The software Architecture of FMOS: a Formal model enabled Multi agent Operating System for distributed real-time Agentic applications in IoT

Flora Amato[1,†], Francesco Moscato[2,*,†]

[1]University of Naples Federico II, Via Claudio, 21, 80125 Naples (NA), Italy
[2]University of Salerno, Via Giovanni Paolo II, 132, 84084 Fisciano (SA), Italy

Abstract

Growing interest in Artificial Intelligence is driving research in distributed autonomous systems, particularly within the Internet of Things (IoT) domain. A key focus is the composition of complex, autonomous, and intelligent services. In the context of IoT and critical systems, this necessitates robust low-level middleware and operating systems capable of automatic, autonomous planning, and scheduling of intelligent agents. This work introduces FMOS, a distributed operating system designed to facilitate the automatic, goal-based composition and scheduling of intelligent agents. FMOS leverages Multi-Agent Systems models to enable goal-based agent planning, encompassing the management of both hard and soft real-time applications. Furthermore, FMOS employs model checking techniques within its planning activities to ensure deadlock-free scheduling.

Keywords

Multi Agent Systems, Agentic Systems, Real-Time Scheduling, Autonomous Planning

1. Introduction

The Internet of Things (IoT) has evolved from a conceptual paradigm into a pervasive reality, with billions of smart objects interconnected to provide critical services across agriculture, healthcare, and industrial sectors. As these systems scale, the traditional centralized cloud computing model faces bottlenecks regarding latency, bandwidth, and reliability, necessitating a shift toward distributed paradigms like Fog and Edge computing.

With continuous advancements in artificial intelligence, as well as in hardware technology whose target are autonomous, embedded systems, operating systems are experiencing many transformative changes.

In parallel, the integration of Neural Processing Units (NPUs)—specialized processors optimized for AI computations—into modern devices has become increasingly prevalent, further accelerating the shift toward intelligent computing, even for applications characterized by low computational resources and by low power consumption requirements. These applications usually deal with distributed resources and software agents: this introduces new challenges for operating systems, particularly in planning cooperation of tasks and resources, as well as in task scheduling.

Moreover, the proliferation of IoT devices has made distributed computing a dominant trend, requiring operating systems to efficiently manage and coordinate distributed resources. In these contexts, in order to enable autonomous computing, automatic composition of tasks and proper resource scheduling should be provided.

Despite rapid progress in both hardware and software, current operating systems exhibit notable limitations. To address these issues and adapt to future trends, operating systems must evolve into smarter, more secure, and efficient architectures.

WOA 2026: the 27th Workshop From Objects to Agents, June 15–17, 2026, Salerno, ITALY
*Corresponding author.
†These authors contributed equally.
flora.amato@unina.it (F. Amato); fmoscato@unisa.it (F. Moscato)
0000-0002-5128-5558 (F. Amato); 0000-0002-7056-4579 (F. Moscato)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-10-30

CEUR-WS.org/Vol-4307/paper38.pdf

Parallel distributed systems offer superior performance and design simplicity for complex task achievement. Multi-Agent Systems (MAS) are a powerful tool for autonomous planning and scheduling, allowing entities to cooperate and negotiate for resources to reach shared objectives.

However, ensuring that these autonomous agents respect strict timing constraints—"real-time compliance"—remains a major problem to solve in safety-critical applications.

We think that a notable improvement in operating system used in these domains consist in the development of a new model for scheduling operations and resources.

In particular, since we are going to provide autonomous capabilities to classical operating systems, we propose here a new model for tasks allocation, cooperation and scheduling, that exploit the concept of *Agent* instead of *task*, providing the architecture of an operating system that explicitly addresses *Agentic* tasks.

The IoT has traditionally been viewed as a network of passive sensors and actuators connected to a centralized cloud for processing. However, a significant paradigm shift has occurred: the emergence of Agentic Systems. These are no longer simple rule-based devices but autonomous entities capable of reasoning, long-term planning, and independent decision-making in dynamic environments [1, 2]

The concept of *Agentic* systems refers to computational frameworks or entities endowed with autonomous capabilities to perceive, reason, decide, and act within dynamic environments. These systems, often underpinned by artificial intelligence (AI) technologies, are designed to perform complex tasks independently, adapt to changing conditions, and interact seamlessly within the Agentic environmnet, with other agents or users, other systems, or their surroundings.

As we use a model for agents based on Beliefs, Desires, Intentions (BDI) logics [3]. Main properties of an Agentic system are:

- Autonomy: we aim at implementing an operating system able to plan execution of *autonomous* agents: this implies that both reactive and pro-active agents will be managed. Agentic systems operate with minimal human intervention, leveraging algorithms to process sensory data, interpret context, and execute actions. Autonomy enables these systems to navigate uncertainty and execute tasks efficiently in diverse scenarios.
- Goal-Oriented Behavior: These systems are designed to achieve specific objectives or satisfy predefined goals. They utilize planning and optimization techniques to identify optimal pathways, resolve conflicts, and adapt their strategies to meet target outcomes.
- Perception and Environment Interaction: Agentic systems perceive their environment through sensors or data inputs, allowing them to construct representations of the external world. This perception enables contextual awareness, facilitating informed decision-making and adaptive behavior.
- Learning and Adaptation: Many agentic systems integrate machine learning algorithms to improve performance over time. By analyzing historical data, identifying patterns, and learning from feedback, these systems refine their operations and enhance their adaptability to novel conditions.
- Interactivity and Collaboration: Agentic systems are often designed to communicate and collaborate, either with humans or other systems. This includes natural language processing for human interaction, as well as protocol-based communication for multi-agent systems or distributed computing frameworks.

Agentic systems are integral to numerous domains, including but not limited to: Autonomous Vehicles, Healthcare, Smart Manufacturing, Virtual Assistants. In this domain, agentic systems have several open research issues like Security and Privacy problems, efficient and safe resource management, and, of course, Explainability of decision-making process that is a "must" in safety-critical systems.

The interest in IoT Ecosystems has moved beyond passive sensor networks toward Agentic AIoT [4], where distributed intelligence is embedded directly into physical assets. Recent surveys [5, 2] highlight that the complexity of these agentic ecosystems requires a fundamental shift in how we manage low-level resources. Traditional middleware often fails to provide the deterministic guarantees needed for

safety-critical applications [6]. This has led to a growing demand for edge orchestration standards that utilize probabilistic reasoning and model checking [7, 8].

FMOS addresses these problems by integrating a formal planner directly into the operating system's scheduling logic, moving from heuristic-based planning to provably correct scheduling via model checking.

