A Compositional Strategy for Verifying Fault-Tolerant Dynamic Task Graph Scheduling in Modern Cloud Environments

2026 TLA+ Workshop

April 12, 2026

Quentin Delamea[1,2]

PhD Student, qdelamea@aneo.fr

Supervisors: Janna Burman[1], Jérôme Gurhem[2], Stéphane Vialle[1,3]

Collaborators: Wilfried Kirschenmann[2], Florian Lemaitre[2], Stephan Merz[4]

[1]LISN laboratory, Paris-Saclay University, Gif-sur-Yvette, France

[2]Aneo, Boulogne-Billancourt, France

[3]CentraleSupelec, Gif-sur-Yvette, France

[4]University of Lorraine, CNRS, Inria, LORIA, Nancy, Franc

I Am Severely Visually Impaired

Your guidance is appreciated! Please help me stay on track with the slides. Thank you!

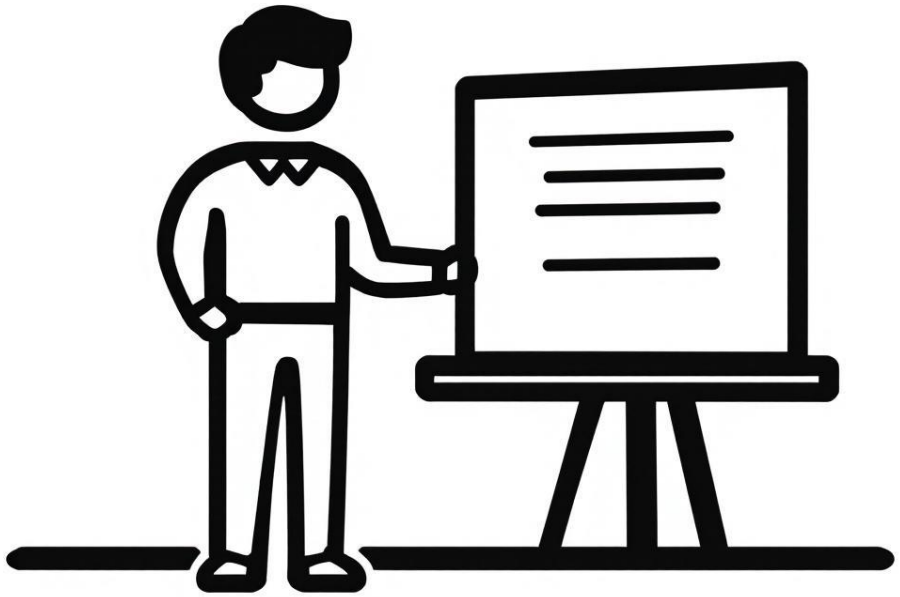


Table of Contents

ArmoniK Abstracts Distributed Infrastructure to Streamline Parallel Workloads

- High-Performance Computing (HPC) is highly complex; specialized developers are scarce[1].
- **The Solution:** A turnkey cloud platform that provides high-level abstractions.
- Allows developers to focus purely on domain algorithms, abstracting away the infrastructure.

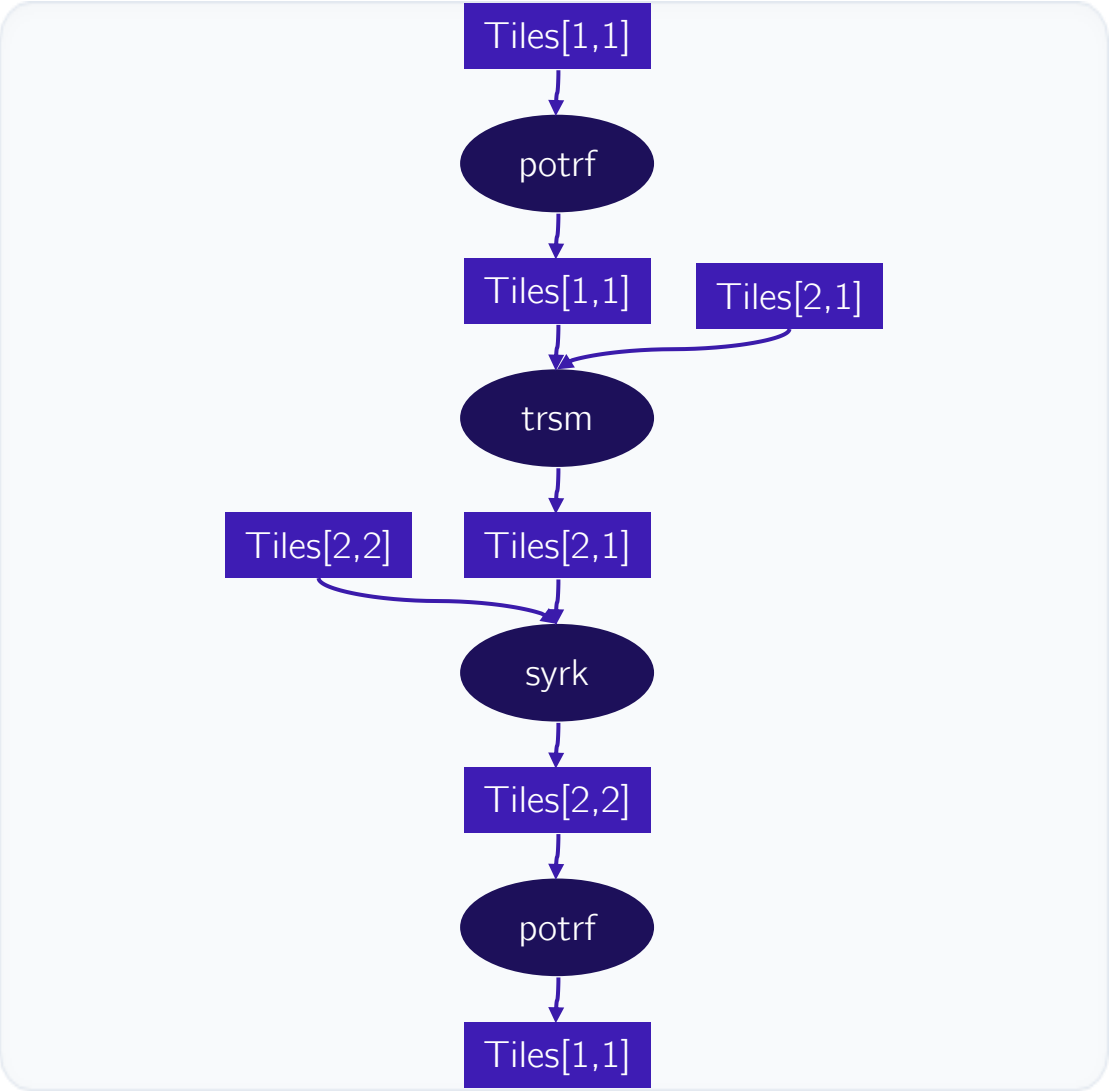
```
@task
def gemm(A, B, C)
    return sp.linalg.blas.dgemm(-1.0, A, B, 1.0, c=C, trans_b=1)

def cholesky_tiled(tiles, n):
    for k in range(n):
        tiles[k, k] = potrf.invoke(tiles[k, k])
        for i in range(k + 1, n):
            tiles[i, k] = trsm.invoke(tiles[k, k], tiles[i, k])
        for i in range(k + 1, n):
            tiles[i, i] = syrk.invoke(tiles[i, k], tiles[i, i])
            for j in range(k + 1, i):
                tiles[i, j] = gemm.invoke(tiles[i, k],
                tiles[j, k], tiles[i, j])
    return tiles
```

