https://orcid.org/0009-0009-9588-9490
https://orcid.org/0000-0001-8163-1119

UDC 004.94:681.5

P.D. OLEKSIIENKO[*], V.V. KAZYMYR[**]

EMBEDDED MODELS OF UAV SWARM CONTROL ALGORITHM

[*]Institute of Mathematical Machines and Systems Problems of the Ukraine National Academy of Science, Kyiv, Ukraine
[**]Chernihiv Polytechnic National University, Chernihiv, Ukraine

Анотація. *У цій статті запропоновано підхід до реалізації програмного забезпечення вбудованої системи на основі архітектури Model-View-Controller (MVC), спеціально модифікованої для мультиагентного домену. Він спрямований на вирішення проблем поганої масштабованості та ризиків єдиної точки відмови, пов'язаних із централізованими архітектурами з циклічним опитуванням у роях безпілотних літальних апаратів (БПЛА). У цій новій інтерпретації сам рій БПЛА переосмислюється як «User» — постійно оновлюваний стан керуючої е-мережі (CEN) (маркування та атрибути токенів) слугує «View», а переходи і слухачі мережі функціонують як «Controller», маніпулюючи моделлю. Модель виражається у вигляді CEN, що є розширенням мереж Петрі для цілей управління. Місця і дані CEN представляють стан агента, але в цілому CEN забезпечує погляд на алгоритм управління як на набір переходів із пов'язаними місцями для реалізації логіки подій. У той же час слухачі інтегрують зовнішні вхідні сигнали і генерують вихідні команди в повністю реактивній манері. В результаті ланцюжкове виконання, орієнтоване на події, усуває циклічне опитування, зменшує навантаження на процесор, одночасно підтримуючи синхронну, асинхронну та паралельну обробку подій у реальному часі. У статті детально описано структури даних на основі Python — місця, переходи, черги/стеки, пул потоків та реактивні слухачі — а також метод динамічної верифікації, який автоматично збирає статистику про активність переходів та їхній час, що дозволяє «на льоту» профілювати та виявляти вузькі місця у продуктивності. Комплексний приклад демонструє програму-агента з тришаровою (реактивною, плановою та кооперативною) моделлю управління, яка паралельно реагує на події датчиків, виконує відкладені дії та синхронізує результати за допомогою переходу приєднання, тим самим підтверджуючи ефективність запропонованого підходу для мультиагентних застосунків.*
Ключові слова: *безпілотний літальний апарат, мультиагентна система, алгоритми керування, Model-View-Controller, керуючі е-мережі, мережі Петрі, обробка подій, Python.*

Abstract. *This paper presents an approach to the embedded system software implementation based on a Model-View-Controller (MVC) architecture, specially modified for the multi-agent domain. It aims to solve the poor scalability and single-point-of-failure risks associated with centralized, cyclic-polling architectures in unmanned aerial vehicle (UAV) swarms. In this novel interpretation, the UAV swarm itself is re-envisioned as the User, a continuously updated state of the Control E-Network (CEN) (token markings and attributes) serves as the View, and the network's transitions and listeners function as the Controller, manipulating the model. The model itself is expressed as a CEN that is an extension of Petri nets for control purposes. The places and data of CEN represent the agent's state, but in general, CEN provides a control algorithm view as a set of transitions and their related places to implement event-driven logic. At the same time, listeners integrate external input signals and generate output commands in a fully reactive manner. The resulting chain-driven, event-oriented execution eliminates cyclic polling, reduces CPU overhead, simultaneously supporting synchronous, asynchronous, and parallel event processing in real time. The paper details the Python-based data structures — places, transitions, queues/stacks, thread pool, and reactive listeners, — together with a dynamic verification method that automatically collects statistics on transition activity and timing, enabling on-the-fly profiling and detection of performance bottlenecks. A comprehensive example demonstrates an agent program with a three-layered (reactive, planning, and cooperative) control model that reacts in parallel to sensor events, performs delayed actions, and synchronizes results through a joining transition, thereby confirming the effectiveness of the proposed approach for multi-agent applications.*

Keywords: *unmanned aerial vehicle, multi-agent system, control algorithms, Model-View-Controller, Control E-Networks, Petri nets, event handling, Python.*

DOI: 10.34121/1028-9763-2025-3-4-90-100

1. Introduction

Unmanned aerial vehicles (UAVs) have become a cornerstone of modern defense doctrine, especially for the protection of civilian and critical infrastructure facilities where permanent human presence is infeasible or too risky. Field deployments show that coordinated swarms can create a dynamic protective «umbrella», rapidly detecting and intercepting threats while providing fault-tolerant coverage over wide areas [1]. Achieving this capability, however, requires real-time cooperation among dozens (or even hundreds) of autonomous agents that must sense, decide, and act under strict timing and communication constraints. Recent surveys of swarm-formation control point out that centralized, cyclic-polling architectures suffer from poor scalability: as the swarm grows, event latency rises, processor time is wasted on needless polling, and single-point failures in the leader jeopardize the whole mission [2].

