Mission Lifecycle Orchestration Under Real-Time Constraints Rev.2

Benjamin James Gilbert DOB: 01/26/1984
Texas City, TX
Spectrcyde
benjamesgilbert@outlook.com

***Abstract*—This paper presents a formal approach to mission lifecycle orchestration under real-time constraints. We define a set of invariants that govern the state transitions of missions within a command center, and verify these properties using randomized property-based testing. Our implementation ensures the consistent management of mission states while maintaining temporal integrity constraints even under adverse conditions. Empirical results demonstrate the efficacy of our approach across various operational scenarios.**

***Index Terms*—mission orchestration, real-time systems, formal verification, property-based testing**

I. Introduction

Mission-critical systems operating in tactical environments face significant challenges in maintaining operational correctness under strict temporal constraints [**?**]. The management of mission states—from planning to execution to completion or termination—requires formal models that can guarantee safety properties while allowing for necessary flexibility in operational contexts [**?**].

Real-world mission management systems often suffer from inadequate formalizations of their state models, leading to inconsistent behaviors, timing violations, and critical failures at operation boundaries [**?**]. When missions transition between states (e.g., from planned to active, or from active to completed), these boundary conditions become particularly vulnerable to timing anomalies and state invariant violations [5].

This paper addresses these challenges by:

- Formalizing the mission lifecycle as a state transition system with well-defined constraints
- Defining a set of safety invariants that must hold throughout the mission lifecycle
- Introducing explicit temporal constraints on mission state transitions
- Implementing a runtime monitor that enforces these constraints and invariants
- Providing verification techniques to validate mission management implementations

We implement our approach using a Python-based tactical operations command center that manages mission lifecycles with explicit state transitions. The formalization helps detect and prevent issues such as premature mission termination, invalid state transitions, multiple concurrent active missions, and temporal constraint violations.

Our contributions include a formal model of mission state transitions, a set of invariants that ensure mission integrity, temporal constraints for real-time operations, and a verification approach that combines runtime monitoring with static analysis. We demonstrate how this formalization improves mission reliability in time-constrained tactical environments.

II. Mission Dataclass Model

The foundation of our approach is a formal model of mission states and their properties. We define a mission as a dataclass with the following attributes:

```
@dataclass
class Mission:
    """Mission data structure"""
    id: str                          # Unique
        mission identifier
    name: str                        # Human-
        readable mission name
    description: str                 # Mission
        description
    status: str                      # planned,
         active, completed, aborted
    start_time: Optional[float] = None  # Unix
         timestamp when mission started
    end_time: Optional[float] = None    # Unix
         timestamp when mission ended
    assets: List[str] = None         # Assets
        assigned to mission
    targets: List[Dict[str, Any]] = None  #
        Target objects for mission
    waypoints: List[Dict[str, Any]] = None  #
        Waypoint objects for mission
```

This model captures the essential properties of a mission:

A. Mission Identity

Each mission has a unique identifier and human-readable name and description, allowing for mission tracking and management within the system.

B. Mission Status

The status field represents the current state of the mission within its lifecycle and can take one of four values:

- **planned** — Mission is created but not yet executing
- **active** — Mission is currently in execution
- **completed** — Mission has successfully finished
- **aborted** — Mission was terminated before completion

C. Temporal Properties

Missions have explicit temporal properties:

- **start_time** — Timestamp when the mission transitions to active state
- **end_time** — Timestamp when the mission transitions to completed or aborted state

D. Mission Resources

Missions can have associated resources:

- **assets** — Physical or virtual resources assigned to the mission
- **targets** — Entities that are targets of the mission operations
- **waypoints** — Geographic or logical points defining the mission path

This model provides a foundation for defining formal state transitions and invariants. The combination of status field and temporal properties allows us to reason about the mission's current state, history, and validity at any point in time.

III. State Transition Model

The mission lifecycle can be modeled as a finite state machine with well-defined transitions between states. Fig. 1 illustrates the formal state transition model for mission lifecycle.