This work describes the overall FMOS architecture, focusing on the methodology used to enact the planning activity. In the following, section 2 examines the transition of Multi-Agent Systems (MAS) toward proactive, goal-based "Agentic" strategies and identifies the critical lack of deterministic real-time support in current general-purpose operating systems. Section 3 details the four-layer architectural stack of FMOS, which bridges high-level BDI agent reasoning with a formal model-enabled real-time kernel to provide deterministic execution on physical hardware. Section 4 describes how FMOS treats scheduling as a reachability problem by transforming agent definitions into synchronized Timed Automata, ensuring that generated plans are provably free of deadlocks. Section 5 validates the system through a remote knee-rehabilitation case study. Finally, section 6 contains concluding remarks.

2. State of the Art

Agentic systems extend the capabilities of traditional AI by integrating mechanisms of autonomy and interaction with the physical environment, allowing them to act as "collaborative partners" rather than mere tools [1, 9]. In critical domains such as industrial automation, autonomous transportation, and healthcare, these systems must not only be "intelligent" but also strictly reliable and responsive in real-time [10].

This evolution is driven by the convergence of MAS and high-level artificial intelligence. Agentic systems extend the capabilities of traditional AI by integrating mechanisms of autonomy and interaction with the physical environment, allowing them to act as "collaborative partners" rather than mere tools [9, 11]. In critical domains such as industrial automation, autonomous transportation, and healthcare, these systems must not only be "intelligent" but also strictly reliable and responsive in real-time.

Agentic Systems are characterized by three core pillars: Autonomy, Sociality, and Reactivity. Recent surveys [1, 2] emphasize that the integration of Deep Reinforcement Learning (DRL) and Large Language Model (LLM)-based planning has allowed agents to transition from reactive behaviors to proactive, goal-based strategies. According to a survey on the frontiers of MAS [1], the "agentic" nature of modern systems allows for the composition of services across fragmented pipelines. For instance, in the context of 6G networks, AI agents are expected to be integrated directly into the scheduling and air interface, facilitating autonomous operations that were previously managed by centralized controllers [12]. This shift necessitates a transition from high-level "middleware" solutions to low-level, embedded operating system support to ensure that these agentic behaviors can be scheduled with microsecond-level precision.

The last few years have seen an increasing interest in MAS and Agentic Systems highlighting how MAS now focuses on complex problem solving and planning in embodied physical environments [1]. In particular in IoT and edge ecosystems for industrial applications, agents are evolving into multi-agent ecosystems where specialized roles and orchestrated collaboration enable teams of agents to tackle complex problems autonomously [9]. Here, MAS systems are a promising framework for deploying autonomous systems [2], where MAS systems with Agentic properties decentralize decision-making across distributed nodes, improving reliability and safety [11].

Despite the intelligence of these agents, a critical gap exists in the underlying infrastructure. Most agentic frameworks operate at the application layer, relying on general-purpose operating systems (GPOS). However, in real-time critical systems, GPOS cannot guarantee deterministic scheduling. The market for Embedded Real-Time Operating Systems (RTOS) for IoT is projected to grow significantly through the next years, driven by the demand for "real-time determinism and low-latency performance" in autonomous vehicles and healthcare monitoring [13].

Table 1
Comparison with State of The Art

Dimension / System	**High-Level Agentic AI Platforms**	**ROS 2.0 Embedded Middleware**	**FMOS**
Foundational Paradigm	Heuristic & LLM-driven execution workflows.	BDI Agent Layer over real-time communication nodes.	Online Formal Model Enabled Kernel
Temporal Logic & Formalisms	Probabilistic routing / Rule-based checks.	Ad-hoc priority levels and callback guards.	Synchronized Timed Automata verified via TCTL.
Verification / Planning Type	Runtime evaluation for accuracy/safety exceptions.	Heuristic, deterministic scheduling paths.	Online Counterexample-Guided Planning
Integration Layer	Application / Cloud Container Layer.	Middleware Layer sitting above a host OS.	Native Operating System Kernel
Hardware Enforcement	Best-effort execution.	Monitored real-time priority distribution.	Dynamic translation of traces to Task Control Blocks.
	IMITATOR4AMAS Strategy Synthesizer	**Krichen's UPPAAL Framework**	**FMOS**
Foundational Paradigm	Continuous-time Asynchronous Multi-Agent Systems (CAMAS).	Supervisory Control Theory via Finite State Automata.	Online Formal Model Enabled Kernel
Temporal Logic & Formalisms	Parametric Timed Automata mapped via STCTL.	Networks of Timed Automata verified via UPPAAL.	Synchronized Timed Automata verified via TCTL.
Verification / Planning Type	Offline Strategy Synthesis (Memoryless imperfect info).	Offline Safety Verification & Access Control Strategy.	Online Counterexample-Guided Planning
Integration Layer	Standalone Software Tool (OCaml-based).	Offline Modeling Environment (UPPAAL Engine).	Native Operating System Kernel.
Hardware Enforcement	None (Purely theoretical/ simulation strategy).	Manual code implementation from verified models.	Dynamic translation of traces to Task Control Blocks.

For an agent to be truly effective in a critical IoT environment, the system must be able to:

- Schedule Intentions: Moving beyond task-based scheduling to goal-based scheduling where the OS understands the agent's intent.
- Distributed Autonomy: Managing resources across multiple nodes without a single point of failure.
- Formal Safety: Ensuring that the autonomous planning of agents does not lead to deadlocks or missed deadlines in safety-critical scenarios.

Recent trends emphasize that "security, safety and real-time processing are no longer options" for embedded IoT devices [14]. This motivates the development of FMOS, a Formal model enabled Multi-agent Operating System. Unlike traditional RTOS, FMOS is designed to schedule the agentic logic itself, using formal verification to ensure that the distributed, autonomous composition of tasks remains within the strict temporal bounds required for real-time safety.

In the domain of cyber-physical systems and robotics, attempts to bridge autonomous decision-making with strict execution deadlines have historically leaned on advanced robotic middleware rather than the operating system kernel. To enforce timing accuracy, these architectures rely on the real-time execution extensions provided by frameworks like Robotic Operating System (ROS)[1] 2.0 [13].

To bypass the non-determinism of probabilistic agent planning, recent research has turned to formal methods, explicitly leveraging Timed Automata (TA) to guarantee behavioral safety [15].

A notable milestone in recent literature is IMITATOR4AMAS [16], a state-of-the-art tool designed for continuous-time asynchronous multi-agent systems (CAMAS). While IMITATOR4AMAS can explicitly construct a memoryless, imperfect-information strategy that satisfies temporal properties, it functions as an offline verification tool. Similarly, Krichen [15] developed a Timed Automata-Based Strategy via UPPAAL to control autonomous unmanned aerial vehicle (UAV) access to critical safety zones. This approach enforces real-time timing invariants and guards to model state transitions securely, validating reachability and safety prior to deployment.

However, across all such recent frameworks, a fundamental literature gap persists: existing formal verification tools operate strictly as external analytical engines.

