- **State Machine Context:**
  - **Pre-Processing:** This step occurs before the state machine initializes.
  - **Transition:** Submission triggers PBF1 (Table A.14.2), transitioning to S0 (Level\_1\_Processing\_Validating\_Resolving) (Table A.14.1).

## Appendix A.17.2. Continent Level (Child Level 3, Grandparent Level 1)

### 1. Hierarchical Structure

TLE Rule Implementation (see Table A.17.1): The continent bitmask is stored as a column value under its parent node—ContinentParent, which resides within the grandparent node—Table ContinentGrandparent (Table A.17.2, Figure A.17.1). This follows the TLE rule for hierarchical data structuring.

**Table A.17.1.** Sample mapping of grandparent, parent, and child nodes at the continent level based on TLE encoding

Child LocationId	ChildId	Child Node	Parent Node (Columns)	Grandparent Node (Table)
2	0	North America	ContinentParent	ContinentGrandparent
4	2	Europe	ContinentParent	ContinentGrandparent
6	4	Asia	ContinentParent	ContinentGrandparent

**Table A.17.2.** Bitmask encoding (Decimal) of selected continent nodes stored in the ContinentGrandparent table

PersonId	ContinentParent
1	21

The ContinentGrandparent and ContinentParent tables are structural artifacts (analogous to sentinel nodes in linked lists) introduced to enable root-level TLE encapsulation. While physically persisted, they represent conceptual hierarchy levels not present in raw geographical data.

Continent Name	Name Type	Select
Africa	Continent	<input type="checkbox"/>
Antarctica	Continent	<input type="checkbox"/>
Asia	Continent	<input checked="" type="checkbox"/>
Europe	Continent	<input checked="" type="checkbox"/>
North America	Continent	<input checked="" type="checkbox"/>
Oceania	Continent	<input type="checkbox"/>
South America	Continent	<input type="checkbox"/>

**Figure A.17.1.** Continent level interface showing checkbox-based selection of continent nodes using bitmask encoding

### 2. Key Workflow

- **Data Retrieval:** The LocationViewModel fetches continent nodes from the Locations table (Table A.14.3) where ParentId = 1.
- **UI Binding:** Continent names (e.g., "North America") are bound to checkboxes in the interface (Figure A.17.1).
- **Bitmask Encoding:** Selected continents are encoded as bitmasks (e.g., 21 for North America + Europe + Asia).
- **Persistence:** Bitmasks are saved in the ContinentGrandparent table (Table A.17.2).

### 3. Continent Level Interface

- **Node Mapping (Figure A.14.1):** Nodes 3.1–3.7 represent continents (e.g., 3.1 = North America).
- **Example:** Selecting Asia (3.5), Europe (3.3), and North America (3.1) generates the bitmask 000000000010101 (decimal 21).

## 4. Interpretation

**Node: ContinentParent**

- **Decimal Value:** 21
  - **Binary Value:** 00010101 (8-bit format)
  - Bit Positions Set:**
    - **Bit 0:** North America (Node 3.1 in Figure A.14.1)
    - **Bit 2:** Europe (Node 3.3 in Figure A.14.1)
    - **Bit 4:** Asia (Node 3.5 in Figure A.14.1)
  - **UI:** North America, Europe, and Asia appear as checked checkboxes in Figure A.17.1.
  - **Storage:** Selected continents are stored as bitmasks in the ContinentGrandparent table (Table A.17.2), with each bit representing a continent.
5. Workflow Impact
- **Selection:** Selections are saved as bitmasks in ContinentGrandparent.
  - **Deselection:** Unchecking North America updates the bitmask to 20 (0000000000010100), while the LocationResetService recursively clears all associated child data within North America (including Country, State, etc.).
  - **UI/Backend Split:** Only child nodes (Continents) are displayed, with grandparent and parent nodes managed by middleware.
6. State Machine Context
- **Current State:** S0 (Level\_1\_Processing\_Validating\_Resolving) (Table A.14.1)
  - **TLE Structure:** Processes Child Level 3 under Grandparent Level 1 (ContinentGrandparent table)
  - **Transition:** On submission, advances to S1 (Level\_2\_Processing\_Validating\_Resolving) via PBF2 (Table A.14.2)

## Appendix A.17.3. Country Level (Child Level 4, Grandparent Level 2)

## 1. Hierarchical Structure

**TLE Rule Implementation:** In the Country Level, Columns in ContinentParent (e.g., 'North America') are dynamically generated only for continents selected at Level 3 (see Table A.17.3). These columns represent parent nodes (continents), while country selections are stored as bitmasks within their respective continent columns (see Table A.17.4 and Figure A.17.2).

Table A.17.3. Sample mapping of grandparent, parent, and child nodes at the country level following TLE rules

Child LocationId	ChildId	Child Node	Parent Node (Columns)	Grandparent Node (Table)
9	0	United States	North America	ContinentParent
10	1	Canada	North America	ContinentParent
19	0	United Kingdom	Europe	ContinentParent
20	1	France	Europe	ContinentParent
24	0	China	Asia	ContinentParent
25	1	India	Asia	ContinentParent

Table A.17.4. Bitmask decimal values representing selected countries persisted in the ContinentParent table

PersonId	North America	Europe	Asia
1	3	3	0

## 2. Key Workflow

- **Parent Nodes:** Columns in the ContinentParent table (e.g., "North America") correspond to selected continents from the previous level (Table A.17.2).
- **Child Bitmasks:** Each column value encodes selected countries using a bitmask (e.g., 00000011 for United States and Canada, as shown under the [North America] column in Table A.17.4).

- **UI Rendering:** The LocationViewModel populates checkboxes for countries under selected continents (Figure A.17.2). Only child nodes (countries) and parent nodes (Continents) are displayed, with grandparent nodes managed by middleware. This hierarchical approach continues consistently down to the city level.

Asia		
Name	Name Type	Select
China	Country	<input type="checkbox"/>
India	Country	<input type="checkbox"/>

Europe		
Name	Name Type	Select
France	Country	<input checked="" type="checkbox"/>
Germany	Country	<input type="checkbox"/>
United Kingdom	Country	<input checked="" type="checkbox"/>

North America		
Name	Name Type	Select
Canada	Country	<input checked="" type="checkbox"/>
Guatemala	Country	<input type="checkbox"/>
Honduras	Country	<input type="checkbox"/>
Mexico	Country	<input type="checkbox"/>
United States	Country	<input checked="" type="checkbox"/>

**Figure A.17.2.** Country level interface with dynamically rendered checkboxes based on selected continents and encoded as bitmasks

### 3. Interpretation

#### Node: North America

- **Bitmask Value:** 3 (binary 00000011 (8-bit format))
- Set Bits:
  - **Bit 0:** United States (Node 4.1 in Figure A.14.1)
  - **Bit 1:** Canada (Node 4.2 in Figure A.14.1)
- **Storage:** Saved in the North America column of the Continent table (Table A.17.4)

#### Node: Europe

- **Bitmask Value:** 3 (binary 00000011(8-bit format))
- Set Bits:
  - **Bit 0:** United Kingdom (Node 4.5 in Figure A.14.1)
  - **Bit 1:** France (Node 4.6 in Figure A.14.1)
- **Storage:** Persisted in the Europe column of the Continent table (Table A.17.4)

#### Node: Asia

- **Bitmask Value:** 0 (binary 00000000(8-bit format))
- **Set Bits:** None (all bits unset)
- **Storage:** Persisted in the Asia column of the Continent table (Table A.17.4)

### 4. Workflow Impact

- **Selection:** Selecting a country (e.g., United States) causes the corresponding state-level tables to be displayed.
- **Deselection:** Unchecking a country (e.g., Canada) invokes the LocationResetService, recursively nullifying child data (states, counties, etc.).

### 5. State Machine Context

- **Current State:** S1 (Level\_2\_Processing\_Validating\_Resolving) (Table A.14.1)
- **TLE Structure:** Processes Child Level 4 under Grandparent Level 2 (ContinentParent table)
- **Transition:** Advances to S2 (Level\_3\_Processing\_Validating\_Resolving) via PBF3 after validation

#### Appendix A.17.4. State Level (Child Level 5, Grandparent Level 3)

## 1. Hierarchical Structure

**TLE Rule Implementation:** In the State Level, columns are dynamically generated in grandparent tables (e.g., North America, Europe, or Asia tables) based on the selected continent-country hierarchy (see Table A.17.5). These columns represent parent nodes (countries), and state selections are stored as bitmasks within the corresponding country columns (see Table A.17.6 and Figure A.17.3).

**Table A.17.5.** Sample mapping of grandparent, parent, and child nodes at the state level using dynamic column generation

Child LocationId	ChildId	Child Node	Parent Node (Columns)	Grandparent Node (Table)
38	11	Virginia	United States	North America
45	18	Maryland	United States	North America
77	0	Ontario	Canada	North America
89	12	Nunavut	Canada	North America

**Table A.17.6.** Bitmask encoding (Decimal) of selected states stored in dynamically generated continent-level (North America) table

PersonId	United States	Canada
1	264192	4097

## 2. Key Workflow

- **Grandparent Tables:** Each grandparent table (e.g., North America in this sample) corresponds to a continent selected at the Country Level (Table A.17.4).
- **Parent Columns:** Columns in the grandparent table (e.g., "United States" in North America) represent selected countries.
- **Child Bitmasks:** Bitmasks in parent columns encode selected states (e.g., 264,192 for Virginia + Maryland in the United States in Table A.17.6)

## 3. Interpretation (Derived from Table A.17.6 and Figure A.17.3)

### Canada

Name	Name Type	Select
Nunavut	State	<input checked="" type="checkbox"/>
Ontario	State	<input checked="" type="checkbox"/>

### France

Name	Name Type	Select
Ile-de-France	State	<input type="checkbox"/>

### United Kingdom

Name	Name Type	Select
England	State	<input type="checkbox"/>
Scotland	State	<input type="checkbox"/>

### United States

Name	Name Type	Select
Alaska	State	<input type="checkbox"/>
California	State	<input type="checkbox"/>
Maryland	State	<input checked="" type="checkbox"/>
Virginia	State	<input checked="" type="checkbox"/>

Submit

**Figure A.17.3.** State level interface illustrating checkboxes for states rendered from selected countries using bitmask storage

### North America (Grandparent Table)

- Parent Column (United States):
  - **Bitmask Value:** 264,192 (binary 10000001000000000000 (20-bit format))
  - Set Bits:
    - **Bit 11:** Virginia (Node 5.2 in Figure A.14.1)
    - **Bit 18:** Maryland (Node 5.1 in Figure A.14.1)
- Parent Column (Canada):
  - **Bitmask Value:** 4,097 (binary 0001000000000001(16-bit format))
  - Set Bits:
    - **Bit 0:** Ontario (Node 5.4 in Figure A.14.1)
    - **Bit 12:** Nunavut (Node 5.3 in Figure A.14.1)

### UI Consistency

- The same LocationViewModel renders checked states (e.g., Maryland, Nunavut) across all grandparent tables (e.g., North America, Europe), as shown in Figure A.17.3.

### Storage

- Selected states are stored as bitmasks in the North America table (Table A.17.6), with columns representing parent countries.

### 4. Technical Note

The bigint data type (64-bit) is used for the United States due to its 50 states, ensuring sufficient bitwise capacity (see Table A.14.3).