[1] https://hyperionresearch.com/wp-content/uploads/2025/11/Hyperion-Research-SC25-Breakfast-Briefing-November-2025-1.pdf

ArmoniK Represents Workloads as Bipartite DAGs of Tasks and Data Objects

- ArmoniK's API intercepts **@task** calls to autonomously build a **bipartite DAG of tasks and objects**
- Exposes massive, native parallelism automatically without manual thread management.
- **Dynamic Execution:** Tasks can seamlessly spawn new tasks or sub-DAGs at runtime.
- Scheduler natively manages data transfers, routing, elasticity, and fault-tolerance.

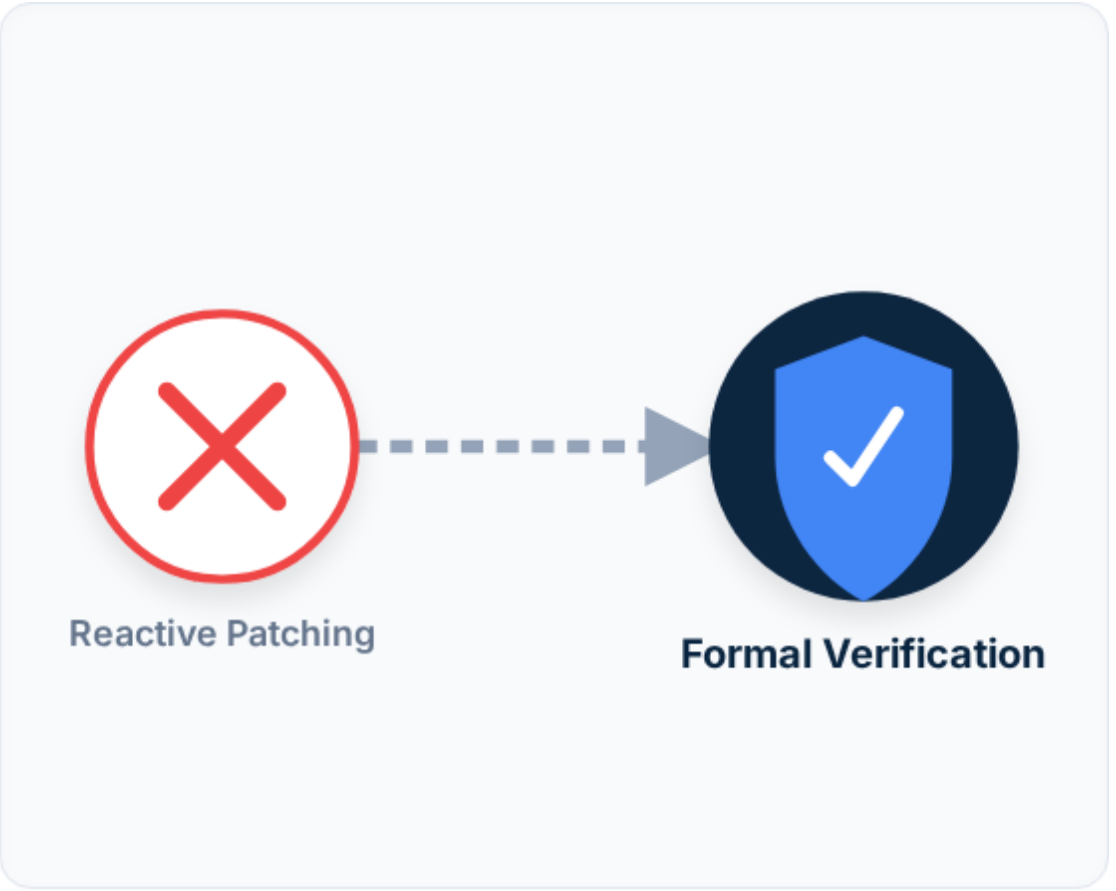


ArmoniK Requires the Adoption of a Rigorous Approach to Distinguish Flawed Logic and Superfluous Mechanisms