The Table 1 contrasts the presented work (FMOS) with the leading state-of-the-art systems and verification frameworks:

Traditional RTOS infrastructures (such as stock FreeRTOS, VxWorks, or Zephyr) are inherently rigid, operating solely on deterministic task execution blocks, completely unaware of high-level task intent,

[1] https://www.ros.org

goals, or cognitive contexts. Advanced frameworks like ROS 2.0 attempt to bridge this by introducing BDI agents at the middleware layer, but the kernel remains stuck executing decoupled, static tasks.

There is no prior witness in scientific literature of an RTOS scheduler that explicitly replaces task-based queues with agentic intent-driven scheduling. FMOS achieves this by directly extending the standard Task Control Block (TCB) with cognitive bounds, making the core operating system kernel the direct orchestrator of multi-agent collaborations.

3. FMOS Architecture

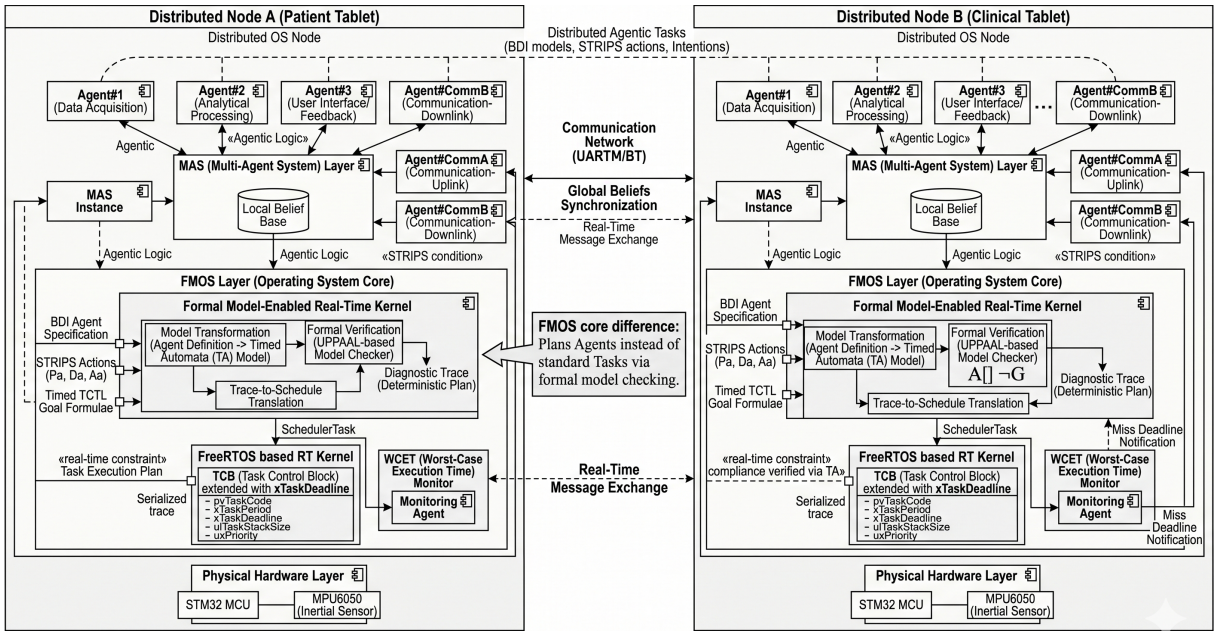

Figure 1: FMOS Architecture

Figure 1 illustrates the architectural model of FMOS . The architecture is explicitly designed to address the challenges of distributed, safety-critical IoT ecosystems by shifting the paradigm from traditional task-based execution to autonomous, goal-oriented agent planning. As highlighted in the central callout of the diagram, the core distinguishing feature of FMOS is its ability to plan Agents instead of standard Tasks via formal model checking and Counterexample-guided generation of plans, as it will be reported in the next Sections.

The architectural stack within each node is vertically integrated, bridging high-level agentic reasoning with low-level deterministic execution across four primary layers: the Distributed Agentic Tasks level, the MAS Layer, the FMOS Layer (Operating System Core), and the Physical Hardware Layer.

At the highest level of abstraction, the system defines Distributed Agentic Tasks utilizing Belief-Desire-Intention (BDI) models, STRIPS actions, and formal Intentions. These conceptual models are instantiated within the MAS Layer as a set of concurrently operating agents.

Central to the MAS Layer on each node is the Local Belief Base, a repository representing the agent's perception of the environment. To maintain a cohesive distributed intelligence, the architecture relies on continuous Global Beliefs Synchronization. This is facilitated by the central Communication Network which enables Real-Time Message Exchange and STRIPS condition synchronization between nodes, ensuring that local planning algorithms operate on an accurate representation of the global state.

The Formal Model-Enabled Real-Time Kernel acts as an online formal planner that guarantees deadlock free and real-time compliant execution.

Once a valid plan is mathematically verified, FMOS must enforce it at the microprocessor level. The Trace-to-Schedule Translation module serializes the diagnostic trace and passes it to the underlying real-time kernel, which is based on FreeRTOS.

To accommodate agentic planning, standard real-time scheduling mechanisms have been fundamentally extended. The diagram illustrates a modified Task Control Block (TCB). Alongside standard FreeRTOS attributes such as ‘pvTaskCode‘ (pointer to task logic), ‘ulTaskStackSize‘, and ‘uxPriority‘, the TCB is explicitly extended with ‘xTaskPeriod‘ and, crucially, ‘xTaskDeadline‘. A dedicated ‘SchedulerTask‘ orchestrates the execution of these modified TCBs, ensuring that the real-time constraints—whose compliance was formally verified via the Timed Automata—are strictly respected during physical execution.

To ensure the safety-critical requirements of the system are maintained dynamically, the FMOS Layer incorporates a WCET (Worst-Case Execution Time) Monitor. A dedicated Monitoring Agent continuously evaluates task execution against the formalized ‘xTaskDeadline‘. Should any anomalous latency occur, the monitor is designed to immediately issue a Miss Deadline Notification, triggering the system’s QoS and replanning protocols via the communication agents.

Finally, the lowest tier of the architecture is the Physical Hardware Layer. As depicted in the specific implementation instance, the distributed OS nodes execute on STM32 Microcontroller Units (MCUs), interfacing directly with Sensors. The formal bridging of this physical layer through to the high-level BDI agent layer encapsulates the holistic, full-stack approach of the FMOS architecture to distributed, real-time IoT orchestration.

4. FMOS Planning and Scheduling

In the FMOS Architecture, scheduling actions enable autonomous planning of Agents involved in a distributed collaboration in order to fulfill common Goals. From the Scheduling perspective, we want to solve the following problem:

Given a set of Agents in a distributed environment with their abilities (i.e. ***actions*** *they are able to perform), is there a way to orchestrate agents with proper communications and actions executions, in order to fulfill a common Goal?*

FMOS, the planning phase is not merely a search for resource allocation but a rigorous process of Planning by Model Checking. While traditional distributed systems rely on static priority-based scheduling or dynamic algorithms like Earliest Deadline First (EDF), FMOS addresses the inherent unpredictability of multi-agent interactions by treating the scheduling problem as a reachability problem within a formal model. This ensures that the resulting plan is not only optimal in terms of sequence but provably free of deadlocks and timing violations before execution begins.