Therefore, research on multi-agent systems (MAS) emphasizes distributed decision-making and formal executable models that expose system behavior to analysis and optimization. Agent-based modeling is widely adopted as an early-stage design tool because it allows developers to explore large parameter spaces and pinpoint performance bottlenecks before committing to expensive hardware prototypes [3]. Nevertheless, most of experimental MAS frameworks remain detached from the resource-constrained world of embedded avionics, hampering technology transfer from laboratory to flight hardware. This is mainly due to the direct separation of the model representation of swarm behavior and the practical implementation of this behavior by software tools that are part of the agent control system.

The aim of the article is to describe a new approach to programming control algorithms based on embedded models, implemented through a modification of the Control E-Network (CEN) formalism [4] by a specific interpretation of the Model-View-Controller (MVC) architecture for embedded systems. This approach focuses on bridging the existing gap in UAV swarm control as a multi-agent system, which currently lacks a comprehensive framework for reactive, planning, and cooperative drone (agent) behavior levels. Specifically, it addresses the need for models that can react to real-time events via listeners, translate real-time events (such as energy alarms, communication status, or sensor triggers) into immediate, localized updates of the swarm-state model through planning listeners, and directly embed drone behavior models into their control circuitry for efficient UAV swarm management based on MAS principles.

2. Related works

The most detailed analysis of modern UAV swarm control strategies is provided in [5, 6], where centralized, decentralized, and behavior-based paradigms are considered. Based on these studies, it can be concluded that centralized control usually enables optimal planning and simpler coordination [5] but suffers from poor scalability and a single point of failure. In contrast, decentralized control is scalable and fault-tolerant, though it may result in sub-optimal or incomplete solutions. As for behavior-based control, it is robust, scalable, and adaptive through the local rules [6], but it lacks guaranteed optimality and requires careful rule design.

As for the implementation of these strategies, current publications on UAV swarm control highlight four complementary threads. The first one is related to route optimization based on graph-based search and expert-system heuristics, but it still relies on top-down scheduling and provides only limited facilities for in-mission re-planning [7]. The second one: endurance-oriented planners now embed explicit battery models and recharge station logistics into mixed-integer or evolutionary routing frameworks; experiments confirm that rotating active and standby vehicles (or coordinating with mobile charging assets) can extend persistent monitoring

by an order of magnitude, yet most of the decision logic remains offline and cannot react swiftly to sudden energy drops or platform loss [8]. The third one: secure low-latency communication remains crucial. Physical-layer security papers propose URLLC-oriented relay schemes that multiplex heterogeneous links, encrypt short packets end-to-end, and dynamically reroute around interference, complicating the command channel against jamming. However, these solutions assume classic command pipelines and do not propagate events into high-level control semantics [9] themselves. And finally, there are safety-critical embedded controllers. In this respect, the work focuses on making flight-control hardware «fault-transparent», for example, using field-programmable gate arrays. These techniques greatly improve observability of hidden faults, but they share a common limitation: control algorithms are predominantly pre-planned and static, with few provisions for fine-grained, event-driven adjustments once a mission is underway.

In general, it can be concluded that despite the advancements in UAV swarm control studies, there are some gaps in the control of UAV swarm as a multi-agent system that includes three levels of drone (agent) behavior: reactive to real-time events, which are implemented through a certain interpretation of the model using listeners, planning listeners that translate real-time events (energy alarms, communication status, and sensor triggers) into immediate, localized updates of the swarm-state model. Addressing these gaps is the core motivation of the approach proposed in the following sections.

3. Proposed approach

Our research is based on the use of CEN that is one of graphical representations of control algorithms enhanced with listener mechanisms, integrated within the Model–View–Controller (MVC) architecture.

3.1. Formal definition of the control algorithm model

Formally, in this study, a CEN, which is a type of Petri nets, is defined as a directed graph with an 8-tuple structure capturing both the Petri net semantics and the integration of external signals:

$$CEN = (P, T, E, M^0, \tau, \varphi, G, L), \quad (1)$$

where each element of the tuple is defined as follows:

– P is a set of positions (places) representing states or steps of the UAV swarm's control process. Each position $p \in P$ can hold a token indicating an active state;

– T is a set of transitions of five types describing a rule for moving tokens between positions. Transitions model the conditions and actions for progressing the control logic from one state to another;

– E is a set of directed edges (arcs) connecting positions to transitions;

– M^0 (initial marking): the initial marking function $M^0(p) \in 0,1$ (a binary marking) to indicate which positions have a token at the process start time;

– τ is a timing function $\tau: T \to R_{\geq 0}$ assigning an execution delay to each transition. This is useful for modelling-controlled delays or timeouts in UAV operations (optional);

– φ is a logical guard condition for transitions: $\varphi: T \to$ {true,false} depending on the state of external variables or probabilistic logic. This allows for modeling the transition fire condition, taking in account the data of global variables updated by a listener;

– G is a set of global variances that save the data about the state of control algorithm (data from physical sensors, messages received from other drones of the UAV swarm);

– L is a set of listeners that handle external events and signals, forming the bridge between the CEN model and its environment.