The Problem: At massive scale, rare failure sequences leave tasks in transient states, causing unpredictable, hard-to-reproduce errors.

The Limitation: Reactive patching ("firefighting") lacks fundamental guarantees and may break the system.

The Solution: Adopting **Formal Methods** to meticulously specify, model-check, and mathematically prove scheduling correctness.



TLA+ Meets Requirements for Concurrency, Complex Data Structures, Liveness, and Formal Proof Support

Concurrency

The language must be capable of handling distributed systems natively.

Expressivity

Must describe high-level data structures (like bipartite DAGs) alongside low-level system components (message queues, networks).

Liveness

Scheduling is inherently linked to liveness—the guarantee that execution eventually progresses.

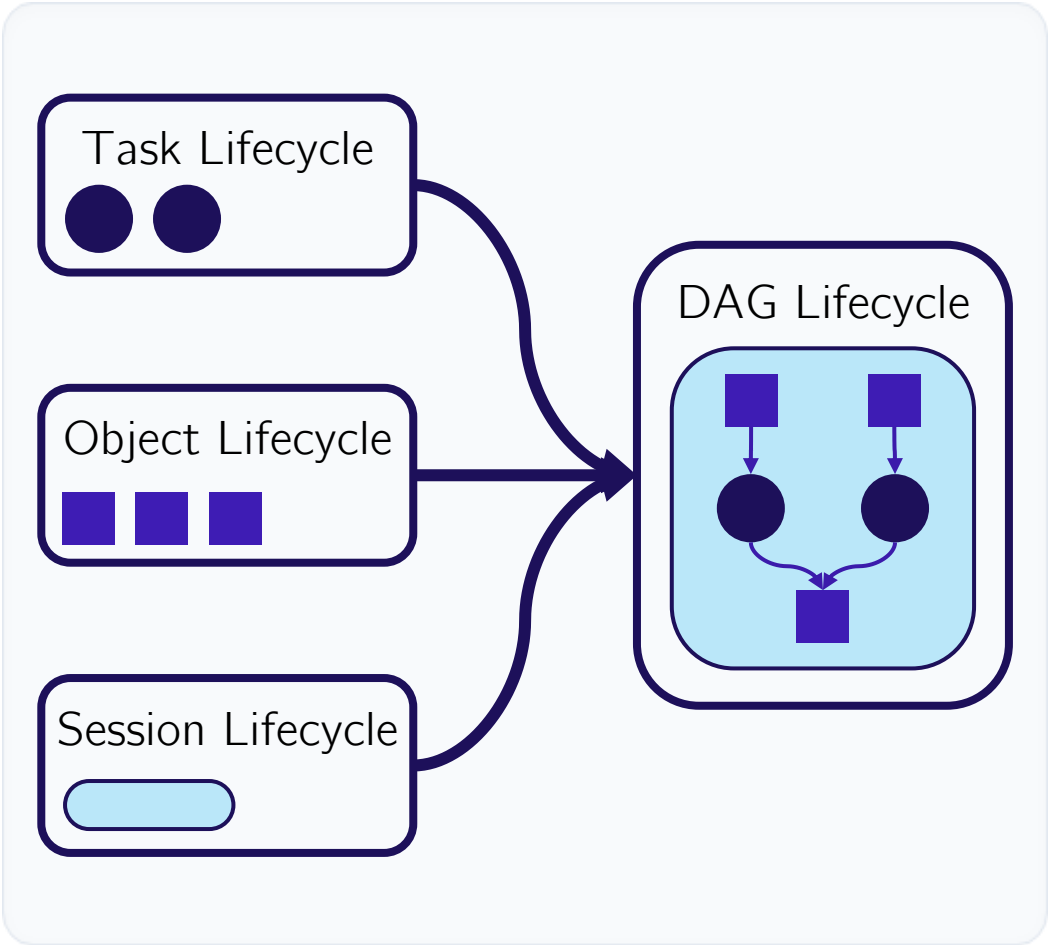
Proof Support

We required a powerful tool to formally prove the reliability of our protocol and component interactions.

Table of Contents

Modular Modeling Specifies Core Aspects Independently Through Successive Refinements Before Composing the Full System

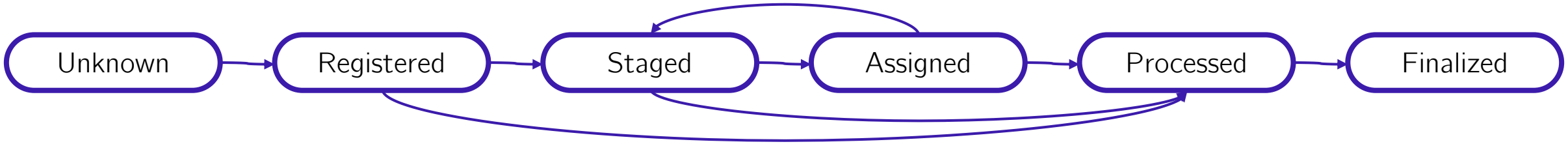
- **Specifying Distributed Scheduling:** Precisely specify scheduling rules and the evolution of the DAG under user and system actions over time,
- **Three Core Scheduling Aspects:** Tasks, Objects, and Sessions
- **Strategy:** Instead of writing a monolithic specification, we decompose the specification by model each aspect independently using refinement before composing them.