### 5. Workflow Impact

- **Selection:** Choosing a state (e.g., Maryland) causes the corresponding county-level tables and user interfaces to be displayed.
- **Deselection:** Unchecking a state (e.g., Virginia) invokes the LocationResetService, recursively nullifying child data (counties, cities).

### 6. State Machine Context

- **Current State:** S2 (Level\_3\_Processing\_Validating\_Resolving) (Table A.14.1)
- **TLE Structure:** Processes Child Level 5 under Grandparent Level 3 (e.g. [North America] table)
- **Transition:**
  - **On success:** Advances to S3 (Level\_4\_Processing\_Validating\_Resolving) via PBFD4
  - **On failure:** Transitions to S5 (Refine\_Level1-3) (Table A.14.1) via PBFD6

### Appendix A.17.5. County Level (Child Level 6, Grandparent Level 4)

#### 1. Hierarchical Structure

TLE Rule Implementation: In the County Level, columns are dynamically generated within Country Level tables (e.g., United States), following the TLE Rule (see Table A.17.7). These columns represent parent nodes (states), while county selections are stored as bitmasks within their respective state columns (see Table A.17.8 and Figure A.17.4).

**Table A.17.7.** Sample mapping of grandparent, parent, and child nodes at the county level using country-specific tables

Child LocationId	ChildId	Child Node	Parent Node (Columns)	Grandparent Node (Table)
92	2	Baltimore County	Maryland	United States
102	12	Howard County	Maryland	United States
120	6	Arlington County	Virginia	United States
186	28	Fairfax County	Virginia	United States

**Table A.17.8.** Bitmask decimal values for selected counties stored in the United States table

PersonId	Virginia	Maryland
----------	----------	----------



a minor string-to-BigInteger conversion overhead, it provides crucial flexibility and scalability for variable-length bitmasks, simplifying schema management and application logic compared to fixed-size integer alternatives.

#### 5. Workflow Impact

- **Selection:** Selected counties trigger the collection of City Level data (e.g., cities under Howard County like Columbia MD), which are stored as bitmasks within the parent county columns of the Country Level tables (e.g., United States).
- **Deselection:** Unchecking a county (e.g., Fairfax County) invokes the LocationResetService, recursively nullifying its child city bitmasks.

#### 6. State Machine Context

- **Current State:** S3 (Level\_4\_Processing\_Validating\_Resolving) (Table A.14.1)
- **TLE Structure:** Processes Child Level 6 embedded in Grandparent Level 4 (e.g. [United States] table)
- **Transition:** Advances to S4 (Level\_5\_Processing\_Validating) via PBFD5

### Appendix A.17.6. City Level (Child Level 7, Grandparent Level 5)

#### 1. Hierarchical Structure

TLE Rule Implementation (see Table A.17.9): In the City Level, columns are dynamically generated within State Level tables (e.g., Maryland, Virginia) to represent parent nodes (counties), and city selections are stored as bitmasks within these dynamically created county columns (see Tables A.17.10, A.17.11, and Figure A.17.5).

**Table A.17.9.** Sample mapping of grandparent, parent, and child nodes at the city level using dynamically generated state tables

Child LocationId	ChildId	Child Node	Parent Node (Columns)	Grandparent Node (Table)
138	0	Arbutus	Baltimore County	Maryland
139	1	Catonsville	Baltimore County	Maryland
146	0	Columbia MD	Howard County	Maryland
147	1	Ellicott City	Howard County	Maryland
149	3	Laurel	Howard County	Maryland
156	0	Arlington	Arlington County	Virginia
164	8	Virginia Square	Arlington County	Virginia

**Table A.17.10.** Bitmask decimal values representing city selections stored in the Maryland table

PersonId	Baltimore County	Howard County
1	3	3

**Table A.17.11.** Bitmask decimal values representing city selections stored in the Virginia table

PersonId	Arlington County	FairFax County
1	257	0

## Arlington County

Name	Name Type	Select
Arlington	City	<input checked="" type="checkbox"/>
Virginia Square	City	<input checked="" type="checkbox"/>

## Baltimore County

Name	Name Type	Select
Arbutus	City	<input checked="" type="checkbox"/>
Catonsville	City	<input checked="" type="checkbox"/>

## Howard County

Name	Name Type	Select
Columbia MD	City	<input checked="" type="checkbox"/>
Ellicott City	City	<input checked="" type="checkbox"/>
Laurel	City	<input type="checkbox"/>

Submit

**Figure A.17.5.** City level interface showing checkbox-based city selections for selected counties using TLE-encoded bitmasks

## 2. Key Workflow

- **Data Retrieval:** The LocationViewModel fetches counties (e.g., Howard County) selected at the County Level (Table A.14.3).
- **UI Binding:** Cities under selected counties (e.g., Columbia MD, Arlington) are bound to checkboxes (Figure A.17.5).
- **Bitmask Encoding:** Selections are stored as bitmasks in county columns (e.g., Howard County = 3).
- **Persistence:** Bitmasks are saved in State Level tables (e.g., Maryland).

## 3. Interpretation

### Node: Howard County

- **Binary:** 00000011 (8-bit format)
- Set Bits:
  - **Bit 0:** Columbia MD (Node 7.3 in Figure A.14.1)
  - **Bit 1:** Ellicott City (Node 7.4 in Figure A.14.1)
- **UI:** Both cities are checked in Figure A.17.5.

### Node: Baltimore County

- **Binary:** 00000011 (8-bit format)
- Set Bits:
  - **Bit 0:** Arbutus (Node 7.1 in Figure A.14.1)
  - **Bit 1:** Catonsville (Node 7.2 in Figure A.14.1)
- **UI:** Both cities are checked in Figure A.17.5.

### Node: Arlington County

- **Binary:** 100000001 (9-bit format)
- Set Bits:
  - **Bit 0:** Arlington (Node 7.5 in Figure A.14.1)
  - **Bit 8:** Virginia Square (Node 7.6 in Figure A.14.1)
- **UI:** Both cities are checked in Figure A.17.5.

### Node: Fairfax County

- **Binary:** 00000000 (8-bit format)
- **Interpretation:** No cities selected
- **UI:** All cities under Fairfax County are unselected and not shown in Figure A.17.5.

### Storage

- Selected cities are stored as bitmasks in State Level tables (e.g., Maryland, Virginia) under county columns (Tables A.17.10. and Tables A.17.11).
- Workflow Impact
    - Selection:** Selected cities are encoded as bitmasks within their respective parent county columns (e.g., Columbia MD, stored in the Howard County column).
    - Deselection:** Unchecking a city (e.g., Virginia Square) updates the bitmask and nullifies its data.
  - State Machine Context
    - Current State:** S4 (Level\_5\_Processing\_Validating) (Table A.14.1)
    - TLE Structure:** Processes Child Level 7 embedded in Grandparent Level 5 (e.g., Maryland table)
    - Transition:** Advances to S6 (Finalize\_All) via PBF9

#### Appendix A.17.7. The Report Page

The LocationReportService generates hierarchical location reports by leveraging the TLE Rule (defined in Section 4.2) to traverse checked nodes in the workflow (Figure A.14.1).

##### Key Components

The LocationReportService leverages the following components to generate hierarchical reports:

- Caching Mechanism:**
  - Metadata Cache:** Preloads table/column names (e.g., ContinentGrandparent, North America)
  - Data Cache:** Stores hierarchical data (e.g., continent-country mappings)
- Recursive CTE Engine:** Constructs hierarchical paths using SQL Common Table Expressions
- Bitwise Decoder:** Resolves selected nodes from stored bitmasks (e.g., Continent = 21 → North America + Europe + Asia)

##### Workflow

- Queue Initialization:**
  - Starts from the root node (ContinentGrandparent, Node 1 in Figure A.14.1) and processes checked nodes breadth-first
- TLE Rule Traversal:**
  - Grandparent:** Active table (e.g., ContinentGrandparent)
  - Parent:** Columns representing child nodes of grandparents (e.g., North America)
  - Child:** Bitmasks encoding grandchild node selections (e.g., United States and Canada under North America)
- Path Generation:**
  - Uses recursive CTEs to build paths (e.g., Continent → North America → United States)
- Aggregation:** Combines visited paths into a unified report (Figure A.17.6)

#### Location Paths Report

- ContinentGrandparent > ContinentParent > Asia
- ContinentGrandparent > ContinentParent > Europe > France
- ContinentGrandparent > ContinentParent > Europe > United Kingdom
- ContinentGrandparent > ContinentParent > North America > Canada > Nunavut
- ContinentGrandparent > ContinentParent > North America > Canada > Ontario
- ContinentGrandparent > ContinentParent > North America > United States > Maryland > Baltimore County > Arbutus
- ContinentGrandparent > ContinentParent > North America > United States > Maryland > Baltimore County > Catonsville
- ContinentGrandparent > ContinentParent > North America > United States > Maryland > Howard County > Columbia MD
- ContinentGrandparent > ContinentParent > North America > United States > Maryland > Howard County > Ellicott City
- ContinentGrandparent > ContinentParent > North America > United States > Virginia > Arlington County > Arlington
- ContinentGrandparent > ContinentParent > North America > United States > Virginia > Arlington County > Virginia Square
- ContinentGrandparent > ContinentParent > North America > United States > Virginia > Fairfax County

**Figure A.17.6.** PBF9 Report Page interface displaying hierarchical output generated from recursive bitmask decoding and TLE traversal

#### Appendix A.17.8. Development with CDD

- Refactoring Journey
  - Initial Approach:**

- **Redundant Components:** Each level (ContinentGrandparent, ContinentParent, and Continent) had dedicated models, views, and controllers.
- **Bottleneck:** Code duplication increased maintenance costs at the Continent Level (grandparent Level 3 in Figure A.14.1).
- **Realization of Shared Logic:**
  - **Hierarchical Symmetry:** Identified recurring patterns (TLE Rule) across levels
  - **Refactoring:**
    - **Shared Models:** LocationViewModel, LocationSaveService
    - **Unified View:** Dynamic UI rendering based on JSON configuration
    - **Centralized Controller:** LocationController handling all levels
- **Impact:**
  - **Workflow Alignment:** Aligns UI-centric child-level workflows with the database's grandparent table hierarchy. Curve a (See Figure A.14.1) depicts this mapping: As UI focus shifts from child data at Level 5 (e.g., States) up to Level 3 (e.g., Continents), the corresponding database operations target grandparent tables from Level 3 (e.g., the Continent table) up to Level 1 (e.g., the ContinentGrandparent table).

This refactoring journey epitomizes effective CDD. By identifying the 'hierarchical symmetry' and consistent 'TLE Rule' patterns across geographical levels, we abstracted level-specific logic into reusable shared components (e.g., LocationViewModel, LocationSaveService, LocationController). This dramatically reduced code duplication, simplified maintenance, and significantly enhanced the system's extensibility. Future hierarchy expansions or rule modifications now primarily involve metadata updates and leverage existing, verified components, substantially lowering long-term total cost of ownership and adapting to evolving data requirements.

