Can Model Checking Assure, Distributed Autonomous Systems Agree? An Urban Air Mobility Case Study

Anubhav Gupta[1]
Department of Computer Science and Engineering
Florida Institute of Technology
Melbourne, Florida-32904

Siddhartha Bhattacharyya[2]
Department of Computer Science and Engineering
Florida Institute of Technology
Melbourne, Florida-32904

S. Vadivel[3]
Department of Computer Science
BITS Pilani Dubai Campus, Dubai
United Arab Emirates

***Abstract*—Advancement in artificial intelligence, internet of things and information technology have enabled the delegation of execution of autonomous services to autonomous systems for civil applications. It is envisioned, that with an increase in the demand for autonomous systems, the decision making associated in the execution of the autonomous services will be distributed, with some of the responsibility in decision making, shifted to the autonomous systems. Thus, it is of utmost importance that we assure the correctness of distributed protocols, that multiple autonomous systems will follow, as they interact with each other in providing the service. Towards this end, we discuss our proposed framework to model, analyze and assure the correctness of distributed protocols executed by autonomous systems to provide a service. We demonstrate our approach by formally modeling the behavior of autonomous systems that will be involved in providing services in the Urban Air Mobility framework that enables air taxis to transport passengers.**

***Keywords*—Formal methods; autonomous systems; distributed algorithms; assurance for distributed protocols; distributed protocol modeling and verification; distributed autonomous systems**

I. Introduction

Advancement in technologies associated with autonomous systems have significantly increased the use of autonomous systems in day to day activities. Additionally, communication capabilities have enabled the use of multiple autonomous systems to be used for executing autonomous missions. Unmanned Aerial Systems (UAS) are used across diverse applications, such as structural health monitoring [1], data driven path planning [2], and object classification [3]. Research by Cesare and Hollinger presented in [4] explores execution of multi-UAS missions under unreliable communication and limited battery life, for search and rescue applications that include urban search and rescue, military reconnaissance, and underground mine rescue operations.

With the increase in UAS applications several research efforts have started focusing on handling contingency scenarios such as investigating emergency landing for UAS by evaluating data available from population census and occupancy estimates from mobile phone activity [5]. Additionally, Automatic Supervisory Adaptive Control (ASAC) method enables the UAS to fly with a damaged wing [6]. As the applications start focusing on safety critical operations it becomes evident that we need to develop and deploy methods and frameworks for assuring multiple autonomous systems working together can complete the operations successfully.

One of the essential elements of an intelligent system design is in the formulation of the logic to intelligently respond to the environment. We in this research effort, focus on representing the logic as in artificial intelligence that enables automated reasoning to verify the correctness of the design. The automated reasoning involves the utilization of theories in formal methods, which is a branch of artificial intelligence that allows the design of logic as models on which we can execute queries, that prove through automated searches if the design satisfies the required properties.

This paper describes work on the verification and assurance of agreement among UASs by designing and implementing a distributed protocol with a case study for Urban Air Mobility (UAM) [7]. The implementation of the logic involved in distributed reasoning and its verification is done using Uppaal [8], a real time model checking tool. In order to accomplish the goal we present a mapping of requirements as identified from UAM model, that is implemented as queries in Uppaal [8].

The rest of this paper is organized as follows. Section II of this paper talks about the previous work that has been done in the area of formal methods and distributed protocols. Section III specifically discusses the framework for the formal modeling and analysis of the behavior of autonomous systems for UAM. It also discusses the expected architecture of distributed autonomous agents providing service in UAM. In Section IV, formal modeling paradigm is discussed in detail. This section elaborates upon the mathematical representation within the modeling paradigm and formal modeling tool Uppaal [8], which is used to build the formal model for the logic involved in distributed protocol for multiple autonomous systems to cooperatively provide a service. This section also states the behavioral model of autonomous systems in Uppaal [8] and the various verification properties used to verify the model. Experimental results are presented in Section V and finally the conclusion along with future work is inferred in Section VI.