Specifications on Github: https://github.com/aneoconsulting/ArmoniK.Spec/

The Task Lifecycle is Modeled Through Iterative Refinements From Abstract To Concrete

Task Processing 1



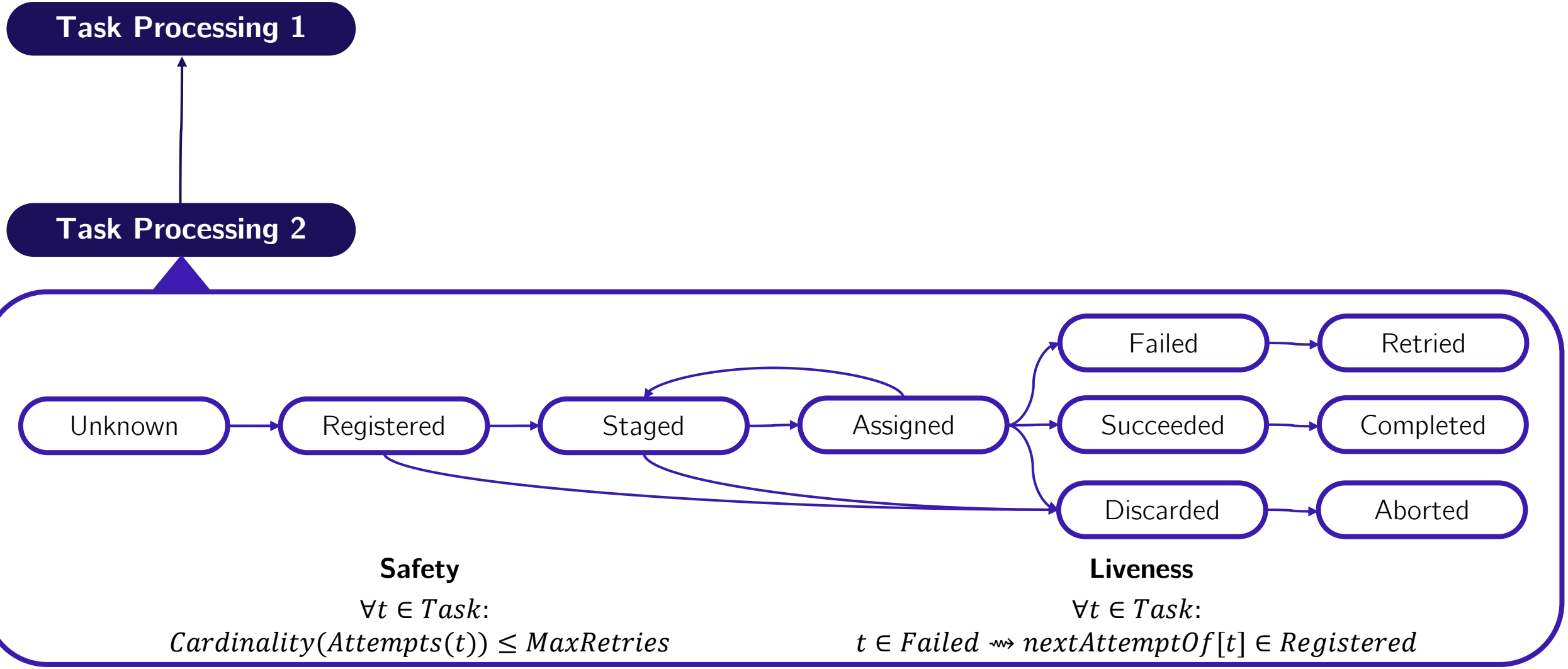
Safety

$$\forall t \in Task : t \in Assigned \Leftrightarrow \exists a \in Agent : t \in agentTaskAlloc[a]$$