## 2. State Machine Context

- **Current State:** S5 (Refine\_Level1-3) (See Table A.14.1)
- **TLE Structure:** Processes Child Levels 3-7 embedded in Grandparent Levels 1-5
- **Transition:** Refactoring prompted a restart from Level 3 (S2) to Level 1 (S0) via S5, reprocessing Levels 1-3 to resolve shared component dependencies

## 3. Formal Validation Takeaways

Validation prioritizes CDD where refinement iterations create unique cyclomatic risks requiring bounded termination ( $R_{\max}=50$ ). Sequential elements inherit correctness from CDD's invariance properties and use conventional verification. The PBFDD state machine's sequential progression (S0 to S4, via Table A.14.2 transitions) benefits from CDD's invariant component design. Core shared components (e.g., LocationViewModel, LocationSaveService, LocationController) are rigorously verified once for their consistent adherence to TLE Rule principles. Consequently, each subsequent level's processing inherits this foundational correctness. Verification then shifts from re-validating component logic to focusing on conventional aspects: data integrity from the Locations dataset (See Table A.14.3) and precise state transition adherence, streamlining validation efforts.

The CDD refinement process adheres to FBFD methodology through these PBFDD-specific invariants:

- Termination Assurance
  - **Per-level refinement limit:**  $\text{refinement\_attempts}[j] \leq R_{\max} = 50$  (See Appendix A.14.3)
  - Error enforcement:
    - **PBFDD6:** Level 1-3 failure after 50 attempts
    - **PBFDD9:** Finalization failure
- State Machine Conformance
  - TLE state mappings:
    - **Continent:** S0 → Grandparent Level 1
    - **City:** S4 → Grandparent Level 5
  - Refinement triggers:
    - Shared component refactoring: PBFDD6 → S5 (See Table A.14.2)

- Parameter Invariance
  - Root-cause level:  $J_i=1$  (Grandparent Level)
  - Refinement scope:
    - $R_i = i - J_i + 1$  (Appendix A.14.3)
    - Example: Level 3 failure  $\rightarrow R_i=3$  (Levels 1-3)
- Complexity Bounds (See Table A.17.12)

**Table A.17.12.** Complexity bounds of the Pbfd MVP system across state machine parameters and refinement limits

Metric	Pbfd Value	Reference
Hierarchy Depth (L)	5	Table A.14.4
States ( $ Q $ )	9	Table A.14.1
Transitions ( $ δ $ )	10	Table A.14.2
Max Attempts Recorded	1 ( $\ll R_{\max}=50$ )	Appendix A.17.8

#### 4. Key Advantage

**Level-Wise Efficiency:** Shared components significantly reduce development effort, scaling exponentially or polynomially with hierarchy depth due to reuse across multiple tiers.

#### Appendix A.17.9. Backtracking to complete the application

This section is not part of the source code referenced in [29], as the Pbfd MVP does not fully implement the complete Pbfd specification. It is included here to provide a comprehensive explanation of the full specification.

#### Sequential Development Process

With the Continent Level fully implemented (Nodes 3.1–3.7 in Figure A.14.1), the Pbfd application uses backtracking to incrementally add missing child nodes under existing parents across subsequent levels to locations.json:

- Country Level Completion
  - **Existing Parents:** Added missing countries under continents (e.g., Japan under Asia)
  - **Validation:** Verified bitmask updates in the ContinentParent table (e.g., Asia's bitmask expanded to include Japan)
- State Level Expansion
  - **Existing Parents:** Added missing states under countries (e.g., Kanto under Japan)
  - **Testing:** Confirmed state bitmasks in the Asia table (e.g., Japan's Kanto = 1)
- County/City Integration
  - **Existing Parents:** Added counties under states (e.g., Tokyo Metropolis under Kanto) and cities under counties (e.g., Tokyo City)
  - **Regression Testing:** Ensured no conflicts with existing data (e.g., Maryland's counties unaffected)

#### State Machine Context

- **Current State:** S6 (Finalize\_All) (Table A.14.1)
- **TLE Structure:** Processes Child Levels 3-7 embedded in Grandparent Levels 1-5
- **Transition:** Finalizes processing, entering completion phase (S7) via Pbfd10
- **Failure Handling:** Exceeding  $R_{\max} = 50$  refinement attempts in S5 transitions to S8 (Validation\_Failure), terminating the workflow

#### Technical Notes

- **Hierarchical Integrity:** Maintains the TLE Rule (e.g., Asia  $\rightarrow$  Japan  $\rightarrow$  Kanto)
- **Testing:**
  - **Bitwise Validation:** Ensures new additions (e.g., Japan) do not corrupt existing selections (e.g., China)
  - **UI Consistency:** Confirms new nodes appear in workflows (Figure A.14.1)

#### Key Advantages

- **Hierarchical Flexibility:** The TLE Rule allows seamless addition of nodes at any level.
- **Efficiency:** Leveraging similarities between neighboring nodes (e.g., Maryland/Virginia counties) reduces redundant coding.

#### Appendix A.18. Comparative Analysis of PDFD and PBFDP MVP Implementations

This section presents a structured comparison between the MVP implementations of Primary Depth-First Development (PDFD) and Primary Breadth-First Development (PBFDP) methodologies. While both approaches share foundational principles—such as hierarchical data modeling, component-driven architecture, and hybrid methodological influences—they diverge significantly in execution strategy, database architecture, and scalability.

##### Appendix A.18.1. Foundational Similarities

- **Hierarchical Data Modeling:** Both approaches structure information using explicit parent–child relationships (e.g., Continent → Country → State). At a finer granularity, nodes are modeled as individual units in a directed graph, supporting localized validation and dependency tracking.
- **Component-Driven Architecture:** Modular MVC components (views, models, and controllers) promote reusability and maintenance across hierarchical levels.
- **User Interaction Workflows:** Dynamic forms and multi-level selection UIs are driven by back-end traversal logic.
- **Hybrid Methodology Integration:** Both leverage elements of DFD, BFD, and CDD to enable top-down progression, subtree resolution, and refinement cycles.

##### Appendix A.18.2. Key Differences in Methodological Strategy

Table A.18.1 contrasts the core methodological strategies of PDFD and PBFDP, highlighting their differences in traversal logic, structural optimizations, and enabling technologies.

**Table A.18.1.** Methodological distinctions between PDFD and PBFDP

Aspect	PDFD	PBFDP
Core Approach	Hybrid Depth-First: Vertical slice traversal with concurrent processing of same-level nodes	Hybrid Breadth-First: Pattern-grouped traversal with selective vertical descent
Key Strategy	Sequential subtrees with bounded vertical depth	Pattern compaction and horizontal aggregation using TLE and bitmasks
Key Technology	Feature-based selective traversal (e.g., BF-by-Two)	Bitmask encoding and Three-Level Encapsulation (TLE)

##### Appendix A.18.3. Graph Traversal Workflow

Table A.18.2 compares the traversal patterns of PDFD and PBFDP, focusing on how nodes are selected, validated, and refined in each methodology.

**Table A.18.2.** Graph traversal strategies in PDFD and PBFDP

Aspect	PDFD	PBFDP
Node Selection	Feature-selected nodes per level	Pattern-based node groups
Progression	Vertical-first traversal	Horizontal-first compaction followed by vertical descent
Refinement Scope	Narrow, vertical chains	Broad pattern groups spanning multiple levels via TLE

##### Appendix A.18.4. Pilot Tunnelling Strategies

Drawing an analogy to pilot tunneling in engineering [143,144], Table A.18.3 illustrates how each method performs risk-aware preliminary development to detect and resolve structural issues.

**Table A.18.3.** Pilot tunneling strategies in PDFD and PBFD

Aspect	PDFD	PBFD
Tunneling Analogy	Small pilot tunnel → feature-driven scaling	Large pilot tunnel → pattern-driven scaling
Focus	Vertical validation with minimal breadth	Horizontal breadth with controlled depth
Efficiency Driver	Early risk detection	Early structural optimization via TLE patterns
Scale	Suitable for small to mid-sized systems	Designed for enterprise-grade and distributed systems

#### Appendix A.18.5. Development Workflow

Table A.18.4 details the contrasting development workflows of the two MVPs, including traversal strategies, refinement cycles, and structural encapsulation.

**Table A.18.4.** Development workflow characteristics in PDFD and PBFD

Aspect	PDFD	PBFD
Core Workflow Pattern	Depth-first exploration with subtree completion	Breadth-first pattern grouping followed by selective descent
Branching Strategy	Narrow branching (few nodes per level)	Wide branching across three-level spans (grandparent-child)
CDD Iterations	Higher (3 iterations during refinement)	Lower (pre-optimized structure reduces iteration count to 1)

#### Appendix A.18.6. Database Architecture

Table A.18.5 outlines the structural and architectural distinctions in the database schemas of PDFD and PBFD, focusing on lookup tables, query complexity, and relational encoding.

**Table A.18.5.** Comparison of database schema design between PDFD and PBFD

Aspect	PDFD	PBFD
Lookup Table	Multiple normalized tables with foreign key relationships	Single adjacency-list table (e.g., Locations table in Table A.14.3)
Base Table	Per-level normalized relational tables	Per-grandparent dynamic tables using TLE
Query Complexity	JOIN-heavy SQL queries	Bitwise queries within denormalized bitmask tables

#### Appendix A.18.7. Data Storage Models

Table A.18.6 compares the storage efficiency and scalability mechanisms used in each methodology's data representation.

**Table A.18.6.** Data storage model comparison for PDFD and PBFD

Aspect	PDFD	PBFD
Data Model	Row-based (1 record per selected node)	Bitmask-based (1 row encodes multiple selections)
Storage Efficiency	Higher overhead due to repeated foreign keys	Compact, bit-level efficiency
Scalability	Limited by relational constraints and locking	Optimized for horizontal scaling and parallel operations

#### Appendix A.18.8. Relational Table Structures

Table A.18.7 contrasts how hierarchical tables are organized, indexed, and accessed in PDFD versus PBFD, emphasizing schema scalability and join complexity.

**Table A.18.7.** Structural comparison of database tables in PDFD and PBF

Aspect	PDFD	PBFD
Schema Design	Dedicated table per hierarchical level	Per-grandparent table generated dynamically via TLE
Scalability	Constrained by row growth and indexing	Scales through distributed grandparent tables
Join Complexity	Multi-table joins for full traversal	Joins only between grandparent tables and the global Person table

#### Appendix A.18.9. MVC Architecture

Table A.18.8 presents the differences in software architecture, focusing on how MVC components are structured and reused across levels.

**Table A.18.8.** MVC architectural comparison of PDFD and PBF

Aspect	PDFD	PBFD
Model	Static models per level (e.g., CountryModel, StateModel)	Unified dynamic view model (LocationViewModel) derived from metadata
View	Level-specific Razor views	Shared Razor view for all hierarchical levels
Controller	Multiple specialized controllers	Single reusable controller (e.g., LocationController)

#### Appendix A.18.10. Performance & Scalability

Table A.18.9 summarizes the runtime characteristics of each approach, including query efficiency, storage cost, and readiness for distributed environments.

**Table A.18.9.** Performance and scalability characteristics of PDFD and PBF

Aspect	PDFD	PBFD
Query Speed	Slower due to multi-join queries ( $O(n)$ )	Faster using in-place bitwise operations ( $O(1)$ )
Write Efficiency	Multiple-row inserts/updates ( $O(n)$ )	Single-row bitmask updates ( $O(1)$ )
Storage Footprint	Higher due to normalized rows	Lower due to compact binary encoding
Distributed Support	Challenging due to ACID across tables	Optimized for horizontal sharding via table-level separation

#### Appendix A.18.11. Comparative Strengths and Tradeoffs

Table A.18.10 presents a summary-level tradeoff analysis of PDFD and PBF, encapsulating key strengths and limitations.

