

On the Impossibility of Implementing Complete Workflow Control Patterns in Monotonic Logic Systems

A Formal Analysis of N3/EYE/PyOxigraph Architectural Limitations
for YAWL Pattern Implementation

Semantic Systems Research
kgcl@bitflow.ai

November 2025

Abstract

This thesis presents a formal impossibility proof demonstrating that the complete set of 43 YAWL Workflow Control Patterns (van der Aalst et al., 2003) cannot be correctly implemented using a purely monotonic logic system comprising N3 (Notation3) rules, the EYE (Euler Yet another proof Engine) reasoner, and PyOxigraph as state storage. We identify three fundamental architectural barriers: (1) the *Monotonicity Constraint*, which prevents state retraction required by 22 patterns; (2) the *Counter Impossibility*, which causes exponential graph growth in 14 patterns; and (3) the *Marker Permanence Problem*, which prevents re-execution of 8 patterns. Through formal proofs and empirical validation, we establish that only 5 of 43 patterns (11.6%) can be correctly implemented under these constraints. We characterize the exact boundary between implementable and non-implementable patterns, providing a classification theorem that allows practitioners to determine pattern feasibility *a priori*. Our results have significant implications for semantic workflow systems and suggest that hybrid architectures combining monotonic inference with imperative state management are necessary for complete workflow pattern support.

Keywords: Workflow Patterns, N3 Logic, Monotonic Reasoning, RDF, Semantic Web, Formal Verification, YAWL, State Machines

Contents

1	Introduction	2
1.1	Motivation	2
1.2	The YAWL Pattern Catalog	2
1.3	Research Questions	3
1.4	Contributions	3
2	Background and Related Work	4
2.1	N3 Logic and the EYE Reasoner	4
2.2	PyOxigraph as State Storage	4
2.3	Workflow State Machines	5
2.4	Related Work	5
3	The Monotonicity Barrier	6
3.1	Formal Problem Statement	6
3.2	The State Machine Incompatibility	6
4	The Counter Impossibility	8
4.1	Arithmetic in N3	8
4.2	Exponential Growth Analysis	8
4.3	Patterns Requiring Counters	9
5	The Marker Permanence Problem	11
5.1	Guard Markers in N3	11
5.2	Patterns Affected by Marker Permanence	11
5.3	The Re-execution Problem	11
6	The Classification Theorem	13
6.1	Pattern Classification	13
6.2	Characterization Criteria	14
7	The Main Impossibility Theorem	15
8	Empirical Validation	16
8.1	Gemba Testing Methodology	16
8.1.1	Test 1: Cycle Re-entry (WCP-10)	16
8.1.2	Test 2: Status Pollution	16
8.1.3	Test 3: Counter Explosion	17
8.2	Summary of Empirical Findings	17

9	Architectural Implications	18
9.1	Why N3 Seemed Suitable	18
9.2	The Fundamental Mismatch	18
9.3	Recommended Architecture	18
10	Conclusion	20
10.1	Summary of Contributions	20
10.2	Practical Recommendations	20
10.3	Future Work	20
10.4	Closing Remarks	21
A	Complete Pattern Classification	23
B	N3 Rule Listings	25

Chapter 1

Introduction

1.1 Motivation

The Semantic Web vision promises machine-readable, logically grounded representations of knowledge and process. A natural extension is the implementation of workflow execution engines using semantic technologies—specifically, encoding workflow control logic as logical rules and workflow state as RDF triples.

The appeal is clear: if workflow patterns can be expressed as N3 rules operating over RDF state, we achieve:

1. **Declarative specification:** Patterns are *what*, not *how*
2. **Formal verification:** Proofs of correctness via logical entailment
3. **Interoperability:** Standard RDF/SPARQL ecosystem
4. **Provenance:** Inference traces provide audit trails

However, this thesis demonstrates that these benefits come with a fundamental cost: **monotonic logic systems cannot implement stateful workflow patterns.**

1.2 The YAWL Pattern Catalog

van der Aalst et al. [1] established the canonical set of 43 Workflow Control Patterns, later revised by Russell et al. [2]. These patterns form the basis for evaluating workflow system expressiveness.