Formally, the enabling condition for a transition at *t* time moment can be written as follows:

$$\text{Enabled}(t) \equiv \bigwedge_{p \in \cdot t}[M(p) \geq 1] \wedge \varphi(t) = \text{true}. \quad (2)$$

When an enabled transition fires, it consumes tokens from its input positions and produces tokens in its output positions ($t \cdot = , p \mid (t, p) \in E$), after an optional delay $\tau(t)$. The firing can also change the token attributes that together with place marking and global variables define the control algorithm state. Listeners are special components that can either (a) inject tokens or trigger transitions immediately upon external events, or (b) update shared variables/flags that are checked by transition conditions.

To sum up, the formal model provides a rigorous yet flexible foundation for modeling UAV swarm control logic, capturing both internal decision flows and external event interactions. This event-driven capability is what embeds the model in a real UAV swarm system, allowing the CEN to react in real time to sensor inputs and to send commands to actuators.

3.2. MVC architecture interpretation

The MVC architecture, traditionally a software design pattern for user-interface applications [10], has already been successfully adapted in unconventional domains such as robotics, simulation, and IoT systems. For example, the MVC paradigm has been introduced in IoT and smart environments to handle distributed sensing and control [11, 12]. Also, in [13], the authors demonstrate how a strict separation between simulation logic and the graphical user interface (GUI) permits the rapid integration of continuous testing of swarm behaviors in a scalable manner. However, all these examples do not address the problem of distinguishing the swarm's state models from reactive logic and external interfaces.

In our study, the MVC architecture gets a new interpretation for embedded systems. Fig. 1 illustrates how the classical MVC triangle is reinterpreted for an embedded CEN controller.

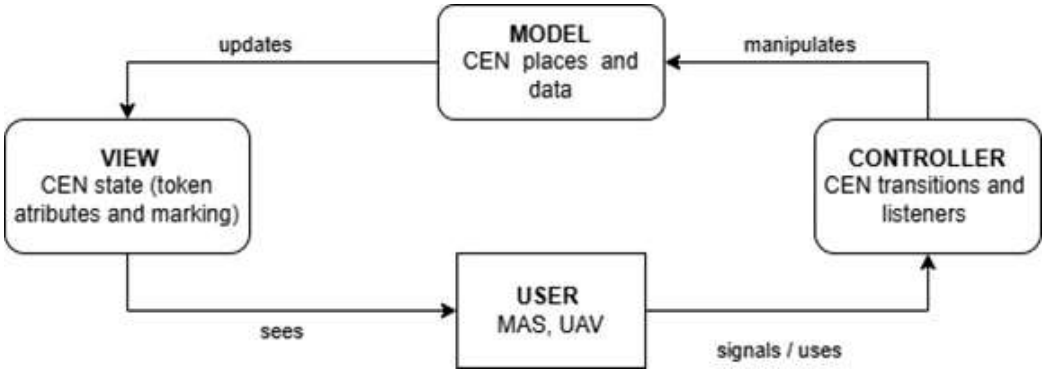

Figure 1 — MVC for the embedded CEN interpretation

The drones like an agent of MAS takes the role of the *User*, the continuously updated CEN state (current CEN marking with tokens' attributers) serves as the *View*, while the Controller (CEN transitions and listeners) manipulates the Model (CEN places and data).

This reinterpretation preserves the separation of concerns typical of the MVC architecture; the model holds the data and logic, the controller manages the execution flow, and the view (system state) provides a clear picture of the model's status to the outside world. By drawing this analogy, we emphasize a two-way integration with the surrounding environment to that of a user interface: the environment injects events via the controller and, in return, perceives the results via the system's observable state. This approach is specifically adapted for real-time operation and external event integration in embedded systems, ensuring that our CEN-based control software remains responsive to incoming signals and updates its state in a way that is transparent to (and monitorable by) the platform it runs on.

3.3. Real-time execution mechanisms

To operate effectively in real time, the CEN model incorporates specialized transition types and execution mechanisms. These mechanisms ensure that both sequential logic and concurrent or asynchronous events are handled properly. The main types of transitions and event-handling strategies are outlined below, aligned with real-time requirements:

– sequential transitions (T-type) are the simplest type of transition, corresponding to the traditional Petri net transition with a possible time delay without any additional conditions;

– synchronous join transitions (J-type) implement a synchronization barrier, firing only when all of its defined input positions have tokens. This corresponds to a logical AND-synchronization (similar to a Petri net transition with multiple input arcs);

– asynchronous fork transitions (F-type) represent a parallel branching (fork) in the control flow;

– conditional transitions are also supported within the model.