**Table A.18.10.** Summary of benefits and limitations of PDFD and PBF methodologies

Approach	Strengths	Limitations
PDFD	Intuitive for traditional developers Simpler debugging workflows	Inefficient for large-scale graphs High storage/query costs
PBFD	High performance and scalability Optimized for modern cloud systems	Higher implementation complexity Limited mainstream tooling support

#### Appendix A.18.12. Example Workflows

##### PDFD (Feature-Driven Traversal)

- **Level 1:** Continents → North America, Asia
- **Level 2:** Countries → USA, Canada
- **Level 3:** States → Maryland, Virginia

**Strategy:** Controlled selection and deselection of hierarchical feature nodes across levels for depth management, ensuring comprehensive combinatorial coverage and uninterrupted user progression.

**PBFD (Pattern-Driven Compaction)**

- **Level 3:** Compact all continents into bitmasks (e.g., `00010101` for North America, Asia, Europe)
  - **Level 4:** Compact countries under selected continents (e.g., North America = `00000011` for USA + Canada)
  - **Level 5:** Compact states under selected countries (e.g., USA = `264,192` for Maryland + Virginia)
- Strategy:** Full bitmask compaction within a TLE table spanning three levels

Appendix A.18.13. Methodology Suitability Guidelines

Choose PDFD or PBFD based on project scale, performance goals, and team capabilities.

- Use PDFD for small-to-medium systems with limited depth, or where team familiarity and debugging clarity are essential
- Use PBFD for complex, deeply nested systems requiring performance, compact storage, and horizontal scalability

Appendix A.19. Real-World Structural Workflow Mermaid Code

```

graph TD
  %% Layer 1 (Single Root)
  N1_1[N1_1]

  %% Layer 2
  N1_1 --> N2_1[N2_1]; N1_1 --> N2_2[N2_2]; N1_1 --> N2_3[N2_3]

  %% Layer 3
  N2_1 --> N3_1[N3_1]; N2_1 --> N3_2[N3_2]; N2_2 --> N3_1; N2_2 --> N3_3[N3_3]; N2_3 --> N3_2; N2_3 --> N3_4[N3_4]

  %% Layer 4
  N3_1 --> N4_1[N4_1]; N3_1 --> N4_2[N4_2]; N3_2 --> N4_1; N3_2 --> N4_3[N4_3]; N3_3 --> N4_2; N3_4 --> N4_4[N4_4]

  %% Layer 5
  N4_1 --> N5_1[N5_1]; N4_1 --> N5_2[N5_2]; N4_2 --> N5_1; N4_2 --> N5_3[N5_3]; N4_3 --> N5_2; N4_4 --> N5_4[N5_4]

  %% Layer 6
  N5_1 --> N6_1[N6_1]; N5_1 --> N6_2[N6_2]; N5_2 --> N6_1; N5_3 --> N6_2; N5_3 --> N6_3[N6_3]; N5_4 --> N6_3

  %% Layer 7
  N6_1 --> N7_1[N7_1]; N6_1 --> N7_2[N7_2]; N6_2 --> N7_1; N6_2 --> N7_3[N7_3]; N6_3 --> N7_2; N6_3 --> N7_4[N7_4]

  %% Layer 8 (Added to meet 8-level requirement)
  N7_1 --> N8_1[N8_1]; N7_2 --> N8_2[N8_2]; N7_3 --> N8_3[N8_3]; N7_4 --> N8_4[N8_4]

  %% Add data labels as annotations
  N1_1 -.-> D1[Claimant]; N2_1 -.-> D2[Incident Location]; N3_1 -.-> D3[Reasons at the Location]; N4_1 -.-> D4[Claimant Organization]; N5_1 -.-> D5[Claimant Role in the Organization];

```

N6\_1 --> D6[Claimant Employment Type]; N7\_1 --> D7[Claimant Employment Period]; N8\_1 --> D8[Specific Period Metric]

```
%% Style the nodes
classDef mainPath fill:#ffcdd2,stroke:#d32f2f,stroke-width:2px,color:#000
classDef dummyNodes fill:#e8f5e8,stroke:#4caf50,stroke-width:1px,color:#666
classDef dataLabels fill:#e3f2fd,stroke:#1976d2,stroke-width:1px,color:#000

class N1_1,N2_1,N3_1,N4_1,N5_1,N6_1,N7_1,N8_1 mainPath
class N2_2,N2_3,N3_2,N3_3,N3_4,N4_2,N4_3,N4_4,N5_2,N5_3,
N5_4,N6_2,N6_3,N7_2,N7_3,N7_4,N8_2,N8_3,N8_4 dummyNodes
class D1,D2,D3,D4,D5,D6,D7,D8 dataLabels
```

#### Appendix A.20. Observational Case Study on Development Effort

**Reviewer Takeaway:** In a longitudinal case study, the Pbfd methodology demonstrated 9–20× reductions in development effort for a complex hierarchical system. Both ratios represent conservative estimates: the 20× comparison involves incomplete OmniScript implementation, while the 9× comparison involved a developer with 25+ years of relational expertise versus concurrent Pbfd invention experience.

##### Appendix A.20.1. Methodological Context and Related Work

Evaluating development efficiency in real-world industrial settings presents significant methodological challenges. Rather than relying on randomized controlled trials—which are rarely feasible for complex software projects due to organizational, ethical, and logistical constraints—empirical software engineering frequently adopts observational, case-based, and design-science methods [97,105,145] to achieve ecological validity. While controlled experiments play a role in validating specific methodological components, they are not the primary vehicle for assessing development practices in production environments.

This appendix presents a longitudinal observational case study (aligned with Table 55) comparing development effort across three implementation strategies—Pbfd, traditional relational schema, and Salesforce OmniScript. Our pragmatic methodology draws from project management artifacts (e.g., Jira, time-tracking systems) and delivered functionality to estimate effort and scope. While less controlled than laboratory experiments, this approach provides high ecological validity and reflects the practical constraints of industrial software development [146].

##### Experimental Design Framework

- **Unit of Comparison:** Development methodology (Pbfd vs. relational vs. OmniScript)
- **Evaluation Focus:** Person-month effort, calendar duration, scope completeness
- **Controlled Variables:** Shared enterprise context, comparable functional requirements, consistent audit logging
- **Independent Variable:** Implementation methodology and platform
- **Study Type:** Longitudinal observational case study with embedded effort estimation

This design emphasizes ecological validity and methodological transparency. Our analysis explicitly acknowledges inherent challenges—such as normalizing effort metrics, accounting for developer expertise [147,148], and comparing projects with differing completion states—and employs conservative estimations to mitigate bias. We therefore interpret the large magnitude of observed differences as a robust indicator of methodological efficiency worthy of further investigation.

##### Appendix A.20.2. Project Characteristics Overview

Table A.20.1 summarizes the scope, methodology, and timeframes of each development effort. The projects were conducted at different times with different primary objectives, which must be considered when interpreting the observational data. Effort A and B involved direct contributions from the author as primary developer, while managerial oversight for Effort B and C was provided by two individuals acknowledged in the Acknowledgements section. All efforts were led by experts.

**Table A.20.1.** Project characteristics for three implementation strategies

Implementation	Methodology/Platform	Team Size	Time Required (Calendar Months)	Year	Scope Delivered
Effort A (PBF Enterprise)	PBFD, bitmask, TLE	1 primary developer	1 (Jun–Jul)	2016	Full System (Production)
Effort B (Relational Port)	Traditional relational schema (SQL Server)	2 part-time developers (0.35 & 0.15 FTE)	9	2021–2022	DB schema and data migration (No UI/Middleware)
Effort C (Salesforce)	Salesforce OmniScript	7 developers	24	2022–2024	UI + logic (undeployed)

All "Time Required" figures exclude separate testing and deployment phases. Effort A's integrated development, however, inherently minimized distinct testing and deployment, allowing rapid production transition.

- **For Effort A:** The "1 primary developer" refers to the PBFD inventor. Two auxiliary developers contributed non-overlapping, sequential efforts (including code development, validation, and training) spanning approximately one to two weeks. The primary developer estimated that replicating this auxiliary work would have required only 1-2 additional days. Because this effort was minimal, non-overlapping, and not part of the core PBFD development activity, it is excluded from the primary metrics. It is a critical threat to validity that the principal developer was also the methodology inventor, a known confound in productivity studies [147,148]. We acknowledge this limits the ability to draw definitive causal inference solely on the methodology.
- **For Effort B:** The same individual who was the primary developer for Effort A contributed 0.35 FTE to Effort B.
- **For Effort C:** Involved a team of 7 developers with varying engagement: 2 core developers (each at ~0.3 FTE) and 5 nominal developers (contributors with assigned roles but limited, sustained effort at ~0.05 FTE each), totaling an estimated 20.4 FTE-months over 24 calendar months. Effort C is included to illustrate platform-specific development challenges and provide context for comparative effort estimation, despite its incomplete status. This effort remained incomplete and undeployed, making direct quantitative comparison challenging.

**Observation on Calendar Time and Person-Month Alignment:** The alignment between calendar time and calculated FTE-months is a key indicator of sustained, continuous development effort. For Effort A, 1 calendar month equated to 1 FTE-month for the primary developer. For Effort C, the 24 calendar months closely approximate the 20.4 FTE-months, accounting for the distributed team structure. This correlation, especially for critical-path foundational work, supports the accuracy of the effort estimation from a project management perspective. The significant discrepancy for Effort B (9 calendar months vs. 4.5 FTE-months) is consistent with its part-time, lower-priority nature.

### Appendix A.20.3. Scope of Delivered Functionality

This section outlines the core functional modules and their delivery status. The varying degrees of completion are a fundamental aspect of this observational comparison.

#### Core Functional Modules:

- Hierarchical question flow (up to 8 hierarchical levels)
- Conditional branching logic with enable/disable rules
- Diverse input types: checkboxes, multi-select dropdowns, text fields

- Real-time validation and navigation
- Secure submission pipeline with persistence and audit logging.
- Storage Optimization

**Table A.20.2.** Key Aspects of Functional Module Delivery across three implementation strategies, showing production readiness and architecture-level support

Key Aspect	Effort A (PBFD)	Effort B (Relational Port)	Effort C (Salesforce OmniScript)
End-to-End Claim Form	☑ Production	✗ (DB schema only, no UI/middleware)	⚠ Incomplete
Full UI/UX Integration	☑ Production	✗ (UI layer not implemented)	⚠ Incomplete
Question Hierarchy Support (8 levels)	☑ (Native PBFD bitmasking)	☑ (via complex SQL JOINS)	⚠ Incomplete
Dynamic Flow + Conditionals	☑ Production	☑ (Logic in DB)	⚠ Incomplete
Storage Optimization	☑ (bitmask encoding)	✗ (normalized schema, higher redundancy)	✗ (Platform-managed)
Deployment Readiness	☑ (in production since 2016)	✗ (no front-end, not deployable)	⚠ In progress (not deployed)

#### Appendix A.20.4. Observed Efficiency Comparison

This analysis provides calculated ratios based on project data. These figures represent observed differences rather than results from a controlled experiment and must be interpreted with caution due to the limitations outlined in A.20.5. Our estimation approach is intentionally conservative to mitigate threats to validity.