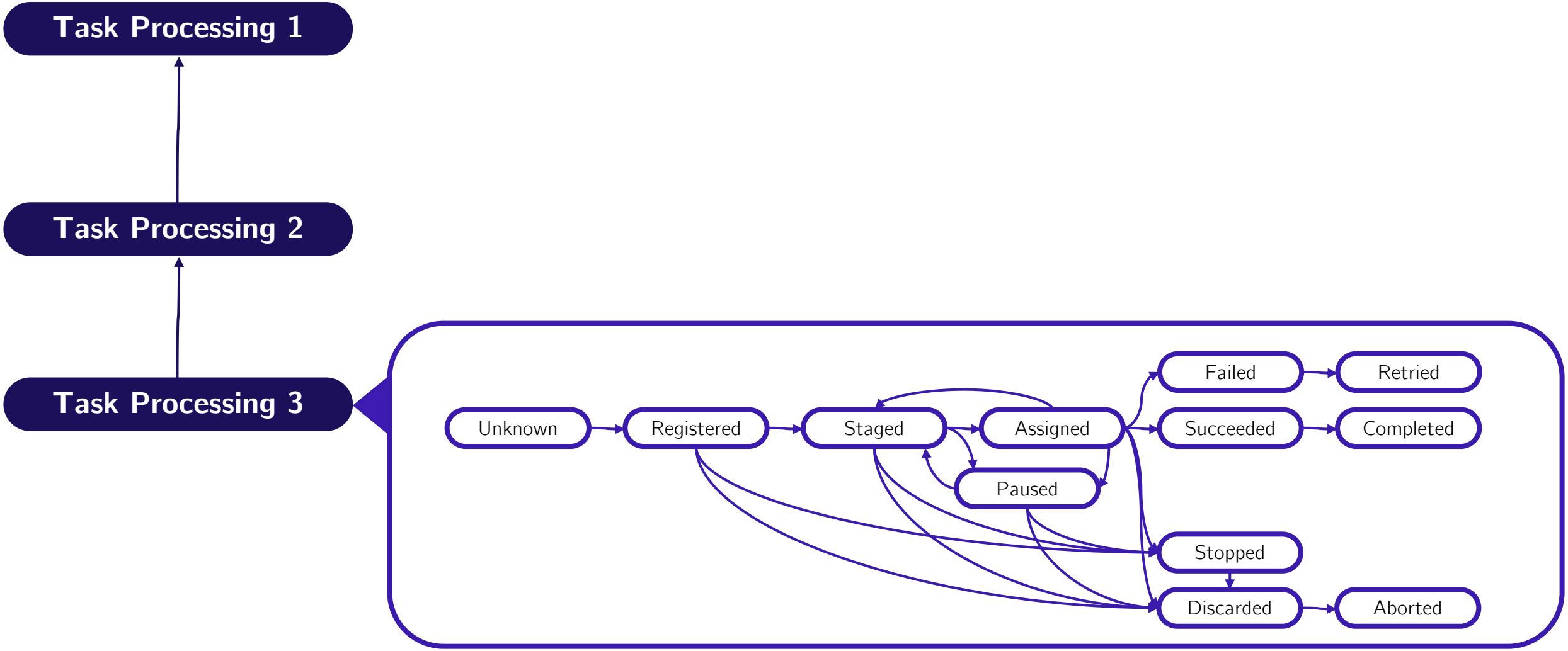
Liveness

$$\forall t \in Task : t \in Processed \rightsquigarrow Finalized$$

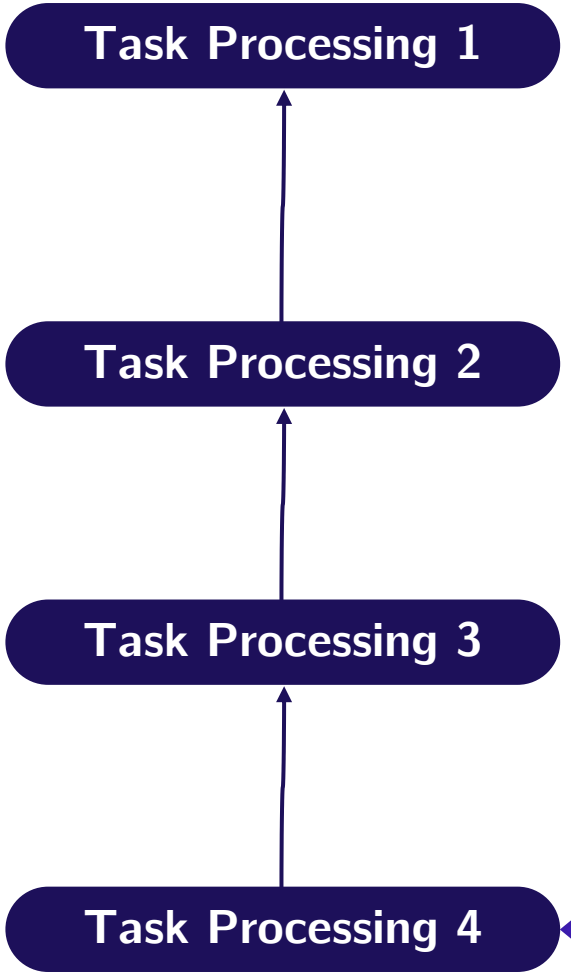
The Task Lifecycle is Modeled Through Iterative Refinements From Abstract To Concrete



The Task Lifecycle is Modeled Through Iterative Refinements From Abstract To Concrete