In our implementation, the X-type transition chooses one of multiple outputs based on a choice function which define the number of outputs that get the token from input place when the transition fires. This is analogous to a switch/case or an exclusive OR (XOR) split in workflow terms. On the contrary, the Y-type transition can be used to model an OR-join, where one of several inputs can pass the token to a single output place. These additional transition types extend the expressiveness of the model, though for the sake of clarity, the core discussion focuses on the fundamental T, J, and F transition types.

One of the most significant features introduced in the CEN modification is the integration of event listeners for handling external events. Here are two modes:

– reactive listeners (immediate mode). In this mode, when an external event or input signal occurs, the corresponding listener in L immediately places a token into a designated position (or directly triggers a specific transition) and invokes the processing of that position without delay. This is analogous to an interrupt service routine in embedded systems. The benefit is real-time responsiveness: the event is handled as soon as it occurs, outside of any fixed scanning cycle;

– polling or flag-based listeners (deferred mode). In some cases, rather than injecting a token immediately, a listener may simply update a global variable or flag when an external event occurs. The model's transitions periodically check these variables as part of their guard conditions. It integrates smoothly with transition which fire periodically.

Both types of listeners ensure that the model remains responsive to the outside world. The reactive approach (A) gives immediate insertion of events into the token flow, while the flag-based approach (B) uses regular transition evaluation to pick up events.

4. Implementation details

The concepts of the CEN model and its real-time execution have been implemented as a Python package. Fig. 2 provides a class diagram of the implementation, illustrating the relationships between the core classes: E_Network, Position, various Transition types, and supporting component.

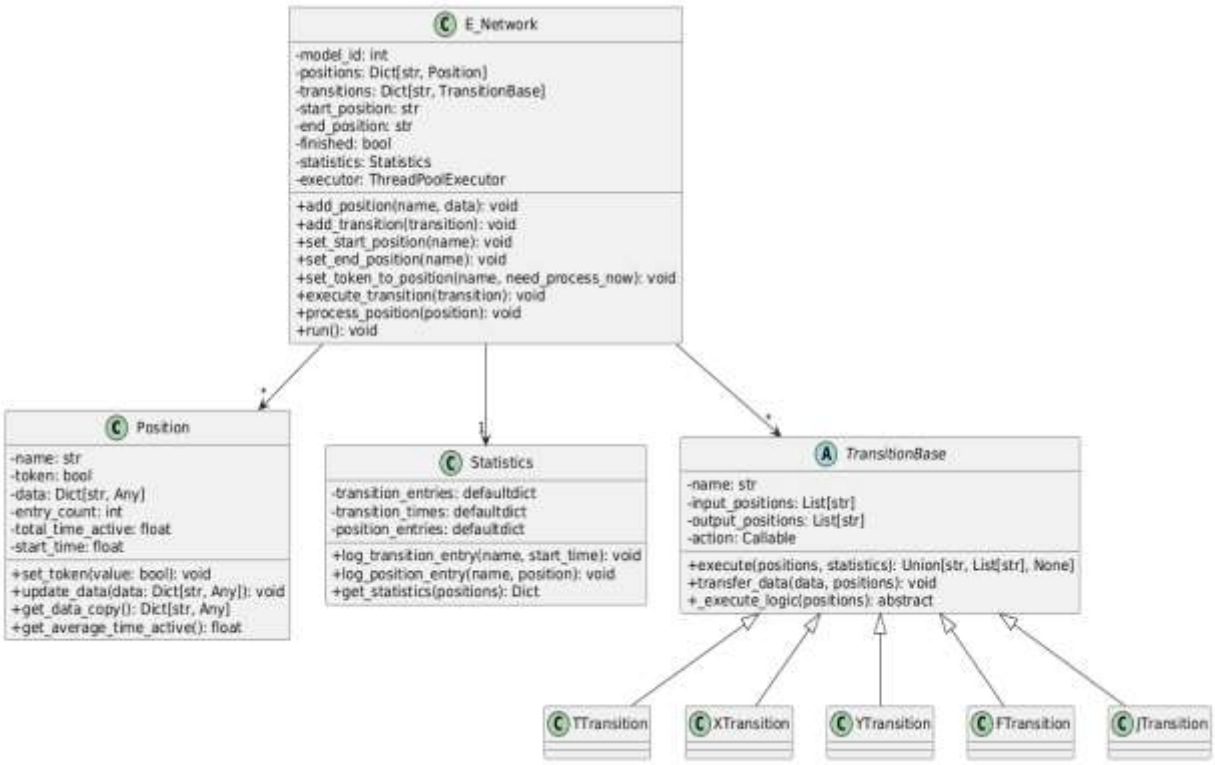

Figure 2 — Class diagram of the implementation

4.1. Core classes and data structures

The implementation defines classes corresponding to the theoretical elements of the model.

