UNSOS

A Complete Computational Operating Framework

Reed Kimble

Contents

1 0000. Project Charter & Scope **14**
- 1.1 Purpose . . . 14
- 1.2 Scope . . . 14
- 1.3 Non-Goals . . . 15
- 1.4 Audience . . . 15
- 1.5 Relationship to UNS, UNS-C, CGP, and Vorticity Space . . . 16
- 1.6 Success Criteria . . . 16
- 1.7 Status . . . 16

2 0001. Terminology, Notation, and Conventions **17**
- 2.1 Purpose . . . 17
- 2.2 Terminology Discipline . . . 17
  - 2.2.1 Canonical Terms . . . 17
  - 2.2.2 Forbidden Overloads . . . 17
- 2.3 UNS-Specific Terminology . . . 18
- 2.4 UNS-C Terminology . . . 18
- 2.5 CGP Terminology . . . 18
- 2.6 Notation Rules . . . 18
  - 2.6.1 General . . . 18
  - 2.6.2 Structural Notation . . . 19
  - 2.6.3 Transformation Notation . . . 19
- 2.7 Representation Conventions . . . 19
  - 2.7.1 Representation Neutrality . . . 19
  - 2.7.2 IR and Code Conventions . . . 19
- 2.8 LLM and Tooling Conventions . . . 19
- 2.9 Change Control . . . 19
- 2.10 Status . . . 20
- 2.11 Decoherence & Outcome Doctrine . . . 20
  - 2.11.1 Core Assertion . . . 20
  - 2.11.2 Outcome Validity . . . 20
  - 2.11.3 Decoherence . . . 20
  - 2.11.4 Pressure and Instability . . . 20
  - 2.11.5 Resolution Semantics . . . 21
  - 2.11.6 Final Boundary Condition . . . 21

3 0003. Application & Distribution Laws **21**
3.1 Purpose . . . 21
3.2 Law A: Single Kernel Compiler (SKC) . . . 22
3.2.1 Statement . . . 22
3.2.2 Implications . . . 22
3.2.3 Unsafe / Development Mode . . . 22
3.3 Law B: Parameter-Scoped Mutation (PSM) . . . 22
3.3.1 Statement . . . 22
3.3.2 Structural Interpretation . . . 22
3.3.3 Consequences . . . 23
3.4 Law C: Transformation-Exact Function Deduplication (TEFD) . . . 23
3.4.1 Statement . . . 23
3.4.2 Canonical Transformation Description (CTD) . . . 23
3.4.3 Multi-Implementation Support . . . 23
3.5 Law D: Sealed Application Capsules . . . 24
3.5.1 Statement . . . 24
3.5.2 Mandatory Capsule Contents . . . 24
3.5.3 IP Protection . . . 24
3.6 Installation Semantics . . . 25
3.7 Ecosystem Consequences . . . 25
3.8 Failure Modes . . . 25
3.9 Status . . . 26

4 0004. Policy Profiles & Distribution **26**
4.1 Purpose . . . 26
4.2 Core Principle: Pre-Authority Narrowing . . . 26
4.2.1 Statement . . . 26
4.3 Policy as Structure . . . 26
4.4 What Policy Profiles May Do . . . 27
4.5 What Policy Profiles May NOT Do . . . 27
4.6 Distribution Model . . . 27
4.7 Mandatory Policy Capsule Contents . . . 28
4.8 Profile Application Semantics . . . 28
4.9 Profile Composition . . . 29
4.10 Storage Policy Narrowing (Extension) . . . 29
4.10.1 Purpose . . . 29
4.10.2 Storage As Policy-Governed Resource Space . . . 29
4.10.3 Interaction With Authority . . . 31
4.10.4 Example Profiles (Non-Normative) . . . 31
4.11 CGP Requirements . . . 32
4.12 Failure Modes . . . 32
4.13 Status . . . 32

5 1000. UNS Foundations for Operating Systems **32**
5.1 Purpose . . . 32
5.2 UNS as a Grammar of Structure . . . 32
5.3 Identity Without Address . . . 33
5.4 Differentiation via Relations . . . 33

5.5 Closure and Kernel Completeness . . . 34
5.6 Reflexivity and Self-Description . . . 34
5.7 Operating Systems as Structural Systems . . . 34
5.8 Why OS Kernels Are a Natural UNS Domain . . . 35
5.9 Constraints Imposed by UNS on UNSOS . . . 35
5.10 Status . . . 35

6 1001. UNS-C: Calculus of Kernel Transformations 35
6.1 Purpose . . . 35
6.2 UNS-C Overview . . . 36
6.3 Transformation Model . . . 36
6.4 Core Properties of UNS-C Transformations . . . 36
6.4.1 C1. Admissibility . . . 36
6.4.2 C2. Locality . . . 37
6.4.3 C3. Compositionality . . . 37
6.4.4 C4. Partiality . . . 37
6.4.5 C5. Non-Invertibility . . . 37
6.5 Invariants Under UNS-C . . . 37
6.6 Equivalence Classes . . . 38
6.7 UNS-C in Kernel Context . . . 38
6.8 Policy Separation . . . 38
6.9 CGP Compatibility . . . 38
6.10 Failure Modes . . . 39
6.11 Status . . . 39

7 1002. CGP as a Kernel Design Diagnostic 39
7.1 Purpose . . . 39
7.2 CGP Recap (Operational Framing) . . . 39
7.3 Representation Families in UNSOS . . . 40
7.4 What CGP Detects in Kernel Design . . . 40
7.5 CGP Divergence Indicators (Kernel-Specific) . . . 40
7.6 CGP as a Design-Time Tool . . . 40
7.7 CGP as a Runtime and Tooling Tool . . . 41
7.8 CGP and UNS / UNS-C Integration . . . 41
7.9 Acceptable CGP Outcomes . . . 41
7.10 Failure Classification . . . 41
7.11 Status . . . 42

8 2000. UNSOS Kernel Overview 42
8.1 Purpose . . . 42
8.2 Definition of the UNSOS Kernel . . . 42
8.3 Kernel Responsibilities . . . 42
8.3.1 1. Structural State Maintenance . . . 42
8.3.2 2. Transformation Application . . . 43
8.3.3 3. Effect Mediation . . . 43
8.3.4 4. Capability Enforcement . . . 43
8.3.5 5. Representation Mediation . . . 43
8.4 What the Kernel Does *Not* Do . . . 43

8.5 Kernel–Policy Boundary . . . 43
8.6 Kernel Interfaces (Abstract) . . . 44
8.7 Kernel as a CGP Convergence Point . . . 44
8.8 Minimality Principle . . . 44
8.9 Status . . . 44

9 2001. Kernel IR & Representation Families 45
9.1 Purpose . . . 45
9.2 Role of the Kernel IR . . . 45
9.3 Kernel IR as Structural Authority . . . 45
9.4 Core Properties of the Kernel IR . . . 45
9.4.1 K1. Representation Invariance . . . 45
9.4.2 K2. Structural Completeness . . . 46
9.4.3 K3. Explicit Effects . . . 46
9.4.4 K4. Stable Identity . . . 46
9.4.5 K5. Local Transformability . . . 46
9.5 Representation Families . . . 46
9.5.1 F1. Expression Family . . . 47
9.5.2 F2. Control-Flow Graph (CFG) Family . . . 47
9.5.3 F3. Dataflow Graph (DFG) Family . . . 47
9.5.4 F4. Relational / Constraint Family . . . 47
9.5.5 F5. Runtime Structural Family . . . 47
9.6 Import Contracts . . . 47
9.7 Export Contracts . . . 48
9.8 CGP Enforcement . . . 48
9.9 What Kernel IR Is Not . . . 48
9.10 Status . . . 48

10 2002. Execution Model 49
10.1 Purpose . . . 49
10.2 Core Commitments . . . 49
10.3 Computation as Structural Evolution . . . 49
10.3.1 Structural State . . . 49
10.3.2 Step Definition . . . 49
10.4 Effect Model . . . 50
10.4.1 Effects Are Explicit . . . 50
10.4.2 Effect Ordering . . . 50
10.4.3 Pure vs Effectful . . . 50
10.5 Task and Continuation Model . . . 50
10.5.1 Tasks as Structures . . . 50
10.5.2 Continuations . . . 50
10.6 Scheduling Neutrality . . . 51
10.7 Nondeterminism . . . 51
10.8 Architecture Realization (x86-64 / AArch64) . . . 51
10.9 CGP Requirements . . . 51
10.10Failure Modes . . . 52
10.11Status . . . 52

11 3000. Heap as UNS Structure **52**
11.1 Purpose . . . 52
11.2 Heap Definition . . . 52
11.3 Identity and Reference . . . 53
11.3.1 Identity Is Structural . . . 53
11.3.2 Handles . . . 53
11.4 Heap Relations . . . 53
11.5 Heap Spaces . . . 54
11.6 Descriptors and Shapes . . . 54
11.7 Root Structure . . . 54
11.8 Heap Invariants (Structural Law) . . . 54
11.8.1 H1. Well-Formedness . . . 55
11.8.2 H2. Edge Validity . . . 55
11.8.3 H3. Handle Soundness . . . 55
11.8.4 H4. Descriptor Integrity . . . 55
11.8.5 H5. Root Closure . . . 55
11.8.6 H6. Architecture Neutrality . . . 55
11.9 Heap Operations as Transformations . . . 55
11.10What This Document Forbids . . . 56
11.11Status . . . 56

12 3001. Garbage Collection as UNS-C Calculus **56**
12.1 Purpose . . . 56
12.2 Design Posture . . . 56
12.3 GC Domain . . . 57
12.4 GC-Invariants (GC-Specific) . . . 57
12.4.1 G1. Reachability Correctness . . . 57
12.4.2 G2. Handle Stability . . . 57
12.4.3 G3. Descriptor-Driven Traversal . . . 57
12.4.4 G4. No Phantom Liveness . . . 57
12.4.5 G5. No Implicit Roots . . . 57
12.5 Admissible Primitive Transformations . . . 58
12.5.1 T1. Shade / Enqueue . . . 58
12.5.2 T2. Mark . . . 58
12.5.3 T3. Scan . . . 58
12.5.4 T4. Evacuate / Move . . . 58
12.5.5 T5. Fixup . . . 59
12.5.6 T6. Unmark / Clear Metadata . . . 59
12.5.7 T7. Sweep / Free . . . 59
12.5.8 T8. Compact (Composed) . . . 60
12.6 Phases as Transformation Sets . . . 60
12.6.1 Minor Collection . . . 60
12.6.2 Major Collection . . . 60
12.7 Incremental and Concurrent GC . . . 60
12.7.1 Required Mechanism: Write Barrier as Transformation . . . 61
12.8 Effect and Capability Considerations . . . 61
12.9 CGP Requirements . . . 61
12.10Failure Modes . . . 61

12.11Status 62

13 3002. Generalized Resource Management 62
13.1 Purpose 62
13.2 Resource Model: Resources as Structures 62
13.3 Resource Identity 62
13.4 Ownership and Authority 63
13.5 Lifetime Structures 63
13.5.1 L1. Reachability-Based Lifetime 63
13.5.2 L2. Ownership-Scoped Lifetime 63
13.5.3 L3. Explicit Lease / Subscription Lifetime 63
13.5.4 L4. Refcount-Compatible Lifetime (Optional) 64
13.6 Reclamation as UNS-C Transformations 64
13.7 Dependency-Driven Reclamation 64
13.8 Resource Spaces and Managers 65
13.9 Policy vs Law 65
13.10Observability and Debugging 65
13.11CGP Requirements 65
13.12Status 66

14 4000. Process Model (Reinterpreted) 66
14.1 Purpose 66
14.2 Process as Structure 66
14.3 Identity and Handles 66
14.4 Address Spaces as Relations 67
14.5 Execution Graph 67
14.6 Process Operations as UNS-C Transformations 67
14.7 Invariants 68
14.7.1 P1. Capability Soundness 68
14.7.2 P2. Ownership Clarity 68
14.7.3 P3. Mapping Consistency 68
14.7.4 P4. Execution Well-Formedness 68
14.7.5 P5. Effect Context Integrity 68
14.8 Kernel Boundary 68
14.9 CGP Requirements 69
14.10Status 69

15 4001. Concurrency as Transformation Space 69
15.1 Purpose 69
15.2 Concurrency as Structure 69
15.2.1 Concurrent State 69
15.3 Partial Orders and Happens-Before 70
15.4 Atomicity and Critical Structure 70
15.5 Synchronization as Structural Patterns 70
15.5.1 S1. Join / Wait 70
15.5.2 S2. Mutex-like Exclusion (Capability-Gated) 71
15.5.3 S3. Semaphores / Counting Access 71
15.5.4 S4. Channels / Message Passing 71

15.5.5 S5. Futures / Promises . . . 71
15.6 Structured Concurrency . . . 71
15.7 Shared Memory and Consistency . . . 71
15.7.1 Consistency as Constraints . . . 72
15.8 Data Race Definition . . . 72
15.9 Policy Separation . . . 72
15.10CGP Requirements . . . 72
15.11Status . . . 73

16 4002. Scheduling via Tracks, Tags, and Time 73
16.1 Purpose . . . 73
16.2 Scheduling as State Sampling . . . 73
16.3 Role of Tags . . . 73
16.3.1 Tags as Admissibility Gates . . . 73
16.4 Role of Tracks . . . 74
16.4.1 Tracks as Preference Signals . . . 74
16.5 Interaction Between Tags and Tracks . . . 74
16.6 Time as an Explicit Effect . . . 74
16.7 Policy Integration . . . 74
16.8 Numeric Metrics (Restricted Role) . . . 75
16.9 Parallelism and Multi-Core Execution . . . 75
16.10Stability and Predictability . . . 75
16.11CGP Requirements . . . 75
16.12Status . . . 75

17 5000. Authority as Structure 76
17.1 Purpose . . . 76
17.2 Authority as Relation . . . 76
17.2.1 Structural Definition . . . 76
17.3 Capabilities . . . 76
17.3.1 Capability Structure . . . 76
17.3.2 No Ambient Authority . . . 76
17.4 Capability Creation . . . 77
17.5 Delegation and Attenuation . . . 77
17.6 Revocation . . . 77
17.7 Authority Over Transformations . . . 78
17.8 Capability Enforcement . . . 78
17.9 Invariants . . . 78
17.9.1 A1. Explicit Authority . . . 78
17.9.2 A2. No Forgery . . . 78
17.9.3 A3. Least Authority . . . 78
17.9.4 A4. Authority Closure . . . 78
17.10CGP Requirements . . . 78
17.11Failure Modes . . . 79
17.12Status . . . 79

18 5001. Isolation & Containment 79
18.1 Purpose . . . 79

18.2 Isolation as Absence of Relations . . . 79
18.2.1 Structural Isolation . . . 79
18.3 Containment . . . 80
18.4 Memory Isolation . . . 80
18.5 Execution Isolation . . . 80
18.6 Module Isolation . . . 80
18.7 Sandboxing . . . 81
18.8 Communication Across Isolation Boundaries . . . 81
18.9 Containment Breakage . . . 81
18.10Revocation and Shrinking . . . 81
18.11CGP Requirements . . . 81
18.12Failure Modes . . . 82
18.13Status . . . 82

19 5002. Verification & Security Invariants 82
19.1 Purpose . . . 82
19.2 Verification Posture . . . 82
19.3 Security Invariants . . . 83
19.3.1 S1. Explicit Authority . . . 83
19.3.2 S2. Capability Non-Forgery . . . 83
19.3.3 S3. Least Authority Preservation . . . 83
19.3.4 S4. Containment Integrity . . . 83
19.3.5 S5. No Ambient Channels . . . 84
19.3.6 S6. Resource Lifetime Safety . . . 84
19.3.7 S7. Effect Authorization . . . 84
19.3.8 S8. Representation-Invariant Enforcement (CGP) . . . 84
19.4 Verification Mechanisms (Structural) . . . 84
19.5 Adapter Verification . . . 85
19.6 Failure Modes . . . 85
19.7 Status . . . 85

20 6000. Unified Resources, Naming, and Projections 85
20.1 6000.1 Purpose . . . 86
20.2 6000.2 Scope . . . 86
20.3 6000.3 Definitions . . . 86
20.4 6000.4 Core Assertions . . . 87
20.4.1 6000.4.1 One ontology . . . 87
20.4.2 6000.4.2 No mounting . . . 87
20.4.3 6000.4.3 Views have shape; storage does not . . . 87
20.4.4 6000.4.4 Outcome-first interaction . . . 87
20.5 6000.5 Temporal Admissibility of Resources . . . 88
20.5.1 6000.5.1 Boot-time only . . . 88
20.5.2 6000.5.2 Boot-time optimized . . . 88
20.5.3 6000.5.3 Runtime-only . . . 88
20.5.4 6000.5.4 Hot-swap . . . 88
20.6 6000.6 Unified Reference Grammar (Selectors) . . . 88
20.6.1 6000.6.1 Selector forms . . . 89
20.6.2 6000.6.2 Candidate resolution as projection . . . 89

20.7 6000.7 Projections: Places, Views, Search . . . 89
20.7.1 6000.7.1 Places . . . 89
20.7.2 6000.7.2 Views . . . 90
20.7.3 6000.7.3 Search . . . 90
20.8 6000.8 Integration Pipeline (External and Internal) . . . 90
20.9 6000.9 Deduplication as Identity (Preview) . . . 90
20.106000.10 Interpretive Resources (Code Pages and Beyond) . . . 91
20.116000.11 Federation Preview: Nodes as Tiles . . . 91
20.126000.12 Requirements Summary . . . 91
20.136000.13 Open Questions (Deferred) . . . 92

21 6001. Storage as Shared Structural Space 92
21.1 6001.1 Purpose . . . 92
21.2 6001.2 Core Assertions . . . 92
21.2.1 6001.2.1 No canonical storage shape . . . 92
21.2.2 6001.2.2 Persistence is structural identity . . . 92
21.2.3 6001.2.3 “Save” is integration . . . 93
21.2.4 6001.2.4 Outcome-first semantics . . . 93
21.3 6001.3 Storage Substrate Model . . . 93
21.4 6001.4 Identity, Versions, and Lineage . . . 93
21.4.1 6001.4.1 Identity classes . . . 93
21.4.2 6001.4.2 Versioning as default . . . 93
21.4.3 6001.4.3 When/Why qualifiers . . . 94
21.5 6001.5 Deduplication as a Consequence of Identity . . . 94
21.5.1 6001.5.1 Multi-granularity dedupe . . . 94
21.6 6001.6 External Media and Device Participation . . . 94
21.6.1 6001.6.1 Integration forms . . . 94
21.6.2 6001.6.2 Removal semantics . . . 94
21.7 6001.7 Reversibility, Pruning, and Collapse . . . 95
21.7.1 6001.7.1 Reversibility by default . . . 95
21.7.2 6001.7.2 Collapse . . . 95
21.7.3 6001.7.3 Pruning . . . 95
21.8 6001.8 Pressure Tracks for Storage . . . 95
21.9 6001.9 Determinism and Auditability . . . 96
21.106001.10 Requirements Summary . . . 96
21.116001.11 Open Questions (Deferred) . . . 96

22 6002. Views, Search, and Projection Mechanics 97
22.1 6002.1 Purpose . . . 97
22.2 6002.2 Core Assertions . . . 97
22.2.1 6002.2.1 Projection-first interaction . . . 97
22.2.2 6002.2.2 Search is a projection operator . . . 97
22.2.3 6002.2.3 Ambiguity is preserved . . . 97
22.3 6002.3 Projection Pipeline Model . . . 97
22.4 6002.4 Views . . . 98
22.4.1 6002.4.1 Definition . . . 98
22.4.2 6002.4.2 View shape classes . . . 98
22.4.3 6002.4.3 View mutability . . . 98

22.5 6002.5 Places . . . 98
22.5.1 6002.5.1 Definition . . . 98
22.5.2 6002.5.2 Places are not locations . . . 98
22.6 6002.6 Search . . . 98
22.6.1 6002.6.1 Search semantics . . . 98
22.6.2 6002.6.2 Ranking . . . 99
22.7 6002.7 Indexing . . . 99
22.8 6002.8 Candidate Sets and Novel . . . 99
22.8.1 6002.8.1 Candidate sets . . . 99
22.8.2 6002.8.2 Novel propagation . . . 99
22.9 6002.9 Determinism and Auditability . . . 99
22.106002.10 Pressure and Instability in Projections . . . 99
22.116002.11 Requirements Summary . . . 99
22.126002.12 Open Questions (Deferred) . . . 100