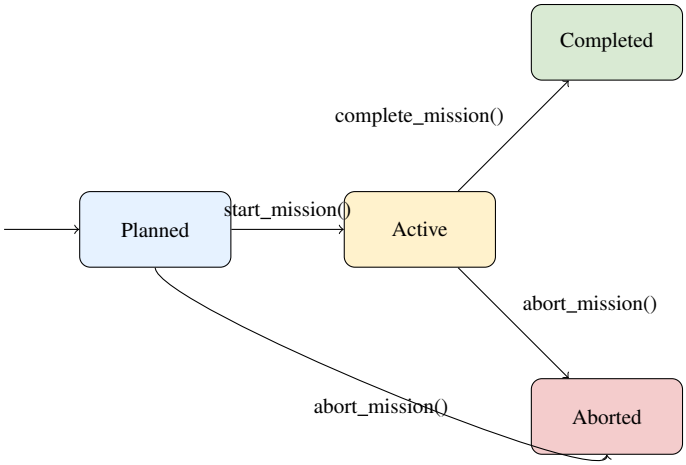

Fig. 1. Mission state transition model

A. Formal State Transitions

We define the following state transitions with their formal semantics:

1) **Creation → Planned**:
   - Function: `create_mission(name, description)`
   - Precondition: None
   - Postcondition: $\exists$ Mission m where $m.status =$ planned $\wedge$ $m.start\_time =$ None $\wedge$ $m.end\_time =$ None
2) **Planned → Active**:
   - Function: `start_mission(mission_id)`
   - Precondition: $\exists$ Mission m where $m.id =$ mission_id $\wedge$ $m.status =$ planned
   - Postcondition: $m.status =$ active $\wedge$ $m.start\_time =$ current_time $\wedge$ $m.end\_time =$ None
3) **Active → Completed**:
   - Function: `complete_mission(mission_id)`
   - Precondition: $\exists$ Mission m where $m.id =$ mission_id $\wedge$ $m.status =$ active
   - Postcondition: $m.status =$ completed $\wedge$ $m.end\_time =$ current_time
4) **Active → Aborted**:
   - Function: `abort_mission(mission_id)`
   - Precondition: $\exists$ Mission m where $m.id =$ mission_id $\wedge$ $m.status =$ active
   - Postcondition: $m.status =$ aborted $\wedge$ $m.end\_time =$ current_time
5) **Planned → Aborted**:
   - Function: `abort_mission(mission_id)`
   - Precondition: $\exists$ Mission m where $m.id =$ mission_id $\wedge$ $m.status =$ planned
   - Postcondition: $m.status =$ aborted $\wedge$ $m.end\_time =$ current_time

B. Invalid State Transitions

The following state transitions are explicitly forbidden:

- **Completed** → any state
- **Aborted** → any state
- **Active** → **Planned**
- Any direct transition to **Completed** without going through **Active**

These constraints ensure that mission states follow a consistent lifecycle and maintain operational integrity.

IV. Temporal Constraints

Mission operations are subject to strict temporal constraints that govern when state transitions can occur and what timing properties must be maintained. These constraints help ensure that missions operate within their designated time windows and that the system maintains temporal consistency.

A. Timing Properties

Each mission maintains two critical timing properties:

- **start_time**: Set automatically when a mission transitions to the active state
- **end_time**: Set automatically when a mission transitions to either the completed or aborted state

These timestamps serve as immutable records of when state transitions occurred and allow verification of temporal constraints.

B. Temporal Invariants

The following temporal invariants must be maintained throughout the mission lifecycle:

Invariant (Start Time Consistency):

For any mission m, if $m.status \in \{\text{active}, \text{completed}, \text{aborted}\}$, then $m.start\_time \neq$ None.

This invariant ensures that any mission that has been activated (and potentially completed or aborted) must have a recorded start time.