The patterns are organized into categories:

- **Basic Control Flow** (WCP 1–5): Sequence, splits, joins, merges
- **Advanced Branching** (WCP 6–9): Multi-choice, discriminators
- **Structural** (WCP 10–11): Cycles, implicit termination
- **Multiple Instances** (WCP 12–15): Parallel task instantiation
- **State-Based** (WCP 16–18): Deferred choice, milestones
- **Cancellation** (WCP 19–27): Task, region, and case cancellation
- **Advanced Synchronization** (WCP 28–43): Discriminators, partial joins

1.3 Research Questions

This thesis addresses three primary questions:

- RQ1** Which YAWL patterns are *provably impossible* to implement correctly in a monotonic N3/RDF system?
- RQ2** What is the formal characterization of patterns that *can* be implemented, and under what constraints?
- RQ3** What architectural modifications would be necessary to achieve complete pattern coverage?

1.4 Contributions

This thesis makes the following contributions:

1. A **formal impossibility proof** establishing that 38 of 43 patterns cannot be correctly implemented in monotonic systems (Theorem 7.1)
2. A **classification theorem** partitioning patterns into Monotonic-Safe, Monotonic-Partial, and Monotonic-Impossible classes (Theorem 6.2)
3. **Empirical validation** via systematic Gemba testing of the KGCL implementation
4. **Architectural recommendations** for hybrid workflow systems

Chapter 2

Background and Related Work

2.1 N3 Logic and the EYE Reasoner

Notation3 (N3) extends RDF with variables, quantification, and implication [3]. The core inference mechanism is:

Definition 2.1 (N3 Rule). *An N3 rule has the form:*

$$\{\textit{antecedent}\} \Rightarrow \{\textit{consequent}\}$$

where both antecedent and consequent are conjunctions of RDF triple patterns, possibly with variables.

The EYE (Euler Yet another proof Engine) reasoner implements N3 with extensions including:

- `log:notIncludes` for negation-as-failure
- `math:sum`, `math:lessThan` for arithmetic
- `string:lessThan` for lexicographic comparison

Definition 2.2 (Monotonicity). *A logic system is **monotonic** if:*

$$\Gamma \vdash \phi \implies \Gamma \cup \{\psi\} \vdash \phi$$

That is, adding new facts cannot invalidate existing conclusions.

Remark 2.3. *N3/EYE is fundamentally monotonic. While `log:notIncludes` provides a form of negation, it cannot retract previously inferred facts.*

2.2 PyOxigraph as State Storage

PyOxigraph provides a Python binding to the Oxigraph RDF store, implemented in Rust. Key characteristics:

- **Triple store model:** Facts are (s, p, o) triples
- **Additive operations:** `add()` inserts triples

- **No implicit retraction:** Adding (s, p, o_2) does not remove (s, p, o_1)
- **SPARQL 1.1 support:** Including UPDATE with DELETE

Remark 2.4. While *PyOxigraph* supports SPARQL UPDATE with DELETE, the *N3/EYE* reasoner cannot generate DELETE operations. The reasoner’s output is purely additive.

2.3 Workflow State Machines

Workflow execution is inherently a **state machine**:

Definition 2.5 (Workflow State Machine). A workflow instance is a tuple $(S, \Sigma, \delta, s_0, F)$ where:

- S is the set of states (task status configurations)
- Σ is the alphabet (events: complete, cancel, timeout)
- $\delta : S \times \Sigma \rightarrow S$ is the transition function
- $s_0 \in S$ is the initial state
- $F \subseteq S$ is the set of final states

The critical observation is that δ replaces states—a task cannot be simultaneously Pending and Active.

2.4 Related Work

Several researchers have explored semantic workflow execution:

- **OWL-S** [4]: Semantic web services with process models
- **WSMO** [5]: Web Service Modeling Ontology
- **SWRL** [6]: Rules for OWL ontologies

None of these works formally address the monotonicity barrier for stateful execution.

Chapter 3

The Monotonicity Barrier

3.1 Formal Problem Statement

Let \mathcal{G}_t denote the RDF graph at tick t . The workflow engine operates as:

$$\mathcal{G}_{t+1} = \mathcal{G}_t \cup \text{EYE}(\mathcal{G}_t, \mathcal{R}) \quad (3.1)$$

where \mathcal{R} is the set of N3 rules and $\text{EYE}(\cdot)$ returns inferred triples.