23 6003. External Resource & I/O Integration 100
23.1 6003.1 Purpose . . . 100
23.2 6003.2 Core Assertions . . . 100
23.2.1 6003.2.1 No external/internal distinction . . . 100
23.2.2 6003.2.2 Integration, not attachment . . . 100
23.2.3 6003.2.3 I/O is structural transformation . . . 101
23.3 6003.3 External Resource Classes . . . 101
23.4 6003.4 Temporal Admissibility (Recap) . . . 101
23.5 6003.5 Integration Pipeline . . . 101
23.6 6003.6 Endpoint Shaping . . . 102
23.6.1 6003.6.1 Endpoint abstraction . . . 102
23.6.2 6003.6.2 Shaping rules . . . 102
23.7 6003.7 Hot-swap and Removal . . . 102
23.7.1 6003.7.1 Hot-swap as default . . . 102
23.7.2 6003.7.2 Removal semantics . . . 102
23.8 6003.8 Performance-Sensitive I/O . . . 102
23.8.1 6003.8.1 Fast-path shaping . . . 102
23.8.2 6003.8.2 Bidirectional translation . . . 103
23.9 6003.9 Security and Authority Boundaries . . . 103
23.106003.10 Determinism and Auditability . . . 103
23.116003.11 Requirements Summary . . . 103
23.126003.12 Open Questions (Deferred) . . . 103

24 6004. Reversibility, Deduplication, and Pruning 104
24.1 6004.1 Purpose . . . 104
24.2 6004.2 Core Assertions . . . 104
24.2.1 6004.2.1 Reversibility is the default state . . . 104
24.2.2 6004.2.2 Deduplication is identity recognition . . . 104
24.2.3 6004.2.3 Pruning is an outcome, not a failure . . . 104
24.3 6004.3 Reversibility Model . . . 104
24.3.1 6004.3.1 Structural reversibility . . . 104
24.3.2 6004.3.2 Explicit irreversibility . . . 105
24.4 6004.4 Deduplication . . . 105

24.4.1 6004.4.1 Identity-driven deduplication . . . 105
24.4.2 6004.4.2 Cross-domain deduplication . . . 105
24.4.3 6004.4.3 Policy and dedupe . . . 105
24.5 6004.5 Pressure-Managed Collapse . . . 106
24.5.1 6004.5.1 Collapse definition . . . 106
24.5.2 6004.5.2 Collapse triggers . . . 106
24.6 6004.6 Pruning . . . 106
24.6.1 6004.6.1 Pruning definition . . . 106
24.6.2 6004.6.2 Pruning rules . . . 106
24.6.3 6004.6.3 Instability-driven pruning . . . 106
24.7 6004.7 Interaction with External Resources . . . 107
24.8 6004.8 Determinism and Auditability . . . 107
24.9 6004.9 Requirements Summary . . . 107
24.106004.10 Open Questions (Deferred) . . . 107

25 6005. Interpretive Resources (Encodings, Code Pages, Decoders) 108
25.1 6005.1 Purpose . . . 108
25.2 6005.2 Core Assertions . . . 108
25.2.1 6005.2.1 Bytes are canonical . . . 108
25.2.2 6005.2.2 Interpretation is projection . . . 108
25.2.3 6005.2.3 No silent substitution . . . 108
25.3 6005.3 Interpretive Resources . . . 108
25.3.1 6005.3.1 Definition . . . 108
25.3.2 6005.3.2 Identity and provenance . . . 109
25.4 6005.4 Code Pages . . . 109
25.4.1 6005.4.1 Respect for existing standards . . . 109
25.4.2 6005.4.2 Code page application . . . 109
25.4.3 6005.4.3 Best projection rule . . . 109
25.5 6005.5 Ambiguity and Novel . . . 109
25.5.1 6005.5.1 Novel in interpretation . . . 109
25.5.2 6005.5.2 Downstream responsibility . . . 110
25.6 6005.6 Policy and Interpretation . . . 110
25.7 6005.7 Determinism and Auditability . . . 110
25.8 6005.8 Interaction with Search and Views . . . 110
25.9 6005.9 Failure Elimination . . . 110
25.106005.10 Requirements Summary . . . 111
25.116005.11 Open Questions (Deferred) . . . 111

26 6100. The UNSOS Console: Interactive Projection Interface 111
26.1 6100.1 Purpose . . . 111
26.2 6100.2 Core Assertions . . . 112
26.2.1 6100.2.1 The console is a projection . . . 112
26.2.2 6100.2.2 Human input is ambiguous by default . . . 112
26.2.3 6100.2.3 Interaction must complete . . . 112
26.3 6100.3 Interpretive Pipeline . . . 112
26.4 6100.4 Interactive Fiction Model . . . 112
26.5 6100.5 Commands as Transformations . . . 113
26.6 6100.6 Disambiguation and Novel . . . 113

26.7 6100.7 Outcome Preview . . . 113
26.8 6100.8 Completion and Boundary Reporting . . . 113
26.9 6100.9 History, Audit, and Reversibility . . . 114
26.106100.10 Learnability and Guidance . . . 114
26.116100.11 Determinism and Safety . . . 114
26.126100.12 Non-Goals . . . 114
26.136100.13 Requirements Summary . . . 115

27 7000. Federation: Tiles, Backplanes, and Translation Layers 115
27.1 7000.1 Purpose . . . 115
27.2 7000.2 Core Assertions . . . 115
27.2.1 7000.2.1 No “distributed system” special case . . . 115
27.2.2 7000.2.2 Local determinism is preserved . . . 115
27.2.3 7000.2.3 Federation is reversible . . . 115
27.3 7000.3 Tiles . . . 116
27.3.1 7000.3.1 Tile definition . . . 116
27.3.2 7000.3.2 Tile autonomy . . . 116
27.4 7000.4 Backplanes . . . 116
27.4.1 7000.4.1 Backplane definition . . . 116
27.4.2 7000.4.2 Backplane neutrality . . . 116
27.5 7000.5 Translation Layers . . . 117
27.5.1 7000.5.1 Translation layer definition . . . 117
27.5.2 7000.5.2 Bidirectional translation . . . 117
27.6 7000.6 Federation Lifecycle . . . 117
27.6.1 7000.6.1 Discovery . . . 117
27.6.2 7000.6.2 Offer and narrowing . . . 117
27.6.3 7000.6.3 Integration . . . 117
27.6.4 7000.6.4 Suspension and removal . . . 117
27.7 7000.7 Consistency and Novel . . . 118
27.7.1 7000.7.1 No global consistency assumption . . . 118
27.7.2 7000.7.2 Novel propagation . . . 118
27.8 7000.8 Pressure and Instability Across Tiles . . . 118
27.9 7000.9 Security and Authority . . . 118
27.9.1 7000.9.1 Authority boundaries . . . 118
27.9.2 7000.9.2 Capability contracts . . . 118
27.107000.10 Determinism and Auditability . . . 119
27.117000.11 Requirements Summary . . . 119
27.127000.12 Open Questions (Deferred) . . . 119

28 8000. Decoherence, Completion, and Outcome Resolution 119
28.1 8000.1 Purpose . . . 119
28.2 8000.2 Core Assertions . . . 120
28.2.1 8000.2.1 Failure does not exist . . . 120
28.2.2 8000.2.2 Completion is mandatory . . . 120
28.2.3 8000.2.3 Decoherence is observable . . . 120
28.3 8000.3 Decoherence . . . 120
28.3.1 8000.3.1 Definition . . . 120
28.3.2 8000.3.2 Decoherence is not an error . . . 120

28.4 8000.4 Completion . . . 120
28.4.1 8000.4.1 Completion definition . . . 120
28.4.2 8000.4.2 Completion pathways . . . 121
28.5 8000.5 Outcome Resolution . . . 121
28.5.1 8000.5.1 Resolution definition . . . 121
28.5.2 8000.5.2 Iterative resolution . . . 121
28.6 8000.6 Pressure and Instability . . . 121
28.6.1 8000.6.1 Pressure as guidance . . . 121
28.6.2 8000.6.2 Instability thresholds . . . 121
28.7 8000.7 Boundary Recognition . . . 122
28.7.1 8000.7.1 Explicit boundaries . . . 122
28.7.2 8000.7.2 No silent abandonment . . . 122
28.8 8000.8 Determinism and Explainability . . . 122
28.9 8000.9 Relationship to Coherence-AI . . . 122
28.108000.10 Requirements Summary . . . 122
28.118000.11 Open Questions (Deferred) . . . 123

29 9000. Posture, Policy, and System Evolution 123
29.1 9000.1 Purpose . . . 123
29.2 9000.2 Core Assertions . . . 123
29.2.1 9000.2.1 Posture is explicit . . . 123
29.2.2 9000.2.2 Policy constrains possibility . . . 123
29.2.3 9000.2.3 Evolution is structural . . . 124
29.3 9000.3 Posture . . . 124
29.3.1 9000.3.1 Definition . . . 124
29.3.2 9000.3.2 Posture transitions . . . 124
29.3.3 9000.3.3 Examples . . . 124
29.4 9000.4 Policy . . . 124
29.4.1 9000.4.1 Policy definition . . . 124
29.4.2 9000.4.2 Policy profiles . . . 125
29.4.3 9000.4.3 Policy evolution . . . 125
29.5 9000.5 System Evolution . . . 125
29.5.1 9000.5.1 Evolution mechanics . . . 125
29.5.2 9000.5.2 Learning without stochasticity . . . 125
29.6 9000.6 Human Interaction . . . 126
29.6.1 9000.6.1 Stability threshold . . . 126
29.6.2 9000.6.2 Override as outcome . . . 126
29.7 9000.7 Federation and Evolution . . . 126
29.8 9000.8 Determinism and Auditability . . . 126
29.9 9000.9 Requirements Summary . . . 126
29.109000.10 Open Questions (Deferred) . . . 127

1 0000. Project Charter & Scope

1.1 Purpose

UNSOS (Universal Number Set Operating System) is a research-driven, implementation-oriented operating system project whose primary goal is to demonstrate that a modern OS kernel, runtime, and system environment can be derived from **structural law**, rather than inherited historical convention.

UNSOS is not an operating system *inspired by* UNS, UNS-C, or CGP. It is an operating system **constrained by them**. These frameworks act as binding design law, not metaphor, analogy, or post hoc justification.

UNSOS further incorporates **Vorticity Space** as a governing model for coherence, pressure, and instability, enabling a system in which outcomes are always valid, failure is abolished as a concept, and escalation is driven solely by recognized decoherence.

The project exists to:

- Test the sufficiency of **UNS** as a grammar for real-world system structure
- Validate **UNS-C** as a calculus for kernel-, runtime-, and system-level transformations
- Apply **CGP** continuously as a diagnostic against representation-dependent design failures
- Demonstrate **pressure, instability, and posture** as viable replacements for error handling, priority, and failure semantics
- Produce a coherent, working operating system whose architecture remains stable across representations, targets, and execution contexts

UNSOS is intended to be *built*, *run*, *examined*, and *federated*, not merely theorized.

1.2 Scope

UNSOS encompasses the full design and implementation of:

- A formally specified kernel architecture constrained by UNS, UNS-C, CGP, and Vorticity Space
- A representation-invariant kernel IR and execution model
- Memory, storage, and resources modeled as unified structural space
- Garbage collection, reclamation, and pruning expressed as UNS-C–admissible transformations
- Concurrency, time, and scheduling expressed as pressure-managed transformation spaces
- Capability-based authority and isolation modeled as structure rather than privilege
- IO, devices, persistence, and external resources expressed as integrated, hot-swappable structure rather than mounted containers
- Explicit treatment of **temporal resource admissibility** (boot-time only, boot-time optimized, runtime)

- Deterministic, system-native compilation and execution targeting **x86-64** and **AArch64 (ARM64)**
- Native support for accelerator-class computation, including GPUs and UNS-C–native analog or hybrid compute backplanes
- A unified resource naming, projection, and view system replacing traditional filesystems and device namespaces
- Deterministic, non-stochastic **Coherence-AI** mechanisms for option narrowing and suggestion
- Federation of multiple UNSOS instances as tiles on a virtual backplane, using UNS-native translation layers
- A complete documentation corpus suitable for LLM-assisted implementation and verification

The scope explicitly includes both **theoretical rigor** and **operational viability**. UNSOS is not a simulation, proof-of-concept, or purely academic artifact.

1.3 Non-Goals

UNSOS explicitly does *not* aim to:

- Recreate or remain compatible with Unix, Linux, POSIX, or Windows semantics
- Provide source or binary compatibility with existing operating systems
- Encode classical notions of failure, exceptions, panics, or priority as kernel concepts
- Optimize prematurely for minimal memory footprint or parity with mature production kernels
- Encode policy decisions (e.g., scheduling heuristics, GC tuning, storage layout) as immutable kernel law
- Serve as a drop-in replacement for any existing OS

Compatibility layers or translation facilities may exist, but they are not design drivers and are considered optional, downstream work.

1.4 Audience

UNSOS is written for:

- Systems and OS researchers
- Kernel, runtime, and low-level infrastructure engineers
- Programming language and compiler designers
- Formal methods and verification practitioners
- Researchers exploring deterministic, non-stochastic intelligence in systems
- Practitioners investigating LLM-assisted system construction

It is *not* intended as an introductory operating systems project or a teaching OS.

1.5 Relationship to UNS, UNS-C, CGP, and Vorticity Space

UNSOS is defined by the following mandatory constraints:

- **UNS** provides the grammatical rules governing structure, identity, differentiation, closure, and reversibility
- **UNS-C** defines which transformations over those structures are admissible, composable, and invariant-preserving
- **CGP** is used as an active diagnostic tool to detect hidden reliance on representational artifacts
- **Vorticity Space** governs coherence, pressure accumulation, instability thresholds, and posture shifts

Any subsystem that cannot be expressed within these constraints is considered out of scope for UNSOS.

1.6 Success Criteria

UNSOS is considered successful if:

- Core kernel mechanisms can be expressed without representation-specific assumptions
- All system evolution can be described in terms of outcomes, pressure, and decoherence rather than failure
- Multiple internal representations converge on identical structural semantics
- Kernel, runtime, and system transformations can be reasoned about locally and compositionally
- The system remains portable across architectures without semantic drift
- Federation preserves local determinism while enabling coherent cross-instance interaction
- The documentation corpus is sufficient for an LLM (e.g., CoPilot) to implement the system faithfully

Success and termination conditions are defined **structurally**, not socially, commercially, or aesthetically.

1.7 Status

This document establishes the authoritative charter and scope for the UNSOS project, updated to reflect accepted axioms and design decisions through the 6xxx document series. All subsequent design, implementation, and documentation decisions are constrained by this charter.

2 0001. Terminology, Notation, and Conventions

2.1 Purpose

This document establishes the **mandatory terminology, notation rules, and representational conventions** for the UNSOS project.

Its role is preventative rather than descriptive: it exists to **eliminate ambiguity, overload, and silent drift** across documents, implementations, representations, and LLM-generated code.

All UNSOS documents, code, diagrams, and machine-generated artifacts are constrained by this document.

2.2 Terminology Discipline

2.2.1 Canonical Terms

The following terms have **fixed meanings** within UNSOS. They must not be redefined locally.

- **Structure**: A set of entities and relations constrained by UNS grammar. Never synonymous with "data structure" unless explicitly stated.
- **Relation / Edge**: A directed, typed connection between entities.
- **Node**: An entity within a structure. Nodes have no intrinsic identity outside their relations.
- **Identity**: Structural position within a relation graph, not memory address, name, or label.
- **Transformation**: A rule-governed change to a structure, defined within UNS-C.
- **Invariant**: A property preserved under all admissible transformations.
- **Kernel Law**: A rule enforced by the kernel grammar or calculus.
- **Policy**: A preference or heuristic that selects among admissible transformations but does not alter admissibility.

2.2.2 Forbidden Overloads

The following terms are **explicitly forbidden** from being overloaded with their traditional meanings unless clearly redefined in context:

- Object
- Process
- Thread
- Address
- Pointer
- Privilege
- Resource
- Ownership

- State

If such a term is required, it must be qualified (e.g., *address-as-relation*, *process-structure*).

2.3 UNS-Specific Terminology

- **UNS**: The Universal Number Set grammar defining admissible structure.
- **Closure**: The property that all constructions remain inside the grammar.
- **Differentiation**: Structural distinction via relations, not labels.
- **Reflexivity**: The ability of structures to include representations of themselves.

UNS terminology is never used metaphorically.

2.4 UNS-C Terminology

- **Admissible Transformation**: A transformation permitted by UNS-C rules.
- **Partial Transformation**: A transformation defined only over a subset of states.
- **Locality**: Transformations operate on bounded structural neighborhoods.
- **Non-invertibility**: Some transformations (e.g., free, commit) cannot be reversed.
- **Equivalence Class**: A set of structures reachable from one another via admissible transformations.

2.5 CGP Terminology

- **Representation Family**: A structurally distinct encoding of the same relations.
- **Convergence**: Structural adequacy preserved across representation families.
- **Divergence Indicator**: Evidence that adequacy depends on representation artifacts.
- **Auxiliary Assumption**: Hidden convention required by one representation but not others.

CGP terminology is used diagnostically, not evaluatively.

2.6 Notation Rules

2.6.1 General

- Mathematical symbols are defined before use.
- Pseudocode is illustrative, not authoritative.
- Diagrams express structure, not flow unless explicitly stated.

2.6.2 Structural Notation

- Nodes: uppercase identifiers (e.g., `N`, `Cell42`)
- Relations/edges: lowercase with type annotation (e.g., `ref`, `owns`, `cap`)
- Directed edges: `A -[rel]-> B`

2.6.3 Transformation Notation

- Transformations are written as `T: S → S'`
- Preconditions are stated explicitly
- Postconditions are invariants + deltas

2.7 Representation Conventions

2.7.1 Representation Neutrality

No document may rely on:

- Textual ordering as semantic ordering
- Memory layout as identity
- Architecture-specific bit patterns
- Implicit evaluation order

Any such reliance must be made explicit and justified as a representation adapter.

2.7.2 IR and Code Conventions

- Kernel IR is the semantic authority
- Surface syntaxes are views, not sources of truth
- Machine code is a lowering, not a reinterpretation

2.8 LLM and Tooling Conventions

- Generated code must reference the document number that authorizes its constructs
- Ambiguity resolution must favor earlier-numbered documents
- When uncertain, tools must fail structurally rather than guess semantically

---

2.9 Change Control

This document may only be amended by explicit revision documents. Silent edits are forbidden.

2.10 Status

This document establishes the binding terminology, notation, and conventions for UNSOS. All subsequent documents inherit and depend upon these definitions.

2.11 Decoherence & Outcome Doctrine

2.11.1 Core Assertion

UNSOS recognizes no concept of failure. All system evolutions are valid outcomes under kernel law. What traditional systems label as "errors" or "failures" are, in UNSOS, simply **outcomes that do not align with the current intent, posture, or coherence objectives**.

There is therefore no exceptional control flow, no panic state, and no ontological distinction between "success" and "failure" at the kernel level.

2.11.2 Outcome Validity

- Every transformation admitted by kernel law produces a valid outcome.
- Undesired outcomes are informative states, not violations.
- Outcome validity is independent of user intent, policy preference, or application expectation.

Outcome evaluation occurs **after** the fact, via coherence analysis, not during execution via exception mechanisms.

2.11.3 Decoherence

Decoherence is the sole negative condition recognized by UNSOS.

Decoherence is defined as:

> A system state in which accumulated pressure indicates declining coherence relative to invariants, intent, or resource stability.

Decoherence is not an error; it is a **geometric and energetic property of state space** under UNS / UNS-C / Vorticity Space principles.

2.11.4 Pressure and Instability

- Undesired outcomes advance one or more **pressure tracks**.
- Pressure is mandatory for:
  - repeated transformations under unchanged conditions
  - non-progressing or no-op transformations

  - unresolved Novel values
- Pressure advancement is monotonic until coherence is restored or posture changes.

Instability represents a threshold beyond which decoherence may not persist.

2.11.5 Resolution Semantics

UNSOS resolves decoherence through lawful state evolution:

1. Re-narrowing the admissible transformation space
2. Selecting alternative coherent transformations
3. Shifting posture (policy, resource allocation, scheduling)
4. Escalating to human override **only when instability thresholds are crossed**

Escalation is not failure; it is **formal recognition of a boundary of manageability**.

2.11.6 Final Boundary Condition

The only terminal condition recognized by UNSOS is:

> *No admissible transformation exists that can reduce pressure further under current law, policy, authority, and resources.*

This condition is fully inspectable, auditable, and explainable.

It represents **recognized decoherence**, not failure.

3 0003. Application & Distribution Laws

3.1 Purpose

This document defines the **binding laws governing application construction, compilation, installation, distribution, and execution** in UNSOS.

These laws elevate application behavior to the same level of rigor as kernel behavior by enforcing:

- a single canonical compiler pipeline
- explicit mutation boundaries
- structural function identity and deduplication
- sealed, rights-bearing application distribution

These are not conventions or tooling preferences. They are **kernel-enforced laws**.