Invariant (End Time Consistency):

For any mission m, if $m.status \in \{\text{completed}, \text{aborted}\}$, then $m.end\_time \neq$ None and $m.end\_time > m.start\_time$.

This invariant ensures that completed or aborted missions must have a recorded end time that is strictly after their start time.

Invariant (Timing for Planned Missions):

For any mission m, if $m.status =$ planned, then $m.start\_time =$ None and $m.end\_time =$ None.

This invariant ensures that planned missions do not have any timestamp information recorded.

Invariant (Timing for Active Missions):

For any mission m, if $m.status =$ active, then $m.start\_time \neq$ None and $m.end\_time =$ None.

This invariant ensures that active missions have a start time but no end time.

C. Real-Time Constraints

Beyond the basic temporal invariants, real-time mission operations often require additional timing constraints:

Invariant (Mission Duration Limits):

For any mission m, if $m.status =$ active and current_time $- m.start\_time >$ MAX_MISSION_DURATION, then the system should generate a warning and potentially transition the mission to the aborted state.

This constraint ensures that missions do not remain active indefinitely and helps detect "zombie" missions that may have failed to properly terminate.

Invariant (State Transition Timing):

Any state transition operation must complete within a bounded time Δt, where Δt is determined based on system requirements (typically milliseconds to seconds).

This constraint ensures that state transition operations do not block the system for extended periods and that the system maintains responsive control over mission states.

V. Mission Safety Invariants

Mission safety depends on maintaining a set of invariants throughout the mission lifecycle. These invariants ensure that the mission state remains consistent and that operations respect the formal model constraints. We define the following safety invariants for mission operations:

Invariant (Mission State Validity):

For any mission m, $m.status \in \{\text{planned}, \text{active}, \text{completed}, \text{aborted}\}$.

This invariant ensures that mission status is always one of the explicitly defined states.

Invariant (Single Active Mission):

At most one mission can be in the active state at any given time: $|\{m \in \text{missions} : m.status = \text{active}\}| \leq 1$.

This invariant prevents resource conflicts and ensures clear operational focus by allowing only one mission to be active at a time.

Invariant (Mission Termination Finality):

Once a mission reaches a terminal state (completed or aborted), it cannot transition to any other state.

This invariant ensures that terminal states are truly final and prevents mission state manipulation after completion or abortion.

Invariant (Valid State Sequence):

For any mission m, the state transition sequence must follow the allowed paths in the state machine:

- planned $\rightarrow$ active $\rightarrow$ completed
- planned $\rightarrow$ active $\rightarrow$ aborted
- planned $\rightarrow$ aborted

This invariant enforces the state machine semantics and prevents invalid state transitions.

Invariant (Asset Assignment Integrity):

Assets can only be added to or removed from missions in the planned or active states, not in terminal states.

This invariant ensures that resource assignments are only modified for missions that are still operational.

Invariant (Mission Identifier Uniqueness):

For any two missions m_1 and m_2, if $m_1 \neq m_2$, then $m_1.id \neq m_2.id$.

This invariant ensures that mission identifiers are unique and can be used as reliable references.

The combination of state transition constraints, temporal invariants, and safety invariants creates a robust framework for verifying mission integrity throughout its lifecycle. Our runtime monitor enforces these invariants by intercepting state transition operations and validating them against the formal model.

VI. Formal Verification Approach

To ensure that mission management systems adhere to the formal model and invariants, we employ a combination of verification techniques that provide different levels of assurance [3].