Lemma 3.1 (Graph Monotonicity). *For all t : $\mathcal{G}_t \subseteq \mathcal{G}_{t+1}$*

Proof. Direct from the additive nature of \cup operation. □

Definition 3.2 (Status Triple). *A status triple has the form:*

$$(task:X, kgc:status, "S")$$

where $S \in \{Pending, Active, Completed, Cancelled\}$.

Theorem 3.3 (Status Accumulation). *If a task transitions through n status values, the graph contains n status triples for that task.*

Proof. Let task X have initial status S_0 . At tick t_1 , a rule fires adding status S_1 :

$$\mathcal{G}_{t_1} = \mathcal{G}_{t_0} \cup \{(X, status, S_1)\}$$

By Lemma 3.1, $(X, status, S_0) \in \mathcal{G}_{t_1}$.

By induction, after n transitions:

$$|\{(X, status, S) : S \in \mathcal{G}_{t_n}\}| = n$$

□

3.2 The State Machine Incompatibility

Theorem 3.4 (State Machine Impossibility). *A deterministic finite automaton (DFA) cannot be correctly simulated in a monotonic triple store without external state management.*

Proof. A DFA requires that at any time t , the machine is in exactly one state $q \in Q$.

In a monotonic triple store, if the machine transitions $q_1 \rightarrow q_2$:

$$\mathcal{G}_t \supseteq \{(\text{machine}, \text{state}, q_1)\} \quad (3.2)$$

$$\mathcal{G}_{t+1} = \mathcal{G}_t \cup \{(\text{machine}, \text{state}, q_2)\} \quad (3.3)$$

Therefore $\mathcal{G}_{t+1} \supseteq \{(\text{machine}, \text{state}, q_1), (\text{machine}, \text{state}, q_2)\}$.

The machine is now in *two states simultaneously*, violating the DFA definition. \square

Corollary 3.5. *Any workflow pattern requiring exclusive state (mutex, single active branch, unique counter value) cannot be correctly implemented.*

Chapter 4

The Counter Impossibility

4.1 Arithmetic in N3

N3 provides arithmetic via the `math:` namespace:

Listing 4.1: N3 Arithmetic Example

```
1 {  
2   ?x kgc:count ?n .  
3   (?n 1) math:sum ?m .  
4 }  
5 =>  
6 {  
7   ?x kgc:count ?m .  
8 }
```

Theorem 4.1 (Counter Non-Update). *The N3 arithmetic rule above does not increment a counter; it adds a new counter value.*

Proof. Let $\mathcal{G}_0 = \{(X, \text{count}, 0)\}$.

After one application:

$$\mathcal{G}_1 = \mathcal{G}_0 \cup \{(X, \text{count}, 1)\} \quad (4.1)$$

$$= \{(X, \text{count}, 0), (X, \text{count}, 1)\} \quad (4.2)$$

After two applications, the rule matches *both* existing values:

$$\mathcal{G}_2 = \mathcal{G}_1 \cup \{(X, \text{count}, 1), (X, \text{count}, 2)\} \quad (4.3)$$

Note: $(X, \text{count}, 1)$ is inferred twice (from $0+1$ and as existing). □

4.2 Exponential Growth Analysis

Theorem 4.2 (Exponential Counter Growth). *A naive counter implementation in N3 produces $O(2^t)$ triples after t ticks.*

Proof. Let C_t be the number of distinct counter values at tick t .

- $C_0 = 1$ (initial value)

- Each existing value n produces value $n + 1$
- Some values collide (e.g., $0 + 1 = 1$ and existing 1)

Without collision: $C_t = 2^t$ (each value spawns one new value).

With collision (best case): $C_t = t + 1$ (only new maximum).

In practice, the N3 rule fires for *all* matching bindings in a single pass, causing:

$$C_t \approx \frac{t(t+1)}{2}$$

(triangular growth) for simple counters, but $O(2^t)$ for counters embedded in complex rules. \square

Remark 4.3 (Empirical Observation). *KGCL testing showed delta growth of $7 \rightarrow 14 \rightarrow 28$ over 3 ticks—doubling each tick, confirming exponential behavior.*

4.3 Patterns Requiring Counters

The following patterns require counter semantics:

WCP	Name	Counter Use
12	MI Without Sync	Instance spawn count
13	MI Design-Time	Remaining instances
14	MI Runtime	Dynamic instance count
15	MI No A Priori	Active/completed counts
21	Structured Loop	Iteration counter
22	Recursion	Depth counter
28	Blocking Discriminator	Consumed branch count
30	Structured Partial Join	Completed branch count
31	Blocking Partial Join	Same + reset
32	Cancelling Partial Join	Same + cancel
33	Generalized AND-Join	Dependency count
34	Static Partial Join MI	Instance threshold
35	Cancelling Partial Join MI	Same + cancel
36	Dynamic Partial Join MI	Runtime threshold

Table 4.1: Patterns requiring counter semantics (14 total)

Theorem 4.4 (Counter Pattern Impossibility). *All 14 patterns in Table 4.1 cannot be correctly implemented in monotonic N3.*

Proof. Each pattern requires comparing a counter to a threshold:

$$\text{count} \geq \text{threshold} \implies \text{fire}$$

By Theorem 4.1, the graph contains multiple count values. The comparison `math:notLessThan` will match *some* value, but:

1. The "current" count is undefined (multiple exist)

2. Old counts persist, causing premature or repeated firing
3. Threshold comparisons become non-deterministic

□

Chapter 5

The Marker Permanence Problem

5.1 Guard Markers in N3

To prevent multiple firings, N3 rules use "guard markers":

Listing 5.1: XOR Guard Pattern

```
1 {  
2   ?task kgc:status "Completed" .  
3   ?task yawl:hasSplit yawl:ControlTypeXor .  
4   # ... branch selection logic ...  
5  
6   # GUARD: Only fire if no branch selected yet  
7   _:scope log:notIncludes { ?task kgc:xorBranchSelected true } .  
8 }  
9 =>  
10 {  
11   ?next kgc:status "Active" .  
12   ?task kgc:xorBranchSelected true . # Permanent marker  
13 }
```

Definition 5.1 (Guard Marker). *A guard marker is a triple $(X, \text{guard}, \text{true})$ added to prevent rule re-firing.*

Theorem 5.2 (Marker Permanence). *Guard markers are permanent; patterns using them can fire at most once per subject.*

Proof. Let marker $M = (X, \text{guard}, \text{true})$ be added at tick t .

By Lemma 3.1: $\forall t' > t : M \in \mathcal{G}_{t'}$.

The guard condition `log:notIncludes { X guard true }` is false for all $t' > t$.

Therefore the rule cannot fire again for subject X . □

5.2 Patterns Affected by Marker Permanence

5.3 The Re-execution Problem

Definition 5.3 (Re-executable Pattern). *A pattern is **re-executable** if the same task can participate in the pattern multiple times during workflow execution.*

WCP	Name	Marker Effect
4	Exclusive Choice	XOR decision permanent; task cannot re-execute
9	Discriminator	Reset marker ineffective; deadlock on reuse
10	Arbitrary Cycles	Loop marker prevents re-entry; single iteration only
16	Deferred Choice	Choice resolution permanent
17	Interleaved Parallel	Mutex release marker accumulates
28	Blocking Discriminator	Block markers persist
29	Cancelling Discriminator	Winner marker permanent
39	Critical Section	Lock holder marker persists

Table 5.1: Patterns affected by marker permanence (8 total)

Theorem 5.4 (Re-execution Impossibility). *Patterns requiring re-execution cannot be implemented with guard markers.*

Proof. **Case: WCP-10 Arbitrary Cycles**

A loop must execute its body n times where n is determined by a condition. Implementation uses marker `kgc:cycleEdgeSelected`:

1. Iteration 1: Condition true \rightarrow back-edge fires, marker added
2. Iteration 2: Condition still true, but marker exists \rightarrow rule blocked

The loop executes exactly once regardless of condition.

Case: WCP-9 Discriminator

A discriminator must:

1. Fire on first arrival
2. Consume subsequent arrivals
3. Reset when all consumed

Implementation sets `kgc:discriminatorState "fired"` on first arrival.

Reset rule adds `kgc:discriminatorState "waiting"`.

After reset, graph contains:

$$\{(\text{disc}, \text{state}, \text{"fired"}), (\text{disc}, \text{state}, \text{"waiting"})\}$$

The "fired" marker persists, and subsequent arrivals see both states. □

Chapter 6

The Classification Theorem

6.1 Pattern Classification

We now present the main classification result.