**Table A.20.3.** Calculated development ratios

Comparison	Observed Ratio (Calculation)	Context and Justification
PBFD vs. Relational Port (A vs B)	~9x ( (4.5 FTE-months * 2) / 1 FTE-month )	Full-stack system (A: 1 FTE-month) vs. backend-only implementation (B: 4.5 FTE-months). A multiplier of 2x was applied to Effort B's DB effort to estimate the missing UI/middleware effort. This multiplier is derived from organizational historical data for projects of similar logic complexity and aligns with conservative expert judgment in software project estimation [149]. This estimates a total ~9 FTE-month effort for a full relational stack.
PBFD vs. OmniScript (A vs C)	~20x (20.4 FTE-months / 1 FTE-month)	Full-stack system (A: 1 FTE-month) vs. incomplete UI+logic (C: ≥20.4 estimated FTE-months). The credibility of this FTE-month estimate is supported by its close alignment with the 24-month calendar timeline (see Section A.20.2). Effort C's incomplete status suggests the actual ratio upon completion would be higher. This comparison is primarily illustrative of the platform-specific challenges encountered.

#### Appendix A.20.5. Summary of Threats to Validity

This section details threats to validity specific to the comparisons made in this appendix. Section 5 of the main text addresses high-level, study-wide threats (e.g., generalizability, observational design), while the appendices contain the specific, methodological threats related to each case study and data source.

##### Construct Validity

Effort measurement is inconsistent across projects (e.g., auxiliary effort excluded in A, all developer time included in C). The "person-month" metric may not reflect effort intensity [146]. The multiplier used for Effort B's UI, while based on historical data, remains an estimation [149].

##### Internal Validity (Mixed Threats)

- **Developer Expertise Variation:** While all implementations were led by expert developers, skill levels and methodology familiarity vary across individuals. Development of both PBF and the relational baseline was led by the methodology's inventor, while OmniScript implementations were carried out by other expert developers, some of whom possessed decades of development experience.
- **OmniScript Incomplete Implementation:** The OmniScript comparison measures effort at an incomplete state, while PBF reached full production deployment. This introduces scope normalization challenges.
- **Same-Developer Learning Asymmetry (PBF vs. Relational):** The same developer led both implementations, possessing 25+ years of relational database expertise, in contrast to concurrent learning while inventing PBF, which created an expertise asymmetry favoring relational approaches.
- **Temporal Span:** Implementations span 2016–2024, introducing potential confounds from evolving tools and practices.
- **Method Invention:** The inventor of PBF/PDF led the PBF implementation, which may introduce bias toward more efficient realization of the methodology. This threat is mitigated by the conservative biases described above.

### External Validity

Findings are from a single case study. Generalizability is limited and requires further replication [97].

### Conclusion Validity

The large magnitude of the observed ratios (~9×, ~20×) persists despite threats to internal validity that bias against PBF. The 20× comparison involves incomplete OmniScript effort (conservative), while the 9× comparison involves a developer with substantially more relational expertise than PBF expertise (conservative).

While these threats prevent definitive causal attribution to methodology alone, the consistency of large efficiency advantages across multiple independent comparisons—each biased conservatively—provides strong evidence that PBF offers substantial methodological benefits when applied by competent practitioners. The results establish a credible lower bound for PBF's efficiency potential rather than precise point estimates [147,148].

### *Appendix A.21. A Longitudinal Performance Evaluation of PBF Versus Traditional Relational Approaches*

**Reviewer Takeaway:** Operating on identical infrastructure, the PBF-based component processed requests 7.6–8.5× faster than traditional relational modules. Tail latency was dramatically reduced, confirming PBF's efficiency for hierarchical workloads under realistic enterprise conditions and sustained production traffic.

#### Appendix A.21.1. Methodology

This analysis employs a longitudinal quasi-experimental study embedded within a production case study [97] to compare the runtime performance of the Primary Breadth-First Development (PBF) methodology against an aggregate baseline of traditional relational patterns. The study spans nearly eight years of continuous production operation (2016 - 2024).

Although embedded in a production case study, the system architecture provided quasi-experimental control over key confounding variables. PBF and traditional modules were implemented within the same ASP.NET MVC solution (Framework v3.5–4.8), compiled into a single assembly, and deployed on the same IIS and SQL Server instances. Both operated concurrently as part of the same running application process, thereby ensuring identical infrastructure, runtime environment, and production traffic.

#### Controlled variables

- **Hardware & OS:** Identical CPU, memory, storage, and Windows Server instance.

- **Database Server:** Shared SQL Server instance with identical configuration, buffer pools, and query execution resources.
- **Network:** No inter-module latency; all communication occurred over the same internal path.
- **Load & Time:** Both modules operated concurrently under the same production traffic and infrastructure conditions, though workload characteristics varied by controller and logic path.

#### Workload definition

- **PBFD operations:** A scoped, read-optimized workload, identified in the audit log as ControllerName = 'MainController' AND ActionName NOT IN ('UpdateX','DeleteX','SaveX'). These operations typically involve multi-level hierarchical navigation and complex pattern matching.
- **Traditional operations:** Traditional operations represent a heterogeneous mix of CRUD operations, reporting queries, and business logic processing across approximately 11 controllers. While not functionally identical to PBFD's read-optimized scope, this aggregate baseline reflects the realistic complexity of enterprise systems against which PBFD must perform.

#### Data collection and filtering

Execution logs were retrieved from the production audit log (AuditEventLog). Events with Duration  $\leq 10$  ms were excluded to minimize noise from lightweight health checks and infrastructure-level overhead. No application-level caching was employed for either module during the observation period, ensuring that measured latencies reflect raw query and processing performance.

#### Analysis metrics

Following established performance guidelines [150][151], latency distributions were computed using continuous percentiles (PERCENTILE\_CONT in SQL Server):

- **P5 (5th percentile):** Infrastructure/middleware floor
- **P50 (median):** Typical user experience
- **P95 (95th percentile):** Tail latency, critical for scalability
- **Average (mean):** Reported for completeness but interpreted with caution due to skew

This methodology integrates the ecological validity of a longitudinal observational study [97] with the internal validity of quasi-experimental comparison, enabled by infrastructure co-location, concurrent execution, and shared production traffic. This evaluation corresponds to the "longitudinal quasi-experimental comparison" design dimension in Table 55, with component architecture and query logic as the independent variable.

#### Appendix A.21.2. Experimental Environment

The platform underwent scheduled upgrades during the study, migrating from Windows Server 2008/SQL Server 2008 R2 to newer environments. For a significant portion of the observation period, including its final configuration, the system operated on infrastructure comparable to the following.

**Table A.21.1.** Example Experimental Environment Specification (Final State)

Component	Specification
Application Framework	ASP.NET MVC on .NET Framework 4.8
Web Server	IIS 10.0 on Windows Server 2016 Std.
Database Server	Microsoft SQL Server 2016
Web Server CPU	Quad-Core, 2.6 GHz (Model 55)
Database Server CPU	8-Core, 2.6 GHz (Model 55)
Web Server RAM	16 GB
Database Server RAM	99 GB
Network	vmxnet3 Ethernet Adapter (~4 Gb/s)
Storage	SSD-backed (RAID configuration)

PBFD and traditional components were always migrated together during upgrades, ensuring identical hardware/software configurations at every stage. This co-location across layers preserved the validity of the relative performance comparison.

#### Appendix A.21.3. SQL Query

```
-- PBFD (System A)
WITH PBFD_Metrics AS (
  SELECT
    PERCENTILE_CONT(0.05) WITHIN GROUP (ORDER BY Duration) OVER () AS P5_A,
    PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY Duration) OVER () AS P50_A,
    PERCENTILE_CONT(0.95) WITHIN GROUP (ORDER BY Duration) OVER () AS P95_A,
    AVG(Duration) OVER () AS Avg_A
  FROM AuditEventLog
  WHERE ControllerName = 'MainController'
    AND ActionName NOT IN ('UpdateX', 'DeleteX', 'SaveX')
    AND Duration > 10
),

-- Traditional Method (System B)
Traditional_Metrics AS (
  SELECT
    PERCENTILE_CONT(0.05) WITHIN GROUP (ORDER BY Duration) OVER () AS P5_B,
    PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY Duration) OVER () AS P50_B,
    PERCENTILE_CONT(0.95) WITHIN GROUP (ORDER BY Duration) OVER () AS P95_B,
    AVG(Duration) OVER () AS Avg_B
  FROM AuditEventLog
  WHERE NOT (
    ControllerName = 'MainController'
    AND ActionName NOT IN ('UpdateX', 'DeleteX', 'SaveX')
  )
  AND Duration > 10
)

-- Comparison
SELECT DISTINCT
  P5_A, P50_A, P95_A, Avg_A,
  P5_B, P50_B, P95_B, Avg_B,
  P5_B / P5_A AS P5_Ratio,
  P50_B / P50_A AS Median_Ratio,
  P95_B / P95_A AS P95_Ratio,
  Avg_B / Avg_A AS Avg_Ratio
FROM PBFD_Metrics, Traditional_Metrics;
```

#### Appendix A.21.4. Results

The dataset includes 46,739,051 logged events. PBFD operations comprised 1,100,375 events (2.4% of total), while traditional operations comprised 45,638,676 events (97.6%).

**Table A.21.2.** Runtime latency comparison (ms) between PBFD and traditional aggregates

Metric (ms)	P5	P50	P95	Average
PBFD	16	47	406	118.46

Traditional	16	359	3469	881.49
(Trad/PBFD)	1	7.64	8.54	7.44

**Notes:**

- A ratio of 1.0 at P5 indicates both methodologies hit the same infrastructural latency floor, confirming that performance differences are due to application- and database-level processing.
- The consistency of performance ratios across all percentiles (P50, P95, average) and the large sample size (46+ million events) provide strong evidence for the observed performance differences, though formal statistical testing was not performed given the complete population data.

## Appendix A.21.5. Key Findings

- **Median Performance (P50):** PBFD processed requests 7.64× faster than the traditional aggregate, improving efficiency for typical operations.
- **Tail Latency (P95):** PBFD reduced slow-response outliers by 8.54×, showing superior scalability under load. In deeply-nested architectures, high tail latencies can cascade and become the dominant factor in overall user-perceived performance, making their mitigation a critical engineering goal [152].
- **Average Latency:** PBFD achieved a 7.44× improvement, confirming consistent performance gains.
- **Performance Floor (P5):** Both shared a 16 ms lower bound, reflecting a common infrastructure/middleware baseline.
- **Effect Size:** The 7–8× performance improvement represents a large effect size by conventional standards in software performance evaluation, particularly notable given that both systems operated under identical environmental constraints.