3.2 Law A: Single Kernel Compiler (SKC)

3.2.1 Statement

All executable application behavior in UNSOS MUST originate from compilation performed by the kernel's canonical compiler pipeline.

No application code executes unless it has passed through SKC and been validated against UNS, UNS-C, and CGP constraints.

3.2.2 Implications

- There is exactly **one authoritative compiler** for UNSOS.
- All front-end languages compile *into* SKC, never around it.
- Native binaries are not executed directly.
- Execution artifacts are kernel-validated transformation objects, not opaque machine code.

SKC is responsible for: - canonical transformation construction - invariant validation - effect and capability declaration - CGP convergence checks

---

3.2.3 Unsafe / Development Mode

An unsafe or development mode MAY exist, but only as a **capability-gated exception**.

- Requires an explicit `cap:AllowUnsafeCompilation` (name illustrative)
- Must execute inside explicit containment/sandbox structures
- May not bypass UNS-C invariant enforcement

Unsafe mode weakens *assumptions*, not *law*.

3.3 Law B: Parameter-Scoped Mutation (PSM)

3.3.1 Statement

An application function may modify **only** the state that is reachable from its declared input parameters.

No function may mutate ambient, implicit, or non-parameter state.

3.3.2 Structural Interpretation

For a function `F(P1, P2, …, Pn)`:

- The admissible mutation footprint is the structural closure of:
  - the parameter graph induced by `P1..Pn`
  - plus any explicitly passed capability or resource parameters

Any attempted transformation outside this footprint is structurally invalid.

3.3.3 Consequences

- Global mutable state does not exist by default
- Side effects must be explicit in parameters
- Concurrency reasoning becomes local
- Sandboxing becomes trivial

This law integrates directly with capability enforcement (5000) and generalized resource management (3002).

3.4 Law C: Transformation-Exact Function Deduplication (TEFD)

3.4.1 Statement

During installation, any application function whose **Canonical Transformation Description (CTD)** exactly matches an existing CTD MUST be abstracted away and bound to the existing implementation.

3.4.2 Canonical Transformation Description (CTD)

A CTD is the canonical, normalized structural description of a function's behavior, including:

- input and output structure
- admissible transformations
- explicit effects and ordering constraints
- capability requirements
- invariants preserved

CTDs are produced and signed by SKC.

Identity comparison is exact, structural, and representation-invariant.

3.4.3 Multi-Implementation Support

Multiple implementations MAY exist for a single CTD under limited circumstances.

- Identity is bound to CTD
- Implementation selection is policy
- Variants may differ by:
  - target architecture
  - performance profile
  - safety tier

Deduplication applies at the CTD level, not at the implementation level.

3.5 Law D: Sealed Application Capsules

3.5.1 Statement

Applications are distributed as **sealed project capsules**, not as loose binaries or libraries.

A capsule is a compressed, encrypted, and signed bundle containing:

- organized source files
- mandatory structural documents
- optional tests or evidence
- rights and licensing metadata n Capsules are inert until processed by SKC.

3.5.2 Mandatory Capsule Contents

At minimum, a capsule MUST contain:

1. **Manifest Document**
   - application identity
   - declared capabilities
   - target constraints
2. **Rights Document**
   - execution rights
   - redistribution permissions
   - expiration or lease terms

Additional documents MAY exist but must not alter law.

3.5.3 IP Protection

- Source code remains encrypted at rest
- SKC operates within kernel authority
- Execution rights do not imply source access

IP protection is intrinsic to the platform, not layered on later.

3.6 Installation Semantics

Installation is defined as:

1. Capsule validation (signature, rights)
2. Compilation via SKC
3. CTD generation
4. CTD matching against installed corpus
5. Deduplication or variant registration
6. Capability binding

No application is installed without completing this pipeline.

3.7 Ecosystem Consequences

These laws eliminate:

- dependency version mismatches
- massive rarely-used libraries
- ambient global state assumptions
- opaque binary execution

They enable:

- emergent shared function ecosystems
- structural package discovery
- precise security reasoning
- reproducible, CGP-stable execution

3.8 Failure Modes

- **Structural Failure**: violation of SKC, PSM, or CTD laws
- **Adapter Failure**: capsule handling or compiler boundary breach
- **Policy Failure**: suboptimal variant selection or rights delegation

Structural and adapter failures invalidate correctness.

3.9 Status

This document establishes the binding laws for application behavior and distribution in UNSOS. All user-space languages, tooling, and package mechanisms must conform to these laws.

4 0004. Policy Profiles & Distribution

4.1 Purpose

This document defines **policy profiles** in UNSOS as distributable, first-class structural artifacts that **narrow the option space** of admissible system behavior *before* user intervention and rights assignment occur.

UNSOS enforces a strict separation:

- **Kernel Law**: what is admissible (UNS / UNS-C / CGP + invariants)
- **Policy**: how choices are selected and constrained within admissible space
- **Authority**: what a principal is permitted to do (capabilities)

Policy profiles are not permissions. They do not grant authority. They constrain and shape the environment in which authority is later assigned.

4.2 Core Principle: Pre-Authority Narrowing

4.2.1 Statement

A policy profile is applied **before** principals are created or user rights are assigned.

This establishes a system posture in which:

- certain admissible transformations are disallowed by policy
- certain transformation selections are preferred
- certain defaults are enforced

Only after this narrowing occurs may principals be instantiated and capabilities delegated.

4.3 Policy as Structure

Policy is represented as explicit structure, not as hidden configuration.

A policy profile `PP` is a structured artifact containing:

- **Constraints**: prohibitions or limits on selectable transformations
- **Preferences**: ordering or weighting over admissible choices
- **Budgets**: resource/time caps and quotas
- **Defaults**: initial configuration of policy modules

- **Trust rules**: allowed signers, capsule constraints, unsafe-mode rules

Policy is therefore:

- inspectable
- auditable
- composable
- representation-invariant

4.4 What Policy Profiles May Do

Policy profiles MAY:

- restrict which admissible transformations may be selected
- restrict when certain transformations may occur
- set scheduling strategies and fairness constraints
- set GC triggering policy, pause budgets, and compaction aggressiveness
- restrict installation behavior (e.g., required signers, forbidden capabilities)
- define default containment and sandboxing posture
- define default capability attenuation rules
- define resource budgets and quotas
- define user experience posture (simplicity, accessibility constraints) as policy

All such actions remain within kernel law.

4.5 What Policy Profiles May NOT Do

Policy profiles MUST NOT:

- alter admissibility rules defined by UNS / UNS-C
- violate or weaken invariants
- grant capabilities or authority
- create ambient privilege
- introduce representation-dependent semantics
- override CGP diagnostics

If a profile attempts to do any of the above, it is structurally invalid.

4.6 Distribution Model

Policy profiles are distributed as **sealed policy capsules**.

A sealed policy capsule is:

- compressed
- encrypted at rest
- signed
- processed and validated by the kernel

Policy capsules parallel application capsules (0003) but contain policy artifacts rather than application code.

4.7 Mandatory Policy Capsule Contents

At minimum, a policy capsule MUST contain:

1. **Policy Manifest**
   - profile identity
   - target constraints
   - declared policy modules affected
2. **Policy Rules Document**
   - constraints
   - preferences
   - budgets
   - defaults
   - trust rules
3. **Rights / Authority to Apply**
   - who may apply or modify this profile
   - whether it may be overridden
   - lease/expiration terms (optional)

4.8 Profile Application Semantics

Applying a policy profile is defined as:

1. validate capsule signature and rights
2. validate structural correctness of policy structure
3. install profile as an active policy root structure
4. activate policy modules with defined defaults
5. enforce constraints for all subsequent selections

Policy application occurs prior to principal creation for initial system posture.

4.9 Profile Composition

Policy profiles support composition via explicit overlay semantics.

Examples:

- `BaseProfile + ChildOverlay`
- `BaseProfile + AccessibilityOverlay`
- `EnterpriseBase + KioskOverlay`

Composition rules must be structural and deterministic:

- constraints compose by intersection
- budgets compose by minimum
- preferences compose by declared priority order
- conflicts must be explicit and fail closed unless policy permits resolution

No implicit "last write wins" is permitted.

4.10 Storage Policy Narrowing (Extension)

4.10.1 Purpose

This section defines **storage behavior** as a first-class target of policy profile narrowing. Storage is treated as a generalized resource space whose admissible transformations are fixed by kernel law, while policy profiles constrain *which storage behaviors are selectable* and *under what conditions*.

Policy-driven storage narrowing occurs **prior to principal creation** and applies uniformly to applications, users, and services instantiated under the profile.

4.10.2 Storage As Policy-Governed Resource Space

Policy profiles MAY narrow storage behavior along the following dimensions:

4.10.2.1 1. Canonicalization Trust & Availability

- Which **canonical content descriptors (CCDs)** are enabled (e.g., raw bytes, UTF-8 strings, structured text)
- Which user-space canonicalizers are trusted
- Whether unsafe or experimental canonicalizers are permitted (typically development-only)

Canonicalization availability directly constrains deduplication and persistence identity.

4.10.2.2 2. Deduplication Posture

Policy profiles MAY define deduplication scope and aggressiveness:

- exact-content deduplication only
- chunked deduplication
- structural / token-based deduplication

Profiles MUST also define **deduplication domains**, such as:

- global dedupe domain
- profile-scoped dedupe domain
- principal-scoped dedupe domain

This prevents unintended cross-boundary information leakage.

4.10.2.3 3. Encryption & Sealing Posture

Policy profiles MAY constrain:

- encryption-at-rest requirements
- acceptable key authorities
- whether decrypted content may be cached
- sealing requirements for persistent objects

These constraints apply uniformly across storage operations.

4.10.2.4 4. Retention, History, and Deletion

Policy profiles MAY define:

- whether version history is mandatory, optional, or forbidden
- snapshot frequency or journaling requirements
- secure deletion posture (e.g., tombstone + key drop)

Retention behavior is policy, not application discretion.

4.10.2.5 5. Namespace & Visibility Rules

Policy profiles MAY constrain:

- creation of global or shared namespaces
- default containment for application storage
- public binding visibility

Names do not grant authority; policy governs which namespaces may exist.

4.10.2.6 6. Quotas & Budgets

Policy profiles MAY impose:

- per-principal storage quotas
- per-namespace limits
- metadata growth budgets

Budgets compose using minimum rules during profile overlay.

4.10.3 Interaction With Authority

Storage policy narrowing does NOT grant access to storage.

Capabilities are still required for:

- reading
- writing
- binding
- deleting

Policy only constrains which storage transformations are selectable once authorized.

4.10.4 Example Profiles (Non-Normative)

4.10.4.1 Child Profile

- structural text canonicalization enabled
- dedupe scoped to profile only
- mandatory encryption-at-rest
- restricted namespace creation

4.10.4.2 Grandma Profile

- conservative dedupe
- strong retention guarantees
- simplified namespace layout

4.10.4.3 Kiosk / Appliance Profile

- no dynamic persistence
- fixed storage graph
- no history retention

These examples illustrate storage narrowing before authority assignment.

4.11 CGP Requirements

Policy profiles must be representation-invariant:

- no semantics derived from textual ordering
- no dependence on encoding artifacts
- equivalent policy structures must behave equivalently across representations

Any divergence indicates policy design failure.

4.12 Failure Modes

- **Structural Failure**: profile attempts to modify law, grant authority, or introduce ambient privilege
- **Adapter Failure**: capsule validation or application boundary breach
- **Policy Failure**: poor posture selection within lawful constraints

Structural and adapter failures invalidate correctness.

4.13 Status

This document establishes policy profiles as distributable structural artifacts that narrow UNSOS option space before user intervention and rights assignment. All policy systems, profiles, and profile composition must conform to these laws.

5 1000. UNS Foundations for Operating Systems

5.1 Purpose

This document establishes **why and how the Universal Number Set (UNS)** is a suitable—and in this project, mandatory—foundation for operating system design.

It translates UNS from a general structural grammar into an **OS-facing foundation**, clarifying what UNS constrains, what it enables, and why traditional operating system concepts map cleanly (and often more rigorously) into UNS terms.

This document is purely foundational. It does not prescribe implementations or policies. Its role is to define **what kinds of structures are even admissible** in UNSOS.

5.2 UNS as a Grammar of Structure

UNS is not a number system in the traditional sense. It is a **grammar of admissible structure**, defining what it means for entities to exist, differ, relate, and compose without appeal to external

semantics.

At its core, UNS asserts:

- Structure is primary; labels and magnitudes are derivative
- Identity arises from relational position, not intrinsic properties
- All valid constructions are closed under the grammar

For an operating system, this immediately reframes the problem:

> An OS is not a collection of privileged objects and procedures; it is a continuously evolving structure of relations constrained by invariants.

5.3 Identity Without Address

Traditional operating systems conflate **identity** with **memory address** or **name**. This creates deep coupling between representation and meaning.

UNS forbids this.

In UNS terms:

- Identity is defined by *structural position*
- Movement, copying, or re-encoding does not alter identity
- Destruction is the only admissible way to remove identity

In UNSOS this leads directly to:

- Handle-based reference models
- Moving and compacting memory as a first-class operation
- Architecture-independent identity

5.4 Differentiation via Relations

UNS rejects differentiation by tagging or labeling alone.

Two nodes are distinct **only if** they participate in different relations.

For OS design this implies:

- Processes differ by their execution and resource relations, not by PID labels
- Authority differs by capability edges, not by mode bits or flags
- Memory regions differ by ownership and reachability, not by address range

This allows UNSOS to eliminate many historical "special cases" that exist solely to preserve labeling schemes.

5.5 Closure and Kernel Completeness

Closure is a defining UNS property:

> All constructions must remain inside the grammar.

For an OS kernel, this means:

- No meta-level escapes to "the machine knows"
- No correctness rules enforced outside structural checks
- No hidden evaluators deciding validity

Every kernel mechanism—memory, scheduling, IO, security—must be expressible as **structure plus admissible transformation**.

If a rule cannot be stated structurally, it is not part of the kernel.

5.6 Reflexivity and Self-Description

UNS permits **reflexive structure**: a system may contain representations of itself without contradiction.

For UNSOS this enables:

- Self-describing kernels
- Introspectable execution state
- In-kernel representations of IR, layouts, capabilities, and invariants

Critically, this does *not* introduce a meta-language. Reflexive structures are governed by the same grammar as all others.

This property underpins later support for:

- Structural reflection
- Proof-carrying transformations
- Deterministic replay and time-travel debugging

5.7 Operating Systems as Structural Systems

From a UNS perspective, an operating system is a **persistent, evolving relational structure** that:

- Mediates interaction between other structures
- Enforces invariants through admissible transformation
- Exposes controlled interfaces as relations

This reframing dissolves several traditional OS dichotomies:

- Kernel vs user space becomes a question of *capability structure*
- Process vs thread becomes a question of *execution graph shape*
- Files vs memory become persistent vs transient substructures

UNS does not remove these distinctions; it **re-expresses them structurally**.

5.8 Why OS Kernels Are a Natural UNS Domain

OS kernels are unusually well-suited to UNS because:

- They already operate on graphs (process trees, memory graphs, dependency graphs)
- They enforce invariants continuously
- They must remain coherent under constant transformation
- They are sensitive to representation artifacts—making CGP diagnostics essential

UNSOS treats the kernel as a **living UNS structure**, not a static artifact.

5.9 Constraints Imposed by UNS on UNSOS

By adopting UNS, UNSOS explicitly accepts the following constraints:

- No privileged identities
- No ambient authority
- No representation-dependent semantics
- No correctness rules outside the grammar

These constraints are not limitations; they are what make UNSOS internally coherent and externally analyzable.

5.10 Status

This document establishes UNS as the foundational grammar for UNSOS. All subsequent kernel, runtime, and user-space designs must be admissible UNS structures or be explicitly declared out of scope.

6 1001. UNS-C: Calculus of Kernel Transformations

6.1 Purpose

This document establishes **UNS-C (Universal Number Set – Calculus)** as the formal framework governing *all admissible change* within UNSOS.

Where UNS defines **what structures may exist**, UNS-C defines **how those structures may change**. Together, they form the complete foundation for kernel dynamics, runtime behavior, and system evolution.

This document is binding on: - Kernel mechanisms - Runtime services - Memory management and GC - Scheduling and concurrency - Capability transfer and revocation - Compilation and lowering passes

Any change not expressible as an admissible UNS-C transformation is **invalid within UNSOS**.

6.2 UNS-C Overview

UNS-C is a **non-semantic calculus of transformation**. It does not encode meaning, preference, optimization, or intent. It defines only:

- Which transformations are admissible
- What invariants must be preserved
- How transformations compose
- When transformations may be partial or non-invertible

UNS-C explicitly separates *correctness* from *desirability*.

6.3 Transformation Model

A transformation is written as:

```
T : S → S'
```

Where: - `S` is a valid UNS structure - `S'` is a valid UNS structure - `T` is admissible under the calculus

Transformations are first-class entities in UNSOS and may themselves be represented as structures.

6.4 Core Properties of UNS-C Transformations

6.4.1 C1. Admissibility

A transformation is admissible **if and only if** it:

- Is defined entirely within the UNS grammar
- Preserves all declared invariants
- Does not rely on external evaluators or hidden semantics

Admissibility is structural, not contextual.

6.4.2 C2. Locality

All transformations must be **local**.

A local transformation: - Operates on a bounded substructure - Does not require global knowledge - Produces globally coherent behavior via composition

Locality enables incremental execution, concurrency, and verification.

6.4.3 C3. Compositionality

If transformations `T1` and `T2` are admissible, then their composition:

`T2  T1`

is also admissible *provided invariants are preserved.*

Global system evolution is defined as a composition of local transformations.

6.4.4 C4. Partiality

Transformations may be **partial**.

A partial transformation: - Is only defined over a subset of valid structures - Must explicitly state its domain of applicability

Example: a `Free` transformation is only admissible over unreachable structures.

6.4.5 C5. Non-Invertibility

Some transformations are inherently **non-invertible**.

Examples: - Deallocation - Commit - Capability revocation

UNS-C permits non-invertibility provided invariants are preserved and the domain rules are explicit.

6.5 Invariants Under UNS-C

An **invariant** is any property preserved under all admissible transformations.

Invariants may include: - Structural well-formedness - Identity preservation - Reachability constraints - Effect ordering - Capability soundness

Invariant preservation is mandatory. No transformation may violate an invariant, regardless of policy motivation.

6.6 Equivalence Classes

UNS-C induces **equivalence classes** over structures.

Two structures are equivalent if one can be transformed into the other via a sequence of admissible transformations.

Equivalence does *not* imply identity.

This concept underpins: - Optimization correctness - Caching and deduplication - Representation switching (CGP)

6.7 UNS-C in Kernel Context

Within UNSOS, UNS-C governs:

- Memory allocation, movement, and reclamation
- Scheduling and execution interleaving
- Capability creation, delegation, and revocation
- IO interaction and state transition
- Compilation passes and lowering

The kernel does not "execute commands"; it **applies transformations**.

6.8 Policy Separation

UNS-C explicitly forbids encoding preference or optimization into the calculus.

Examples of *policy* (not calculus): - Which task to schedule next - When to trigger GC - How aggressively to compact memory

Policy selects *when* to apply transformations, never *which transformations are legal*.

6.9 CGP Compatibility

UNS-C transformations must be **representation-invariant**.

A transformation that: - Only works in one representation - Relies on layout or encoding artifacts - Requires representation-specific auxiliary assumptions

is invalid under CGP and therefore invalid in UNSOS.

6.10 Failure Modes

Failures are classified as:

- **Structural Failure**: violation of UNS grammar or UNS-C invariants
- **Policy Failure**: poor choice among admissible transformations
- **Implementation Failure**: incorrect realization of a valid transformation

Only structural failures invalidate the design.

6.11 Status

This document establishes UNS-C as the binding calculus of change for UNSOS. All kernel and runtime behavior must be expressible as admissible UNS-C transformations.

7 1002. CGP as a Kernel Design Diagnostic

7.1 Purpose

This document establishes **CGP (Convergence–Generalization Principle)** as an active, mandatory diagnostic framework for the design, verification, and evolution of UNSOS.

CGP is not a philosophy, optimization technique, or proof of correctness. Within UNSOS it serves a precise role:

> **To detect when kernel correctness or expressiveness depends on representational artifacts rather than structure.**

CGP is applied continuously, not retroactively. Any subsystem that fails CGP diagnostics is considered structurally suspect, regardless of apparent functionality.

7.2 CGP Recap (Operational Framing)

CGP evaluates a system relative to a **family of representations** of the same underlying relations.