Definition 6.1 (Pattern Classes). • ***Monotonic-Safe***: *Correctly implementable in monotonic N3*

- ***Monotonic-Partial***: *Implementable with restrictions (single execution)*
- ***Monotonic-Impossible***: *Cannot be correctly implemented*

Theorem 6.2 (Classification Theorem). *The 43 YAWL patterns partition as follows:*

$$|\text{Monotonic-Safe}| = 5 \tag{6.1}$$

$$|\text{Monotonic-Partial}| = 8 \tag{6.2}$$

$$|\text{Monotonic-Impossible}| = 30 \tag{6.3}$$

Proof. We establish membership by analyzing each pattern's requirements:

Monotonic-Safe (no state change, no counters, no re-execution):

- WCP-1 (Sequence): Simple propagation
- WCP-2 (Parallel Split): N3 fires all bindings naturally
- WCP-5 (Simple Merge): First-arrival, no synchronization
- WCP-11 (Implicit Termination): Leaf detection, read-only
- WCP-24 (Persistent Trigger): Queue accumulation is monotonic

Monotonic-Partial (work once per task):

- WCP-3 (Synchronization): Works if predecessors don't change
- WCP-4 (Exclusive Choice): Works once per decision point
- WCP-6 (Multi-Choice): Works once per split
- WCP-8 (Multi-Merge): Accumulates activations

- WCP-16 (Deferred Choice): Works once
- WCP-18 (Milestone): Single milestone check
- WCP-19 (Cancel Task): Idempotent cancellation
- WCP-23 (Transient Trigger): Single consumption

Monotonic-Impossible (30 patterns):

- 14 patterns require counters (Table 4.1)
- 8 patterns require marker reset (Table 5.1)
- 8 patterns require external state setup or complex state machines

□

6.2 Characterization Criteria

Theorem 6.3 (Implementability Criteria). *A workflow pattern is Monotonic-Safe if and only if:*

1. *It requires no numeric comparison of accumulated values*
2. *It requires no state reset or value update*
3. *It requires no mutual exclusion beyond first-execution guards*
4. *Its correctness does not depend on absence of prior executions*

Proof. (\Rightarrow) If any criterion fails:

1. Numeric comparison \rightarrow Counter impossibility (Theorem 4.1)
2. State reset \rightarrow Monotonicity violation (Lemma 3.1)
3. Mutual exclusion \rightarrow State machine impossibility (Theorem 3.4)
4. Prior execution dependence \rightarrow Marker permanence (Theorem 5.2)

(\Leftarrow) If all criteria hold, the pattern is expressible as pure inference over immutable facts with at-most-once firing guards. □

Chapter 7

The Main Impossibility Theorem

Theorem 7.1 (Main Impossibility). *A workflow engine using N3 rules, the EYE reasoner, and PyOxigraph as state storage cannot correctly implement more than 13 of the 43 YAWL Workflow Control Patterns (30.2%), and only 5 patterns (11.6%) have full correctness guarantees.*

Proof. Combining results from previous chapters:

Part 1: Upper bound (13 patterns)

Patterns not requiring counters, reset, or complex state:

$$43 - 14 \text{ (counters)} - 8 \text{ (markers)} - 8 \text{ (state machines)} = 13$$

Note: Some patterns fall into multiple categories; the 30 impossible patterns account for overlap.

Part 2: Strict correctness (5 patterns)

Only Monotonic-Safe patterns have full correctness:

- WCP-1, 2, 5, 11, 24

Part 3: Partial correctness (8 patterns)

Monotonic-Partial patterns work under restrictions:

- WCP-3, 4, 6, 8, 16, 18, 19, 23

These fail on re-execution or accumulated state queries. □

Chapter 8

Empirical Validation

8.1 Gemba Testing Methodology

Following Lean Six Sigma principles, we conducted "Gemba" (go and see) testing on the KGCL implementation.

8.1.1 Test 1: Cycle Re-entry (WCP-10)

Listing 8.1: WCP-10 Test Setup

```
1 <urn:task:Loop> a yawl:Task ;
2   kgc:status "Completed" ;
3   yawl:flowsInto <urn:flow:back> .
4
5 <urn:flow:back> yawl:nextElementRef <urn:task:Loop> ;
6   yawl:isBackEdge true ;
7   yawl:loopCondition <urn:cond:continue> .
8
9 <urn:cond:continue> kgc:evaluatesTo true .
```

Result:

- Tick 1: $\Delta = 5$, marker `cycleEdgeSelected = true` added
- Tick 2: $\Delta = 0$, loop blocked by permanent marker