A Position has a token attribute (a boolean in our implementation, indicating whether a token is present) and may hold data attributes. This data payload travels with the token, allowing for the transfer of information between states (for example, one UAV may deposit sensor data in a token that moves to the next state for processing). The Position class also keeps simple metrics: it counts how many times a token has entered that position, thereby allowing the model to maintain stateful information as tokens move.

The Transitions Base class and specialized subclasses encapsulate the logic of moving tokens from input to output positions under certain conditions or actions. In this implementation, the abstract class TransitionBase defines the common interface and transition properties (name, the lists of input and output positions, and an optional action to perform during firing). Specific types of transitions are implemented as subclasses.

All transition classes implement an execute method (inherited from TransitionBase) which in turn calls an internal_execute_logic specific to each subclass. The execute method also hooks into a Statistics component (logging transition execution times). Notably, transitions use the transfer_data helper to copy data to output positions, ensuring that each new token carries forward the data from its origin.

The E_Network class represents an entire CEN model instance (essentially, it corresponds to the Controller in the MVC analogy, managing the Model's state). It contains dictionaries of positions and transitions within the CEN model, place markings, a thread pool executor for running transitions concurrently, and a Statistics object for collecting performance metrics. In fact, it implements the logic required to run the network.

The concurrency model of E_Network is partly event-driven (trigger transitions as soon as input tokens appear) and partly multi-threaded (parallel branches). This hybrid ensures that no busy-wait polling is needed for normal transition firing — each token insertion triggers checks immediately — and parallelism is exploited where the model design calls for it. The careful use

of locking in J-type transitions and thread pooling in F-type transitions addresses real-time concerns: threads can run simultaneously to handle different UAVs or tasks, but synchronization points are handled safely.

To analyze performance (important in real-time systems for verifying compliance with timing constraints), the implementation includes a Statistics utility. It records how many times each transition is fired, the distribution of their execution times, how often each position is activated, and how long tokens averagely stay there. While not part of the model's core logic, these are useful for debugging and tuning the system (for instance, for identifying bottlenecks if a transition is slow or often gets queued). Logging is also integrated into the engine (with model ID tagging) to trace execution for debugging.

4.2. Integration with UAV hardware/software

While the prototype is implemented as a Python project, it is mapped to actual UAV swarm control systems and hardware platforms. All experiments are conducted on a standard X-type quadrotor (450 mm, 4S Li-Po), whose avionics combine a real-time flight core with a Python-driven mission computer. Low-level attitude control is managed by a Holybro Pixhawk 6C (STM32H743, triple-redundant IMU, PX4 v1.14), while high-level CEN logic runs on a Jetson Nano B01 (quad-core A57 @ 1.43 GHz, 4 GB LPDDR4, JetPack 4.6). The Pixhawk handles the 400 Hz inner control loop and streams MAVLink telemetry over USB-C to the Jetson, which runs the event-driven swarm controller at 20–50 Hz.

The sensors feeding reactive listeners include an 8 MP IMX219 front camera for target recognition, a MEMS directional microphone for acoustic bearing cues, a downward-facing optical flow camera for hover stabilization, and integrated GNSS/barometer modules for global positioning and battery telemetry. Inter-agent coordination packets are exchanged via a Digi XBee 3 2.4 GHz mesh radio with sub-15 ms latency.

Hard real-time control loops remain on the MCU, while the Jetson injects events into the CEN in two forms: (i) immediate tokens triggered by the camera or mesh radio, and (ii) flag updates for slower variables such as battery voltage or acoustic levels. This split yields < 8 ms end to end event latency, demonstrating that the reactive, model-centric architecture fits within the resource constraints of a typical embedded UAV platform.

The listeners are implemented as callback hooks tied to UAV sensor interfaces or network message handlers. They invoke set_token_to_position method on the CEN engine when the UAV sends a telemetry update or a ground station issues a command. The output signals correspond to function calls or messages sent to the UAVs. For example, when the model generates an output signal, the code would translate it into an API call to a drone (such as changing a drone's flight mode or sending a new waypoint). In our case, an output transition (like a *T-type* transition designated as an output action) can directly invoke such code via its action callback.

4.3. Real-time implementation

The implementation choices above have been made to satisfy real-time constraints and the needs of embedded UAV control:

– Low latency event handling. In traditional PLC or polling systems, an external event might wait until the next cycle tick to be handled, which could be tens of milliseconds or more. In our case, by using an event-driven approach, we avoid the latency of a fixed cycle loop as soon as a critical event occurs (e.g., proximity alert from a UAV's sensor), the listener directly triggers the relevant part of the model. This is crucial for ensuring swarm safety and reactivity.