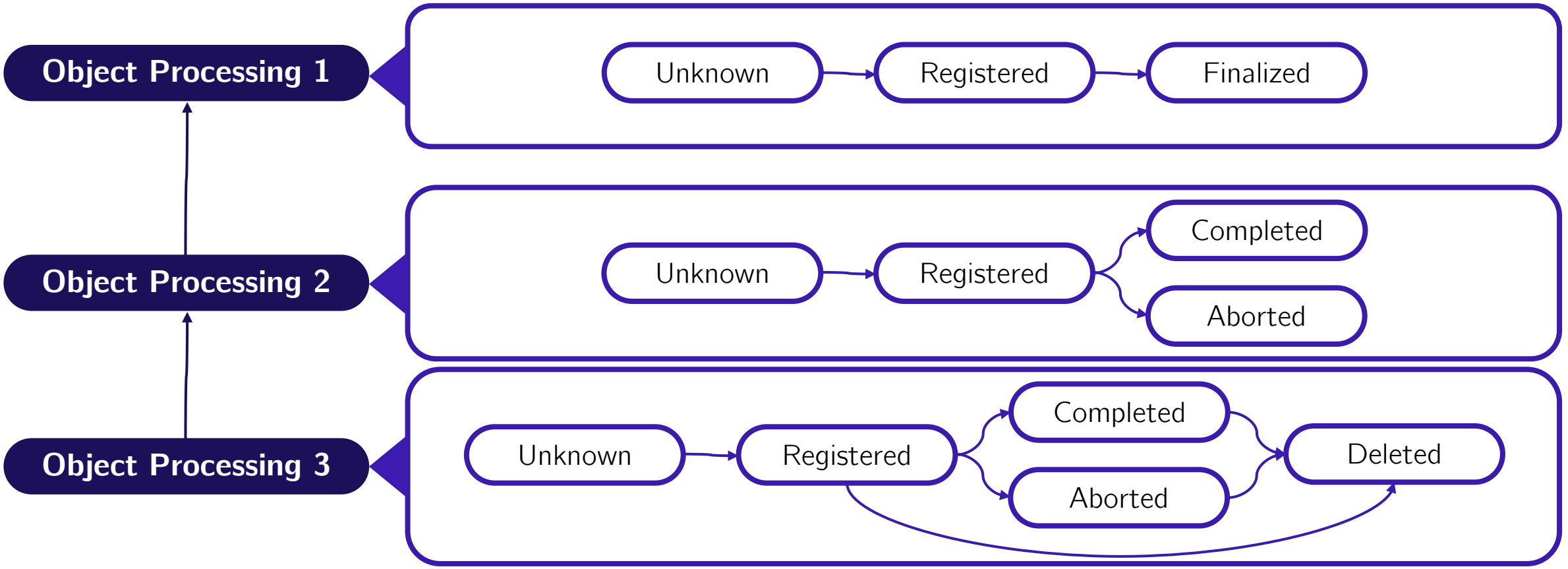


The Task Lifecycle is Modeled Through Iterative Refinements From Abstract To Concrete

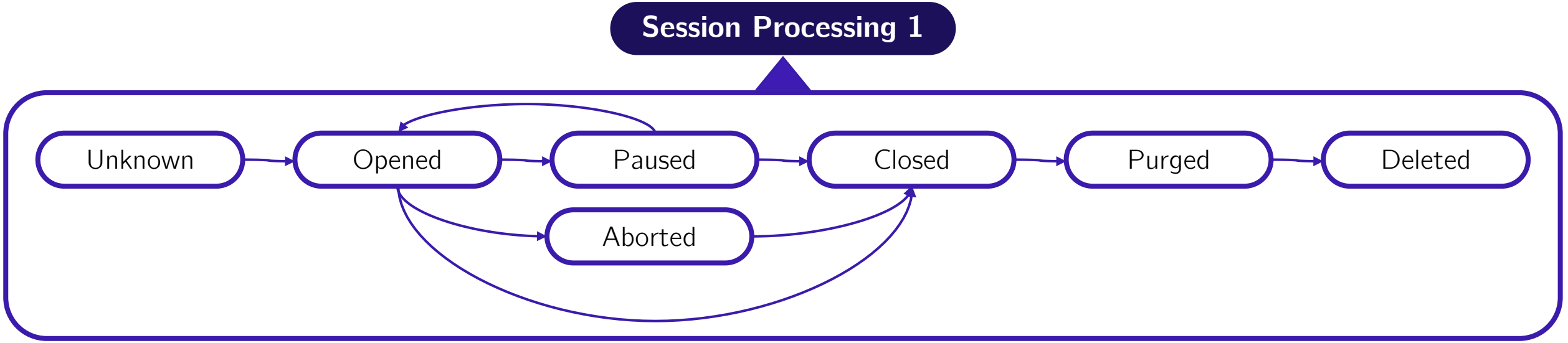


Extends Task Processing 3 with task deletion considerations.

Data Objects Are Modeled Through a Similar Iterative Refinement Process



A Single Specification Captures the Lifecycle of Sessions



Lifecycle Compositions Iteratively Refine the Specification of Scheduling for Workloads Composed of Tasks and Objects Grouped within Sessions

Composition adds couplings rules

- Tasks can transition to *Staged* if all its input objects are *Finalized*
- Objects can transition to *Finalized* if at least one of its parent tasks is *Processed*.

Compositions define DAG-related properties

- Tasks-Objects dependencies always form a bipartite DAG which sources and sinks are objects.
- If a task is *Aborted* then eventually all its dominated tasks and objects will be *Aborted*.