## Appendix A.21.6. Threats to Validity

- **Construct Validity (Workload heterogeneity):** The traditional baseline encompassed ~11 controllers with diverse workloads, not all directly comparable to PBFD's read-optimized scope. This heterogeneity—which includes simpler operations alongside complex ones—may understate PBFD's efficiency but provides a realistic enterprise baseline. Reported ratios should be interpreted as conservative lower-bound estimates.
- **Internal Validity (Implementation factors):** While infrastructure was controlled, minor differences in query patterns or transient load conditions may exist. The long (8-year) observation window helps mitigate transient effects. Furthermore, the use of percentiles over means reduces the impact of outlier events on the overall results [150][151].
- **External Validity (Generalizability):** Results stem from a single large-scale enterprise deployment. While ecologically valid [97], replication in other environments is necessary to establish generalizability.

## Appendix A.21.7. Conclusion

This longitudinal case study, conducted under tightly controlled production conditions, shows that PBFD consistently achieved 7–8× latency reductions across median, tail, and average measures compared to traditional relational approaches. By co-locating both systems on identical infrastructure, these improvements can be attributed directly to the underlying methodology rather than environmental factors.

PBFD's demonstrated efficiency for read-heavy hierarchical workloads positions it as a scalable, latency-reducing alternative for enterprise systems.

*Appendix A.22. A Comparative Analysis of Storage Efficiency: PBFD vs. Traditional Relational Deployment*

**Reviewer Takeaway:** PBFDF achieves 11.7× storage reduction and operational performance gains through TLE-based bitmask encoding, validated via a controlled schema-level experiment.

#### Appendix A.22.1. Methodology

This appendix presents a controlled schema-level experiment embedded within a production case study [145], comparing the storage efficiency of the Primary Breadth-First Development (PBFDF) methodology against a traditional Third Normal Form (3NF) relational schema. The analysis uses production data from a long-term deployment, following the same longitudinal case study approach outlined in Appendix A.21.

PBFDF leverages Three-Level Encapsulation (TLE) for hierarchical data management; its formal model is described in Section 4.2. This experiment isolates schema structure as the independent variable, evaluating how TLE's bitmask encoding and PBFDF's schema design contribute to operational and storage efficiency compared to conventional relational approaches.

#### Experimental Design Context (aligned with Table 55)

- **Unit of Comparison:** Two alternative schema architectures instantiated over the same dataset:
  - Traditional 3NF (multi-table, join-based)
  - PBFDF/TLE (wide-form, bitmask-encoded, minimal table count)
- **Evaluation Focus:**
  - Structural reduction (tables, rows, junctions, indexing strategy)
  - Physical storage usage (reserved space, index size, unused space, row volume)
- **Controlled Variables:**
  - Same DBMS
  - Same hardware and configuration
  - Same source dataset used for schema population
  - Same total record volume mapped according to each schema's structure
- **Independent Variable:** Schema design paradigm (join-centric 3NF vs. compact PBFDF/TLE)
- **Data Source Handling:** The dataset is identical in origin, but table counts and row distributions differ due to schema architecture (e.g., 4.7M rows normalized vs. 170K rows in PBFDF per Table A.22.2)
- **Study Type:** Controlled schema-level experiment focused on structural and storage efficiency

#### Experimental Environment

The storage analysis was conducted on the system's final, stable configuration: a Microsoft SQL Server 2016 instance running on Windows Server 2016 Standard. Both schemas operated on the same shared database instance, ensuring that observed differences are attributable solely to schema design—not to hardware, storage subsystem, or platform configuration (see A.21).

#### Schema Design Comparison

The fundamental architectural differences between the two approaches are summarized in Table A.22.1. PBFDF's use of bitmask encoding for hierarchical relationships, as formalized in Section 4.2, is the primary differentiator.

**Table A.22.1.** Fundamental Schema Architecture Comparison

Feature	Traditional 3NF	PBFDF
Core Transactional Tables	6	2 (Wide-form, bitmask-encoded)
Explicit Junction Tables	7	0
Indexing Strategy	Per-entity and per-relationship (join-focused)	Minimal (payload- and query-focused)

**Note:** PBFDF's bitmask encoding mechanism and table layout are formalized in Section 4, linking storage design to the formal methodology.

#### Functional Equivalence

Both implementations were rigorously designed to support identical production requirements:

- Complex hierarchical structures (8-level nested claims).
- Dynamic validation and conditional branching logic.
- Comprehensive, timestamped audit logging and versioning.

#### Data Collection Protocol

Storage metrics were collected following a reproducible protocol to ensure accuracy and minimize measurement bias:

- **Tool:** `sp_spaceused` executed via `sp_msforeachtable` across all user-defined tables [153]
- **Timing:** Immediately after scheduled index maintenance to standardize fragmentation
- **Scope:** User-defined tables and indexes only; system metadata excluded
- **Dataset:** 8 years of production data (Traditional: 4.7M rows across all tables; PBF: 170K rows in core tables).

#### Reproducible T-SQL

-- Reproducible T-SQL

```
CREATE TABLE #StorageMetrics (
  TableName NVARCHAR(128),
  Rows BIGINT,
  ReservedKB NVARCHAR(50),
  DataKB NVARCHAR(50),
  IndexKB NVARCHAR(50),
  UnusedKB NVARCHAR(50)
);
INSERT INTO #StorageMetrics EXEC sp_msforeachtable 'EXEC sp_spaceused "?";'
SELECT * FROM #StorageMetrics ORDER BY ReservedKB DESC;
```

#### Appendix A.22.2. Results

Aggregated storage usage metrics, presented in Table A.22.2, demonstrate significant efficiency gains from the PBF architecture.

**Table A.22.2.** Aggregated Storage Usage Metrics

Metric	Traditional	PBF	Ratio (Trad/PBF)
Core Tables	6	2	3.0×
Total Rows	4.7M	170K	27.6×
Reserved Space (KB)	658,768	56,168	11.7×
Index Size (KB)	37,040	432	85.7×
Unused Space (KB)	5,448	48	113.5×

**Note:** Ratios reflect core transactional tables only; auxiliary lookup tables excluded.

#### Appendix A.22.3. Key Findings

- **Structural Simplification:** PBF's schema required 3× fewer core tables and eliminated all 7 junction tables, drastically simplifying the data model and query execution paths.
- **Storage Efficiency:** PBF achieved 11.7× reduction in reserved space, 85.7× reduction in index overhead, and 113.5× improvement in page utilization.
- **Operational Performance Linkage:** The drastic reduction in row count and index size directly lowers I/O pressure and improves buffer pool cache locality. This optimized data footprint complements bitmask encoding as a key contributor to the 7–8× faster query performance documented in Appendix A.21, as query processing involves scanning fewer data pages.
- **Methodological Traceability:** This experiment isolates schema structure as the independent variable, aligning with the controlled design dimensions in Table 55.
- **Formal Integration:** PBF's schema design is consistent with the TLE model in Section 4.2, linking empirical outcomes to theoretical guarantees.

#### Appendix A.22.4. Threats to Validity

- **Construct Validity:** Metrics focus exclusively on user data storage. System metadata is excluded. Lookup tables are omitted from comparison ratios due to their optional role in downstream functionality and inconsistent presence across implementations.
- **Internal Validity:** Traditional schema may include legacy optimizations. Post-maintenance measurements minimize index fragmentation bias.
- **External Validity:** The results are most directly applicable to systems managing complex hierarchical data. The efficiency gains for flat, transactional data may differ. Furthermore, the absolute savings are influenced by SQL Server's storage engine (e.g., 8KB page size), though the relative gains are expected to hold across relational platforms.

#### Appendix A.22.5. Conclusion

This controlled schema-level experiment provides strong empirical evidence that the PBF method—via its TLE-based bitmask encoding—achieves order-of-magnitude storage efficiency improvements for hierarchical workloads.

By achieving an 11.7× storage reduction (a 91.5% decrease), the experiment grounds the theoretical model in production-scale data. The elimination of all junction tables and the 85.7× reduction in index overhead directly reduce I/O pressure and improve cache locality, contributing to the query performance gains reported in Appendix A.21.

Overall, this experiment effectively links the formal PBF methodology to its industrial implementation, demonstrating that PBF's architectural choices provide predictable and substantial advantages for managing complex hierarchical data in enterprise relational systems.

## References

1. Skillcrush. 8 Full-Stack Development Trends to Look Out for in 2025. Skillcrush 2025. <https://skillcrush.com/blog/full-stack-developer-trends/> (accessed May 15, 2025).
2. GeeksforGeeks. Top 10 Full Stack Development Trends in 2025. GeeksforGeeks 2025. <https://www.geeksforgeeks.org/blogs/full-stack-development-trends/> (accessed May 15, 2025).
3. IBM. IBM Full Stack Software Developer Professional Certificate. Coursera 2024. <https://www.coursera.org/professional-certificates/ibm-full-stack-cloud-developer> (accessed May 15, 2025).
4. Talent500. Full Stack Developer Roadmap 2025: Skills & Guide. Talent500 2025. <https://talent500.com/blog/full-stack-developer-roadmap-2025> (accessed May 15, 2025).
5. Stack Overflow. Developer Survey 2025. Stack Overflow 2025. <https://survey.stackoverflow.co/2025> (accessed May 15, 2025).
6. Beck, K.; Beedle, M.; van Bennekum, A.; Cockburn, A.; Cunningham, W.; Fowler, M.; et al. Manifesto for Agile Software Development. Agile Alliance 2001. <https://agilemanifesto.org> (accessed May 15, 2025).
7. Tsilionis, K.; Ishchenko, V.; Wautelet, Y.; Simonofski, A. Scaling Agility in Large Software Development Projects: A Systematic Literature Review. In: Visvizi, A.; Troisi, O.; Corvello, V., Eds.; Research and Innovation Forum 2023; Springer Proceedings in Complexity; Springer: Cham, 2024; pp. 1-15.
8. Santos, P.d.; de Carvalho, M.M. Exploring the challenges and benefits for scaling agile project management to large projects: a review. *Require. Eng.* 2022, 27, 117-134.
9. Stojanovic, Z.; Dahanayake, A.; Sol, H.G. Modeling and Architectural Design in Agile Development Methodologies. In Proceedings of the 8th CAISE/IFIP8.1 International Workshop on Evaluation Methods in System Analysis and Design; Velden, M., Ed.; 2003; pp. 180-189.
10. Mognon, F.; C. Stadzisz, P. Modeling in Agile Software Development: A Systematic Literature Review. In Agile Methods; Silva da Silva, T., Estácio, B., Kroll, J., Mantovani Fontana, R., Eds.; Communications in Computer and Information Science, Vol. 680; Springer: Cham, 2017; pp. 1-15.
11. Northwood, C. *The Full Stack Developer: Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer*; Apress: New York, 2018.