A design exhibits **convergence** if: - The same relations can be expressed across multiple, structurally distinct representations - No representation requires extra expressive power, hidden assumptions, or ad hoc extensions - Observable behavior and admissible transformations remain invariant

A design exhibits **divergence** if: - Expressiveness depends on one representation's artifacts - One representation requires auxiliary scaffolding not present in others - Correctness silently relies on layout, ordering, or encoding conventions

CGP does not claim truth or optimality. It diagnoses *representation dependence*.

7.3 Representation Families in UNSOS

Within UNSOS, CGP is applied across the following primary representation families:

- **Expression Form**: functional / term-based representations
- **CFG Form**: block-structured control-flow graphs
- **DFG Form**: dataflow graphs with explicit dependencies
- **Relational Form**: constraint- and relation-based descriptions
- **Structural Runtime Form**: heap, capability, and execution graphs

No representation is privileged. Kernel law must survive translation between them.

7.4 What CGP Detects in Kernel Design

CGP is used to detect:

- Implicit evaluation order
- Hidden global state
- Architecture-dependent semantics
- Address- or layout-derived identity
- IR-specific correctness assumptions
- Optimizations that change meaning when representation changes

These are historically common OS failure modes.

7.5 CGP Divergence Indicators (Kernel-Specific)

A kernel mechanism fails CGP if any of the following are true:

1. It can only be expressed in one IR or representation family
2. It requires representation-specific annotations to remain correct
3. Its correctness depends on textual order, memory layout, or encoding
4. It gains power from implicit conventions
5. It cannot be translated without semantic loss

Any such mechanism must be redesigned or rejected.

7.6 CGP as a Design-Time Tool

CGP is applied during design by:

- Designing mechanisms in at least two distinct representations
- Forcing translation between representations

- Identifying where expressiveness or correctness is lost

Designs that survive this process are considered structurally sound.

7.7 CGP as a Runtime and Tooling Tool

CGP is also applied operationally:

- Multi-view IR optimizers must preserve structure
- GC and scheduling transformations must be representation-invariant
- Debugging and tracing must not alter semantics by representation choice

Tooling is expected to surface CGP failures explicitly.

7.8 CGP and UNS / UNS-C Integration

CGP operates *on top of* UNS and UNS-C:

- **UNS** defines what structures may exist
- **UNS-C** defines how they may change
- **CGP** tests whether those definitions are representation-stable

A system may be UNS-admissible and UNS-C-correct yet still fail CGP. Such failures indicate hidden dependence on representation artifacts.

7.9 Acceptable CGP Outcomes

CGP does not require global convergence.

Acceptable outcomes include: - Local convergence with documented divergence elsewhere - Explicit representation adapters at system boundaries - Declared out-of-scope representations

Unacceptable outcomes are *implicit* divergence.

7.10 Failure Classification

CGP-related failures are classified as:

- **Structural CGP Failure**: Kernel law depends on representation
- **Tooling CGP Failure**: Compiler or debugger introduces divergence
- **Policy CGP Failure**: Optimization strategy depends on representation artifacts

Structural CGP failures invalidate the design.

7.11 Status

This document establishes CGP as a mandatory diagnostic instrument for UNSOS. All kernel mechanisms, transformations, and tooling must withstand CGP analysis or be explicitly excluded from scope.

8 2000. UNSOS Kernel Overview

8.1 Purpose

This document defines **what the UNSOS kernel is**, **what it is responsible for**, and—equally important—**what it explicitly is not**.

The kernel is the first layer where UNS, UNS-C, and CGP are realized as an *operational system*. This document establishes kernel boundaries before any mechanism-level detail is introduced, preventing accidental policy leakage or historical OS assumptions.

8.2 Definition of the UNSOS Kernel

The UNSOS kernel is a **structural transformation engine**.

It does not: - "run programs" in the traditional sense - interpret high-level semantics - encode optimization preferences - enforce policy decisions beyond admissibility

Instead, the kernel: - Maintains a set of **admissible structures** (UNS) - Applies **admissible transformations** to those structures (UNS-C) - Enforces **structural invariants** - Exposes controlled interfaces for external interaction

---

8.3 Kernel Responsibilities

The UNSOS kernel is responsible for the following domains, expressed structurally:

8.3.1 1. Structural State Maintenance

- Maintain kernel-visible structures representing:
  - Execution graphs
  - Memory and resource graphs
  - Capability and authority graphs
- Enforce well-formedness invariants at all times

8.3.2 2. Transformation Application

- Accept requests for transformation
- Validate admissibility under UNS-C
- Apply transformations locally and compositionally
- Reject non-admissible transformations deterministically

8.3.3 3. Effect Mediation

- Represent effects (IO, time, interaction) as explicit structure
- Preserve effect ordering constraints
- Prevent implicit effect sequencing

8.3.4 4. Capability Enforcement

- Enforce authority purely via structural capability relations
- Prevent ambient or implicit privilege
- Support delegation and revocation as transformations

8.3.5 5. Representation Mediation

- Serve as the convergence point between representation families
- Ensure semantics survive translation
- Surface CGP divergence explicitly

8.4 What the Kernel Does *Not* Do

The UNSOS kernel explicitly does **not**:

- Choose scheduling policies
- Decide memory reclamation heuristics
- Interpret user-space language semantics
- Optimize for performance beyond structural correctness
- Encode device-specific behavior

All such decisions live in **policy layers**, **user-space services**, or **representation adapters**.

8.5 Kernel–Policy Boundary

A strict boundary exists between:

- **Kernel Law** (grammar + calculus + invariants)
- **Policy** (preference, optimization, heuristics)

The kernel provides: - Admissible operations - Structural visibility

Policy provides: - Selection strategies - Optimization goals

Policy may *request* transformations but may not alter admissibility rules.

8.6 Kernel Interfaces (Abstract)

The kernel exposes a minimal set of abstract interfaces:

- **Transformation Interface**: submit, validate, apply transformations
- **Structural Query Interface**: inspect kernel-visible structure
- **Capability Interface**: create, transfer, revoke authority
- **Effect Interface**: request effectful transformations

These interfaces are structural, not semantic APIs.

8.7 Kernel as a CGP Convergence Point

The kernel is the **primary CGP convergence point** in UNSOS.

All of the following must converge structurally at the kernel boundary:

- Compiler IRs
- Runtime execution models
- Memory and resource management
- Debugging and introspection tools

Any divergence detected here indicates a design failure upstream.

8.8 Minimality Principle

The UNSOS kernel is intentionally minimal.

A mechanism belongs in the kernel *only if*: - It enforces an invariant - It defines admissibility - It mediates representation convergence

Everything else is pushed outward.

8.9 Status

This document establishes the scope and responsibility of the UNSOS kernel. Subsequent documents will refine *how* these responsibilities are realized without altering the boundaries defined here.

9 2001. Kernel IR & Representation Families

9.1 Purpose

This document defines the **Kernel Intermediate Representation (Kernel IR)** as the semantic authority of UNSOS and specifies the **representation families** that must converge upon it.

Kernel IR is not merely an implementation artifact. It is the **structural contract** between: - theory (UNS / UNS-C), - diagnostics (CGP), and - execution (kernel, runtime, compilation).

This document establishes what the Kernel IR *is*, what it must guarantee, and how multiple representations relate to it without semantic drift.

9.2 Role of the Kernel IR

The Kernel IR serves as:

- The **canonical structural form** for kernel-visible computation
- The convergence point for multiple representation families
- The reference semantics for correctness, transformation, and lowering

No surface syntax, compiler front-end, runtime view, or machine representation is authoritative over the Kernel IR.

9.3 Kernel IR as Structural Authority

The Kernel IR encodes:

- Explicit control structure
- Explicit data dependencies
- Explicit effect ordering
- Explicit capability usage
- Explicit resource relations

Nothing implicit is permitted. If a property cannot be expressed structurally in the IR, it does not exist at the kernel level.

9.4 Core Properties of the Kernel IR

The Kernel IR must satisfy the following properties:

9.4.1 K1. Representation Invariance

The Kernel IR must be equally expressive when imported from any supported representation family.

No family may require additional expressive power, annotations, or auxiliary constructs beyond what the Kernel IR provides.

9.4.2 K2. Structural Completeness

All kernel-relevant behavior must be representable in the IR:

- Control flow
- Data flow
- Effects
- Capabilities
- Resource ownership

Partial encodings or side channels are forbidden.

9.4.3 K3. Explicit Effects

All effects (IO, time, interaction, nondeterminism) are represented explicitly.

The IR must prevent: - Implicit sequencing - Accidental reordering - Representation-dependent effect behavior

---

9.4.4 K4. Stable Identity

All IR entities have stable structural identity independent of layout or address.

Rewriting, lowering, and relocation must preserve identity unless explicitly destroyed by an admissible transformation.

9.4.5 K5. Local Transformability

IR constructs must admit local UNS-C transformations.

Global rewrites must be decomposable into local steps.

9.5 Representation Families

UNSOS recognizes the following representation families as **first-class and non-privileged**:

9.5.1 F1. Expression Family

Characteristics: - Nested expressions - Functional or term-based form - Implicit control expressed structurally

Requirements: - All evaluation order must be made explicit on import - All effects must be lifted into explicit structure

---

9.5.2 F2. Control-Flow Graph (CFG) Family

Characteristics: - Explicit basic blocks - Explicit branching and joining - SSA-style value flow

Requirements: - Dominance and joining must be explicit - No reliance on fallthrough or textual ordering

---

9.5.3 F3. Dataflow Graph (DFG) Family

Characteristics: - Nodes represent operations - Edges represent dependencies - Scheduling is external

Requirements: - Control dependencies must be made explicit - Effect ordering must be represented structurally

---

9.5.4 F4. Relational / Constraint Family

Characteristics: - Relations describe valid states - Execution arises from constraint satisfaction

Requirements: - Chosen execution schedules must be represented explicitly - Nondeterminism must be modeled as an explicit effect

---

9.5.5 F5. Runtime Structural Family

Characteristics: - Heap graphs - Capability graphs - Execution and scheduling graphs

Requirements: - Must be directly representable in Kernel IR terms - No runtime-only semantics permitted

---

9.6 Import Contracts

Each representation family must satisfy an **import contract** when producing Kernel IR:

- All implicit structure must be made explicit

- No semantics may be introduced or removed
- Any loss of information is a CGP failure

Importers are representation adapters, not semantic interpreters.

9.7 Export Contracts

Kernel IR may be exported into other representations for:

- Optimization
- Visualization
- Debugging
- Lowering

Exports must preserve: - Structural identity - Admissible transformation sets - Observable behavior

---

9.8 CGP Enforcement

CGP is enforced at the Kernel IR boundary:

- Multiple representations of the same program must map to equivalent IR structures
- Divergence is treated as a design error

Kernel IR is the **test harness** for CGP convergence.

9.9 What Kernel IR Is Not

Kernel IR is not:

- A user-facing language
- A syntax tree
- A machine IR
- An optimization playground

It is a **structural contract**.

9.10 Status

This document establishes the Kernel IR as the semantic authority and defines the representation families UNSOS must support without privileging any single form.

10 2002. Execution Model

10.1 Purpose

This document defines the UNSOS **Execution Model**: how computation proceeds as **structural evolution** under UNS and UNS-C, while remaining **representation-invariant** under CGP.

UNSOS does not define execution as "a CPU runs instructions." That is a target-dependent realization.

Instead, UNSOS defines execution as:

> **A sequence (or partial order) of admissible transformations applied to kernel-visible structures, subject to explicit effect constraints and capability constraints.**

This execution model is the semantic bridge between Kernel IR and machine-level behavior.

10.2 Core Commitments

The execution model is constrained by the following commitments:

1. **Explicitness**: All semantics relevant to correctness must be explicit in structure.
2. **Effect Visibility**: Effects are first-class and ordered only by explicit constraints.
3. **Scheduling Neutrality**: Execution order is not inherently total; scheduling is policy.
4. **Capability Soundness**: Operations occur only when structurally authorized.
5. **Representation Invariance**: Equivalent structures behave equivalently across representations.

10.3 Computation as Structural Evolution

10.3.1 Structural State

At any moment, the system state consists of a collection of kernel-visible structures, including:

- Execution graphs (tasks, continuations, pending work)
- Memory graphs (handle mappings, heap spaces, reachability)
- Capability graphs (authority relations)
- Effect graphs (ordered constraints over effectful actions)

10.3.2 Step Definition

A single **execution step** is the application of one admissible transformation:

```
T : S → S'
```

Where: - `S` and `S'` are valid kernel states - `T` is admissible under UNS-C

Execution is therefore defined as a chain (or DAG) of transformations.

10.4 Effect Model

10.4.1 Effects Are Explicit

All effectful actions (IO, time, randomness, external interaction, nondeterminism) must be represented as explicit structure.

UNSOS forbids: - Implicit IO - Hidden time reads - Ambient nondeterminism

10.4.2 Effect Ordering

Effect ordering is represented by explicit constraints, typically as:

- Effect tokens (linear or partially ordered)
- Effect edges in a constraint graph

No ordering exists unless expressed structurally.

10.4.3 Pure vs Effectful

The execution model distinguishes:

- **Pure transformations**: do not alter effect structure
- **Effectful transformations**: consume/produce effect structure

This distinction is structural and must remain stable across all representation families.

10.5 Task and Continuation Model

10.5.1 Tasks as Structures

A "task" is a structural unit of pending or active computation:

- It owns or references a continuation structure
- It holds a capability context
- It participates in an execution graph

Tasks are not privileged threads. Threads are a target-level scheduling implementation.

10.5.2 Continuations

Continuations are explicit structures encoding "what remains to be done."

This supports: - suspension/resumption - structured concurrency - deterministic replay - time-travel debugging

---

10.6 Scheduling Neutrality

UNSOS does not define a single execution order.

Instead: - The kernel defines **admissible next transformations** (law) - Policy selects among them (preference)

This naturally supports: - cooperative scheduling - preemptive scheduling - work-stealing - actor scheduling

All are policies over the same calculus.

10.7 Nondeterminism

Nondeterminism is not forbidden, but must be explicit.

Sources of nondeterminism include: - external interrupts - timers - device input - concurrent interleavings

All nondeterminism is modeled as explicit effectful structure.

If a choice is not represented structurally, it does not exist.

10.8 Architecture Realization (x86-64 / AArch64)

Machine execution (instruction streams, registers, traps, interrupts) is a **representation adapter**.

The adapter is responsible for:

- Root enumeration and stack maps
- Trap/interrupt translation into effectful transformations
- Lowering continuations into machine contexts
- Enforcing atomic and memory-order requirements

Architecture differences may change implementation details, but must not change kernel semantics.

10.9 CGP Requirements

The execution model must remain stable under representation change.

Therefore: - A CFG representation and a DFG representation of the same Kernel IR must yield equivalent admissible transformation sets - No representation may gain or lose effect ordering constraints - No representation may imply a total order that is not structurally present

Any violation is a CGP failure.

10.10 Failure Modes

Execution model failures are classified as:

- **Structural Failure**: transformation violates invariants or introduces implicit semantics
- **Adapter Failure**: target-level execution diverges from kernel semantics
- **Policy Failure**: poor scheduling choices within admissible space

Structural and adapter failures invalidate correctness.

10.11 Status

This document establishes UNSOS execution as admissible structural evolution under explicit effect and capability constraints, independent of representation family and target architecture.

11 3000. Heap as UNS Structure

11.1 Purpose

This document specifies the UNSOS heap as a **UNS-admissible structural system**.

It defines the heap not as "allocated blocks at addresses," but as a **relational structure** whose correctness is determined by grammatical constraints, not conventions.

This document establishes: - The heap's structural model (nodes, relations, substructures) - Identity and reference rules (handles, not addresses) - Required invariants for well-formed memory state - The division between heap law (structure) and GC policy (preference)

This document does not define GC algorithms; that is the role of 3001.

11.2 Heap Definition

The UNSOS heap is a directed, typed relational structure composed of:

- **Cells**: storage nodes containing payload and relational fields
- **Handles**: stable references used by computation to denote cells
- **Relations**: typed edges between cells (references)

- **Spaces**: substructures representing allocation domains
- **Root Structure**: a designated root substructure that defines reachability

The heap is correct if it remains inside UNS grammar: well-typed, well-formed, closed.

11.3 Identity and Reference

11.3.1 Identity Is Structural

A cell's identity is not its physical address.

Identity is defined by: - Structural position within the heap's relation graph - Persistence of handle identity under relocation

11.3.2 Handles

A handle is the only valid reference form visible to kernel-visible computation.

A minimal handle model:

- `handle_id`: stable identity token
- `generation`: prevents stale reuse

The mapping:

`Handle → Cell`

is a kernel-maintained structure.

Raw pointers are permitted only in representation adapters and must never carry semantic identity.

11.4 Heap Relations

Heap relations are typed edges. Examples:

- `ref`: standard strong reference
- `weak`: non-owning reference
- `owns`: ownership relation (space → cell)
- `desc`: descriptor relation (cell → descriptor)
- `fwd`: forwarding relation (cell → replacement)

Relations are part of kernel law; their existence and meaning are not implied.

11.5 Heap Spaces

The heap is partitioned into **spaces**, which are substructures with distinct allocation and reclamation regimes.

Spaces are still UNS structures and must obey the same invariants.

Minimal recommended spaces:

- **Nursery Space**: fast allocation, frequent evacuation
- **Tenured Space**: longer-lived cells, less frequent compaction
- **Large Object Space**: special handling for large allocations
- **Metadata Space**: descriptors, handle tables, shape tables

Space design influences policy, but not heap law.

11.6 Descriptors and Shapes

Every allocated cell must have a descriptor relation:

```
Cell -[desc]-> Descriptor
```

Descriptors define: - Field layout and reference slots - Type/shape identity - Traversal rules for GC and introspection

Descriptors are structural nodes, not compiler-only metadata.

11.7 Root Structure

Roots are not implicit.

The root set is represented as a **root structure**:

- A designated node (or small structure) holding edges to root handles
- Roots may originate from:
  - stacks and registers (via adapters)
  - globals
  - runtime metadata

The kernel treats roots as ordinary edges in a designated substructure.

11.8 Heap Invariants (Structural Law)

The heap must preserve the following invariants at all times.

11.8.1 H1. Well-Formedness

Every cell is exactly one of:

- **Free**: owned by a free-structure
- **Allocated**: owned by exactly one space and has a valid descriptor

No cell may be multiply owned or unclassified.

11.8.2 H2. Edge Validity

Every edge must target:

- A valid handle
- A valid cell
- A null/sentinel

No dangling edges.

11.8.3 H3. Handle Soundness

Every live handle maps to exactly one allocated cell.

Stale handles must be rejected by generation checks.

11.8.4 H4. Descriptor Integrity

Every allocated cell has exactly one descriptor relation.

Descriptor graphs must be acyclic or explicitly cycle-safe.

11.8.5 H5. Root Closure

All reachable computation must be representable via reachability from the root structure.

11.8.6 H6. Architecture Neutrality

Heap correctness must not depend on address bit patterns, alignment tricks, or pointer tagging.

Representation adapters may use such techniques internally, but the heap model must not.

11.9 Heap Operations as Transformations

All heap operations are expressed as admissible transformations:

- `Alloc`: introduce new allocated cell with descriptor
- `Link`: add/update a reference edge
- `Unlink`: remove a reference edge

- `Move`: relocate cell, preserve handle identity
- `Free`: return unreachable cell to free-structure

The admissibility and sequencing of these operations are governed by UNS-C and specified in 3001.

11.10 What This Document Forbids

This heap model forbids:

- Address identity as semantic identity
- Implicit roots
- Implicit traversal rules
- Hidden metadata required for correctness
- GC algorithms that rely on representational artifacts

11.11 Status

This document establishes the UNSOS heap as a UNS-admissible relational structure with explicit identity, relations, and invariants. It defines heap law and prepares the ground for GC as a UNS-C calculus in 3001.

12 3001. Garbage Collection as UNS-C Calculus

12.1 Purpose

This document defines garbage collection (GC) in UNSOS as a **UNS-C admissible transformation system** operating over the heap structure defined in 3000.

GC is not treated as a monolithic algorithm. Instead, it is defined as:

> A closed set of **admissible, local transformations** that preserve heap invariants while permitting reclamation, relocation, and compaction.

GC *policy* (when to collect, how aggressively, pause budgets, heuristics) is explicitly out of scope for this document.

12.2 Design Posture

GC in UNSOS is governed by these commitments:

1. **Calculus before policy**: legality is structural; preference is external.
2. **Locality**: all GC actions decompose into bounded local transformations.
3. **Compositionality**: global collection behavior arises from composition.

4. **Explicitness**: reachability and effect ordering constraints are structural.
5. **Representation invariance (CGP)**: GC correctness must not depend on address artifacts or representation-specific assumptions.