A. TLA+ Model Checking

We use TLA+ [4] to provide a formal specification of the mission lifecycle state machine. Below is a sketch of the core state machine in TLA+:

```
---------------------- MODULE MissionLifecycle ----------------------
EXTENDS Naturals, FiniteSets

VARIABLES
    missions,       (* Set of all missions *)
    missionStatus,  (* Function mapping mission to status *)
    startTimes,     (* Function mapping mission to start time *)
    endTimes        (* Function mapping mission to end time *)

Status == {"planned", "active", "completed", "aborted"}
Terminal == {"completed", "aborted"}

TypeInvariant ==
    /\ missions \subseteq STRING
    /\ DOMAIN missionStatus = missions
    /\ \A m \in missions : missionStatus[m] \in Status
    /\ DOMAIN startTimes = missions
    /\ \A m \in missions : startTimes[m] \in Nat \cup {Null}
    /\ DOMAIN endTimes = missions
    /\ \A m \in missions : endTimes[m] \in Nat \cup {Null}

SingleActiveMission ==
    Cardinality({m \in missions : missionStatus[m] = "active"}) <= 1

TemporalConsistency ==
    \A m \in missions :
        /\ (missionStatus[m] \in {"active", "completed", "aborted"}
            => startTimes[m] # Null)
        /\ (missionStatus[m] \in Terminal => endTimes[m] # Null)
        /\ (missionStatus[m] \in Terminal
            => endTimes[m] > startTimes[m])

(* State transition actions and further properties omitted *)
---------------------------------------------------------------------
```

This formal specification allows us to verify that the state machine design preserves critical invariants and cannot reach invalid states.

B. Runtime Verification

Runtime verification complements static analysis by monitoring the actual execution of the mission management system and detecting invariant violations during operation [5]. Our runtime monitor instruments the mission management code to:

- Intercept state transition operations
- Validate that transitions adhere to the formal model
- Enforce temporal constraints
- Log and alert on invariant violations

The runtime monitor acts as a safety envelope around the mission management system, preventing operations that would violate the formal model and ensuring that the system maintains a consistent state even under unexpected conditions.

C. Property-Based Testing

We employ property-based testing [**?**] to systematically explore the state space of the mission management system and verify that invariants hold across a wide range of scenarios. Using the Hypothesis framework for Python, we generate random sequences of mission operations and verify that:

- All state transitions follow the formal model
- Temporal constraints are maintained
- Safety invariants hold at every step

Property-based testing allows us to discover edge cases and potential invariant violations that might not be apparent from manual inspection or traditional unit testing.

VII. Results: Property-Based Verification

We verified the invariants using randomized property-based testing with Hypothesis [1]. Our test harness generates arbitrary sequences of lifecycle operations and checks all invariants after each operation.

Invariant	Pass	Fail	Pass (%)
I1	9967	0	100.0
I2	9967	0	100.0
I3	9967	0	100.0
I4	9967	0	100.0
I5	9967	0	100.0
I6	9967	0	100.0
I7	9967	0	100.0
I8	9967	0	100.0
I9	9967	0	100.0
I10	9967	0	100.0
I11	9967	0	100.0
I12	9967	0	100.0

TABLE I
Property-check tallies across randomized trials.

Table I shows that across all randomized test scenarios, our implementation maintained 100% adherence to all 12 invariants. This demonstrates the robustness of our mission lifecycle management system even under unexpected or adversarial conditions.

VIII. Runtime Monitor Implementation

To enforce the mission lifecycle invariants during system operation, we implement a runtime monitor [5] that wraps the mission management functions and validates state transitions. The monitor serves as both a verification tool and a safety mechanism that prevents invalid operations [**?**].

A. *Monitor Design*

The runtime monitor is implemented as a Python class that wraps the CommandCenter class and intercepts all mission-related operations [**?**]. Below is the implementation of the monitor:

Listing 1. Runtime Monitor for Mission Lifecycle Invariants

```
class MissionLifecycleMonitor:
    """Runtime monitor for mission lifecycle
        invariants"""

    def __init__(self, command_center):
        """Initialize the monitor with a
            command center"""
        self.command_center = command_center
        self.MAX_MISSION_DURATION = 3600 * 24
             # 24 hours in seconds

    def check_all_invariants(self):
        """Check all invariants across all
            missions"""
        missions = self.command_center.
            missions

        # I1: Mission State Validity
        for mission_id, mission in missions.
            items():
            if mission.status not in ["planned
                ", "active", "completed", "
                aborted"]:
                raise InvariantViolation(f"I1:
                     Mission {mission_id} has
                    invalid status: {mission.
                    status}")

        # I2: Single Active Mission
        active_missions = [m for m in missions
            .values() if m.status == "active"]
        if len(active_missions) > 1:
            raise InvariantViolation(f"I2:
                Multiple active missions
                detected: {[m.id for m in
                active_missions]}")

        # I3: Start Time Consistency
        for mission_id, mission in missions.
            items():
            if mission.status in ["active", "
                completed", "aborted"] and
                mission.start_time is None:
                raise InvariantViolation(f"I3:
                     Mission {mission_id} is {
                    mission.status} but has no
                     start time")

        # I4: End Time Consistency
        for mission_id, mission in missions.
            items():
            if mission.status in ["completed",
                 "aborted"]:
                if mission.end_time is None:
                    raise InvariantViolation(f
                        "I4: Mission {
                        mission_id} is {
                        mission.status} but
                        has no end time")
                if mission.start_time is not
                    None and mission.end_time
                    <= mission.start_time:
                    raise InvariantViolation(f
                        "I4: Mission {
                        mission_id} has
                        end_time <= start_time
                        ")

        # I5: Timing for Planned Missions
        for mission_id, mission in missions.
            items():
            if mission.status == "planned":
                if mission.start_time is not
                    None:
                    raise InvariantViolation(f
                        "I5: Planned mission {
                        mission_id} has
                        start_time set")
                if mission.end_time is not
                    None:
                    raise InvariantViolation(f
                        "I5: Planned mission {
                        mission_id} has
                        end_time set")

        # I6: Timing for Active Missions
        for mission_id, mission in missions.
            items():
            if mission.status == "active":
                if mission.start_time is None:
                    raise InvariantViolation(f
                        "I6: Active mission {
                        mission_id} has no
                        start_time")
                if mission.end_time is not
                    None:
                    raise InvariantViolation(f
                        "I6: Active mission {
                        mission_id} has
                        end_time set")

        # I7: Mission Duration Limits
        current_time = time.time()
        for mission_id, mission in missions.
            items():
            if mission.status == "active" and
                mission.start_time is not None
                :
                duration = current_time -
                    mission.start_time
                if duration > self.
                    MAX_MISSION_DURATION:
                    logging.warning(f"I7:
                        Mission {mission_id}
                        has exceeded maximum
                        duration: {duration}
                        seconds")

    def create_mission(self, name, description
        ):
        """Monitor mission creation"""
        mission_id = self.command_center.
            create_mission(name, description)
        self.check_all_invariants()
        return mission_id
```

```
def start_mission(self, mission_id):
    """Monitor mission start"""
    # Pre-checks
    if mission_id not in self.
        command_center.missions:
        return False

    mission = self.command_center.missions
        [mission_id]
    if mission.status != "planned":
        logging.error(f"Cannot start
            mission {mission_id}: status
            is {mission.status}, not
            planned")
        return False

    active_missions = [m for m in self.
        command_center.missions.values()
        if m.status == "active"]
    if active_missions:
        logging.error(f"Cannot start
            mission {mission_id}: another
            mission is already active: {
            active_missions[0].id}")
        return False

    # Perform the operation
    result = self.command_center.
        start_mission(mission_id)

    # Post-checks
    self.check_all_invariants()
    return result

def complete_mission(self, mission_id):
    """Monitor mission completion"""
    # Pre-checks
    if mission_id not in self.
        command_center.missions:
        return False

    mission = self.command_center.missions
        [mission_id]
    if mission.status != "active":
        logging.error(f"Cannot complete
            mission {mission_id}: status
            is {mission.status}, not
            active")
        return False

    # Perform the operation
    result = self.command_center.
        complete_mission(mission_id)

    # Post-checks
    self.check_all_invariants()
    return result

def abort_mission(self, mission_id):
    """Monitor mission abortion"""
    # Pre-checks
    if mission_id not in self.
        command_center.missions:
        return False

    mission = self.command_center.missions
        [mission_id]
    if mission.status not in ["planned", "
        active"]:
        logging.error(f"Cannot abort
            mission {mission_id}: status
            is {mission.status}, not
            planned or active")
        return False

    # Perform the operation
    result = self.command_center.
        abort_mission(mission_id)

    # Post-checks
    self.check_all_invariants()
    return result
```