12. Zammetti, F. *Modern Full-Stack Development: Using TypeScript, React, Node.js, Webpack, Python, Django, and Docker*; Apress: New York, 2022.
13. Mkaouer, W.; Kessentini, M.; Sahraoui, H.; Bechikh, S.; Deb, K. Many-objective software modularization using NSGA-III. *ACM Trans. Softw. Eng. Method.* 2015, 24, 1-45.
14. Recker, J. Opportunities and constraints: the current struggle with BPMN. *Bus. Process Manag. J.* 2010, 16, 181-201.
15. Kandogan, E.; Kraska, T.; Li, F.; Wu, E. Orchestrating Agents and Data for Enterprise: A Blueprint Architecture for Compound AI. In *Proceedings of the 2025 IEEE 41st International Conference on Data Engineering Workshops*; IEEE: New York, 2025; pp. 18-27.
16. Liu, D. Primary Breadth-First Development (Pbfd): An Approach to Full Stack Software Development. *arXiv* 2025, arXiv:2501.10624.
17. Liu, D. Pbfd and Pdfd: Formally Defined and Verified Methodologies and Empirical Evaluation for Scalable Full-Stack Software Engineering. *Zenodo* 2025. <https://doi.org/10.5281/zenodo.16883985>.
18. Besker, T.; Martini, A.; Bosch, J. Software developer productivity loss due to technical debt. *J. Syst. Softw.* 2019, 156, 41-61.
19. Perera, J.; Tempero, E.; Tu, Y.-C.; Blincoe, K. A systematic mapping study exploring quantification approaches to code, design, and architecture technical debt. *ACM Trans. Softw. Eng. Method.* 2024, 1, 1-35.
20. Kretschmer, R.; Khelladi, D.E.; Lopez-Herrejon, R.E.; Egyed, A. Consistent change propagation within models. *Softw. Syst. Model.* 2021, 20, 539-555.
21. Tkalich, A.; Klotins, E.; Moe, N.B. Identifying critical dependencies in large-scale continuous software engineering. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*; ACM: New York, 2025; pp. 157-168.
22. Behutiye, W.N.; Rodriguez, P.; Oivo, M.; Tosun, A. Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Inf. Softw. Technol.* 2017, 82, 139-158.
23. Arulraj, A.; Pavlo, A.; Menon, V. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*; ACM: New York, 2016; pp. 583-598.
24. Meyer, A.N.; Fritz, T.; Murphy, G.C.; Zimmermann, T. Software developers' perceptions of productivity. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*; ACM: New York, 2014; pp. 19-29.
25. Etikyala, S.P.; Etikyala, V. Efficiency in Cloud-Enabled Asynchronous Services: Analysis of Workflow Orchestrators. In *Proceedings of the World Congress on Computer and Information Technology*; WCCIT: 2023.
26. University of Oxford. FDR Documentation. University of Oxford 2025. <https://cocotec.io/fdr/manual/> (accessed May 15, 2025).
27. Gibson-Robinson, T.; Armstrong, P.; Boulgakov, A.; Roscoe, A.W. FDR3 — A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*; Ábrahám, E., Havelund, K., Eds.; Lecture Notes in Computer Science, Vol. 8413; Springer: Berlin, 2014; pp. 1-15.
28. Liu, D. Pdfd-MVP. *GitHub* 2025. <https://github.com/IBM-Consulting-Formal-Methods/Pdfd-MVP> (accessed May 15, 2025).
29. Liu, D. Pbfd-MVP. *GitHub* 2025. <https://github.com/IBM-Consulting-Formal-Methods/Pbfd-MVP> (accessed May 15, 2025).
30. Lenarduzzi, V.; Taibi, D. MVP Explained: A Systematic Mapping Study on the Definitions of Minimal Viable Product. In *Proceedings of the 2016 42th Euromicro Conference on Software Engineering and Advanced Applications*; IEEE: New York, 2016; pp. 112-119.
31. Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*; Addison-Wesley: Boston, 2003.
32. Brandolini, A. *Introducing EventStorming: An Act of Deliberate Collective Learning*; Leanpub: Victoria, BC, Canada, 2025.
33. Vernon, V. *Domain-Driven Design Distilled*; Addison-Wesley: Boston, 2016.

34. Ihirwe, F.; Di Ruscio, D.; Mazzini, S.; Pierini, P.; Pierantonio, A. Low-code engineering for Internet of Things: A state of research. In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings; ACM: New York, 2020; pp. 1-8.
35. Sahay, A.; Indamutsa, A.; Di Ruscio, D.; Pierantonio, A. Supporting the understanding and comparison of low-code development platforms. In Proceedings of the 2020 46th Euromicro Conference on Software Engineering and Advanced Applications; IEEE: New York, 2020; pp. 171-178.
36. Goguen, J.A.; Burstall, R.M. Introducing institutions. In Proceedings of the Carnegie Mellon Workshop on Logic of Programs; Springer: New York, 1984; pp. 221-256.
37. Spivey, J.M. The Z Notation: A Reference Manual; Prentice Hall: New York, 1992.
38. Jackson, D. Software Abstractions: Logic, Language, and Analysis; MIT Press: Cambridge, 2016.
39. Woodcock, J.; Larsen, P.G.; Bicarregui, J.; Fitzgerald, J. Formal methods: Practice and experience. *ACM Comput. Surv.* 2009, 41, 1-36.
40. Chechik, M.; Combemale, B.; Gray, J.; et al. Formal methods in the scope of the Software and Systems Modeling journal. *Softw. Syst. Model.* 2025, 24, 271-272.
41. Schmidt, D.C. Model-driven engineering. *Computer* 2006, 39, 25-31.
42. France, R.; Rumpe, B. Model-driven development of complex software: A research roadmap. In 2007 Future of Software Engineering; IEEE: New York, 2007; pp. 37-54.
43. Brambilla, M.; Cabot, J.; Wimmer, M. Model-Driven Software Engineering in Practice, Second Edition; Morgan & Claypool: San Rafael, 2017.
44. Hutchinson, J.; Rouncefield, M.; Whittle, J. Model-driven engineering practices in industry. In Proceedings of the 2011 33rd International Conference on Software Engineering; ACM: New York, 2011; pp. 633-642.
45. Hoare, C.A.R. Communicating Sequential Processes; Prentice Hall: New York, 1985.
46. Clarke, E.M.; Grumberg, O.; Peled, D.A. Model Checking; MIT Press: Cambridge, 1999.
47. Hopcroft, J.E.; Ullman, J.D. Introduction to Automata Theory, Languages, and Computation; Addison-Wesley: Boston, 1979.
48. Peterson, J.L. Petri Net Theory and the Modeling of Systems; Prentice Hall: New York, 1981.
49. Zimmermann, T.; Weissgerber, P.; Diehl, S.; Zeller, A. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.* 2005, 31, 429-445.
50. McIntosh, S.; Kamei, Y.; Adams, B.; Hassan, A.E. An empirical study of the impact of modern code review practices on software quality. *Empir. Softw. Eng.* 2016, 21, 2146-2189.
51. Abadi, D.J.; Madden, S.R.; Ferreira, M. Integrating compression and execution in column-oriented database systems. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data; ACM: New York, 2006; pp. 671-682.
52. Elmasri, R.; Navathe, S. Fundamentals of Database Systems, 7th Edition; Pearson: New York, 2016.
53. Stonebraker, M.; et al. C-Store: A column-oriented DBMS. In Proceedings of the 31st International Conference on Very Large Data Bases; VLDB: 2005; pp. 553-564.
54. Garcia-Molina, H.; Ullman, J.D.; Widom, J. Database Systems: The Complete Book, 2nd Edition; Pearson: New York, 2008.
55. Abadi, D.J.; Boncz, P.A.; et al. Column-Stores vs. Row-Stores: How Different Are They Really? In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data; ACM: New York, 2008; pp. 967-980.
56. van der Aalst, W.M.P. The application of Petri nets to workflow management. *J. Circuits Syst. Comput.* 1998, 8, 21-66.
57. Milner, R. Communicating and Mobile Systems: The  $\pi$ -Calculus; Cambridge University Press: Cambridge, 1999.
58. Liskov, B.; Zilles, S. Specification techniques for data abstractions. *ACM SIGPLAN Notices* 1975, 10, 72-87.
59. Harel, D. Statecharts: a visual formalism for complex systems. *Sci. Comput. Program.* 1987, 8, 231-274.
60. Pnueli, A. The temporal logic of programs. In Proceedings of the 18th Annual Symposium on Foundations of Computer Science; IEEE: New York, 1977; pp. 46-57.
61. Dijkstra, E.W. A Discipline of Programming; Prentice-Hall: New York, 1976.

62. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. Introduction to Algorithms, 4th Edition; MIT Press: Cambridge, 2022.
63. Knuth, D.E. The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd Edition; Addison-Wesley: Boston, 1997.
64. Moore, E.F. The shortest path through a maze. In Proceedings of an International Symposium on the Theory of Switching; Harvard University Press: Cambridge, 1959; pp. 285-292.
65. Bass, L.; Clements, P.; Kazman, R. Software Architecture in Practice, 3rd Edition; Addison-Wesley: Boston, 2012.
66. Poppendieck, M.; Poppendieck, T. Lean Software Development: An Agile Toolkit; Addison-Wesley: Boston, 2003.
67. Jones, C. Software Methodologies: A Quantitative Guide; Auerbach Publications: New York, 2018.
68. Edison, H.; Wang, X.; Conboy, K. Comparing Methods for Large-Scale Agile Software Development: A Systematic Literature Review. *IEEE Trans. Softw. Eng.* 2022, 48, 2709-2731.
69. Verdecchia, R.; Kruchten, P.; Lago, P. Architectural Technical Debt: A Grounded Theory. In *Software Architecture*; Springer: Cham, 2020; pp. 202-219.
70. Curran, G.M.; Bauer, M.; Mittman, B.; Pyne, J.M.; Stetler, C. Effectiveness-Implementation Hybrid Designs: Combining Elements of Clinical Effectiveness and Implementation Research to Enhance Public Health Impact. *Med. Care* 2022, 50, 217-226.
71. Holzmann, G. The SPIN Model Checker: Primer and Reference Manual; Addison-Wesley: Boston, 2004.
72. McCreesh, C.; Prosser, P. The shape of the search tree for the maximum clique problem and the implications for parallel branch and bound. *ACM Trans. Parallel Comput.* 2015, 2, 1-27.
73. Womack, J.P.; Jones, D.T. Lean Thinking: Banish Waste and Create Wealth in Your Corporation; Free Press: New York, 2003.
74. Larman, C.; Basili, V.R. Iterative and Incremental Development: A Brief History. *Computer* 2003, 36, 47-56.
75. van der Aalst, W. Process Mining: Data Science in Action; Springer: Berlin, 2016.
76. Derrick, J.; Boiten, E. Refinement: Semantics, Languages and Applications; Springer: Cham, 2018.
77. Wiratunga, N.; Craw, S. Incorporating Backtracking in Knowledge Refinement. In *Validation and Verification of Knowledge Based Systems*; Springer: Boston, 1999; pp. 1-15.
78. Boehm, B.W. A spiral model of software development and enhancement. *Computer* 1988, 21, 61-72.
79. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design patterns: Elements of reusable object-oriented software; Addison-Wesley: Boston, 1994.
80. Parnas, D.L. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM* 1972, 15, 1053-1058.
81. Yourdon, E.; Constantine, L.L. Structured Design: Fundamentals of a Discipline of Computer Program and System Design; Prentice Hall: New York, 1979.
82. Ruijters, E.; Stoelinga, M. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Comput. Sci. Rev.* 2015, 15, 29-62.
83. Boehm, B.; Turner, R. Using risk to balance agile and plan-driven methods. *Computer* 2003, 36, 57-66.
84. Clements, P.; Bachmann, F.; Bass, L.; Garlan, D.; Ivers, J.; Little, R.; Merson, P.; Nord, R.; Wood, B. Documenting Software Architectures: Views and Beyond, 2nd Edition; Addison-Wesley: Boston, 2010.
85. Martin, R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design; Prentice Hall: New York, 2017.
86. Lehman, M.M. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 1980, 68, 1060-1076.
87. ISO/IEC/IEEE 12207:2017. Systems and software engineering — Software life cycle processes. International Organization for Standardization 2017. (accessed May 15, 2025).
88. Lamport, L. The Temporal Logic of Actions (TLA). *ACM Trans. Program. Lang. Syst.* 1994, 16, 872-923.
89. Feathers, M.C. Working Effectively with Legacy Code; Prentice Hall: New York, 2004.
90. Sadalage, P.J.; Fowler, M. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence; Addison-Wesley: Boston, 2012.
91. Silberschatz, A.; Korth, H.F.; Sudarshan, S. Database System Concepts, 7th Edition; McGraw-Hill: New York, 2019.