The core of this planner lies in its ability to bridge high-level **Practical-Reasoning Agents (PRA)** from the BDI model, with low-level real-time constraints.

The FMOS *planner* utilizes the BDI model to represent the internal decision-making process of its distributed agents, where:

- Beliefs: they represent agents perceptions of the environment. In an IoT environment these include value of sensors acquired by single agents and their state space representation, as well as values they believe other agents have in their own state space representation.
- Actions: they consist in possible agent operations, and a formal description about *when* actions are enabled to execute and *what* are their preconditions and effects.
- Goals: they are the target state of the whole environment agents want reach.
- Intentions: they represent persistent wills that drive planning, constrain future deliberation, and influence beliefs.

Each agent is defined as a rational system capable of reactive and proactive behavior.

In particular, **Actions definition** follow a S.T.R.I.P.S.(STanford Research Institute Problem Solver)-like model [17], where, given an action $\alpha \in Actions$, it can be described by a triple:

$$\langle P\alpha, D\alpha, A\alpha \rangle$$

with:

- $P\alpha$ are actions **Preconditions**, i,e, a set of First Order Logic predicates that have to evaluate true in Agent actual state in order to execute the action;
- $D\alpha$ and $A\alpha$. Delete List, and Add List respectively: these lists are the Effects an action produces after its execution. They are the set of predicates that respectively hold no more and newly in agents state spaces. We call **Effects** or **PostContitions** predicates in aforementioned lists.

A **Planning Problem** is therefore represented in the ST.R.I.P.S. formalism as the triple:

$$\langle \Delta, \mathcal{O}, \gamma \rangle$$

Where:

- Δ is the set of values of the agent's Beliefs in the initial state;
- $\mathcal{O} = \{\langle P\alpha, D\alpha, A\alpha \rangle | \alpha \in Ac\}$, with Ac being the set of actions, is the set of descriptions of all possible actions that the agent can perform;
- γ is a set of formulas in FOL that represent the goal (i.e.the formulas that holds in the state agent choose as planning target)

The output of the planning process will therefore be a Plan expressible as:

$$\pi = (\alpha_1, \alpha_2, \alpha_3, \cdots, \alpha_n)$$

That is, the list of actions the agent must perform to move from the initial state to the goal.

In FMOS **Scheduling is a Continuous Process** where all nodes in the distributed environment perform a *Continous Planning Loop*(Listed in Algorithm1).

Algorithm 1 Practical-Reasoning Agent Scheduling Loop

1: Evaluates the list of actions of *Agents in the whole distributed environment*
2: Identifies the actions to perform on local Agents
3: Prepares synchronization messages to interact with other nodes
4: Execute local actions (synchronized by messages in previous step)
5: Evaluates local state after actions
6: Collects Beliefs from other agents in other nodes
7: Control if the Goal is still feasible (if not, it performs a re-planning action)
8: Produce an agreement on a global plan from local plans

In particular, more in detail, first and last step of Algorithm1 are detailed in 2
This algorithm manages the lifecycle of an agent through the following phases:

- *Deliberation* (Lines 0–6): The process where the agent updates its beliefs based on environmental perception ($brf(B, p)$) (where $B$is the set of Beliefs of all agents on the node, and p is the set of perceptions of agents during the current loop) and selects its current Desire from available options ($options(B, I)$, where I are the Intentions of agents scheduled in the current loop). Actions to execute depend on the Intentions that can be executed in the current loop depending on Preconditions ($filter(B, D, I)$)
- *Planning* (Line 7): The reasoning engine generates an ordered list of actions (π) to reach the goal state from the current belief state.

Algorithm 2 Practical-Reasoning Agent Scheduling Loop

1: $B \leftarrow \Delta$; ▷ Δ is the set of initial Beliefs
2: $I \leftarrow I_0$; ▷ I_0 are initial intentions
3: **while** true **do**
4: get next percept p through *see*(...) function;
5: $B \leftarrow brf(B, p)$;
6: $D \leftarrow options(B, I)$;
7: $I \leftarrow filter(B, D, I)$;
8: $\pi \leftarrow plan(B, D, Ac)$;
9: **while not** ($empty(\pi)$ **or** $succeded(\pi)$ **or** $impossible(I, B)$) **do**
10: $\alpha \leftarrow head(\pi)$;
11: $execute(\alpha)$;
12: $\pi = tail(\pi)$;
13: get next percept p through *see*(...) function;
14: $B \leftarrow brf(B, p)$;
15: **if** ($reconsider(I, B)$) **then**
16: $D \leftarrow options(B, I)$;
17: $I \leftarrow filter(B, D, I)$;
18: **endif**
19: **if** ($notsound(\pi, I, B)$)
20: $\pi \leftarrow plan(B, I, Ac)$;
21: **endif**
22: **EndWhile**
23: **EndWhile**

- *Execution* (Lines 8–11): The agent performs actions one at a time until the plan is complete, the goal is achieved, or the plan becomes impossible.
- *Reconsideration* (Lines 12–17): After each action, the agent updates its beliefs. If the current goal is no longer reachable or optimal, the agent reconsiders its intentions.
- *Replanning* (Lines 18–20): If the intention remains valid but the current plan is no longer sound due to environmental changes, a new plan is generated.

Some questions arise here:

- What policy is the Deliberation process based on?
- When can we define a goal as no longer achievable and a plan no longer feasible, and how often should we ask ourselves this question? This point is in turn of fundamental importance; in particular when dealing with real time constraints.

Anyway answers provided in FMOS to these questions are beyond the scope of this work and will be reported in future works.

Here, we focus on the *planning* phase, that FMOS implements by transforming the distributed environment into a network of synchronized Timed Automata (TA), the system can exhaustively explore the state space to identify a valid execution trace.

4.1. Real Time Planning Phase

The planner's role In FMOS is to convert these high-level Intentions into a sequence of atomic actions that satisfy the timing constraints of the Goal to fulfill in the distributed environment.

Since Step 6. in Algorithm1 updates the Beliefs of Agents on each Node with the Beliefs of all agents in the distributed environment, We can enact planning on each node independently from other nodes, because they have a local representation of the whole system, and they will agree on a common plan in Step 8 of Algorithm1

The planner's role is to convert these high-level Intentions into a sequence of atomic actions (e.g., ReadAccelerations, SendData) that satisfy the timing constraints of the real-time kernel.

4.2. Formalization via Timed Automata

To enable automated verification, the system state is modeled as a tuple (L, l_0, C, A, E, Inv), where: L is the set of locations (states).