12.3 GC Domain

GC operates on the following structures:

- Heap graph (cells, relations, spaces)
- Handle mapping structure
- Root structure
- Descriptor/shape structure
- Optional mark/worklist structures

GC transformations must preserve heap invariants H1–H6 from 3000.

12.4 GC-Invariants (GC-Specific)

In addition to heap invariants, GC preserves these:

12.4.1 G1. Reachability Correctness

A cell is considered live if and only if it is reachable from the root structure via strong reference relations.

12.4.2 G2. Handle Stability

Handle identities remain stable across all GC transformations.

12.4.3 G3. Descriptor-Driven Traversal

Traversal of references is defined by descriptor structure, not by implicit layout.

12.4.4 G4. No Phantom Liveness

No cell may remain live solely due to GC internal bookkeeping once external reachability is removed.

12.4.5 G5. No Implicit Roots

GC may not introduce implicit roots. Any temporary protection must be explicit in a GC structure.

12.5 Admissible Primitive Transformations

The GC calculus is defined as a set of admissible primitives. Each primitive is:

- Local
- Domain-restricted
- Invariant-preserving

12.5.1 T1. Shade / Enqueue

Purpose: introduce a cell into a work structure without changing heap reachability.

- Input: cell `C`, work structure `W`
- Action: add relation `W -[work]-> C`
- Preconditions: `C` is allocated; `W` is valid
- Postconditions: invariants preserved; reachability unchanged

---

12.5.2 T2. Mark

Purpose: record that a cell has been reached during traversal.

- Action: attach mark annotation (bit or relation) to `C`
- Preconditions: `C` allocated
- Postconditions: does not alter heap relations; only GC metadata changes

Mark is a structural annotation and must be representable as structure.

12.5.3 T3. Scan

Purpose: expand traversal frontier.

- Input: marked cell `C`
- Action:
  - enumerate outgoing strong-reference edges by descriptor rules
  - for each target `D`, apply `Shade/Enqueue` and/or `Mark` as required

Scan is defined by descriptor structure; scanning rules must not be hardcoded layout assumptions.

12.5.4 T4. Evacuate / Move

Purpose: relocate a cell while preserving identity.

- Input: cell `C` in source space, target space `S'`

- Action:
  - allocate new cell `C'`
  - copy payload and edges (as defined by descriptor)
  - install forwarding relation `C -[fwd]-> C'`
  - update handle mapping so `Handle(C) ↠ C'`
- Preconditions:
  - `C` is allocated
  - descriptor supports relocation
- Postconditions:
  - handle stability preserved
  - references remain correct under forwarding rule

12.5.5 T5. Fixup

Purpose: repair edges that point to moved cells.

- Input: edge `A -[ref]-> B`
- Action:
  - if `B` has forwarding relation, rewrite edge to `A -[ref]-> fwd(B)`

Fixup is local and descriptor-guided.

12.5.6 T6. Unmark / Clear Metadata

Purpose: remove GC-specific annotations.

- Action: remove mark and work relations.
- Preconditions: GC phase boundaries satisfied.

12.5.7 T7. Sweep / Free

Purpose: reclaim unreachable cells.

- Input: cell `C`
- Domain restriction:
  - `C` is allocated
  - `C` is not reachable from root structure (as determined by admissible traversal state)
- Action:
  - remove all incoming/outgoing edges as required by space law
  - return `C` to free-structure

Sweep is explicitly **non-invertible**.

12.5.8 T8. Compact (Composed)

Compaction is not a primitive.

It is defined as a composed sequence of:

- Evacuate/Move
- Fixup
- Sweep/Free

Compaction remains admissible because it composes admissible local steps.

12.6 Phases as Transformation Sets

GC phases are not global states; they are **allowed transformation subsets**.

Examples:

12.6.1 Minor Collection

- Shade
- Mark
- Scan
- Evacuate (nursery)
- Fixup
- Clear metadata

12.6.2 Major Collection

- Shade
- Mark
- Scan
- Sweep (tenured)
- Optional compaction

Phase boundaries are policy-controlled but must remain structurally representable.

12.7 Incremental and Concurrent GC

UNS-C locality and compositionality make incremental/concurrent GC natural.

12.7.1 Required Mechanism: Write Barrier as Transformation

A write barrier is a mandatory structural rule:

If during concurrent marking: - a marked (black) cell gains a reference to an unmarked (white) cell

then an admissible compensating transformation must occur: - Shade/Enqueue the target, or - re-shade the source depending on barrier variant

Barrier choice is policy; barrier correctness is calculus.

12.8 Effect and Capability Considerations

GC itself is a kernel activity and must respect:

- effect ordering constraints (it may not reorder externally visible effects)
- capability boundaries (it may only traverse and modify structures it is authorized to)

GC transformations operate within a capability-limited kernel context.

12.9 CGP Requirements

GC correctness must not depend on representational artifacts:

Forbidden dependencies: - pointer tagging assumptions - address identity - alignment tricks - implicit stack scanning without maps

Adapters may use architecture-specific mechanisms, but the calculus and invariants must not depend on them.

12.10 Failure Modes

GC failures are classified as:

- **Structural Failure**: invariants violated, improper domain rule, implicit root introduced
- **Adapter Failure**: root enumeration or barriers mis-implemented
- **Policy Failure**: poor timing or heuristic choice

Structural and adapter failures invalidate correctness.

12.11 Status

This document defines GC in UNSOS as a UNS-C admissible transformation calculus over the heap structure. It establishes primitives, domain rules, invariants, and compositional phase behavior while leaving policy selection to downstream layers.

13 3002. Generalized Resource Management

13.1 Purpose

This document generalizes UNSOS memory management into a unified **resource management framework**.

In UNSOS, memory is not a special case. It is one instance of a broader kernel responsibility:

> Maintain and evolve structures that represent **resources**, their **ownership**, their **reachability**, and their **reclamation**, under explicit invariants.

This document defines a shared structural model for managing:

- memory
- capabilities
- file descriptors / handles
- device channels
- IPC endpoints
- time- and event-subscriptions
- kernel-managed objects of any kind

It also defines a unified reclamation posture: **reclamation is a UNS-C admissible transformation**, with legality defined structurally and policy defined externally.

13.2 Resource Model: Resources as Structures

A **resource** in UNSOS is a node (or structured subgraph) that participates in:

- ownership relations
- authority relations
- lifetime relations
- dependency relations

Resources are not privileged by type. They differ only by their descriptors and relations.

13.3 Resource Identity

Resource identity is structural, not numeric.

Traditional OS identifiers (FDs, PIDs, handles) are treated as **handles**:

- Stable reference tokens
- Backed by a mapping structure
- Protected against stale reuse via generation

This unifies memory handles and non-memory handles under one model.

13.4 Ownership and Authority

UNSOS distinguishes:

- **Ownership**: who is responsible for lifecycle and reclamation
- **Authority**: who is permitted to use or transform the resource

Ownership and authority are separate relation types.

Examples:

- `owns`: space / manager → resource
- `cap`: principal / context → resource

No ambient authority exists. Authority always flows by explicit edges.

13.5 Lifetime Structures

Resource lifetimes are represented structurally using one or more of the following patterns:

13.5.1 L1. Reachability-Based Lifetime

A resource remains live if it is reachable from a designated root structure via strong relations.

Example: - managed memory objects - dynamically created endpoints

13.5.2 L2. Ownership-Scoped Lifetime

A resource remains live while owned by an owner structure.

Example: - kernel allocations bound to a task - per-process address space structures

13.5.3 L3. Explicit Lease / Subscription Lifetime

A resource remains live while a lease relation is valid.

Example: - timers - event subscriptions - capability leases

13.5.4 L4. Refcount-Compatible Lifetime (Optional)

Refcount may exist as a *policy implementation*, but the kernel does not treat refcount as a primitive truth mechanism.

If refcount is used, it must remain representable structurally and must not introduce implicit ordering assumptions.

13.6 Reclamation as UNS-C Transformations

Reclamation is defined as a family of admissible transformations that remove resources from live structures.

Reclamation transformations are typically:

- **partial**: only defined when domain conditions hold (e.g., unreachable)
- **non-invertible**: reclamation destroys identity

Examples:

- `FreeMemoryCell`
- `CloseChannel`
- `RevokeCapability`
- `CancelSubscription`

Each reclamation transformation must declare:

- its domain restrictions
- which invariants it preserves
- which identity is destroyed

13.7 Dependency-Driven Reclamation

Many resources depend on other resources.

UNSOS models dependencies structurally:

- `depends_on`: A $\rightarrow$ B

Reclamation must respect dependency invariants:

- A may not outlive B unless explicitly permitted
- Reclaiming B may induce reclamation or transformation of A

Dependency rules are expressed structurally, not in procedural cleanup code.

13.8 Resource Spaces and Managers

Resources may be grouped into **spaces**, each managed by a resource manager structure.

A resource manager provides:

- allocation transformations
- reclamation transformations
- invariants for its space

Memory spaces are one instance of this pattern.

13.9 Policy vs Law

This document defines resource **law**.

Policy includes:

- when to reclaim
- how aggressively to compact
- which caches to drop
- which resources to evict under pressure

Policy may select among admissible transformations but cannot alter admissibility.

13.10 Observability and Debugging

Because resources and lifetimes are structural, UNSOS can expose:

- a resource graph snapshot
- ownership chains
- capability edges
- reclamation candidates and proofs

This enables: - leak diagnosis - lifetime auditing - deterministic replay support

---

13.11 CGP Requirements

Resource management must not depend on representational artifacts.

Forbidden patterns:

- meaning derived from numeric ranges (e.g., “small FD means system FD”)
- authority derived from mode bits without capability edges
- cleanup behavior dependent on textual control flow

All lifetime and authority behavior must survive translation across representations.

13.12 Status

This document establishes a unified structural framework for resource identity, ownership, authority, lifetime, and reclamation in UNSOS. Memory management is a special case of generalized resource management; all other kernel-managed resources must conform to the same structural laws.

14 4000. Process Model (Reinterpreted)

14.1 Purpose

This document defines the UNSOS **process model**, reinterpreted under UNS, UNS-C, and CGP.

UNSOS does not adopt the historical process abstraction as a privileged entity defined by: - a PID - an address space - a thread list - a syscall boundary

Instead, UNSOS defines a process as a **structured execution graph** with explicit relations to:

- capabilities (authority)
- resources (ownership)
- memory structures (addressing and mapping)
- execution continuations (pending computation)
- effect context (explicit ordering constraints)

This document establishes the structural definition of "process" and the invariants it must preserve.

14.2 Process as Structure

A process is a substructure `P` that includes (at minimum):

- **Execution Graph**: tasks/continuations representing pending and active computation
- **Capability Context**: the authority edges that determine what `P` may do
- **Resource Ownership Graph**: resources whose lifetime is scoped to `P`
- **Memory Mapping Structure**: relations that define which memory is accessible and how
- **Effect Context**: explicit effect tokens/constraints relevant to `P`

No component is privileged by name; the process is defined by the presence of these relations.

14.3 Identity and Handles

Process identity is structural.

A “PID” is treated as a handle:

- stable reference token
- mapped to a process structure
- protected by generation

A PID does not confer authority.

14.4 Address Spaces as Relations

Traditional systems treat an address space as a privileged, implicit mapping.

UNSOS treats an address space as a **memory mapping structure** consisting of relations:

- `maps`: `AddressSpace → Mapping`
- `range`: `Mapping → RangeDescriptor`
- `backs`: `Mapping → BackingResource` (page store / object / file)
- `perm`: `Mapping → PermissionDescriptor` (capability-limited)

The meaning of “address” is therefore not intrinsic. It is a coordinate within a mapping relation.

This allows: - multiple addressing schemes - region-based memory - object-capability style addressing

All without semantic drift.

14.5 Execution Graph

A process contains an execution graph consisting of:

- tasks
- continuations
- scheduling relations
- synchronization structures (defined in 4001)

Tasks are structural units of computation, not necessarily OS threads.

Machine threads are representation adapters that realize execution policies.

14.6 Process Operations as UNS-C Transformations

All process lifecycle operations are admissible transformations:

- `CreateProcess`: introduce process structure with initial relations
- `SpawnTask`: add a task/continuation substructure
- `Join/Wait`: add and resolve synchronization relations

- `TransferCapability`: modify capability edges
- `MapMemory`: modify mapping relations
- `DestroyProcess`: reclaim process structure (non-invertible)

Each transformation: - declares domain restrictions - preserves invariants - is locally decomposable

---

14.7 Invariants

The process model preserves these invariants.

14.7.1 P1. Capability Soundness

A process may only perform transformations for which it possesses explicit capability edges.

14.7.2 P2. Ownership Clarity

All process-scoped resources must be reachable from `P` via ownership relations.

No ambient cleanup.

14.7.3 P3. Mapping Consistency

Every accessible memory region must be derivable from explicit mapping structure.

14.7.4 P4. Execution Well-Formedness

The execution graph must remain well-formed under scheduling transformations.

14.7.5 P5. Effect Context Integrity

Effect ordering constraints must not be altered implicitly by process-level changes.

14.8 Kernel Boundary

UNSOS does not require a syscall boundary defined by tradition.

Instead, the kernel boundary is structural:

- User-space computation is computation lacking kernel-only capabilities
- Kernel actions are transformations requiring kernel capabilities

A “system call” is therefore:

> A request to perform a transformation that requires authority not held by the caller.

The interface is capability-mediated, not instruction-mediated.

14.9 CGP Requirements

The process model must survive representation changes:

- A process must remain the same structure under different IR views
- No process semantics may depend on PID numbering, address layout, or textual ordering
- Translation between CFG, DFG, and relational views must preserve process behavior

Any representation-specific process assumptions are CGP failures.

14.10 Status

This document establishes the UNSOS process model as a structured execution and authority graph, with address spaces and system boundaries expressed as relations and capabilities. Subsequent documents define concurrency (4001) and time/scheduling (4002) over this foundation.

15 4001. Concurrency as Transformation Space

15.1 Purpose

This document defines concurrency in UNSOS as a **transformation space** rather than as a fixed set of historical primitives (threads, locks, preemption).

In UNSOS:

> Concurrency is the structural coexistence of multiple admissible transformation sequences over shared or related structures.

The kernel does not privilege any one concurrency paradigm. Instead, it defines a lawful space of concurrent evolution under: - explicit effect constraints - explicit capability constraints - explicit happens-before relations

Scheduling and execution strategies remain policy.

15.2 Concurrency as Structure

15.2.1 Concurrent State

A concurrent system state contains:

- multiple tasks / continuations
- shared resource graphs
- synchronization structures

- effect ordering constraints
- explicit partial-order relations

Concurrency is present whenever more than one task can legally progress via admissible transformations.

15.3 Partial Orders and Happens-Before

UNSOS models ordering as explicit structure.

A **happens-before relation** is an edge in an ordering graph:

- `A -[hb]-> B`

Where `A` and `B` are transformation events or effect tokens.

No ordering exists unless represented.

This enables: - deterministic reasoning - multiple scheduling strategies - CGP-safe representation changes

---

15.4 Atomicity and Critical Structure

Atomicity is not defined as "CPU instruction atomic."

In UNSOS, atomicity means:

> A transformation (or composed sequence) appears as a single admissible step with respect to specified invariants.

Atomic regions are expressed structurally:

- A region of transformations guarded by capability + ordering constraints
- Or a transformation declared indivisible at the kernel law level

Machine-level atomic operations are adapters that realize kernel-declared atomic transformations.

15.5 Synchronization as Structural Patterns

UNSOS defines synchronization in terms of explicit structural patterns that constrain admissible transformation interleavings.

15.5.1 S1. Join / Wait

A task may declare a dependency relation:

- `TaskA -[waits_on]-> TaskB`

Admissibility rules restrict TaskA's progress until TaskB reaches a defined state.

15.5.2 S2. Mutex-like Exclusion (Capability-Gated)

Mutual exclusion is modeled by exclusive possession of a capability edge:

- `Task -[cap]-> LockToken`

Only the holder may perform transformations requiring that token.

15.5.3 S3. Semaphores / Counting Access

Counting access is modeled as a resource with N token relations.

Transformations consume/produce tokens explicitly.

15.5.4 S4. Channels / Message Passing

Message passing is modeled as a channel structure:

- `Channel -[queue]-> Message`

Send and receive are admissible transformations that modify the queue structure.

15.5.5 S5. Futures / Promises

A promise is a structure that can transition from unresolved to resolved.

Dependent tasks hold relations to the promise and are restricted by domain rules.

15.6 Structured Concurrency

UNSOS favors structured concurrency as a **policy-friendly structural shape**, not a forced paradigm.

Structured concurrency is expressed by containment relations:

- `Scope -[contains]-> Task`

Invariants: - Tasks do not outlive their scope unless explicitly detached by transformation.

This integrates naturally with generalized resource management (3002).

15.7 Shared Memory and Consistency

Shared memory is permitted, but all consistency assumptions must be explicit.

15.7.1 Consistency as Constraints

Memory ordering is represented structurally via:

- effect ordering edges
- atomic transformation declarations
- explicit barriers as transformations

No memory model is implied by architecture.

Architecture-specific fences are adapter realizations of structural constraints.

15.8 Data Race Definition

A data race is defined structurally as:

- two or more admissible transformations
- operating on the same mutable structure
- without sufficient ordering or exclusion constraints

Race detection is therefore a structural analysis problem.

15.9 Policy Separation

The kernel defines: - what interleavings are admissible - what ordering constraints exist

Policy defines: - which admissible step to take next - how to allocate CPU time - which tasks to prioritize

This supports: - cooperative scheduling - preemptive scheduling - work stealing - actor scheduling

All without changing kernel law.

15.10 CGP Requirements

Concurrency must remain representation-invariant:

- No ordering may be implied by textual form
- No synchronization may rely on IR-specific artifacts
- CFG/DFG/Rel forms must preserve the same happens-before constraints

Any representation-dependent interleaving behavior is a CGP failure.

15.11 Status

This document establishes concurrency in UNSOS as a lawful transformation space governed by explicit ordering, capability, and effect constraints. Synchronization is expressed structurally, while scheduling remains a policy choice.

16 4002. Scheduling via Tracks, Tags, and Time

16.1 Purpose

This document defines scheduling in UNSOS as **qualitative state sampling** over execution space, driven by **Tracks**, **Tags**, and explicit time effects.

UNSOS explicitly rejects scheduling models based primarily on raw numeric priorities, counters, or fixed time slices. Instead:

> Scheduling is the selection of admissible execution transformations based on the system's current qualitative posture.

This document operationalizes the design principles established in **0002. Design Principles & Invariants**, particularly: - P1a. Tracks & Tags Over Raw Numbers - P1b. Pay Structure Costs Once; Amortize via Identity, Sharing, and Policy

---

16.2 Scheduling as State Sampling

Scheduling is not a periodic tick-driven mechanism. It is a **state sampling operation**:

1. Observe the current kernel-visible structural state
2. Evaluate relevant Tracks and Tags
3. Apply policy constraints and preferences
4. Select the next admissible execution transformation

Time enters scheduling only as an **explicit effect**, not as an implicit driver.

16.3 Role of Tags

16.3.1 Tags as Admissibility Gates

Tags express qualitative truths that **gate whether execution is admissible**.

Examples: - `task:blocked` - `task:non-preemptible` - `system:maintenance-mode` - `core:offline`

If a Tag forbids execution, no amount of numeric pressure may override it.

Tags therefore define *may / may not* conditions.

16.4 Role of Tracks

16.4.1 Tracks as Preference Signals

Tracks express **directional or progressive system state** that influences *preference*, not admissibility.

Examples: `- fairness:stable → degraded → unstable - latency:nominal → sensitive → critical - gc-pressure:low → rising → high → critical - thermal:nominal → constrained → throttled`

Tracks bias scheduling decisions without introducing hard numeric thresholds.

16.5 Interaction Between Tags and Tracks

UNSOS enforces a strict separation:

- **Tags** determine what *can* run
- **Tracks** influence what *should* run next

This prevents numeric priority inversion and brittle heuristics.

16.6 Time as an Explicit Effect

Time is not ambient. Scheduling decisions that depend on time must:

- consume explicit time-related capabilities
- respect effect ordering constraints

Examples: - timer expiration enabling a waiting task - deadline Tags attached to tasks

No hidden clock-driven behavior is permitted.

16.7 Policy Integration

Scheduling policy operates entirely within admissible space defined by Tags and Tracks.

Policy MAY: - select latency-first vs throughput-first posture - bias toward GC or IO work under pressure - prioritize interactive tasks when tagged

Policy MUST NOT: - override Tag-based admissibility - introduce implicit numeric thresholds

---

16.8 Numeric Metrics (Restricted Role)

Numeric measurements (CPU time, run counts, wait duration) MAY be used only:

- as evidence for Track transitions
- as diagnostic output
- as bounded accounting values

Numeric values must not be used directly as primary scheduling selectors.