– Deterministic synchronization. The use of J-type transitions with locks ensures that even in a highly concurrent scenario the model's behavior is deterministic and consistent with the for-

mal specification. The cost is a brief locking period, but this is limited to the duration of the joined operation (typically negligible compared to sensor or network I/O latencies).

– Parallel execution. Real UAV swarms inherently involve parallel activities (each UAV operates independently yet cooperatively). The thread pool allows our model to reflect this parallelism. This improves throughput and aligns with multi-core processing capabilities of modern embedded platforms. We do impose a limit (e.g., max 10 threads) to avoid overwhelming the system, but this can be tuned based on hardware.

– Resource management. An additional queuing mechanism is implemented to handle situations where tasks accumulate, ensuring that no data is lost and that processing occurs in order. In real UAV systems, this may correspond to queuing incoming requests or telemetry if the system is temporarily busy. By incorporating this within the model, we allow the control logic to include buffering behavior explicitly.

– Timing and delays. The model's inclusion of τ for transitions and the timing recorded by Statistics help to analyze and design the system with real-time deadlines in mind.

5. Experimental evaluations

To evaluate the proposed architecture, a simplified CEN-based control model of a UAV agent has been developed. In Fig. 3, it is presented in a simplified form.

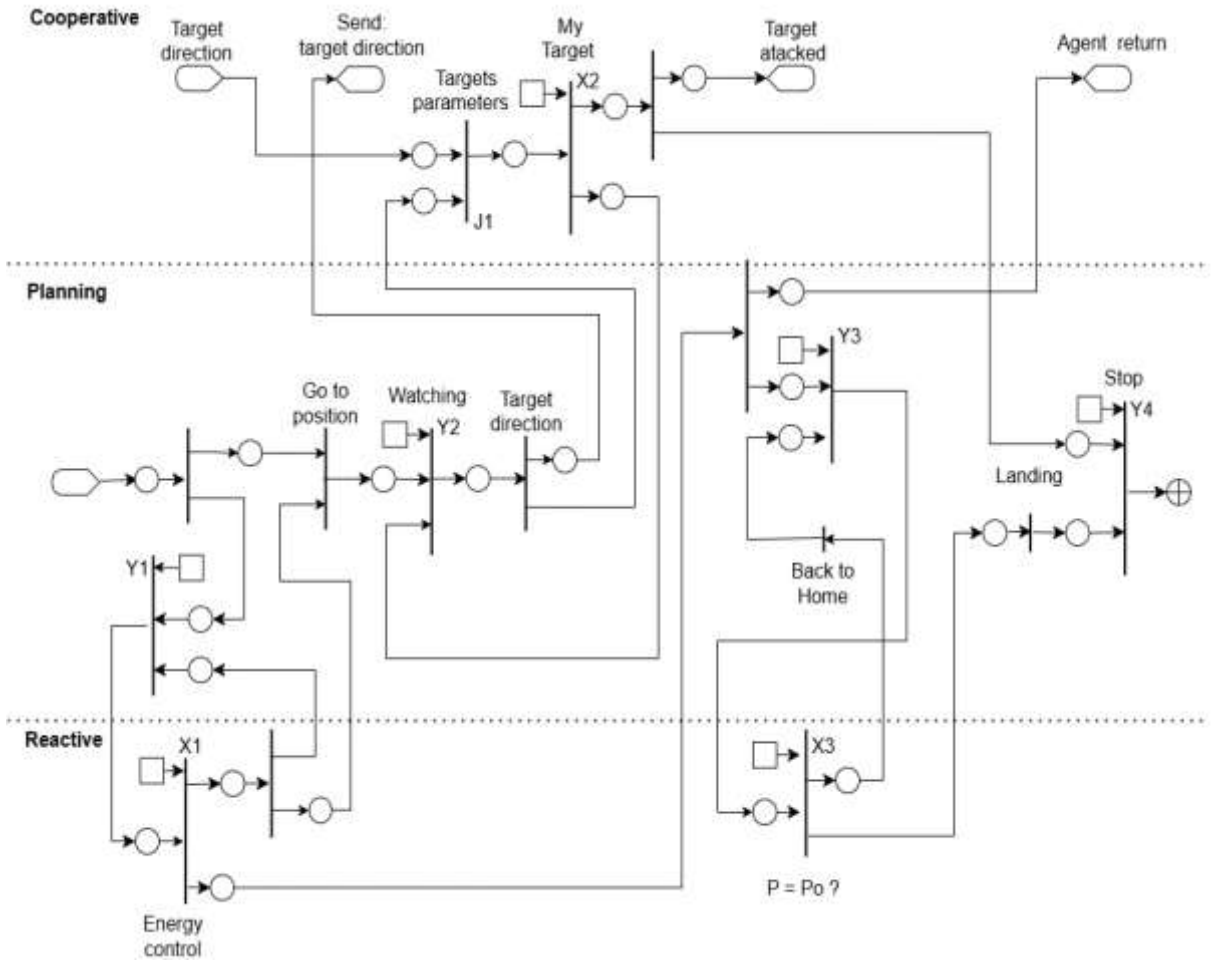

Figure 3 — Simplified UAV agent CEN model

This model implements three-layered control levels — reactive, planning, and cooperative, each showcasing key mechanisms of our approach.