$l_0 \in L$ is the initial location.

C is a set of clocks that progress synchronously.

A is the set of actions and synchronization channels.

$E \subseteq L \times A \times Bounds(C) \times 2^C \times L$ is the set of edges (transitions) defined by guards and clock resets. $B(C)$ is a set of conjunctions over non-negative real valued clocks.

$Inv : L \rightarrow Bounds(C)$ is the set of invariants that assign an upper bound on time for specific locations.

From a formal perspective, a timed automaton constitutes a finite-state machine augmented with a set of real-valued clock variables. The framework adopts a dense-time model wherein each clock variable evaluates to a real number, and all clocks within the system progress synchronously.

A complex system is represented as a network of multiple timed automata operating in parallel. This model is further enhanced by the inclusion of bounded discrete variables, which are integrated into the global system state. Consequently, a comprehensive state of the system is defined by the following components:

The specific locations (nodes) of all constituent automata.

The current clock values.

The values of the discrete variables.

In this concurrent architecture, every automaton may execute an edge—alternatively referred to as a transition—independently or through synchronization with other automata in the network. The integration of these formalisms allows for the rigorous verification of system properties. The primary properties of interest include:

Reachability: Determining whether a specific state S is accessible via a valid execution run from the initial state.

Safety: Ensuring that the system avoids "failure" states by guaranteeing no path exists to prohibited configurations.

Liveness: Validating that a specified condition will eventually be satisfied during the system's operational lifecycle.

Deadlock Absence: Verifying that the system remains progressive and does not enter a state from which no further transitions can be executed.

These queries are processed by a reasoning engine that returns a boolean result or a diagnostic trace, which serves as an exemplary path to illustrate the property under investigation.

In FMOS, we exploit the ability of Model Checkers for Timed Automata to find Counterexamples when Reachability properties cannot be verified on a Model.

Practically, it is possible to check **Reachability** on *Goals* if we can model agents with their actions with a Timed Automata (with their timing constraints as well). Anyway, this is a time-consuming operation and the simple verification of a Logical property on a Model does not imply that a concrete plan is found to reach the Goal.

Hence FMOS enact a *Counterexample-Guided planning Strategy*(3).

Counterexample may not be unique, but, if they exists, they are evaluated as fast as possible, without exploding and examining the whole state space of the model, making available a plan in a short time.

4.3. Agents to Timed Automata Model Transformation

Agents defined at FMOS systems level have to be translated into timed automata in order to compute a plan to reach common Goals. In brief, Algorithm4 resumes the main phases of the procedure used to translate Agents definition into a Timed Automata.

Algorithm 3 Counterexample-Guided planning Strategy

1: Define a Reachability Formula on The Timed Automata: Given an Agent Model, Does It Exist a scheduling of actions π that reaches the global Goal $\mathcal{G} : E <> G$ (that in CTL Logic means: "Is There Any π in all possible plans that reaches the goal $\mathcal{G}$?")
2: Negate the property to check defined in the previous step: $A[]\neg G$ (that in CTL Logic sounds like: "Is it always true that we **cannot** find any π among all possible plans that reaches the goal $\mathcal{G}$?")
3: **if** the model verifies the last property **then**
4: the Goal is not reachable and we cannot find any plan to reach the Goal
5: **else**
6: the model checker finds a counterexample that negates the formula. The counterexample here is a plan that negates the property: This means that a π exists that reach the goal, **and the counterexample** is this π
7: **end if**

Algorithm 4 Counterexample-Guided planning Strategy

1: Create a TA sub-model for each agent;
2: Agents Beliefs are assigned to variables in sub-models;
3: For all actions in Agents definition, create a proper transition in its sub-model;
4: Action preconditions become the edge guards;
5: Effects become the edge updates;
6: The goal, and in particular the goal condition, is properly translated so that it is understandable to the reasoning engine that reasons on the TA.

In FMOS, a distributed application is modeled as a network of several such automata running in parallel. For example, the Session Automaton coordinates the lifecycle of an entire rehabilitation session, while individual Agent Automata (e.g., Agent1, Agent2) model the specific logic and timing of distributed tasks on their respective nodes.

4.4. Synchronization and Communication Mechanics

Communication between distributed nodes is a critical source of non-determinism in IoT systems. FMOS models this using urgent channels and broadcast synchronization in Timed Automata.

Handshake Synchronization: When one automaton performs a send action (e.g., start_agent1!), another must be ready for a receive action (start_agent1?). This models the coordination between the global Session manager and the local Node controllers.

Timing Invariants: To enforce real-time compliance, locations are assigned invariants such as $x \leq WCET$, where x is a clock variable and $WCET$ is the Worst-Case Execution Time of the corresponding task. If the clock exceeds this value without a transition occurring, the state becomes invalid, signaling a timing failure during the planning phase.

4.5. The Formal Specification Language: Timed CTL (TCTL)

The planner uses Timed Computation Tree Logic (TCTL) to define the properties the schedule must satisfy. These are interpreted over continuous computation trees where paths represent the evolution of the system over time.

Reachability ($E <> \phi_{goal}$): Specifies that there exists at least one path where the goal state (e.g., session completion) is eventually reached.

Safety ($A[]\phi_{safe}$): Ensures that a property ϕ is satisfied in all possible states, primarily used to verify that no task ever exceeds its deadline.

Deadlock-Freeness ($A[]$not deadlock): Confirms that the system will never enter a state where no further transitions are possible, a common issue in distributed MAS due to circular wait conditions.

4.6. Real-Time Constraints and Overhead Analysis

A critical aspect of the formal planner is accounting for the overhead introduced by the scheduling logic itself. In FMOS, the SchedulerTask is assigned the highest priority to act as a "fictitious scheduler" that moves other tasks between Suspended and Ready states based on the validated trace.

The total execution time on a node is calculated as:

$$\Delta T_{total} = \Delta T_{tasks} + \Delta T_{comm} + \text{overhead}$$

Experimental analysis of the FMOS prototype showed that for a typical 60-second telerehabilitation session, the planning and scheduling overhead remained minimal, approximately 116 ms (roughly 0.19% of the session duration). This confirms that the formal planning approach provides the necessary safety and liveness guarantees without compromising the system's real-time performance.

5. Experimental Results

5.1. Case Study: remote rehabilitation scenario

The planner's effectiveness is best demonstrated during distributed execution. In the remote rehabilitation scenario, the formal model includes a monitoring agent that halts the system if a deadline miss is detected on any node.

Let us consider a longitudinal knee remote re-habilitation protocol specifically designed for geriatric populations. The intervention comprises a scheduled series of therapeutic sessions supported by a dual-component technological architecture:

Kinematic Data Acquisition: A knee-mounted inertial measurement unit (IMU) is utilized to capture high-fidelity triaxial accelerometry data. This enables the quantitative assessment of biomechanical variables and movement trajectories associated with each prescribed exercise.