16.9 Parallelism and Multi-Core Execution

Tracks and Tags are per-structure, not global.

This enables: - per-core scheduling posture - localized congestion handling - reduced global contention

Parallel execution emerges naturally from local qualitative state.

16.10 Stability and Predictability

Because scheduling decisions are based on qualitative posture:

- behavior is explainable
- oscillations are reduced
- performance cliffs are avoided

Schedulers can report *why* a task ran, not just *that* it ran.

16.11 CGP Requirements

Scheduling semantics must be representation-invariant:

- no dependence on tick frequency
- no reliance on instruction timing
- no representation-specific ordering assumptions

Any divergence across representations or architectures is a CGP failure.

16.12 Status

This document establishes scheduling in UNSOS as track/tag–driven state sampling over execution space, integrating explicit time effects and policy-driven preference selection while rejecting fragile numeric-first heuristics.

17 5000. Authority as Structure

17.1 Purpose

This document defines **authority** in UNSOS as a purely **structural property**, not a mode, flag, or ambient condition.

UNSOS rejects historical security models based on implicit privilege (e.g., kernel/user mode, UID-based trust, ring hierarchies). Instead:

> Authority exists *only* where it is explicitly represented as a relation.

This document establishes the structural model of authority, capability flow, and enforcement that underpins all security properties in UNSOS.

17.2 Authority as Relation

17.2.1 Structural Definition

Authority is represented as a directed, typed relation:

```
Principal -[cap]-> Resource
```

Where: - **Principal** is a structural node (task, process, module, kernel context) - **Resource** is any kernel-visible structure (memory region, device, channel, transformation) - **cap** is a capability edge encoding permitted actions

No authority exists outside these relations.

17.3 Capabilities

17.3.1 Capability Structure

A capability is not a token alone; it is a **relation with constraints**:

- permitted operations
- scope and attenuation rules
- optional expiration or revocation relations

Capabilities are first-class structures subject to UNS-C transformations.

17.3.2 No Ambient Authority

UNSOS forbids ambient authority.

The following do *not* confer authority:

- execution mode (kernel/user)
- memory location
- call stack position
- process identity

All authority must be explicitly represented.

17.4 Capability Creation

Capabilities are created only via admissible transformations:

- initial kernel bootstrap
- explicit delegation
- controlled minting by authorized principals

Each capability creation must declare:

- source of authority
- scope of delegation
- invariants preserved

17.5 Delegation and Attenuation

Capabilities may be delegated by creating new capability relations:

- `PrincipalA -[cap]-> Resource`
- `PrincipalA -[delegate]-> PrincipalB`
- resulting in `PrincipalB -[cap]-> Resource` (possibly attenuated)

Attenuation (reduction of authority) is encouraged and structurally enforced.

17.6 Revocation

Revocation is an admissible, non-invertible transformation.

Revocation mechanisms may include:

- explicit revocation edges
- indirection nodes (revocation lists)
- lease expiration structures

Revocation must: - be representable structurally - not require global scans unless explicitly declared

---

17.7 Authority Over Transformations

In UNSOS, **transformations themselves are resources**.

Performing a transformation requires authority to do so.

This enables: - fine-grained control over kernel operations - sandboxing of kernel-adjacent services - secure extensibility without new modes

---

17.8 Capability Enforcement

The kernel enforces authority by:

- checking the presence of required capability relations
- validating scope and constraints
- rejecting unauthorized transformation requests

Enforcement is structural and deterministic.

17.9 Invariants

17.9.1 A1. Explicit Authority

Every successful operation must be justified by an explicit capability relation.

17.9.2 A2. No Forgery

Capabilities cannot be fabricated by computation; they can only be obtained via admissible transformations.

17.9.3 A3. Least Authority

Delegation should minimize authority by default.

17.9.4 A4. Authority Closure

All authority changes must remain within UNS grammar and UNS-C calculus.

17.10 CGP Requirements

Authority semantics must be representation-invariant:

- no authority inferred from numeric ranges
- no privilege derived from layout or call depth

- no representation-specific enforcement shortcuts

Any such dependency is a CGP failure.

17.11 Failure Modes

Authority failures are classified as:

- **Structural Failure**: implicit or ambient authority introduced
- **Adapter Failure**: incorrect enforcement at representation boundary
- **Policy Failure**: overly broad delegation choices

Structural and adapter failures invalidate security correctness.

17.12 Status

This document establishes authority in UNSOS as an explicit structural relation enforced by the kernel. All security, isolation, and containment properties derive from this model.

18 5001. Isolation & Containment

18.1 Purpose

This document defines **isolation and containment** in UNSOS as emergent properties of **structural authority, ownership, and transformation constraints**, not as special execution modes or privileged memory boundaries.

UNSOS achieves isolation by *removing the possibility of interaction*, not by policing interaction after the fact.

18.2 Isolation as Absence of Relations

18.2.1 Structural Isolation

In UNSOS, two structures are isolated **if and only if** there exists no admissible path of relations or transformations between them.

Isolation is therefore:

- structural (graph separation)
- verifiable (reachability analysis)
- representation-invariant (CGP-safe)

No runtime checks are required once isolation is structurally established.

18.3 Containment

Containment is a *directed* form of isolation.

A structure `A` contains `B` if:

- `A` owns `B`
- `A` controls the capabilities that reach `B`
- `B` cannot acquire relations outside `A` without explicit delegation

Containment is enforced by ownership and capability topology, not by execution context.

18.4 Memory Isolation

Memory isolation is achieved through:

- absence of capability edges granting access
- explicit memory mapping relations (4000)
- handle-based identity (3000)

Address spaces do not imply isolation. Isolation arises from *who can reach what*.

18.5 Execution Isolation

Execution isolation is achieved by:

- disjoint execution graphs
- absence of shared effect tokens
- absence of shared synchronization structures

Preemption or scheduling does not affect isolation; structure does.

18.6 Module Isolation

Modules (drivers, services, plugins) are isolated by:

- capability-limited authority
- resource ownership scoping
- explicit interfaces as capability sets

A module cannot escalate privilege because no implicit escalation path exists.

18.7 Sandboxing

A sandbox is a constrained substructure defined by:

- limited initial capabilities
- restricted resource ownership
- controlled delegation paths

Sandboxes are created by admissible transformations that *do not grant* forbidden relations.

18.8 Communication Across Isolation Boundaries

Communication requires explicit structure:

- message channels
- shared buffers with explicit capability edges
- effect-mediated transformations

There is no implicit shared state.

18.9 Containment Breakage

Breaking containment is impossible unless:

- a new capability relation is introduced
- ownership relations are modified
- a representation adapter violates invariants

Containment violations are therefore **structural failures**, not bugs of logic.

18.10 Revocation and Shrinking

Containment can be tightened over time by:

- revoking capabilities
- reclaiming resources
- collapsing scopes

All are UNS-C admissible transformations.

18.11 CGP Requirements

Isolation and containment must not rely on:

- memory layout
- address ranges
- execution modes
- textual scoping rules

Any such reliance is a CGP failure.

18.12 Failure Modes

- **Structural Failure**: unintended relation or capability introduced
- **Adapter Failure**: enforcement bypassed at representation boundary
- **Policy Failure**: overbroad initial delegation

Structural and adapter failures invalidate isolation guarantees.

18.13 Status

This document establishes isolation and containment in UNSOS as purely structural properties derived from authority, ownership, and transformation constraints.

19 5002. Verification & Security Invariants

19.1 Purpose

This document defines the **security invariants** that must hold in UNSOS and the verification posture required to maintain them.

UNSOS security is not based on "trusted code behaving correctly." It is based on:

- explicit authority as structure (5000)
- isolation/containment as topology (5001)
- admissible transformations as the only way state can change (UNS-C)
- representation invariance as a correctness requirement (CGP)

Therefore, security is verified by proving **structural properties** and enforcing them at transformation boundaries.

19.2 Verification Posture

UNSOS adopts the following verification posture:

1. **Invariant-first**: define security as invariants, not as policies.
2. **Local reasoning**: transformations must be verifiable locally.

3. **Compositional correctness**: global security arises from composition of locally-correct steps.
4. **Fail structurally**: ambiguity or uncertainty results in rejection, not best-effort allowance.
5. **Adapters are suspect**: representation adapters are treated as high-risk boundaries.

19.3 Security Invariants

The following invariants must hold under all admissible transformations.

19.3.1 S1. Explicit Authority

Every successful operation must be justified by explicit capability relations.

There is no ambient privilege.

19.3.2 S2. Capability Non-Forgery

No principal may create a capability relation except through admissible transformations that themselves require authority to do so.

Capabilities cannot be manufactured by computation.

19.3.3 S3. Least Authority Preservation

Delegation transformations must preserve attenuation constraints:

- a delegate may not gain authority not derivable from the delegator
- default delegation must be attenuating unless explicitly overridden by authority

---

19.3.4 S4. Containment Integrity

A contained structure may not acquire relations outside its containment boundary without explicit delegation.

Containment boundaries are preserved under all transformations except those explicitly designated as boundary-modifying.

19.3.5 S5. No Ambient Channels

Communication and influence across principals may only occur through explicit structures:

- channels
- shared buffers with explicit capability edges
- effect-mediated transformations

No hidden side channels are permitted at the kernel law level.

19.3.6 S6. Resource Lifetime Safety

Reclamation, revocation, and closure must not create dangling authority:

- a capability to a reclaimed resource must become invalid
- stale handles must be rejected via generation
- dependent resources must transition coherently

19.3.7 S7. Effect Authorization

Effectful transformations (IO, time, external interaction) must require explicit capabilities.

No principal may observe or influence the external world without explicit authorization.

19.3.8 S8. Representation-Invariant Enforcement (CGP)

Security correctness must not depend on representation artifacts.

Enforcement must survive:

- representation switching (Expr/CFG/DFG/Rel)
- target lowering (x86-64/AArch64)
- optimization passes

Any security rule that only works in one representation is invalid.

19.4 Verification Mechanisms (Structural)

UNSOS verification mechanisms are expressed structurally:

- capability graph checks
- reachability/topology checks
- invariant preservation proofs for transformations

- CGP convergence tests across representations

These mechanisms may be implemented as:

- compile-time proofs
- runtime checks at transformation boundaries
- proof-carrying transformations (8001)

The kernel is allowed to enforce invariants dynamically, but enforcement must be explicit and structurally justified.

19.5 Adapter Verification

Representation adapters (hardware, ABI, traps, root enumeration, driver boundaries) are treated as critical.

Adapter correctness requires:

- explicit mapping from adapter events to kernel transformations
- audits for implicit privilege leakage
- strict separation of adapter-only assumptions from kernel law

Adapters must never introduce new authority.

19.6 Failure Modes

Security failures are classified as:

- **Structural Failure**: invariant violation or ambient authority introduced
- **Adapter Failure**: boundary leaks authority or changes semantics
- **Policy Failure**: overly broad delegation or poor operational choices

Structural and adapter failures invalidate security correctness.

19.7 Status

This document establishes UNSOS security as a set of structural invariants and defines a verification posture based on local, compositional reasoning and CGP-stable enforcement.

20 6000. Unified Resources, Naming, and Projections

> **Normative dependencies:** This document is constrained by `0000. Project Charter & Scope` and `0002. Design Principles & Invariants` (including **Decoherence &**

Outcome Doctrine, **pressure**, and **instability**).

20.1 6000.1 Purpose

This document defines the UNSOS model for:

- a **single unified ontology** for *all* system resources
- a **unified naming and reference grammar** (no paths-as-authority)
- **projections (views)** as first-class, explicitly-labeled interpretations of underlying structure
- **search** as a projection operator, not a separate subsystem

The goal is to replace traditional separations (filesystem vs device namespace vs process space vs "external media") with one coherent structure that can be narrowed, shaped, integrated, and projected under law.

20.2 6000.2 Scope

This document covers:

- Resource ontology and lifecycle
- Temporal admissibility of resources (boot-time only / boot-time optimized / runtime)
- Unified reference grammar (qualifiers: where/what/how/when/why/who)
- Projections: Places, Views, Search, and their reorderability
- External resource integration (high-level, not bus-specific)
- Hot-swap semantics (whenever electrically possible)
- Interpretive resources (including code pages) as projections
- Federation preview: multiple UNSOS instances as tiles on a virtual backplane

This document does **not** fully specify:

- storage internals (see 6001+)
- I/O transport specifics (USB, PCIe, network packetization)
- accelerator compute semantics (GPU / UNS-C-AC execution) beyond naming and integration

20.3 6000.3 Definitions

- **Resource:** Any structured entity admitted into UNSOS such that it can be referenced, projected, transformed, and audited.
- **Integration:** The act of ingesting an external or internal structure into the unified resource space.
- **Projection (View):** An explicitly labeled interpretation of one or more resources.
- **Selector:** A structured reference expression used to identify a resource or resource-set.
- **Temporal admissibility:** The declared timing constraints under which a resource may participate in narrowing/shaping/integration.

20.4 6000.4 Core Assertions

20.4.1 6000.4.1 One ontology

From the kernel's perspective, **all resources are the same class of thing**. This includes (non-exhaustive):

- data and persistent state
- executable functions and applications
- devices and interfaces
- buses and fabrics
- accelerators (GPU, UNS-C-AC)
- remote nodes and virtual instances
- encodings, codecs, and code pages
- policies, profiles, and constraints

UNSOS does not treat these as separate namespaces.

20.4.2 6000.4.2 No mounting

UNSOS does not "mount" external containers into a parallel hierarchy. Instead, it:

1. **ingests** external resources,
2. **narrows** admissible integration forms via policy,
3. **shapes** the resource into native structure,
4. **integrates** it into the unified resource space,
5. and exposes it only through explicit projections.

20.4.3 6000.4.3 Views have shape; storage does not

UNSOS does not assign a single canonical "shape" to storage. **Shape is a property of projections.**

A user may select or define view-shapes such as: tree, flat, graph, Venn/tag, timeline, search-first, place-first, or hybrids. Switching view-shape is a *state sampling choice* that may carry a pressure cost.

20.4.4 6000.4.4 Outcome-first interaction

All resource discovery, naming, projection, and integration operate under the **Decoherence & Outcome Doctrine**:

- No "errors" exist as a kernel concept.
- Undesired outcomes become pressure.
- Instability thresholds govern escalation.

This applies equally to "missing files," unplugged devices, ambiguous encodings, remote disconnections, and inconsistent projections.

20.5 6000.5 Temporal Admissibility of Resources

UNSOS classifies each resource by **when** it may participate in narrowing and shaping.

> **Law:** A resource's temporal availability is an explicit property that governs **when** it may participate in narrowing and shaping, not whether it may exist.

20.5.1 6000.5.1 Boot-time only

Definition: Resources whose participation is limited to boot, and whose continued availability must not be required after handoff.

- sampled to produce facts (manifest/measurements)
- not integrated as live dependencies
- influence initial posture and narrowing only

20.5.2 6000.5.2 Boot-time optimized

Definition: Resources that benefit from early discovery and shaping but remain valid runtime resources.

- integrated early to improve posture selection and scheduling
- may arrive late or disappear later without "failure"
- removal/arrival advances pressure and triggers reshaping

20.5.3 6000.5.3 Runtime-only

Definition: Resources that cannot be assumed at boot and are integrated dynamically.

- always treated as hot-swappable
- never prerequisites for kernel validity

20.5.4 6000.5.4 Hot-swap

> **Requirement:** Hot-swap must be supported whenever electrically possible.

Resource arrival/removal is a normal state transition:

- no broken paths (paths are not authoritative)
- dependent projections may become Novel
- dependent processes accumulate pressure and re-narrow

20.6 6000.6 Unified Reference Grammar (Selectors)

UNSOS replaces path-based naming with **selectors**: structured, composable reference expressions.

Selectors MUST support the following qualifier classes:

- **Where:** origin, locality, scope, place

- **What:** type, structure family, identity class
- **How:** interface, projection, interpretation, protocol, decoding
- **When:** version, timestamp, state slice
- **Why:** provenance, intent, derivation, history
- **Who:** authority context, ownership, policy profile (when relevant)

Notes:

- "When" and "Why" are often implied by the system's history/provenance model, but MUST be expressible explicitly.
- Selectors do not confer authority. Authority is determined by policy and capability invariants.

20.6.1 6000.6.1 Selector forms

Selectors may be expressed through multiple user-facing syntaxes, but MUST compile to a canonical internal form.

Canonical selector features:

- composable filters (AND/OR/NOT)
- typed fields (e.g., `what:function`, `what:device`, `how:utf8`)
- stable identity references (content/structure identity, not location)
- ranked candidate sets (projection may return multiple candidates)

20.6.2 6000.6.2 Candidate resolution as projection

When a selector matches multiple candidates:

- the result is a **candidate-set projection**, not an error
- policy/profile may narrow automatically
- unresolved ambiguity remains explicit (Novel)

20.7 6000.7 Projections: Places, Views, Search

UNSOS exposes resources through projections that may be ordered by user preference.

Supported interaction orders include:

- Places → Views → Search
- Search → Views → Places
- Views → Search → Places

These are not different ontologies; they are different *projection pipelines*.

20.7.1 6000.7.1 Places

A **place** is a stable, human-meaningful anchor for a projection pipeline (e.g., "Work," "Family Photos," "Hardware," "Build Artifacts," "Research"). A place is not a mount point; it is a named

selector + view defaults.

20.7.2 6000.7.2 Views

A **view** is a shape and lens applied to a resource set (tree/flat/graph/Venn/timeline, etc.).

Views are explicitly labeled as projections.

20.7.3 6000.7.3 Search

Search is not a separate subsystem; it is a projection operator over selectors and indexes.

Search results may be:

- exact identity matches
- structural similarity matches (policy-gated)
- candidate sets with explicit Novel when ambiguous

20.8 6000.8 Integration Pipeline (External and Internal)

Any resource entering UNSOS MUST pass through a unified pipeline:

1. **Detect / Observe** (arrival as an outcome)
2. **Narrow** admissible integration forms (policy profile first)
3. **Shape** into native structural representations
4. **Integrate** into unified resource space
5. **Project** through Places/Views/Search
6. **Audit** (provenance, history, rights)

This applies equally to:

- USB/HID/COM devices
- external drives and media
- GPUs and accelerators
- remote nodes and VMs
- imported archives/app capsules
- encoded text and code pages

20.9 6000.9 Deduplication as Identity (Preview)

UNSOS treats deduplication as a consequence of identity, not a storage optimization.

- identical functions need not be installed twice
- identical data need not be saved twice
- common structures (including frequently repeated strings) may be represented once, referenced many times

Deduplication MUST preserve provenance and intent via metadata rather than duplicating content.

(Implementation and storage mechanics are specified in 6001+.)

20.10 6000.10 Interpretive Resources (Code Pages and Beyond)

UNSOS must interoperate with existing encodings and code pages without redefining them.

Rule:

- raw bytes are canonical
- decoding is a projection (`how:encoding`)
- the system may present a "best projection," but MUST label it as such
- ambiguity remains explicit (Novel), not replaced by "??" or silent substitution

Code pages and decoders are treated as first-class resources and may be narrowed/shaped by policy.

20.11 6000.11 Federation Preview: Nodes as Tiles

UNSOS supports multi-node configurations by treating each UNSOS instance as a **tile on a virtual backplane**.

- the fabric may be LAN/WAN/PCIe/shared memory/VM substrate
- cross-instance interaction is mediated by a **deterministic translation adapter** expressed in UNS terms
- remote resources are integrated through the same pipeline as local external resources

Partitions and disconnections are outcomes that:

- advance pressure
- produce Novel until resolved
- trigger posture change under instability thresholds

Full federation semantics are specified in the 7xxx series.

20.12 6000.12 Requirements Summary

UNSOS MUST:

- model all entities as resources in a unified ontology
- reject mounting as a primitive; use ingestion/narrowing/shaping/integration
- expose shape only via projections; storage has no canonical shape
- support temporal admissibility classes and hot-swap whenever electrically possible
- provide selector-based naming with where/what/how/when/why/who qualifiers
- treat ambiguity as explicit candidate-set projections with Novel
- treat encodings/code pages as interpretive resources with labeled projections
- preserve outcome-first semantics: no errors, only outcomes; pressure/instability govern escalation

20.13 6000.13 Open Questions (Deferred)

- Canonical internal selector representation family (syntax is flexible; semantics are not)
- Default view-shape sets per policy profile
- Cross-instance selector resolution semantics (federation adapter design)
- Indexing strategies that avoid "brittle global tables" while supporting fast projection

---

End of 6000.

21 6001. Storage as Shared Structural Space

> **Normative dependencies:** This document is constrained by `0000. Project Charter & Scope`, `0002. Design Principles & Invariants` (including **Decoherence & Outcome Doctrine**, **pressure**, **instability**, **reversibility by default**), and `6000. Unified Resources, Naming, and Projections`.