II. Literature Survey

A. Formal Methods or Assurance Methods

There has been considerable previous work done in the area of formal methods for assurance [9], [10], [11]. In [9] the research discusses a method to perform run-time assurance for learning systems with an assurance architecture designed in Architecture Analysis and Design Language (AADL) and formal contracts for each of the components modeled and verified in Assume Guarantee (AGREE) annex. In [12] Davis discusses an approach to use architectural analysis to prove that the protocol designed for multi-agents satisfies the specified properties. This effort also emphasized the use of AADL and AGREE for formal assurance. For formal assurance of cooperative agents [10], discusses the development of a framework to represent cognitive architecture which is then translated into a formal environment Uppaal to verify that the autonomous agent along with interaction with the human achieves the objective. These studies emphasizes on the fact that how the use of formal methods can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected.

Further, Kern and Greenstreet utilize the emergence of formal methods as an alternative approach to ensuring the quality and correctness of hardware designs [13]. Also, they emphasize the two main aspects to the application of formal methods in a design process which are modeling a formal framework that specifies the desired properties of a design. The second and more important aspect is the verification process and tools that are used to reason about the relationship between a specification and a corresponding relationship. In [14], Devillers et al. present a formal modeling and verification approach for a leader election algorithm. It describes how formal methods is used to formally model the leader election algorithm as an I/O automaton, and then it describes the verification process to prove that the implementation matches the specification. The authors emphasize the importance and use of formal methods to increase confidence in the correctness of protocols, hardware and software systems [14].

The above-cited works depict the evolution of formal methods as a formal modeling technique over the years and why it is of utmost importance to model any hardware or software specification before deploying them in a real-world environment. Formal methods have been used over the years not only for modeling designs and software but also for verification and validation of these complex designs that help in identifying subtle errors during the design process which can be later eliminated during the implementation stage.

In formal methods, model checking or theorem proving are two of the prominent methods, that are used to verify satisfaction of properties within a designed system, where model checking is automated. Model checking is a method for checking whether a model of a system meets a given specification (correctness). This is mainly associated with hardware or software systems, where we want to check liveliness requirements, as well as safety requirements. To algorithmically solve this, both the model as well as its specifications are formulated in a precise mathematical language. A model, is generally a graph such as a state machine diagram, representing the behavior of a system. The state machine diagram includes, states, transitions, condition checks and actions associated with the transitions.

The main purpose of model checking is to examine whether the evolving traces of a model, generated as an execution tree satisfies the user-given property specification. Model checking for formal verification has been used as a successfully adjunct to simulation-based verification and testing.

B. Distributed Protocols and Analysis

Phillips in [15] describes the characteristics of distributed systems and their protocols. It specifically focuses on the client-server model which is used to develop a set of requirements for a distributed system along with a description of the architecture [15]. With the advancement of networking technologies such distributed systems have significantly grown in numbers so, it has become really important to apply formal methods to the field of distributed protocols [16] to prove that the distributed systems correctly operate to achieve the required functionality.

In [17], Bhattacharyya et al. discuss the formal modeling and verification of distributed systems modeled with quasi-synchrony. It mainly provides an intuitive modeling environment that allows specification of high-level architecture and synchronization logic of quasi-synchronous systems [17]. As an example a leader selection problem is discussed where the objective had been to verify a leader is elected among a set of autonomous systems. A more elaborate explanation of verification of quasi synchronous systems is described by Miller et al. in [18] where they discuss the importance of distributing critical systems to make them redundant and fault-tolerant so that they can meet the reliability requirements. The authors specifically describe the integration and enhancement of distributed systems with innovative formal verification tools such as Satisfiability Modulo Theories (SMT) based model checkers for timed automata to provide system engineers with immediate feedback on the correctness of their designs. This work mainly focuses on the design of distributed complex systems using formal method techniques, but our approach proposes the modeling and verification of the distributed logic required for successfully executing distributed operations autonomously. Also, [18] uses examples of quasi-synchronous systems to model and verify the Pilot Flying System, the Leader Selection Case, the Active-Standby System, and the Wheel Breaking System (WBS). In a presentation [19] by Thomas Ball from Microsoft at the NUS university recently, he explains the importance of formal methods as model checking tool for distributed systems. The presentation mainly focuses on automated checking of the complex design implementation using formal methods for infinite-state systems. It also shows the importance of automatically verifying distributed systems before they can be deployed so that they are provably correct. It also talks about how formal methods find bugs in system designs that cannot be found through any other known technique.