Bidirectional Communication Interface: The system utilizes a distributed network architecture involving two synchronized tablet devices. These interfaces facilitate real-time, internet-mediated communication between the patient and the therapist, ensuring remote supervision, clinical feedback, and adherence monitoring throughout the rehabilitative process.

The acquisition and transmission of kinematic data within the remote re-habilitation framework follow a structured, post-process paradigm. Triaxial acceleration vectors captured by the inertial sensor are buffered and locally archived on the patient's interface upon the completion of each constituent time interval within a session. Following the conclusion of the entire therapeutic session, the aggregated raw datasets are transmitted via an asynchronous uplink to the therapist's workstation (tablet).

Upon receipt, the data undergo computational processing to facilitate clinical movement analysis. The therapist utilizes these data to evaluate the biomechanical fidelity of the patient's knee movements against prescribed standards.

The system architecture is structured as a distributed network comprising two primary nodes, as illustrated in Figure 2:

- Node A (Patient Interface): This node, instantiated on the patient's tablet, serves as the primary local data repository and visualization interface. It is responsible for the initial buffering and storage of triaxial acceleration vectors captured by the IMU. Furthermore, Node A functions as the terminal for the delivery of longitudinal session results, enabling the patient to access post-hoc clinical feedback regarding the biomechanical accuracy of their knee movements.
- Node B (Clinical Interface): This node represents the therapist's tablet, which acts as the central processing unit for the session data. Node B ingests the raw triaxial acceleration data transmitted from Node A to perform the necessary computational analyses. These operations generate the definitive session metrics and diagnostic results, which form the empirical basis for the therapist's evaluation and subsequent clinical guidance.

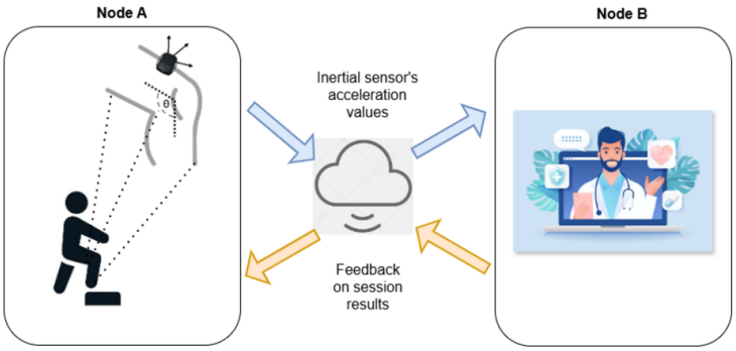

Figure 2: Remote rehabilitation scenario

To formalize the functional components of the system for a scientific publication, the software architecture is modeled as a Multi-Agent System (MAS) distributed across the network nodes. The agents are characterized by their specific operational responsibilities, data processing tasks, and synchronization protocols. In particular:

Agent#1 (Data Acquisition - Node A): Facilitates the ingestion of raw kinematic data. It interfaces with the inertial sensor to retrieve triaxial acceleration vectors in real-time, subsequently committing these values to the global belief base of Node A for local persistence.

Agent#2 (Analytical Processing - Node B): Serves as the primary computational engine. It retrieves the triaxial datasets to calculate the pitch and roll orientation angles for each discrete movement. This agent performs a comparative analysis between the derived angles and predefined safety thresholds to determine movement fidelity. The synthesis of these evaluations constitutes the formal session results.

Agent#3 (User Interface/Feedback - Node A): Manages the visualization of clinical outcomes. It retrieves the processed data to present the patient with a comprehensive summary of their performance and movement accuracy.

The synchronization and exchange of data between the distributed nodes are managed by a specialized communication layer, whose agents are:

Agent#CommA (Node A): Oversees the uplink of global beliefs (raw acceleration data) to Node B and the subsequent downlink of the processed session results. Furthermore, it implements a Quality of Service (QoS) monitoring protocol; it transmits an alert to Node B regarding any local deadline misses of critical functions and listens for reciprocal temporal notifications from the clinical node.

Agent#CommB (Node B): Acts as the counterpart to Agent CommA. It handles the reception of the patient-side belief base and the transmission of the finalized therapeutic results. It maintains architectural synchronization by exchanging temporal status notifications, ensuring that any computational latency or missed deadlines on Node B are reported to Node A for system-wide transparency.

Figure 3 depicts Sub-models generated by the Model Transformation Procedure before the planning starts. The system's formal verification and behavioral modeling are implemented via a Network of Timed Automata (NTA). The NTA consists of the following discrete automata:

Session: The orchestrator of the therapeutic protocol.

CurrentNode: Represents the active hardware state.

Agent Automata: Individual models for **Agent#1**, **Agent#2**, **Agent#3**, **Agent#CommA**, and **Agent#CommB**.

To optimize the model's architectural scalability, these agents are implemented as specific instances of a generic Agent class. This modular approach allows the network to be generalized through a single parameterized Agent(id) automaton, where the id variable dictates the functional identity and behavior of the instance.

The *Session* automaton serves as the high-level controller that dictates the sequential execution of the therapeutic workflow. It utilizes a synchronization primitive mechanism (signaling) to manage the