21.1 6001.1 Purpose

This document specifies how UNSOS treats persistence and storage as a **single shared structural space** rather than:

- a hierarchical filesystem
- a collection of mounted containers
- a set of device-local allocation domains

Storage in UNSOS is the persistent continuation of the unified resource space. "Files," "directories," "volumes," and "devices" are projections over this shared structural substrate.

21.2 6001.2 Core Assertions

21.2.1 6001.2.1 No canonical storage shape

Storage has **no canonical user-visible shape**. Any apparent shape (tree, flat, graph, tag/Venn, timeline) is a projection defined in `6000`.

21.2.2 6001.2.2 Persistence is structural identity

Persistence is the act of committing **structural identity** into durable media.

- location is not identity
- device is not identity
- path is not identity

A persisted object is referenced by its identity and provenance, not by where it happens to reside.

21.2.3 6001.2.3 "Save" is integration

A save operation is not "write bytes to a file." It is:

1. shaping transient state into admissible persistent structure
2. integrating that structure into shared storage space
3. producing one or more projections for user interaction

21.2.4 6001.2.4 Outcome-first semantics

Storage operations follow the **Decoherence & Outcome Doctrine**:

- there is no storage "failure," only outcomes
- ambiguous states are explicit (Novel)
- pressure and instability govern escalation and pruning

21.3 6001.3 Storage Substrate Model

UNSOS storage is modeled as a persistent graph of structured entities:

- **Nodes:** typed structures (data objects, functions, policies, projections, manifests)
- **Edges:** typed relationships (contains, derives, references, depends-on, projects-as)
- **Annotations:** provenance, intent, rights, timestamps, version lineage

This graph MUST be representation-invariant (CGP): multiple internal encodings may exist, but they must converge on identical structural semantics.

21.4 6001.4 Identity, Versions, and Lineage

21.4.1 6001.4.1 Identity classes

UNSOS distinguishes (at minimum) the following identity classes:

- **Structural identity:** the structure and content of an object
- **Provenance identity:** how the object was derived and by whom/what
- **Intent identity:** why the object exists (declared purpose, policy context)

The canonical reference identity MAY be composite, but MUST remain stable under representation changes.

21.4.2 6001.4.2 Versioning as default

Because UNSOS is reversible by default, **version lineage** is intrinsic to storage.

- every commit appends to lineage
- destructive overwrite is an explicit, audited act
- pruning is an explicit user act unless instability forces it

21.4.3 6001.4.3 When/Why qualifiers

Selectors (`6000`) MUST be able to refer to:

- current projection
- prior versions n- branch/lineage forks (policy-gated)
- snapshots ("state slices")

21.5 6001.5 Deduplication as a Consequence of Identity

UNSOS treats deduplication as a *structural consequence*:

- if an object's structural identity already exists, it is not duplicated
- saving produces new lineage metadata, not new content
- identical substructures (including repeated strings) may be represented once and referenced many times

Deduplication MUST preserve provenance and intent by attaching distinct annotations, not by duplicating the underlying structure.

21.5.1 6001.5.1 Multi-granularity dedupe

Dedupe may occur at multiple granularities:

- whole-object identity
- substructure identity
- canonical atom identity (e.g., common tokens/strings)

The system MUST avoid a brittle "global lookup table" mentality. Identity resolution is structural, policy-scoped, and pressure-managed.

21.6 6001.6 External Media and Device Participation

External media is not mounted. It participates as a resource with temporal admissibility (`6000.5`).

21.6.1 6001.6.1 Integration forms

A storage-capable device MAY be integrated in forms such as:

- **Contributing store:** adds durable capacity to the shared space
- **Imported artifact:** ingested as a bounded subgraph (archive, capsule)
- **Ephemeral cache:** contributes temporary capacity (policy-gated)

The chosen form is determined by policy profile and user posture.

21.6.2 6001.6.2 Removal semantics

Device removal is an outcome:

- objects whose durability depended solely on the device become Novel until resolved
- projections remain, but may degrade to candidate sets
- pressure advances until the system re-shapes or the user intervenes

No “broken path” exists because location is not identity.

21.7 6001.7 Reversibility, Pruning, and Collapse

21.7.1 6001.7.1 Reversibility by default

UNSOS assumes nothing is lost. Practically, physical limits require collapse and pruning.

21.7.2 6001.7.2 Collapse

Collapse reduces active state to a minimum viable baseline:

- summarizes inactive history
- retains reversible anchors
- reduces resident working set

Collapse is encouraged by policy and driven by pressure.

21.7.3 6001.7.3 Pruning

Pruning is the explicit elimination of lineage segments and/or substructures.

Rules:

- pruning is user-initiated unless instability forces it
- forced pruning targets obvious non-essentials first
- ambiguous pruning requires human override
- all pruning is audited and reversible *until* the physical act is committed

21.8 6001.8 Pressure Tracks for Storage

Storage MUST maintain pressure tracks at appropriate scopes, including:

- growth pressure (unbounded accumulation)
- access pressure (hot vs cold divergence)
- projection pressure (excessive reshaping without coherence)
- durability pressure (too many single-device dependencies)
- reversibility depth pressure (history volume)

Pressure is relieved by:

- deduplication
- collapse
- pruning
- posture change

- device integration form changes

Instability thresholds govern when the system must seek human guidance.

21.9 6001.9 Determinism and Auditability

Given identical:

- integrated storage graph
- policy profile
- posture
- pressure tracks

…the system MUST produce identical:

- identity resolutions
- deduplication decisions
- collapse/prune candidate sets
- default projections

All decisions MUST be auditable via provenance and pressure history.

21.10 6001.10 Requirements Summary

UNSOS storage MUST:

- be a persistent continuation of unified resource space
- model persistence as structured identity + provenance + intent
- provide intrinsic version lineage under reversibility-by-default
- deduplicate as a consequence of identity at multiple granularities
- integrate external media without mounting
- treat device removal as outcome producing Novel and pressure, not failure
- support collapse and pruning as pressure-driven coherence mechanisms
- remain deterministic and auditable

21.11 6001.11 Open Questions (Deferred)

- Canonical internal storage encoding families and their CGP proofs
- Indexing strategies compatible with structural identity (avoid brittle global tables)
- Policies for collapse/prune default behavior under different profiles
- Cross-node persistence and federation resolution (7xxx)

End of 6001.

22 6002. Views, Search, and Projection Mechanics

Normative dependencies: This document is constrained by `0000. Project Charter & Scope`, `0002. Design Principles & Invariants`, `6000. Unified Resources, Naming, and Projections`, and `6001. Storage as Shared Structural Space`.

22.1 6002.1 Purpose

This document defines how UNSOS exposes resources to users and applications through **projections**, and how **views** and **search** operate as projection mechanics rather than independent subsystems.

The goal is to ensure that: - no projection is mistaken for ground truth, - ambiguity is preserved explicitly (Novel), - interaction remains deterministic and auditable, - and navigation scales with complexity rather than collapsing under it.

22.2 6002.2 Core Assertions

22.2.1 6002.2.1 Projection-first interaction

All user-visible representations of system state are projections. There is no privileged "raw" or "true" view available to users or applications.

22.2.2 6002.2.2 Search is a projection operator

Search is not a separate service layered atop storage. It is a **projection operator** applied to the unified resource space using selectors and indexes.

22.2.3 6002.2.3 Ambiguity is preserved

When multiple valid interpretations exist: - the system MUST not silently select one, - the result MUST be expressed as a candidate-set projection, - Novel MUST remain explicit until resolved.

22.3 6002.3 Projection Pipeline Model

All projections follow the same abstract pipeline:

1. **Selector construction** (explicit or implicit)
2. **Candidate discovery** (structural identity matching)
3. **Policy narrowing** (profile-driven admissibility)
4. **View shaping** (structural lens application)
5. **Ranking (optional)** (deterministic, auditable)
6. **Presentation** (explicitly labeled projection)

Each stage may contribute pressure or Novel.

22.4 6002.4 Views

22.4.1 6002.4.1 Definition

A **view** is a structural lens that: - selects a subset of resources, - imposes a shape (tree, graph, flat, Venn, timeline, etc.), - defines grouping and ordering rules, - and specifies default interaction affordances.

A view does **not** own resources and does **not** define identity.

22.4.2 6002.4.2 View shape classes

UNSOS does not prescribe a fixed set of view shapes, but implementations MUST support at least:

- hierarchical (tree)
- flat list
- graph / dependency
- tag / Venn-style overlap
- temporal (timeline / lineage)

Hybrid and user-defined shapes are admissible.

22.4.3 6002.4.3 View mutability

Views are mutable projections: - reshaping a view does not alter underlying storage - switching views is a state sampling choice - excessive reshaping without convergence advances pressure

22.5 6002.5 Places

22.5.1 6002.5.1 Definition

A **place** is a stable, named anchor for a projection pipeline. It bundles: - a base selector, - default view shape(s), - policy hints, - and interaction affordances.

Examples: “Work”, “Media”, “Hardware”, “Research”, “Build Outputs”.

22.5.2 6002.5.2 Places are not locations

A place is not a directory, mount point, or path. It is a reusable projection definition.

22.6 6002.6 Search

22.6.1 6002.6.1 Search semantics

Search operates by: - constructing selectors from user intent, - matching against structural identity and annotations, - returning candidate sets as projections.

Search results are always labeled as projections.

22.6.2 6002.6.2 Ranking

Ranking MAY be applied, but MUST: - be deterministic, - be auditable, - never collapse ambiguity without justification.

Policy profiles may influence ranking strategies.

22.7 6002.7 Indexing

Indexes exist to accelerate projection, not to define truth.

Rules: - indexes are derivative and disposable - index inconsistency produces pressure, not failure - rebuilding an index is always admissible

Index selection and maintenance is pressure-managed.

22.8 6002.8 Candidate Sets and Novel

22.8.1 6002.8.1 Candidate sets

When multiple resources satisfy a selector: - the result is a candidate set - the projection MUST surface ambiguity explicitly

22.8.2 6002.8.2 Novel propagation

Novel values propagate through projections: - unresolved ambiguity remains visible - downstream transformations must acknowledge Novel

22.9 6002.9 Determinism and Auditability

Given identical: - unified resource space - selectors - views - policy profiles

…the system MUST produce identical projections and rankings.

All projection decisions MUST be traceable via provenance and pressure history.

22.10 6002.10 Pressure and Instability in Projections

Projection-related pressure includes: - excessive ambiguity without resolution - frequent reshaping without convergence - over-broad selectors - conflicting policy constraints

Instability thresholds may trigger: - suggestion of narrower views - request for user clarification - posture shifts

22.11 6002.11 Requirements Summary

UNSOS MUST:

- treat all user-visible representations as projections
- implement search as a projection operator

- preserve ambiguity explicitly via candidate sets and Novel
- support mutable, user-definable view shapes
- provide places as reusable projection anchors
- maintain deterministic, auditable projection behavior

22.12 6002.12 Open Questions (Deferred)

- Default projection ergonomics per policy profile
- UI affordances for candidate-set interaction
- Index eviction strategies under memory pressure

End of 6002.

23 6003. External Resource & I/O Integration

> **Normative dependencies:** This document is constrained by `0000. Project Charter & Scope`, `0002. Design Principles & Invariants` (including **Decoherence & Outcome Doctrine**, **pressure**, **instability**), and the 6000–6002 series.

23.1 6003.1 Purpose

This document defines how UNSOS integrates **external resources and I/O**—including devices, interfaces, buses, networks, and accelerators—into the unified resource space without introducing special-case namespaces, mounting semantics, or failure modes.

External resources are treated as **participants in system structure**, not peripherals attached to it.

23.2 6003.2 Core Assertions

23.2.1 6003.2.1 No external/internal distinction

From the kernel's perspective, there is no fundamental distinction between internal and external resources. Externality is a property of **origin and temporal admissibility**, not of ontology.

23.2.2 6003.2.2 Integration, not attachment

External resources are never "attached," "mounted," or "opened." They are:

1. detected as outcomes,
2. narrowed by policy and posture,
3. shaped into admissible structural forms,
4. integrated into unified resource space,
5. exposed only through projections.

23.2.3 6003.2.3 I/O is structural transformation

I/O is not a side-channel. It is a **structural transformation** between resource domains, governed by UNS-C rules.

23.3 6003.3 External Resource Classes

UNSOS does not hard-code device categories, but implementations MUST support integration of at least:

- human interface devices (HID)
- storage-capable devices
- communication interfaces (serial, network, logical streams)
- accelerators (GPU, UNS-C-AC)
- sensors and actuators
- remote UNSOS instances (federation preview)

All classes follow the same integration law.

23.4 6003.4 Temporal Admissibility (Recap)

Each external resource declares or is assigned a temporal admissibility class:

- **Boot-time only**
- **Boot-time optimized**
- **Runtime-only**

Temporal admissibility governs *when* a resource may participate in narrowing and shaping, not whether it may exist.

23.5 6003.5 Integration Pipeline

Every external resource MUST pass through the following pipeline:

1. **Detect / Observe** – arrival is an outcome
2. **Describe** – capabilities, endpoints, signals
3. **Narrow** – policy profile constrains admissible forms
4. **Shape** – resource mapped into native structural representations
5. **Integrate** – admitted into unified resource space
6. **Project** – exposed via places/views/search
7. **Audit** – provenance, history, rights recorded

Failure at any stage produces Novel and pressure, never kernel failure.

23.6 6003.6 Endpoint Shaping

23.6.1 6003.6.1 Endpoint abstraction

External devices often expose endpoints (e.g., HID reports, USB interfaces, network sockets). UNSOS treats endpoints as **raw signal surfaces** that must be shaped before integration.

23.6.2 6003.6.2 Shaping rules

Endpoint shaping MUST:

- preserve signal semantics
- eliminate transport-specific artifacts
- produce deterministic structural representations
- declare uncertainty explicitly

Multiple shaped forms MAY exist concurrently under different projections.

23.7 6003.7 Hot-swap and Removal

23.7.1 6003.7.1 Hot-swap as default

Hot-swap is the default behavior for all non–boot-time-only resources and MUST be supported whenever electrically possible.

23.7.2 6003.7.2 Removal semantics

Resource removal is an outcome:

- dependent projections may become Novel
- dependent processes accumulate pressure
- no broken paths or invalid handles exist

Escalation occurs only under instability thresholds.

23.8 6003.8 Performance-Sensitive I/O

23.8.1 6003.8.1 Fast-path shaping

For high-throughput devices (e.g., GPUs, displays, network interfaces), UNSOS MAY employ fast-path shaping:

- bypassing unnecessary metadata layers
- using pre-shaped buffers
- deferring full integration when appropriate

Fast-paths MUST remain auditable and reversible.

23.8.2 6003.8.2 Bidirectional translation

UNSOS supports bidirectional translation layers where appropriate:

- high-speed rendering and display
- accelerator offload (GPU, UNS-C-AC)
- signal-domain computation

Such translation layers are treated as UNS-C–admissible transformations.

23.9 6003.9 Security and Authority Boundaries

External resources do not receive implicit authority.

Rules:

- all authority is conferred by policy and capability
- shaped endpoints declare permissible state effects
- no external computation may alter state not declared as part of its integration contract

This applies equally to accelerators and remote nodes.

23.10 6003.10 Determinism and Auditability

Given identical:

- external signals
- integration pipeline configuration
- policy profiles

…the system MUST produce identical shaped resources and projections.

All integration decisions MUST be auditable via provenance and pressure history.

23.11 6003.11 Requirements Summary

UNSOS MUST:

- integrate external resources into unified resource space
- reject mounting/attachment as primitives
- treat I/O as structural transformation
- support hot-swap whenever electrically possible
- shape endpoints deterministically before integration
- support fast-paths without violating auditability
- preserve outcome-first semantics

23.12 6003.12 Open Questions (Deferred)

- Bus-specific descriptor normalization (USB, PCIe, etc.)

- Performance bounds of fast-path shaping
- Cross-node I/O federation semantics

End of 6003.

24 6004. Reversibility, Deduplication, and Pruning

> **Normative dependencies:** This document is constrained by `0000. Project Charter & Scope`, `0002. Design Principles & Invariants` (including **Decoherence & Outcome Doctrine**, **pressure**, **instability**, **reversibility by default**), and the 6000–6003 series.

24.1 6004.1 Purpose

This document specifies how UNSOS enforces **reversibility by default**, and how **deduplication** and **pruning** emerge as lawful, pressure-managed mechanisms for maintaining coherence under physical and operational constraints.

UNSOS does not treat data loss, overwrite, or deletion as implicit or convenient actions. Any irreversible act is explicit, auditable, and justified by pressure or instability.

24.2 6004.2 Core Assertions

24.2.1 6004.2.1 Reversibility is the default state

All system evolution is assumed reversible unless explicitly declared otherwise.

24.2.2 6004.2.2 Deduplication is identity recognition

Deduplication is not an optimization pass; it is the recognition of identical structure.

24.2.3 6004.2.3 Pruning is an outcome, not a failure

Pruning is a managed response to pressure and instability, not an error condition.

24.3 6004.3 Reversibility Model

24.3.1 6004.3.1 Structural reversibility

Structural reversibility means:

- prior states remain referenceable
- transformations preserve lineage
- collapse does not erase identity

Reversibility applies to:

- data
- functions
- policies
- projections
- integration forms

24.3.2 6004.3.2 Explicit irreversibility

Irreversible actions MUST:

- be explicitly declared
- be policy-gated
- be audited
- produce pressure acknowledging loss of reversibility

Examples include cryptographic destruction, physical media disposal, and user-confirmed permanent erasure.

24.4 6004.4 Deduplication

24.4.1 6004.4.1 Identity-driven deduplication

UNSOS deduplicates when structural identity matches:

- whole-object identity
- substructure identity
- canonical atom identity

Deduplication produces shared references, not merged provenance.

24.4.2 6004.4.2 Cross-domain deduplication

Deduplication applies across:

- applications
- data
- functions
- policies
- imported artifacts

No domain is privileged.

24.4.3 6004.4.3 Policy and dedupe

Policy profiles MAY:

- limit dedupe scope
- require isolation despite identity

- trade storage pressure for provenance separation

24.5 6004.5 Pressure-Managed Collapse

24.5.1 6004.5.1 Collapse definition

Collapse is the reduction of active state into a minimum viable baseline while preserving reversibility anchors.

Collapse:

- summarizes inactive lineage
- evicts cold state from residency
- retains reconstruction paths

24.5.2 6004.5.2 Collapse triggers

Collapse is triggered by pressure tracks including:

- memory pressure
- storage growth pressure
- reversibility depth pressure
- projection churn pressure

Collapse is automatic, deterministic, and auditable.

24.6 6004.6 Pruning

24.6.1 6004.6.1 Pruning definition

Pruning is the explicit removal of lineage segments or structures such that reversibility is no longer possible.

24.6.2 6004.6.2 Pruning rules

Rules:

- pruning is user-initiated unless instability forces it
- forced pruning targets obvious non-essentials first
- ambiguous pruning requires human intervention
- all pruning decisions are audited

24.6.3 6004.6.3 Instability-driven pruning

When instability thresholds are exceeded:

- the system MUST seek resolution
- if no resolution is available, controlled pruning is admissible
- pruning MUST minimize irreversible loss

24.7 6004.7 Interaction with External Resources

Pruning MUST account for:

- removable media
- transient devices
- federated resources

External removal does not constitute pruning; it produces Novel and pressure until resolved or collapsed.

24.8 6004.8 Determinism and Auditability

Given identical:

- unified resource space
- policy profiles
- pressure history

…the system MUST produce identical:

- deduplication decisions
- collapse candidates
- pruning candidate sets

All irreversible acts MUST be traceable via audit logs.

24.9 6004.9 Requirements Summary

UNSOS MUST:

- assume reversibility by default
- deduplicate as a consequence of identity
- collapse state under pressure while preserving anchors
- prune only explicitly or under instability
- audit all irreversible acts
- remain deterministic and explainable

24.10 6004.10 Open Questions (Deferred)

- User interaction ergonomics for pruning decisions
- Granularity of irreversible confirmation thresholds
- Cross-node pruning semantics (federation)

End of 6004.

25 6005. Interpretive Resources (Encodings, Code Pages, Decoders)

Normative dependencies: This document is constrained by `0000. Project Charter & Scope`, `0002. Design Principles & Invariants` (including **Decoherence & Outcome Doctrine**, **pressure**, **instability**), and the 6000–6004 series.

25.1 6005.1 Purpose

This document defines how UNSOS treats **interpretation itself** as a first-class resource. Encodings, code pages, decoders, parsers, renderers, and similar mechanisms are not implicit system behaviors; they are **explicit interpretive resources** applied through projections.

The goal is to eliminate silent misinterpretation, lossy decoding, and undefined behavior when encountering ambiguous or underspecified data.

