ARTOS: Airborne Radar Test & Orchestration System

Table of Contents

Technical Design Specification (v2.1)

1. Executive Summary

Il sistema **ARTOS** è un framework di orchestrazione per l'automazione dei test radar. L'architettura separa l'interfacciamento hardware (Moduli), la logica di analisi (Algoritmi Dinamici) e la sequenza operativa (Test Plans). Il cuore del sistema è il **TestContext**, un'interfaccia unificata che permette sia ai test che agli algoritmi di accedere a dati e comandi in tempo reale.

---

2. Architettura a Livelli (Layered Architecture)

1. **Level 0 - Hardware Abstraction Layer (HAL)**: Moduli `1553`, `SFP`, `TargetSim`.
2. **Level 1 - Orchestrator Core**: Gestisce il ciclo di vita dei moduli e la sincronizzazione.
3. **Level 2 - Dynamic Algorithm Library**: Tool di analisi che utilizzano il `TestContext` per elaborare dati cross-modulo.
4. **Level 3 - Test Execution Layer**: Script (JSON/Python) che orchestrano la chiamata ai moduli e l'esecuzione degli algoritmi.

---

3. Il TestContext: Il Motore di Integrazione

Il `TestContext` non è solo un contenitore, ma fornisce i metodi per l'interazione sicura con l'hardware e lo scambio di dati.

3.1 Definizione del TestContext

```
from typing import Any, Dict, Optional

class TestContext:
    """
    Unified interface to access hardware modules and system state.
    Passed to both Test Scripts and Dynamic Algorithms.
    """

    def __init__(self, modules: Dict[str, Any]):
        self._modules = modules
        self.bus1553 = modules.get("bus1553")
        self.video = modules.get("video")
        self.target_sim = modules.get("target_sim")
        self.results_cache: Dict[str, Any] = {}

    def get_module_data(self, module_name: str) -> Any:
        """Returns the current data buffer or status from a specific module."""
        module = self._modules.get(module_name)
        if module:
            # Assume modules implement a get_current_data method
            return module.get_data_stream()
        return None

    def log_event(self, message: str, level: str = "INFO"):
        """Centralized logging for reports."""
        print(f"[{level}] {message}")
```

---

4. Level 2: Dynamic Algorithm Library

Gli algoritmi sono ora concepiti come "Agenti di Analisi" che operano sul contesto.

4.1 Interfaccia Algoritmo Aggiornata

```
import abc

class BaseAlgorithm(abc.ABC):
    """
    Interface for dynamic algorithms.
    Algorithms use the context to pull data and perform cross-module validation.
    """

    def __init__(self):
        self.name = self.__class__.__name__

    @abc.abstractmethod
    def execute(self, context: TestContext, params: Dict[str, Any]) -> Dict[str, Any]:
        """
        Executes the logic.
```

```
        :param context: Access to all HW modules and system data.
        :param params: Specific parameters for this execution (e.g. thresholds).
        :return: Analysis results with status (PASS/FAIL).
        """
        pass
```

4.2 Esempio di Algoritmo Cross-Modulo: `TargetValidationAlgo`

Questo algoritmo dimostra come accedere a due moduli diversi tramite il contesto.

```
class TargetValidationAlgo(BaseAlgorithm):
    """
    Verifies if a target injected via TargetSim appears on the 1553 Bus.
    """

    def execute(self, context: TestContext, params: Dict[str, Any]) -> Dict[str, Any]:
        # 1. Get injected target info from TargetSim
        injected_targets = context.target_sim.get_active_targets()

        # 2. Get current messages from 1553 Bus
        bus_data = context.bus1553.get_message("B6") # Example: Radar Track Message

        # 3. Perform comparison logic (Algorithm-specific)
        match_found = self._compare_data(injected_targets, bus_data, params['tolerance'])

        return {
            "status": "PASS" if match_found else "FAIL",
            "details": f"Target match status: {match_found}",
            "timestamp": context.bus1553.get_system_timestamp()
        }

    def _compare_data(self, injected, bus, tolerance):
        # Implementation of the mathematical comparison
        return True
```

---

5. Level 3: Test Execution (JSON & Python)

5.1 Test Plan (Logica Operativa)

L'utente scrive il test decidendo *cosa* fare e *quale* algoritmo di libreria usare per validare l'azione.

Esempio di Test Script (Python):

```
def run_radar_track_test(context: TestContext):
    # Step 1: Inject target
    context.target_sim.inject_target(id=101, distance=50, azimuth=0)

    # Step 2: Use an algorithm from the dynamic library to verify
    # The TestManager handles the call passing the context
    result = context.run_algorithm("TargetValidationAlgo", tolerance=0.5)

    if result["status"] == "FAIL":
        context.log_event("Target not detected by radar!", "ERROR")
```

---

6. Meccanismo di Data Flow e Visibilità

Per garantire che gli algoritmi abbiano i dati necessari, l'Orchestratore implementa le seguenti regole:

1. **Data Pull**: Gli algoritmi "pescano" (pull) i dati dai moduli hardware tramite il `TestContext`. I moduli devono quindi mantenere un buffer degli ultimi dati ricevuti.
2. **Shared State**: Il `TestContext` mantiene una `results_cache`. Se l'Algoritmo A produce un risultato, l'Algoritmo B può leggerlo dal contesto nello step successivo.
3. **Concurrency**: Mentre i test e gli algoritmi girano nel thread principale (o thread di esecuzione test), i moduli HW continuano a riempire i loro buffer in thread separati, garantendo che il `TestContext` veda sempre dati aggiornati.

---

7. Workflow per i Collaboratori

- **Sviluppatore Modulo (L0)**: Deve assicurarsi che il modulo esponga metodi "Getter" (es: `get_last_frame()`, `get_bus_state()`) accessibili dal `TestContext`.
- **Sviluppatore Algoritmo (L1)**: Riceve il `TestContext` e scrive la logica di calcolo. Non deve preoccuparsi di come i dati arrivano, solo di come elaborarli.

- **Sviluppatore Test (L2)**: Utilizza i comandi dei moduli per muovere il sistema e gli algoritmi per validare i requisiti.