92. Novotný, P.; Wild, J. Relational modeling of hierarchical data in biodiversity databases. Database 2024, 2024, baae107.
93. Selinger, P.G.; Astrahan, M.M.; Chamberlin, D.D.; Lorie, R.A.; Price, T.G. Access Path Selection in a Relational Database Management System. In Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data; ACM: New York, 1979; pp. 23-34.
94. Knuth, D.E. Bitwise Tricks & Techniques. In The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1; Addison-Wesley: Boston, 2011; pp. 1-62.
95. Warren, H.S. Jr. Hacker's Delight, 2nd Edition; Addison-Wesley: Boston, 2013.
96. Angles, R.; Gutierrez, C. Survey of graph database models. ACM Comput. Surv. 2008, 40, 1:1-1:39.
97. Runeson, P.; Höst, M. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 2009, 14, 131-164.
98. Kitchenham, B.A.; Charters, S. Guidelines for performing systematic literature reviews in software engineering. Keele University Technical Report 2007, EBSE-2007-01.
99. Basili, V.R.; Rombach, H.D. The TAME project: towards improvement-oriented software environments. IEEE Trans. Softw. Eng. 1988, 14, 758-773.
100. Sittig, D.F.; Singh, H. Design and Evaluation of a Structured Incident Reporting System for Healthcare. Int. J. Med. Inform. 2013, 82, 1188-1195.
101. Knuth, D.E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd Edition; Addison-Wesley: Boston, 1997.
102. Cover, T.M.; Thomas, J.A. Elements of Information Theory, 2nd Edition; Wiley: New York, 2006.
103. Solingen, R.; Basili, V.; Caldiera, G.; Rombach, H.D. The Goal Question Metric Approach. In Encyclopedia of Software Engineering; John Wiley & Sons: New York, 2002.
104. Easterbrook, S.; Singer, J.; Storey, M.A.; Damian, D. Selecting Empirical Methods for Software Engineering Research. In Guide to Advanced Empirical Software Engineering; Shull, F., Singer, J., Sjøberg, D.I.K., Eds.; Springer: London, 2008; pp. 1-25.
105. Kitchenham, B.; Pfleeger, S.L.; Pickard, L.M.; Jones, P.W.; Hoaglin, D.C.; El Emam, K.; Rosenberg, J. Preliminary guidelines for empirical research in software engineering. IEEE Trans. Softw. Eng. 2002, 28, 721-734.
106. Shadish, W.R.; Cook, T.D.; Campbell, D.T. Experimental and Quasi-Experimental Designs for Generalized Causal Inference; Cengage Learning: Boston, 2002.
107. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A. Experimentation in Software Engineering; Springer: Berlin, 2012.
108. LaToza, T.D.; Myers, B.A. Hard-to-answer questions about code. In Evaluation and Usability of Programming Languages and Tools; ACM: New York, 2010; pp. 1-8.
109. Stonebraker, M. SQL databases v. NoSQL databases. Commun. ACM 2010, 53, 10-11.
110. Beck, K. Extreme Programming Explained: Embrace Change, 2nd Edition; Addison-Wesley: Boston, 2004.
111. Sommerville, I. Software Engineering, 10th Edition; Pearson: New York, 2015.
112. Pressman, R.S.; Maxim, B.R. Software Engineering: A Practitioner's Approach, 9th Edition; McGraw-Hill: New York, 2019.
113. Robinson, I.; Webber, J. Graph Databases, 2nd Edition; O'Reilly: Sebastopol, 2015.
114. Florescu, D.; Kossmann, D. Storing and Querying XML Data Using an RDMBS. IEEE Data Eng. Bull. 1999, 22, 27-34.
115. Wu, K.; Otoo, E.J.; Shoshani, A. Using Bitmap Indexing Technology for Combined Numerical and Text Queries. LBNL Technical Report 2006, LBNL-59254.
116. Roscoe, A.W. Understanding Concurrent Systems; Springer: London, 2010.
117. Emerson, E.A. Temporal and modal logic. In Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics; Elsevier: Amsterdam, 1990; pp. 995-1072.
118. Elmasri, R.; Navathe, S.B. Fundamentals of Database Systems, 7th Edition; Pearson: New York, 2015.
119. Jackson, M. Problem Frames: Analysing and Structuring Software Development Problems; Addison-Wesley: Boston, 2001.
120. Rumpe, B. Modeling with UML: Language, Concepts, Methods; Springer: Berlin, 2016.

121. Stahl, T.; Voelter, M. *Model-Driven Software Development: Technology, Engineering, Management*; Wiley: New York, 2006.
122. Fitzgerald, B.; Stol, K.-J. Continuous software engineering: A roadmap and agenda. *J. Syst. Softw.* 2017, 123, 176-189.
123. Leite, L.; Rocha, C.; Kon, F.; Milojevic, D.; Meirelles, P. A survey of DevOps concepts and challenges. *ACM Comput. Surv.* 2020, 52, 1-35.
124. Podelski, A.; Rybalchenko, A. A Complete Method for the Synthesis of Linear Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation*; Steffen, B., Levi, G., Eds.; *Lecture Notes in Computer Science*, Vol. 2937; Springer: Berlin, 2004; pp. 239-251.
125. Bradley, C.; Manna, Z.; Sipma, H. Linear Ranking with Reachability. In *Computer Aided Verification*; Etessami, K., Rajamani, S.K., Eds.; *Lecture Notes in Computer Science*, Vol. 3576; Springer: Berlin, 2005; pp. 491-504.
126. Colón, M.A.; Sipma, H.B. Synthesis of Linear Ranking Functions. In *Tools and Algorithms for the Construction and Analysis of Systems*; Margaria, T., Yi, W., Eds.; *Lecture Notes in Computer Science*, Vol. 2031; Springer: Berlin, 2001; pp. 1-15.
127. Cook, B.; Podelski, A.; Rybalchenko, A. Termination Proofs for Systems Code. *ACM SIGPLAN Notices* 2006, 41, 415-426.
128. Larraz, D.; Oliveras, A.; Rodríguez-Carbonell, E.; Rubio, A. Proving termination of imperative programs using Max-SMT. In *Proceedings of the 2013 Formal Methods in Computer-Aided Design*; IEEE: New York, 2013; pp. 218-225.
129. Chatterjee, K.; Goharshady, E.K.; Novotný, P.; Zárevúcky, J.; Žikelić, Đ. On Lexicographic Proof Rules for Probabilistic Termination. In *Formal Methods*; Huisman, M., Păsăreanu, C., Zhan, N., Eds.; *Lecture Notes in Computer Science*, Vol. 13047; Springer: Cham, 2021; pp. 1-20.
130. Roscoe, A.W. *The Theory and Practice of Concurrency*; Prentice-Hall: New York, 2005.
131. Vardi, M.Y. The Complexity of Relational Query Languages. In *Proceedings of the 14th ACM SIGACT Symposium on Theory of Computing*; ACM: New York, 1982; pp. 137-146.
132. Celko, J. *Joe Celko's Trees and Hierarchies in SQL for Smarties*, 2nd Edition; Morgan Kaufmann: Burlington, 2012.
133. Tropashko, V. Nested Intervals Tree Encoding in SQL. *ACM SIGMOD Rec.* 2006, 35, 47-52.
134. Hellerstein, J.M.; Stonebraker, M.; Hamilton, J. Architecture of a Database System. *Found. Trends Databases* 2007, 1, 141-259.
135. Knebl, H. *Algorithms and Data Structures: Foundations and Probabilistic Methods for Design and Analysis*; Springer: Cham, 2020.
136. Date, C.J. *Database Design and Relational Theory: Normal Forms and All That Jazz*, 2nd Edition; Apress: New York, 2019.
137. Griffiths, P.P.; Wade, B.W. An Authorization Mechanism for a Relational Database System. *Commun. ACM* 1976, 19, 429-437.
138. Abadi, D.J. Query execution in column-oriented database systems. PhD Dissertation, Massachusetts Institute of Technology: Cambridge, MA, 2006.
139. Neumann, T. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 2011, 4, 539-550.
140. Bauer, C.; King, G. *Java Persistence with Hibernate*; Manning Publications: New York, 2006.
141. Verbitski, A.; Gupta, A.; Saha, D.; Brahmadesam, M.; Gupta, K.; Mittal, R.; et al. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*; ACM: New York, 2017; pp. 1041-1052.
142. Fowler, M. *Patterns of Enterprise Application Architecture*; Addison-Wesley: Boston, 2002.
143. Kuesel, T.R.; King, E.H.; Bickel, J.O. *Tunnel Engineering Handbook*, 2nd Edition; Springer: New York, 1996.
144. Li, S.; Zhang, Y.; Cao, M.; Wang, Z. Study on excavation sequence of pilot tunnels for a rectangular tunnel using numerical simulation and field monitoring method. *Rock Mech. Rock Eng.* 2022, 55, 3507-3523.

145. Basili, V.R. The Role of Controlled Experiments in Software Engineering Research. In Empirical Software Engineering Issues; Basili, V.R., Rombach, D., Schneider, K., Kitchenham, B., Pfahl, D., Selby, R.W., Eds.; Lecture Notes in Computer Science, Vol. 4336; Springer: Berlin, 2007; pp. 1-12.
146. Sjoberg, D.I.; Hannay, J.E.; Hansen, O.; Kampenes, V.B.; Karahasanovic, A.; Liborg, N.K.; et al. A survey of controlled experiments in software engineering. *IEEE Trans. Softw. Eng.* 2005, 31, 733-753.
147. Sackman, H.; Erikson, W.J.; Grant, E.E. Exploratory experimental studies comparing online and offline programming performance. *Commun. ACM* 1968, 11, 3-11.
148. Forsgren, N.; Storey, M.A.; Maddila, C.; Zimmermann, T.; Houck, B.; Butler, J. The SPACE of developer productivity. *Commun. ACM* 2021, 64, 46-53.
149. Jørgensen, M.; Shepperd, M. A systematic review of software development cost estimation studies. *IEEE Trans. Softw. Eng.* 2007, 33, 33-53.
150. Jain, R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*; Wiley: New York, 1991.
151. Georges, A.; Buytaert, D.; Eeckhout, L. Statistically Rigorous Java Performance Evaluation. In Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications; ACM: New York, 2007; pp. 57-76.
152. Dean, J.; Barroso, L.A. The Tail at Scale. *Commun. ACM* 2013, 56, 74-80.
153. Microsoft Docs. `sp_spaceused` (Transact-SQL). Microsoft 2024. <https://learn.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-spaceused-transact-sql> (accessed May 15, 2025).

**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.