Reactive control level provides the monitoring of critical global state variables and trigger safety actions when needed. Transition X1 checks the UAV's global battery status whereas transition X3 uses position data to determine landing readiness. The UAV's altitude listener updates a global position status. Both X1 and X3 embody reactive safeguards, implemented via global state checks. The planning layer encapsulates the high-level mission sequence that consists of initialization, navigation, target engagement, return, landing, and mission stop while remaining continuously sensitive to events coming from the reactive and cooperative levels. Within this chain, all major CEN transition types are exercised: sequential segments (e.g., initialization → navigation) are implemented with T-type transitions; parallel observation and navigation tasks run under F-type transitions; conditional progression hinges on *Y*-type transitions whose guards depend on global variables set by listeners; and synchronization with teammate messages is handled by the J-type transition. Thus, the planning layer is not merely a linear script: it is a hybrid flow in which reactive (battery, position) and cooperative (target confirmation) events dynamically steer the mission path.

Cooperative control level handles synchronization with external agents or commands. Here, a J1 transition is used to merge local and remote event streams, enabling multi-UAV coordination. It has two input places: one receives a local token (representing the UAV's readiness or a local observation, such as detecting a potential target), and the other receives an external token delivered via an inter-agent communication. In our scenario, when an incoming message arrives, the reactive listener immediately creates a token in the corresponding place, prompting the model to evaluate transition J1 without delay. Transition J1 will only fire once both required inputs are present, thereby synchronizing the UAV's local state with the received data. This demonstrates that the architecture can efficiently coordinate multi-agent behavior with minimal latency and correct data fusion.

Previously, this model has been tested in a semi-nature simulation environment Gazebo [14] to observe how the UAV agent reacts to events and coordinates with a notional swarm. After that, runtime statistics have been collected to assess performance. In total, the model included 26 positions and 16 transitions, 3 listeners (communication messages, battery status, and current position), and 3 global variables (battery level, current position, and drone status). It was embedded in JetsonNano on UAV platform with communication module and two cameras that provided positioning and target recognition. A fragment of the Python implementation of the CEN model is shown in Fig. 4.

```
358  def setup_model(model_id):
359      model = E_Network(model_id)
360      # setup positions
361      model.add_position("P1")
362      model.add_position("P2")
363      ...
364
365      # setup transactions
366      j1 = JTransition("J1", ["P1", "P2"], "P3")
367      x1 = XTransition("X1", "P3", ["P4", "P12"], condition=lambda: condition_x1())
368      t4 = TTransition("T4", "P8", "P5", delay=1, action=action_increment_repeat)
369      ...
370      model.add_transition(j1)
371      ...
372      model.set_start_position("P1")
373      model.set_end_position("P15")
374      return model
```

Figure 4 — Fragment of CEN model Python implementation

Experimental runs have shown that the model-centric architecture accurately captures complex mission logic while maintaining determinism and event-driven responsiveness.

6. Conclusions

In conclusion, this work is aimed at developing an event-driven, model-centric architecture for UAV swarm control that integrates formal verifiability with the practical demands of real-time operation. Our objective, outlined in Section 2.4, was to bridge the gap between rigorous modeling techniques and the realities of embedded multi-agent flight. To achieve this, we have proposed a layered Control E Network (CEN) model, enhanced by an event-driven execution engine and reactive listener mechanisms for external inputs. Together, the diverse set of CEN transition types (T, J, F, X, Y) and the listener infrastructure enable the controller to express and reliably execute sequential logic, parallel branching, synchronization barriers, conditional decision paths, and asynchronous interrupts within a unified formalism. This capability is essential for UAV swarms, where many drones must operate concurrently, occasionally synchronize at waypoints, respond to sensor events or new commands, and still satisfy strict real-time constraints.

We have successfully designed and implemented a prototype of this architecture and validated it on representative real-time embedded hardware. The high-level CEN model operated within the tight resource constraints of onboard flight computers, coordinating multiple agents without compromising performance or reliability. In practice, the system responded immediately to environmental events, orchestrated parallel tasks without conflicts, and incurred only minimal overhead. Adopting an MVC-adapted software architecture further enhanced maintainability: the Model maintains the CEN graph and state data, the View renders the graph for debugging and mission design, while the Controller (the event-driven execution engine) maps sensor inputs and inter-UAV messages to CEN transitions. This clear separation of concerns simplifies reasoning about and extending complex swarm logic, while decentralized event handling (with no global polling loop) ensures that adding new UAVs or mission components does not degrade performance.