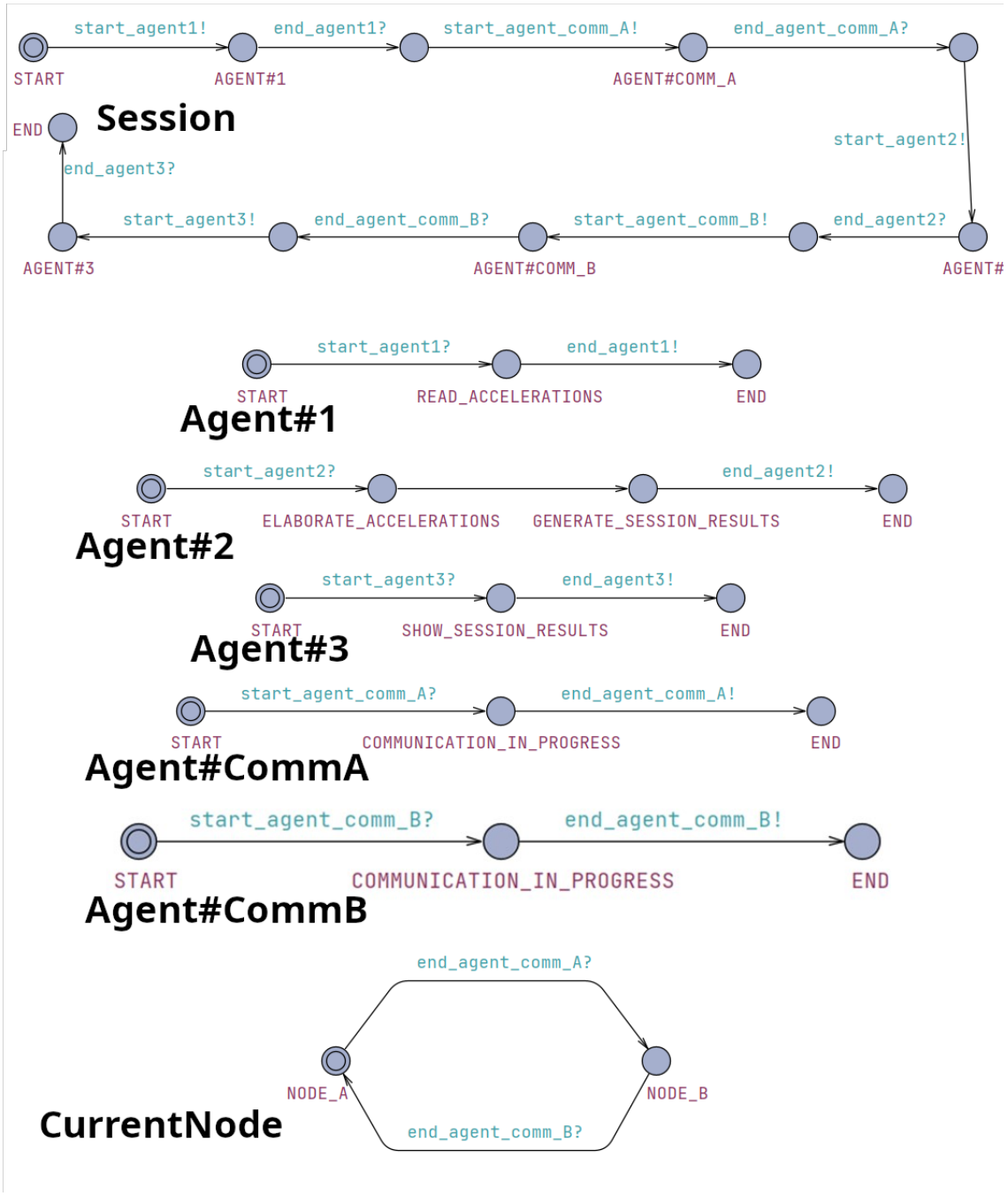

Figure 3: Planning Model Timed Automata

lifecycle of each agent:

Initialization: The process commences with the broadcast of the $start_agent1!$ synchronization signal, transitioning Agent#1 into an active state.

Sequential Execution: The Session automaton enters a wait state until it receives the reciprocal $end_agent1?$ signal. This handshake protocol is strictly adhered to for all subsequent agents in the pipeline, ensuring that data dependencies (e.g., acquisition before analysis) are respected.

Termination: Upon receiving the completion signal from Agent#3, the Session automaton transitions to the *END* location, formally concluding the therapeutic session and resetting the system state.

The core logic for data acquisition, kinematic analysis, and feedback visualization is distributed

across three primary agent automata:

Agent#1 : Upon receiving the $start_agent1?$ synchronization signal from the Session orchestrator, the automaton transitions to the$READ_ACCELERATIONS$ location. This state models the ingestion of triaxial acceleration vectors from the IMU. Once the reading operation is finalized, the automaton executes an $end_agent1!$ handshake with the Session controller and terminates at the END location.

Agent#2: Triggered by $start_agent2?$, this agent moves to the $ELABORATE_ACCELERATIONS$ location to derive the pitch and roll angles. It subsequently transitions to the $GENERATE_SESSION_RESULTS$ state, where diagnostic metrics are synthesized for clinical evaluation. The lifecycle concludes with the $end_agent2!$ signal and a move to the END location.

Agent#3: Following activation via $start_agent3?$, the automaton enters the $SHOW_SESSION_RESULTS$ state, modeling the rendering of therapeutic feedback on the patient interface. It synchronizes its termination via $end_agent3!$ before reaching the END location.

To facilitate distributed execution, the network employs dedicated agents for *data transmission and a global monitor for node tracking*:

Agent#CommA and **Agent#CommB**: These automata manage the bidirectional transfer of the global belief base. Agent#CommA (Node A to B) enters the $COMMUNICATION_IN_PROGRESS$ state upon $start_agent_comm_A?$ and signals completion via the broadcast $signalend_agent_comm_A!$. Agent#CommB (Node B to A) follows an identical structural logic, utilizing the dedicated $start_agent_comm_B?$ and $end_agent_comm_B!channels$.

CurrentNode: This automaton provides a formal abstraction of the active hardware environment. It initializes at $NODE_A$ to reflect the session's start on the patient's tablet. It monitors the $end_agent_comm_A?$ broadcast to transition to $NODE_B$, effectively shifting the execution context to the clinical workstation. Upon the $end_agent_comm_B?$ signal, the automaton reverts to $NODE_A$, signifying the return of the control flow to the patient.

5.2. Planning and Goal Reachability Specification

To determine if the goal state is attainable, we utilized counter-example plan generation strategy described in the previous sections.

Property Formulation: In order to produce a plan, we define a safety property:

$A[]\neg GoalCondition$

CounterExample Generation If the property is satisfied, it indicates that no valid execution path (plan) exists for the specified goal.

If the property is not satisfied, the verification engine identifies a counter-example. This counter-example constitutes a specific *plan* from the initial state to a state where the goal condition is met.

Specifically, we evaluated the reachability of the terminal state within the Session automaton using the following CTL specification:

$A[]\neg(Session.END)$

The verification engine concluded that this property is not satisfied, thereby proving that a valid plan exists (the Counterexample generated by the model checker) for the remote rehabilitation session. This trace serves as a formal representation of the desired system scheduling.

5.3. Experimental Setup and Empirical Results

The experimental setup included:

- Two STM32 NUCLEO-WB55RG board with ARM Cortex MCU and Bluetooth (BLE) embedded communication system;
- MPU6050 inertial sensors
- FreeRTOS as Base Operating System for FMOS.

The empirical evaluation of the implemented distributed scheduling system focused on two critical metrics: *inter-node communication latency* and the *computational overhead* introduced by the scheduling algorithm. While measurements were localized to Node A, the symmetrical nature of the architecture allows these findings to be extrapolated to Node B.

The protocol utilizes two classes of messages: control signals (fixed at 1 byte) and data structures (variable based on the global belief base). The Global Beliefs structure is composed of kinematic arrays defined over $\mathcal{N}$ intervals (in this example $\mathcal{N} = 6$):

Floating-Point Data: Five arrays (for IMU data) containing 8-byte double values, for $\mathcal{N} = 6$:

$(N) \times 8 \quad bytes \times 5 \quad fields = 240 bytes$

Integer Data: One array for results in 1-byte values, for $\mathcal{N} = 6$: **6 bytes**

$(N) \times 1 \quad byte \times 1 \quad field = 240 bytes$

Total Payload: 246 *bytes* per structure.