25.2 6005.2 Core Assertions

25.2.1 6005.2.1 Bytes are canonical

Raw byte sequences are canonical. Meaning is never assumed.

25.2.2 6005.2.2 Interpretation is projection

All interpretation of bytes into symbols, structures, or signals occurs through **explicit projections** using interpretive resources.

25.2.3 6005.2.3 No silent substitution

UNSOS MUST NOT silently substitute placeholder glyphs, replacement characters, or inferred values that erase ambiguity.

25.3 6005.3 Interpretive Resources

25.3.1 6005.3.1 Definition

An **interpretive resource** is any resource that maps:

- raw data → structured representation
- signal → symbol
- bytes → characters
- stream → frames

Examples include:

- text encodings (UTF-8, UTF-16, legacy code pages)
- binary parsers
- media codecs

- protocol decoders
- format interpreters

25.3.2 6005.3.2 Identity and provenance

Interpretive resources have:

- structural identity (ruleset)
- provenance (origin, version, authority)
- declared admissibility (what they may interpret)

They are subject to deduplication and versioning like any other resource.

25.4 6005.4 Code Pages

25.4.1 6005.4.1 Respect for existing standards

UNSOS MUST interoperate with existing code pages and encodings. It does not redefine their structure or semantics.

25.4.2 6005.4.2 Code page application

Applying a code page is a projection:

- `how:encoding=utf8`
- `how:encoding=cp1252`
- `how:encoding=unknown`

If a data source does not declare its encoding, the system MAY offer candidate projections but MUST preserve ambiguity explicitly.

25.4.3 6005.4.3 Best projection rule

UNSOS MAY present a **best projection** based on context, history, and policy, but MUST:

- label it explicitly as a projection
- preserve access to raw bytes
- expose uncertainty (Novel)

25.5 6005.5 Ambiguity and Novel

25.5.1 6005.5.1 Novel in interpretation

When interpretation is ambiguous:

- Novel MUST be produced
- ambiguity MUST be visible to downstream consumers
- pressure accumulates if ambiguity remains unresolved

25.5.2 6005.5.2 Downstream responsibility

Consumers of interpreted data MUST acknowledge Novel or explicitly constrain interpretation.

25.6 6005.6 Policy and Interpretation

Policy profiles MAY:

- restrict admissible encodings
- prefer specific decoders
- forbid heuristic inference
- require human confirmation for ambiguous interpretation

Policy does not remove ambiguity; it governs how it is handled.

25.7 6005.7 Determinism and Auditability

Given identical:

- raw data
- interpretive resources
- policy profiles

…the system MUST produce identical interpreted structures and Novel markings.

All interpretive decisions MUST be auditable via provenance and pressure history.

25.8 6005.8 Interaction with Search and Views

Interpretation affects:

- search indexing
- projection shapes
- rendering

Indexes MAY exist per-interpretation but MUST be labeled accordingly.

25.9 6005.9 Failure Elimination

Misinterpretation is not a failure; it is an outcome.

UNSOS replaces:

- decode errors
- replacement glyphs
- silent truncation

with:

- explicit Novel

- pressure-managed resolution
- auditable interpretive choice

25.10 6005.10 Requirements Summary

UNSOS MUST:

- treat interpretation as a first-class resource
- preserve raw data as canonical
- apply encodings and decoders only through projections
- forbid silent ambiguity erasure
- surface Novel explicitly
- remain deterministic and auditable

25.11 6005.11 Open Questions (Deferred)

- UI ergonomics for interpretive ambiguity
- Heuristic suggestion boundaries under different policy profiles
- Cross-node interpretive consistency (federation)

End of 6005.

26 6100. The UNSOS Console: Interactive Projection Interface

> **Normative dependencies:** This document is constrained by `0000. Project Charter & Scope`, `0002. Design Principles & Invariants`, and the UNSOS corpus (6000–6005, 8000, 9000). It defines the mandatory human-facing console for UNSOS without introducing a general UI/UX layer.

26.1 6100.1 Purpose

This document defines the UNSOS console as a **human interaction projection** over the unified resource space.

The console is not a shell, REPL, or keyword-driven command interpreter in the classical sense. It is an **interactive projection interface** that applies interpretation, narrowing, and outcome resolution to human intent.

Its role is to provide a deterministic, explainable, and safe human control surface consistent with UNSOS laws.

26.2 6100.2 Core Assertions

26.2.1 6100.2.1 The console is a projection

The console presents no ground truth. All interactions occur through projections applied to the unified resource space.

26.2.2 6100.2.2 Human input is ambiguous by default

Human utterances are inherently ambiguous. Ambiguity is treated as Novel, not error.

26.2.3 6100.2.3 Interaction must complete

All console-initiated actions are subject to the same completion guarantees as any other system action.

26.3 6100.3 Interpretive Pipeline

The console operates as a deterministic interpretive pipeline:

1. **Ingress**: raw input bytes are captured verbatim
2. **Tokenization**: symbols, phrases, and structural markers are extracted
3. **Intent framing**: verb–subject–object candidates are constructed
4. **Candidate expansion**: possible target resources and transformations are enumerated
5. **Policy narrowing**: inadmissible actions are removed
6. **Outcome preview**: remaining candidates are presented as labeled projections
7. **Execution**: selected transformation is applied and resolved

No stage may silently discard ambiguity.

26.4 6100.4 Interactive Fiction Model

The console intentionally adopts principles from interactive fiction (IF) systems:

- context-sensitive parsing
- disambiguation dialogues ("Which one do you mean?")
- state-dependent meaning
- explicit explanation of why an action cannot proceed

This model aligns naturally with UNSOS concepts of Novel, pressure, and completion.

26.5 6100.5 Commands as Transformations

There are no intrinsic commands.

Every console action resolves to one or more **candidate transformations** expressed in UNS-C terms.

The console does not execute text; it executes **selected transformations**.

26.6 6100.6 Disambiguation and Novel

When multiple interpretations or targets exist:

- the console MUST present candidate sets
- each candidate MUST be explainable
- Novel MUST be visible

Automatic selection is permitted only when policy explicitly allows it.

26.7 6100.7 Outcome Preview

Before execution, the console SHOULD present an outcome preview:

- affected resources
- reversibility depth impact
- pressure implications
- potential instability risks

Preview is the default mode of interaction.

26.8 6100.8 Completion and Boundary Reporting

If an action cannot complete:

- the console MUST report the explicit boundary
- the explanation MUST reference constraints and pressure
- the system state MUST remain recoverable

Messages such as “permission denied” or “invalid command” are insufficient.

26.9 6100.9 History, Audit, and Reversibility

Console history is not textual. It is a structured sequence of transformations with provenance.

Users may:

- inspect prior actions
- replay projections
- reverse admissible transformations

26.10 6100.10 Learnability and Guidance

The console MAY expose projections such as:

- "what can I do here?"
- "why is this not allowed?"
- "what would reduce pressure?"

These are projections over state, not hard-coded help text.

26.11 6100.11 Determinism and Safety

Given identical:

- system state
- policy profile
- input sequence

…the console MUST produce identical candidate sets and outcomes.

No console action may mutate state outside declared transformation scope.

26.12 6100.12 Non-Goals

The console does not:

- define a graphical UI
- perform probabilistic language inference
- guess user intent beyond declared candidates
- bypass policy or authority

26.13 6100.13 Requirements Summary

The UNSOS console MUST:

- operate as a projection and interpretation interface
- surface ambiguity explicitly
- narrow choices deterministically
- present outcome previews
- guarantee completion or boundary declaration
- integrate fully with audit and reversibility

End of 6100.

27 7000. Federation: Tiles, Backplanes, and Translation Layers

> **Normative dependencies:** This document is constrained by `0000. Project Charter & Scope`, `0002. Design Principles & Invariants` (including **Decoherence & Outcome Doctrine**, **pressure**, **instability**), and the 6000–6005 series.

27.1 7000.1 Purpose

This document defines how multiple UNSOS instances cooperate without collapsing determinism, authority, or coherence. Federation is treated as **structural integration**, not as distributed exception handling or shared-state illusion.

UNSOS federation allows multiple instances to function as **tiles on a virtual backplane**, regardless of the physical or logical substrate connecting them.

27.2 7000.2 Core Assertions

27.2.1 7000.2.1 No "distributed system" special case

UNSOS does not introduce a separate class of rules for distributed systems. Federation follows the same ingestion, narrowing, shaping, integration, and projection laws as all other resources.

27.2.2 7000.2.2 Local determinism is preserved

Each UNSOS instance remains locally deterministic. Cross-instance interaction is treated as **external input**, never as implicit shared state.

27.2.3 7000.2.3 Federation is reversible

Federation relationships are reversible by default. Disconnecting a node is an outcome that advances pressure but does not invalidate local state.

27.3 7000.3 Tiles

27.3.1 7000.3.1 Tile definition

A **tile** is a complete UNSOS instance operating under its own policy profile, posture, pressure history, and authority boundaries.

Tiles may represent:

- physical machines
- virtual machines
- embedded systems
- accelerator-backed compute domains
- sandboxed UNSOS partitions

27.3.2 7000.3.2 Tile autonomy

Each tile:

- owns its local state
- enforces its own policies
- evaluates pressure independently
- may accept or reject federation offers

No tile is required to trust another tile.

27.4 7000.4 Backplanes

27.4.1 7000.4.1 Backplane definition

A **backplane** is any substrate that permits signal exchange between tiles. Examples include:

- local networks
- wide-area networks
- PCIe fabrics
- shared memory buses
- VM hypervisor channels

The backplane does not define semantics; it merely transports signals.

27.4.2 7000.4.2 Backplane neutrality

UNSOS does not assume reliability, ordering, or latency guarantees from the backplane. Such properties are surfaced as Novel and pressure, not hidden assumptions.

27.5 7000.5 Translation Layers

27.5.1 7000.5.1 Translation layer definition

A **translation layer** is a deterministic adapter that maps:

- outbound structures → transmissible signals
- inbound signals → admissible local structures

Translation layers are expressed as UNS-C–admissible transformations.

27.5.2 7000.5.2 Bidirectional translation

Translation is bidirectional and symmetric:

- outbound translation declares what state may be affected remotely
- inbound translation declares what effects are admissible locally

No undeclared state mutation is permitted.

27.6 7000.6 Federation Lifecycle

27.6.1 7000.6.1 Discovery

Tiles may discover each other via explicit configuration or dynamic observation. Discovery is an outcome and may produce Novel.

27.6.2 7000.6.2 Offer and narrowing

Federation begins with an offer:

- declared capabilities
- declared intent
- declared authority scope

Policy profiles narrow admissible federation forms.

27.6.3 7000.6.3 Integration

Accepted federation offers are integrated as external resources:

- remote capabilities become referenceable resources
- access is projection-based
- authority is strictly bounded

27.6.4 7000.6.4 Suspension and removal

Federation may be suspended or removed:

- voluntarily

- due to pressure
- due to instability

Suspension produces Novel and pressure, not failure.

27.7 7000.7 Consistency and Novel

27.7.1 7000.7.1 No global consistency assumption

UNSOS does not assume global consistency. Divergence is expected and tracked.

27.7.2 7000.7.2 Novel propagation

Unacknowledged remote effects produce Novel:

- pending transformations
- uncertain reads
- delayed commits

Resolution collapses Novel deterministically.

27.8 7000.8 Pressure and Instability Across Tiles

Pressure sources include:

- latency
- partition duration
- conflicting projections
- repeated unresolved Novel

Instability thresholds may trigger:

- posture changes
- federation narrowing
- human intervention
- controlled disengagement

27.9 7000.9 Security and Authority

27.9.1 7000.9.1 Authority boundaries

Federation does not imply shared authority. Each tile enforces its own authority model.

27.9.2 7000.9.2 Capability contracts

All cross-tile effects MUST be declared in capability contracts enforced by translation layers.

27.10 7000.10 Determinism and Auditability

Given identical:

- tile state
- translation layers
- policy profiles
- signal history

…the system MUST produce identical federation outcomes.

All federation interactions MUST be auditable.

27.11 7000.11 Requirements Summary

UNSOS MUST:

- treat federation as structural integration
- preserve local determinism
- model nodes as autonomous tiles
- treat transport substrates as neutral backplanes
- enforce explicit translation layers
- propagate Novel and pressure across tiles
- avoid global consistency assumptions

27.12 7000.12 Open Questions (Deferred)

- Federation posture presets
- Multi-tile projection ergonomics
- Long-lived partition handling strategies

End of 7000.

28 8000. Decoherence, Completion, and Outcome Resolution

> **Normative dependencies:** This document is constrained by `0000. Project Charter & Scope`, `0002. Design Principles & Invariants` (including **Decoherence & Outcome Doctrine**, **pressure**, **instability**), and all prior series (2000–7000).

28.1 8000.1 Purpose

This document defines how UNSOS replaces classical notions of *failure*, *error handling*, *exceptions*, and *recovery* with a single, coherent framework based on **decoherence recognition**, **completion**, and **outcome resolution**.

UNSOS does not attempt to prevent undesirable outcomes. Instead, it guarantees that **all system evolution completes meaningfully**, either by restoring coherence, escalating posture, or explicitly identifying an unmanageable boundary.

28.2 8000.2 Core Assertions

28.2.1 8000.2.1 Failure does not exist

UNSOS has no concept of failure as a terminal or exceptional state.

All events are outcomes. Outcomes may be undesired, but they are still valid system states.

28.2.2 8000.2.2 Completion is mandatory

Every initiated system action MUST complete in one of the following forms:

- coherence restoration
- posture change
- explicit boundary declaration

Infinite silent non-completion is structurally impossible.

28.2.3 8000.2.3 Decoherence is observable

Decoherence is not an abstract notion. It is detected via tracked pressure, Novel persistence, and instability thresholds.

28.3 8000.3 Decoherence

28.3.1 8000.3.1 Definition

Decoherence is the condition in which system evolution no longer converges toward stable, internally consistent state.

Decoherence is indicated by:

- unrelieved pressure accumulation
- repeated ineffective transformations
- unresolved Novel
- conflicting invariants

28.3.2 8000.3.2 Decoherence is not an error

Decoherence represents information about system limits, not malfunction.

28.4 8000.4 Completion

28.4.1 8000.4.1 Completion definition

Completion is the act of bringing an initiated action to a meaningful terminus.

A completed action:

- leaves the system in a well-defined state
- preserves determinism
- advances or resolves pressure

28.4.2 8000.4.2 Completion pathways

Completion may occur through:

1. successful coherence restoration
2. transformation of the problem space
3. escalation to posture change
4. explicit declaration of an unmanageable boundary

28.5 8000.5 Outcome Resolution

28.5.1 8000.5.1 Resolution definition

Outcome resolution is the process by which UNSOS evaluates an outcome and determines how to proceed toward completion.

Resolution is iterative and pressure-guided.

28.5.2 8000.5.2 Iterative resolution

UNSOS MAY attempt multiple transformations to resolve an outcome, analogous to an intelligent compilation loop:

- attempt transformation
- observe result
- adjust approach

Repeated ineffective attempts increase pressure.

28.6 8000.6 Pressure and Instability

28.6.1 8000.6.1 Pressure as guidance

Pressure tracks guide resolution by:

- discouraging repetition
- prioritizing relief of instability
- bounding iteration

28.6.2 8000.6.2 Instability thresholds

When instability thresholds are exceeded:

- automatic resolution halts
- human intervention is requested
- controlled collapse or pruning may be triggered

28.7 8000.7 Boundary Recognition

28.7.1 8000.7.1 Explicit boundaries

When resolution is not possible within admissible constraints, UNSOS MUST explicitly declare:

- what outcome occurred
- why it cannot be managed further
- which constraints are violated

This declaration itself is a completed outcome.

28.7.2 8000.7.2 No silent abandonment

UNSOS MUST NOT abandon actions silently or mask unresolvable states.

28.8 8000.8 Determinism and Explainability

Given identical:

- initial state
- pressure history
- policy profile

UNSOS MUST produce identical resolution paths and boundary declarations.

All resolution steps MUST be explainable and auditable.

28.9 8000.9 Relationship to Coherence-AI

Outcome resolution is the primary manifestation of **Coherence-AI**.

It is:

- deterministic
- non-stochastic
- explainable
- pressure-governed

UNSOS does not guess. It narrows.

28.10 8000.10 Requirements Summary

UNSOS MUST:

- abolish failure as a system concept

- guarantee completion of all initiated actions
- detect and track decoherence
- resolve outcomes iteratively under pressure guidance
- escalate under instability rather than looping indefinitely
- declare explicit boundaries when resolution is impossible

28.11 8000.11 Open Questions (Deferred)

- Ergonomics of boundary declaration for users
- Resolution strategy tuning per policy profile
- Federation-wide decoherence handling

End of 8000.

29 9000. Posture, Policy, and System Evolution

> **Normative dependencies:** This document is constrained by `0000. Project Charter & Scope`, `0002. Design Principles & Invariants`, and the complete UNSOS corpus (2000–8000).

29.1 9000.1 Purpose

This document defines how UNSOS evolves over time without sacrificing determinism, coherence, or auditability. It formalizes **posture**, **policy**, and **system evolution** as first-class, governed mechanisms rather than ad-hoc configuration, tuning, or administrative intervention.

UNSOS does not "optimize itself" opportunistically. It **narrows and reshapes its behavior lawfully** in response to pressure, instability, and accumulated history.

29.2 9000.2 Core Assertions

29.2.1 9000.2.1 Posture is explicit

At any moment, UNSOS occupies a **posture**: a coherent configuration of constraints, priorities, and admissible transformations.

Posture is: - explicit - inspectable - auditable - reversible (unless explicitly collapsed)

29.2.2 9000.2.2 Policy constrains possibility

Policy does not dictate outcomes. It constrains the **space of admissible outcomes**.

29.2.3 9000.2.3 Evolution is structural

System evolution occurs through lawful transitions between postures, driven by pressure and resolved through completion.

29.3 9000.3 Posture

29.3.1 9000.3.1 Definition

A **posture** is a bounded operating stance that determines:

- resource admissibility
- scheduling and execution bias
- storage and reversibility behavior
- projection defaults
- federation openness

29.3.2 9000.3.2 Posture transitions

Posture transitions:

- are deterministic
- may be automatic or user-initiated
- advance pressure if resisted
- are auditable

Posture change is the primary mechanism for system adaptation.

29.3.3 9000.3.3 Examples

Illustrative (non-exhaustive) postures:

- exploratory
- conservative
- constrained
- archival
- high-throughput
- isolation-heavy

These are descriptive labels, not hard-coded modes.

29.4 9000.4 Policy

29.4.1 9000.4.1 Policy definition

Policy is a structured set of constraints applied during narrowing, shaping, and resolution.

Policy governs:

- authority boundaries
- admissible interpretations
- federation scope
- deduplication and pruning behavior
- ambiguity tolerance

29.4.2 9000.4.2 Policy profiles

Policies may be grouped into **profiles** applied before user interaction:

- child profile
- guest profile
- conservative profile
- experimental profile

Profiles perform early narrowing, not late enforcement.

29.4.3 9000.4.3 Policy evolution

Policy itself is a resource:

- versioned
- auditable
- reversible

Policy change produces pressure proportional to its impact.

29.5 9000.5 System Evolution

29.5.1 9000.5.1 Evolution mechanics

UNSOS evolves through:

- posture transitions
- policy refinement
- integration of new resources
- pruning under instability

No evolution occurs implicitly.

29.5.2 9000.5.2 Learning without stochasticity

UNSOS adapts by:

- recording outcome history
- tracking pressure relief effectiveness
- preferring transformations that restore coherence

This produces **deterministic adaptation**, not probabilistic learning.

29.6 9000.6 Human Interaction

29.6.1 9000.6.1 Stability threshold

A stable system resists human override.

An unstable system requests it.

29.6.2 9000.6.2 Override as outcome

Human override is:

- explicit
- audited
- pressure-relieving

Overrides do not violate determinism; they enter the system as declared outcomes.

29.7 9000.7 Federation and Evolution

In federated systems:

- each tile maintains its own posture
- shared policy may be negotiated
- divergence is expected and tracked

Global posture is never assumed.

29.8 9000.8 Determinism and Auditability

Given identical:

- posture history
- policy versions
- pressure history

…the system MUST produce identical evolution paths.

All evolution decisions MUST be explainable.

29.9 9000.9 Requirements Summary

UNSOS MUST:

- treat posture as a first-class operating state
- constrain behavior via policy rather than imperative control
- evolve only through explicit, auditable transitions
- adapt deterministically using pressure and history
- resist unnecessary human intervention

29.10 9000.10 Open Questions (Deferred)

- Canonical posture taxonomies
- UX for posture inspection and transition
- Long-term federation policy drift handling

End of 9000.