By embracing a model-centric design, we also gained inspectability: the swarm's control logic can be statically analyzed, unit tested, and formally verified at design time, providing stronger correctness guarantees than ad hoc code alone. At runtime, the same model remains agile and adaptive, integrating a new behavior (e.g., a collision-avoidance listener) or retuning a transition condition requires only localized modifications to the CEN graph, rather than a complete rewrite of the control software. Thus, the proposed approach delivers a verifiable, scalable, and responsive swarm control system that is ready for mission requirements to evolve.

Looking ahead, there are several promising avenues to extend this research. A key priority is the development of auto-adaptation mechanisms, enabling the swarm to dynamically adjust its control model in response to environmental changes or internal performance metrics. For instance, the system could autonomously tune operational thresholds (such as battery warning levels) or reconfigure the transition network when a UAV fails or mission objectives evolve—without requiring human intervention.

Another important direction is the evaluation of the architecture in more complex multi-agent missions. This includes scaling to larger swarms and tackling more intricate tasks, such as cooperative search-and-rescue or distributed mapping, to assess the model's robustness under increasing scenario complexity. We plan to introduce heterogeneous agents and more advanced communication patterns in simulation, allowing us to rigorously test the limits of the model's scalability and adaptability.

REFERENCES

1. Dashkevych A., Vorontsova D., Rosokha S. An Approach for the Determination of Drone Positions Set with Maximal Terrain Visibility. *ICT in Education, Research and Industrial Applications (ICTERI 2021)*. 2021. Vol. 3013. P. 64–73.
2. Wang X., Zhao Z., Yi L., Ning Z., Guo L., Yu F.R., Guo S. A Survey on Security of UAV Swarm Networks: Attacks and Countermeasures. *ACM Computing Surveys*. 2024. Vol. 57 (3). P. 1–37. DOI: https://doi.org/10.1145/3703625.
3. Macal C.M., North M.J. Tutorial on Agent-Based Modelling and Simulation. *Journal of Simulation*. 2010. N 4. P. 151–162. DOI: https://doi.org/10.1057/jos.2010.3.
4. Kazymyr V., Prila O., Usik A., Sysa D. New Paradigm of Model-Oriented Control in IoT. *Information and Software Technologies.* 2019. Vol. 1078. P. 605–614. DOI: https://doi.org/10.1007/978-3-030-30275-7_46.
5. Alqudsi Y., Makaraci M. UAV Swarms: Research, Challenges, and Future Directions. *Journal of Engineering and Applied Science.* 2025. Vol. 72 (12). P. 1–20.
6. Bu Y., Yan Y., Yang Y. Advancement Challenges in UAV Swarm Formation Control: A Comprehensive Review. *Drones.* 2024. Vol. 8 (7). P. 320.
7. Shmelova T., Sterenharz A., Burlaka O. Optimization of Flows and Flexible Redistribution of Autonomous UAV Routes in Multilevel Airspace / Ermolayev V. et al. (eds.) ICTERI 2019 Workshops, CEUR-WS. 2019. Vol. 2393. P. 704–715.
8. Phalapanyakoon K., Siripongwutikorn P. Route Planning of Unmanned Aerial Vehicles under Recharging and Mission-Time Constraints. *International Journal of Mathematical Engineering and Management Sciences.* 2021. Vol. 6 (5). P. 1439–1459. DOI: https://doi.org/10.33889/IJMEMS.2021.6.5.087.
9. Sheng Z., Cui L., Nasir A.A., Wang R., Fang Y. URLLC-Oriented Secure Communication for UAV Relay-Assisted Networks. *Physical Communication*. 2023. Vol. 59. DOI: https://doi.org/10.1016/j.phycom. 2023.102063.
10. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994. 416 p.
11. Yadawad R., Kulkarni U.P. MVC Architecture and C2C File-Transfer Protocol for Smart-Home IoT Appliance Control. *SN Computer Science*. 2023. Vol. 4. P. 369. DOI: https://doi.org/10.1007/s41870-023-01349-w.
12. Khan S., Sharma N. Design and Development of an IoT-Based Web Application for an Intelligent Remote SCADA System. *International Journal of Advanced Computer Science and Applications*. 2018. Vol. 9 (3). P. 41–49.
13. Estivill-Castro V., Hexel R., Lusty S. Continuous Integration for Testing Full Robotic Behaviours in a GUI-Stripped Simulation. CEUR-WS. 2018. Vol. 2245. Paper 3. P. 1–12.
14. Gazebo Simulator. URL: https://gazebosim.org (last accessed on May 4, 2025).

Стаття надійшла до редакції 17.07.2025