Accounting for all control messages and the 1-byte acknowledgment required for each transmission, the cumulative data volume for a single session is 502 *bytes*.

The measured Average Communication time (ΔT_{com}), that comprises the whole process for all intervals is

$\Delta T_{com} = 54ms$

The overhead represents the temporal cost of the scheduling logic itself, distinct from task execution and communication. Execution time was quantified using the FreeRTOS tick interrupt mechanism with *tick period set to 1 ms*. Based on the Worst-Case Execution Time (WCET) profiles, the total duration for tasks within the process waiting queue—primarily dominated by the exercise sessions—was calculated as:

In this experiment, the measured average execution time of the whole process is: $T_{ex} = 60.692ms$ (i.e about 1 minute). The scheduling overhead is isolated by subtracting the total task execution time and communication latency from the total observed runtime of the scheduler task. In the experiment, the Average Overhead is :

$Average_Overhead = 116ms$.

The analysis demonstrates that the distributed planning algorithm introduces a marginal overhead of 116 ms relative to a 1 minute session (i.e. less than 0.2% of the whole execution time). This minimal computational cost confirms the efficiency of the proposed MAS-based architecture for real-time remote re-habilitation applications.

6. Conclusions

This work has introduced FMOS, a formal-model-enabled Multi-Agent Operating System designed to bridge the gap between high-level autonomous agentic reasoning and the rigorous determinism required for real-time IoT environments . By integrating a formal planner directly into the operating system's scheduling logic, FMOS transitions from traditional heuristic task management to a paradigm of goal-oriented execution . The architecture successfully leverages the Belief-Desire-Intention (BDI) model to manage distributed autonomy, enabling agents to perceive, reason, and act within dynamic contexts while adhering to strict temporal constraints.

A fundamental contribution of this research is the implementation of a counterexample-guided planning strategy utilizing Timed Automata (TA) and the UPPAAL model-checking engine . This framework treats the scheduling problem as a reachability task, ensuring that the resulting plans are provably free of deadlocks and timing violations before execution commences . The feasibility of this approach was demonstrated through a longitudinal knee telerehabilitation case study, where complex multi-agent interactions were automatically orchestrated across distributed nodes.

Empirical evaluation on STM32 hardware confirmed that the formal planning approach is highly efficient for safety-critical IoT applications . The results indicated that the computational overhead introduced by the formal scheduler remained minimal—averaging approximately 116 ms—representing a negligible fraction of the total session duration . Furthermore, the system maintained low inter-node

communication latency, with an average time of 57 ms, validating the protocol's scalability for real-time remote healthcare monitoring.

In conclusion, FMOS provides a robust foundation for the next generation of "Agentic AIoT," where distributed intelligence is embedded directly into physical assets . Future work will expand the deliberation policies of the scheduling loop and explore the explainability of the decision-making process in diverse safety-critical scenarios, ensuring that autonomous agentic behaviors remain both intelligent and provably safe.

Declaration on Generative AI

During the preparation of this work, the authors used Gemini and Grammarly in order to: Grammar and spelling check, Paraphrase and reword. Further, the authors used Gemini for figures 1 in order to: Generate the image. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] A. Nguyen-Thi-Mai, H. Tran, M. Voznak, T. D. Ho, V. N. Van, S. CHATZINOTAS, H. TRAN DINH, A survey on challenges and emerging frontiers of multi-agent systems, in: SOICT 2025 (The 14th International Symposium on Information and Communication Technology), 2025.

[2] D. Maldonado, E. Cruz, J. A. Torres, P. J. Cruz, S. d. P. G. Benitez, Multi-agent systems: A survey about its components, framework and workflow, IEEE Access 12 (2024) 80950–80975.

[3] M. Wooldridge, An introduction to multiagent systems, John wiley & sons, 2009.

[4] L. Wu, W. Lu, Y. Zou, H. An, B. Wang, Agentic artificial intelligence of things (aiot) for just-in-time (jit) logistics and supply chain: a case of modular construction, International Journal of Logistics Research and Applications (2026) 1–37.

[5] G. T. Tran, T. Le Dinh, T. D. Le, Agentic ai across technologies, applications, and development domains: A systematic literature review, in: International Conference on Data Analytics & Management, Springer, 2026, pp. 587–601.

[6] A. K. Pati, Agentic ai: a comprehensive survey of technologies, applications, and societal implications, IEEE Access (2025).

[7] C. Chandler, Model checking for trustworthy reasoning in autonomous driving, Ph.D. thesis, University of Glasgow, 2025.

[8] H. M. Ul Haque, Z. Saleem, M. Ul Haque, G. Sidorov, et al., A probabilistic reasoning-based multi-agent formalism for smart environment, Computación y Sistemas 29 (2025) 2359.

[9] M. Abou Ali, F. Dornaika, J. Charafeddine, Agentic ai: a comprehensive survey of architectures, applications, and future directions, Artificial Intelligence Review 59 (2025) 11.

[10] N. Safir, M. Albezuirat, N. Ahmed, Multi-agent systems: A review, Int. J. Eng. Artif. Intell 5 (2024) 48–55.

[11] L. Hughes, Y. K. Dwivedi, T. Malik, M. Shawosh, M. A. Albashrawi, I. Jeon, V. Dutot, M. Appanderanda, T. Crick, R. De', et al., Ai agents and agentic systems: A multi-expert analysis, Journal of Computer Information Systems 65 (2025) 489–517.

[12] C. Feng, A. Zhang, G. Min, Y. Huang, T. Q. Quek, X. You, Towards 6g native-ai edge networks: A semantic-aware and agentic intelligence paradigm, arXiv preprint arXiv:2512.04405 (2025).

[13] M. Altoumi, N. A. S. Ahmed, Real-time operating systems (rtos) for iot applications: A comprehensive comparative study, in: 2025 IEEE International Multi-Conference on Smart Systems & Green Process (IMC-SSGP), IEEE, 2025, pp. 1–6.

[14] R. K. Baliyar Singh, J. K. Dash, K. H. K. Reddy, The role of edge-ai in edge enabled iot systems: A comprehensive performance analysis, Peer-to-Peer Networking and Applications 19 (2026) 35.

[15] M. Krichen, Timed automata-based strategy for controlling drone access to critical zones: A uppaal modeling approach, Electronics 13 (2024) 2609.

[16] D. Catta, A. Lacroix, W. Penczek, L. Petrucci, T. Sidoruk, Imitator4amas: Strategy synthesis for stctl, arXiv preprint arXiv:2602.10810 (2026).

[17] A. Natan, R. Stern, M. Kalech, W. Yeoh, T. C. Son, Diagnosing multi-agent strips plans, in: 35th International Conference on Principles of Diagnosis and Resilient Systems (DX 2024), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024, pp. 8–1.