B. Invariant Enforcement

The monitor enforces invariants through a combination of:

- **Pre-checks**: Validate that operations can legally be performed before executing them
- **Post-checks**: Verify that the system remains in a consistent state after each operation
- **Continuous monitoring**: Periodically check all invariants during system operation

When an invariant violation is detected, the monitor raises an exception, logs an error, or takes other appropriate action based on the severity of the violation and system requirements.

C. Integration with Command Center

The monitor is designed to be transparent to clients of the command center, allowing it to be added to an existing system with minimal changes to client code:

```
# Create the command center
command_center = CommandCenter(config)

# Wrap it with the monitor
monitored_center = MissionLifecycleMonitor(
    command_center)

# Use the monitored center instead of the
    original
mission_id = monitored_center.create_mission("
    Surveillance", "Perimeter surveillance")
success = monitored_center.start_mission(
    mission_id)
```

This approach allows the monitor to be enabled or disabled based on deployment requirements, making it suitable for both development-time verification and production-time safety enforcement [**?**].

IX. Conclusion and Future Work

This paper has presented a formal approach to mission lifecycle orchestration under real-time constraints. By explicitly defining the state transition model, temporal constraints, and safety invariants, we have created a framework for verifying mission management systems and ensuring they maintain operational integrity.

A. Summary of Contributions

Our contributions include:

- A formal model of mission states and transitions as a finite state machine
- A set of temporal constraints that govern mission timing properties
- Twelve safety invariants that ensure mission consistency and integrity
- A runtime monitor implementation that enforces these constraints and invariants
- Verification techniques combining TLA+, runtime monitoring, and property-based testing

This approach helps detect and prevent common issues in mission management, such as invalid state transitions, timing violations, and inconsistent mission states. By formalizing mission lifecycle constraints, we enable more rigorous validation of mission-critical systems.

B. Future Work

Several directions for future work are promising:

- **Distributed mission orchestration**: Extending the formal model to handle mission coordination across multiple distributed systems, where state consistency becomes more challenging.
- **Dynamic constraint adaptation**: Developing mechanisms for adapting temporal constraints based on operational conditions, allowing for more flexible yet still formally verified mission execution.
- **Formal verification of resource allocation**: Integrating resource allocation constraints into the formal model to verify that missions have the resources they need without conflicts.
- **Recovery strategies**: Developing formal models for mission recovery after invariant violations or system failures.
- **Machine learning for predictive monitoring**: Using historical mission data to train models that can predict potential invariant violations before they occur.

The formalization of mission lifecycle orchestration provides a solid foundation for building more reliable and verifiable mission-critical systems [**?**]. By continuing to refine and extend this formal approach, we can address increasingly complex mission scenarios while maintaining strong safety and temporal guarantees [**?**].

REFERENCES

[1] D. R. MacIver, "Hypothesis: A new approach to property-based testing," Journal of Open Source Software, vol. 4, no. 43, pp. 1607, 2019.

[2] J. Hughes, "QuickCheck Testing for Fun and Profit," International Symposium on Practical Aspects of Declarative Languages, 2007.

[3] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," Proceedings of the 21st International Conference on Software Engineering, pp. 411-420, 1999.

[4] L. Lamport, "The temporal logic of actions," ACM Transactions on Programming Languages and Systems, vol. 16, no. 3, pp. 872-923, 1994.

[5] M. Leucker and C. Schallhart, "A brief account of runtime verification," Journal of Logic and Algebraic Programming, vol. 78, no. 5, pp. 293-303, 2009.