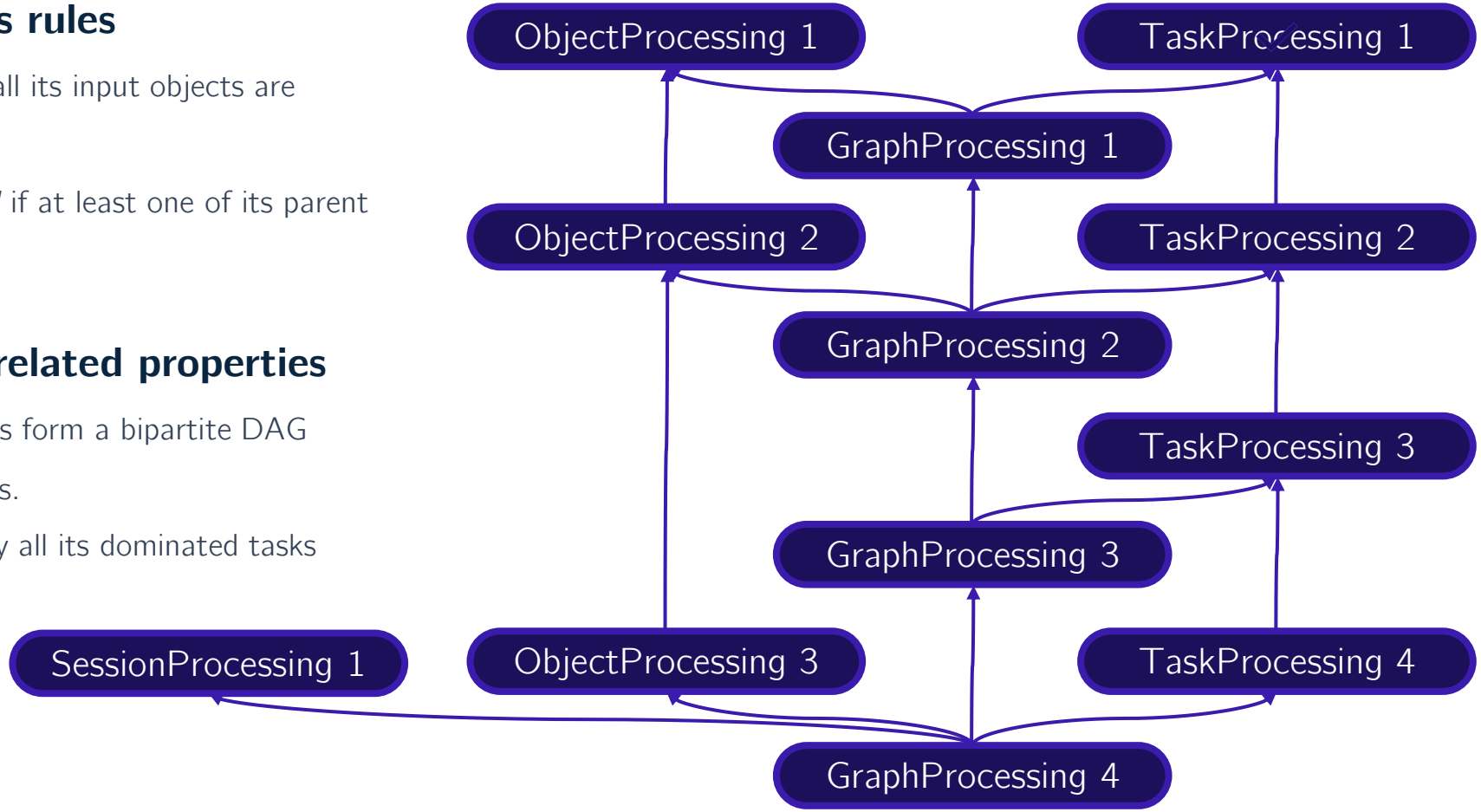


Table of Contents

Modular Modeling Enables Domain Experts to Review Specifications Without Formal Methods Expertise

A specification is only valuable if it is mathematically correct AND faithful to the real-world system.

ArmoniK development team had no prior FM experience.

Breaking the system into focused sub-aspects builds cross-functional team expertise gradually.

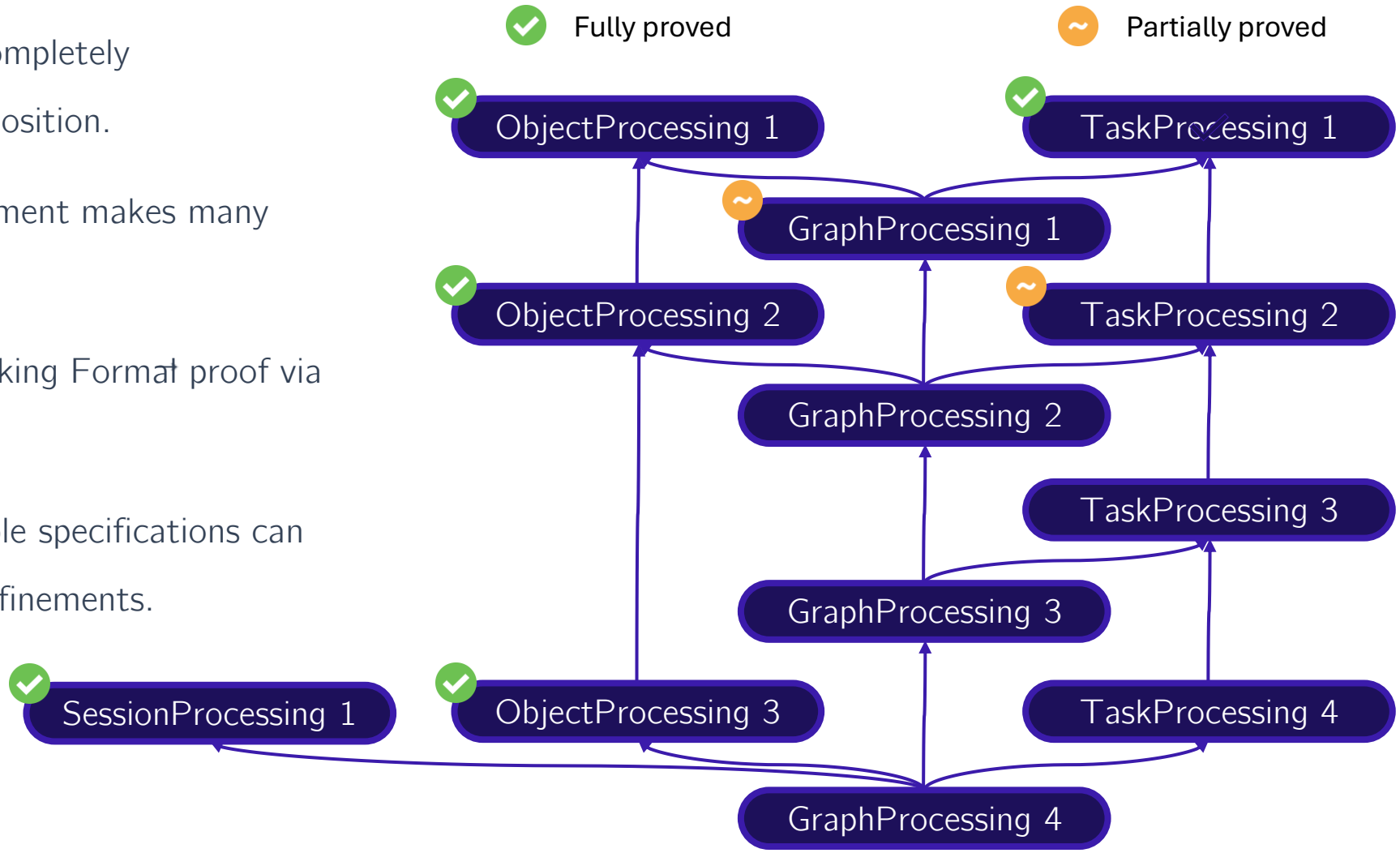
Modular Modeling Enables Targeted Optimizations and Early Property Verification for TLC