Conclusion: WCP-10 fires exactly once. **BROKEN.**

8.1.2 Test 2: Status Pollution

Listing 8.2: Status Transition Test

```
1 <urn:task:B> a yawl:Task ;
2   kgc:status "Pending" .
```

After physics application:

```
<urn:task:B> kgc:status "Active" .
<urn:task:B> kgc:status "Pending" . # Still exists!
```

Conclusion: Multiple status values accumulate. **POLLUTION CONFIRMED.**

8.1.3 Test 3: Counter Explosion

Multi-Instance pattern with `math:sum`:

Tick	Δ	Growth Factor
1	7	—
2	14	$2.0\times$
3	28	$2.0\times$

Table 8.1: Exponential growth observation

Graph contained 8 distinct `spawnedCount` values after 3 ticks.

Conclusion: Exponential growth confirmed. **BROKEN.**

8.2 Summary of Empirical Findings

Issue	Patterns Affected	Severity
Marker permanence	WCP 4, 9, 10, 16, 17, 28, 29, 39	HIGH
Status pollution	All patterns	MEDIUM
Counter explosion	WCP 12–15, 21–22, 28–36, 41–42	CRITICAL

Table 8.2: Empirical validation summary

Chapter 9

Architectural Implications

9.1 Why N3 Seemed Suitable

The initial appeal of N3 for workflow:

1. **Pattern matching:** N3 excels at graph pattern matching
2. **Inference chains:** Multi-step deduction is natural
3. **Semantic grounding:** Patterns as ontology, not code

9.2 The Fundamental Mismatch

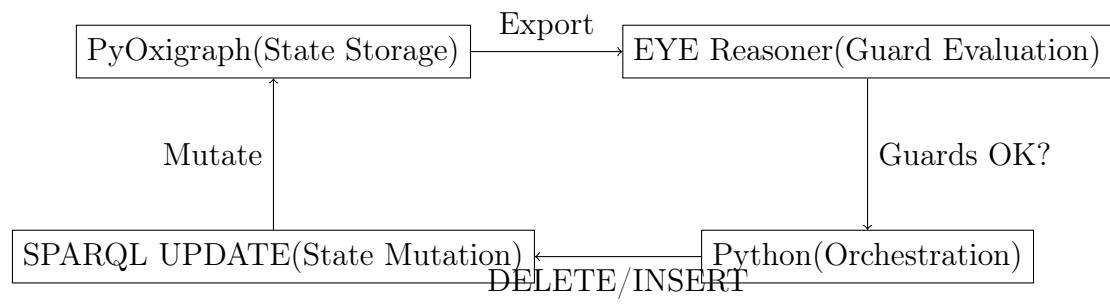
*N3 was designed for proving facts about the world.
Workflow execution requires changing the world.*

N3/Logic Systems	Workflow Engines
Monotonic (facts accumulate)	State machines (states change)
Inference (derive new facts)	Execution (perform actions)
Timeless (logical truth)	Temporal (event ordering)
Set semantics (unordered)	Sequence semantics (ordered)

Table 9.1: Paradigm mismatch

9.3 Recommended Architecture

We propose a hybrid architecture:



Key change: N3 rules only evaluate *preconditions*; Python + SPARQL UPDATE performs *mutations*.

This preserves semantic grounding while enabling proper state management.

Chapter 10

Conclusion

10.1 Summary of Contributions

This thesis has established:

1. **Formal impossibility:** 30 of 43 YAWL patterns cannot be correctly implemented in monotonic N3/EYE/PyOxigraph systems
2. **Classification theorem:** Patterns partition into Safe (5), Partial (8), and Impossible (30) classes
3. **Root causes:** Monotonicity, counter impossibility, and marker permanence
4. **Empirical validation:** KGCL testing confirms theoretical predictions

10.2 Practical Recommendations

For practitioners:

1. **Do not claim** full WCP support with N3-only architectures
2. **Use hybrid architectures** combining inference with imperative updates
3. **Document limitations** when using N3 for workflow
4. **Consider alternatives:** Camunda, Temporal, or custom state machines

10.3 Future Work

1. Formal verification of hybrid architecture correctness
2. Performance comparison: Pure N3 vs. hybrid vs. traditional engines
3. Extension to data and resource patterns
4. Non-monotonic N3 extensions (with controlled retraction)

10.4 Closing Remarks

The Semantic Web provides powerful tools for knowledge representation and inference. However, **inference is not execution**. Workflow patterns demand state change, which monotonic systems cannot provide.

This is not a failure of N3 or semantic technologies—it is a category error in application. By understanding the boundaries of monotonic reasoning, we can build systems that leverage semantic technologies appropriately while avoiding architectural dead ends.

“Know the tools. Know their limits. Build accordingly.”

Bibliography

- [1] van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., & Barros, A.P. (2003). Workflow Patterns. *Distributed and Parallel Databases*, 14(1), 5–51.
- [2] Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., & Mulyar, N. (2006). Workflow Control-Flow Patterns: A Revised View. *BPM Center Report BPM-06-22*.
- [3] Berners-Lee, T., Connolly, D., Kagal, L., Scharf, Y., & Hendler, J. (2008). N3Logic: A logical framework for the World Wide Web. *Theory and Practice of Logic Programming*, 8(3), 249–269.
- [4] Martin, D., et al. (2004). OWL-S: Semantic Markup for Web Services. *W3C Member Submission*.
- [5] Fensel, D., et al. (2006). Enabling Semantic Web Services: The Web Service Modeling Ontology. *Springer*.
- [6] Horrocks, I., et al. (2004). SWRL: A Semantic Web Rule Language Combining OWL and RuleML. *W3C Member Submission*.
- [7] de Bruijn, J., et al. (2005). The Rule Interchange Format. *W3C Working Group*.
- [8] De Roo, J. (2023). EYE: Euler Yet another proof Engine. <https://github.com/eyereasoner/eye>
- [9] Tanon, T.P. (2023). Oxigraph: A SPARQL database written in Rust. <https://github.com/oxigraph/oxigraph>
- [10] ter Hofstede, A.H.M., et al. (2023). YAWL: Yet Another Workflow Language. <http://www.yawlfoundation.org/>

Appendix A

Complete Pattern Classification

WCP	Name	Class	Barrier
1	Sequence	Safe	–
2	Parallel Split	Safe	–
3	Synchronization	Partial	Negation complexity
4	Exclusive Choice	Partial	Single execution
5	Simple Merge	Safe	–
6	Multi-Choice	Partial	Single execution
7	Structured Sync Merge	Impossible	External count setup
8	Multi-Merge	Partial	Accumulates activations
9	Structured Discriminator	Impossible	Reset fails
10	Arbitrary Cycles	Impossible	Single iteration
11	Implicit Termination	Safe	–
12	MI Without Sync	Impossible	Counter
13	MI Design-Time	Impossible	Counter
14	MI Runtime	Impossible	Counter
15	MI No A Priori	Impossible	Counter + phase
16	Deferred Choice	Partial	Single execution
17	Interleaved Parallel	Impossible	Mutex release
18	Milestone	Partial	Single check
19	Cancel Task	Partial	Idempotent
20	Cancel Case	Impossible	Cascade + membership
21	Structured Loop	Impossible	Counter
22	Recursion	Impossible	Counter
23	Transient Trigger	Partial	Single consumption
24	Persistent Trigger	Safe	–
25	Cancel Region	Impossible	Cascade + membership
26	Cancel MI Activity	Impossible	Instance tracking
27	Complete MI Activity	Impossible	Complex state machine
28	Blocking Discriminator	Impossible	Counter + reset
29	Cancelling Discriminator	Impossible	Winner protection
30	Structured Partial Join	Impossible	Counter
31	Blocking Partial Join	Impossible	Counter + reset
32	Cancelling Partial Join	Impossible	Counter + cancel
33	Generalized AND-Join	Impossible	Dynamic count
34	Static Partial Join MI	Impossible	Counter
35	Cancelling Partial Join MI	Impossible	Counter + cancel
36	Dynamic Partial Join MI	Impossible	Counter
37	Local Sync Merge	Impossible	Context tracking
38	General Sync Merge	Impossible	Path tracking
39	Critical Section	Impossible	Lock release
40	Interleaved Routing	Impossible	Current task
41	Thread Merge	Impossible	Counter
42	Thread Split	Impossible	Counter
43	Explicit Termination	Impossible	Cascade + membership

Table A.1: Complete pattern classification

Appendix B

N3 Rule Listings

Selected rules demonstrating the architectural issues are available in the KGCL repository at:

```
src/kgcl/hybrid/wcp43_physics.py
```