	State space explosion	Formula evaluation complexity
Challenges		
Mitigations	▪ Symmetry reduction: exploits system symmetries to reduce space	▪ Formula Re-writing (e.g., using recursive instead of existential formulations).Mitigation 2: ▪ Operator Overloading with Java
Drawback	▪ Incompatible with verifying liveness properties	▪ Encourages computer-oriented code over mathematical-oriented code ▪ Introduces unverified black-boxed code

Mitigations can make proofs more difficult and should be used sparingly. Modular modeling allows delaying their use and applying them only where necessary.

Modular Modeling Enhances Tractability of Formal Proofs by Decomposing Them into Elementary Sub-Steps

- System decomposition allows completely independent proofs before composition.
- Eliminating monolithic entanglement makes many sub-theorems trivial.
- **Workflow:** Minimal model-checking Format proof via TLAPS.
- Proved results in high-level simple specifications can be reused in subsequent logic refinements.



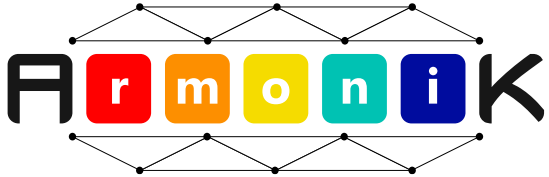
Conclusion & Future Works

01 Reliable Dynamic Distributed Scheduling Demands Rigorous Mathematical Foundations

02 Modular Modeling First Specifies Core Scheduling Aspects Independently Before Composing The Full System

03 Decomposing Specifications Benefits Team Reviews Alongside Model Checking And Formal Proof

Conclusion & Future Works



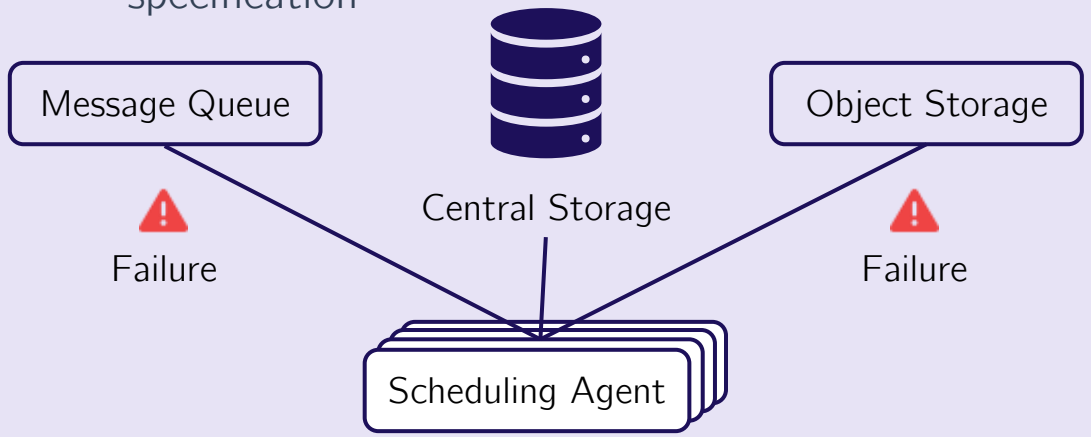
Reliable Dynamic Distributed Scheduling Demands Rigorous Mathematical Foundations

Modular Modeling First Specifies Core Scheduling Aspects Independently Before Composing The Full System

Decomposing Specifications Benefits Team Reviews Alongside Model Checking And Formal Proof

Toward the Full Specificatio

- Current specification describes consistent interactions with the central database.
- Extend system modeling to remaining components and their interactions
- Compose models into a unified, full-system specification



THANK YOU!

Do you have any question?

I can't see your raised hands. Feel free to speak up directly!