The work in [20] exhibits a methodology to develop mathematically checkable parameterized proofs of the correctness of fault-tolerant round-based distributed algorithms. It focuses on how to replace informal and incomplete pseudo code by syntax-free formal and complete definitions of a global-state transition system. In [21], Fakhfakh et al. discuss various formal verification approaches for distributed algorithms. The study shows how there has been a rapid increase in the field of distributed algorithms due to the advances in networking technologies. It also provides information for researchers and developers to understand the contributions and challenges of the existing formal verification technologies for distributed algorithms and paves the way to enhance the reliability of these distributed algorithms [21]. In [22], the work focuses on how formal methods can be used to analyze, design, and verify security protocols over open networks and distributed systems.

As we can see that there has been considerable work done in the field of distributed protocols and formal methods [16]. But none of the work specifically focuses on modeling the logic of distributed autonomous systems using formal methods for UAM. Our **contribution** has been in the design of a framework that can be applied to the formal modeling and verification of logic designed for distributed autonomous systems to successfully execute services. We have also formally mapped the requirements for autonomous services to prove that the distributed autonomous systems have a consensus among themselves. We also propose an architectural representation of how autonomous services can be designed and verified before deployment.

III. Framework

Fig. 1 shows the process flow diagram for formal verification of distributed protocol for multiple distributed autonomous systems. The process starts with stating the requirements i.e. the goal that needs to be satisfied by the distributed autonomous systems. The requirements in our research flow from the emerging services provided by autonomous systems such as, Last Mile Delivery [23], Air Taxi and Air Metro [7]. Among these services Urban Air Mobility [7] is a futuristic concept that is being researched and developed all around the world. As a result, there is an immediate need for research thoroughly investigating possible scenarios for such emerging technologies which are agnostic to the actual implementation, but helps the process of identifying the infrastructure and correctly specifying the logic involved in the successfully deploying distributed autonomous operations. Our framework describes such an approach to design and implement behavioral models for autonomous systems, that can be formally verified and is independent of the technology to implement it. The formally verified models will help to deploy trusted, secure and reliable autonomous systems in real-world environment.

These requirements led to the generation of a Formal Model designed as automata and formal properties defined or stated in temporal logic. The formal model is developed using a formal verification tool called Uppaal [8]. Uppaal has an inbuilt simulator and verifier to simulate and verify the behavior of models (in our case autonomous system). Verifier aids the process of identifying errors in the model by executing properties that generate a counterexample along with a simulation trace if the property is violated, which helps to rectify the generated errors. This process is repeated until all the errors along are identified and corrections made. The verifier also helps to list and model many path properties that help in verifying various behavioral characteristics of the stated model, which otherwise is hard to identify and verify. Once the final verification is done and all the errors have been removed, we have a model that is formally verified and the logic of which can be trusted based on the formal verification. It is envisioned that this model can then be translated into a graphical simulation environment in order to see the exact behavior of autonomous system and also to generate real-time data. The simulation environment can be any environment that supports the integration of multiple autonomous agents, multiple drones or VTOL Planes such as X-Plane [24], AirSim [25] or Robot Operating System (ROS) [26]. This translation from formal model to simulation environment is not realized in this research.

Fig. 2 below graphically shows a hypothesized distributed architecture to support services as expected in UAM. Fig. 2(a) shows how a city can be decomposed into zones supported by multi agent environments and it's various components. Each zone is further composed of several drones that are distributed in nature, managed by a server. There is constant interaction between the drone modules and the server within a zone. Each zone interacts with other zones present within the city with the help of servers present inside each zone. There can be multiple servers based on the requirements but, for simplicity we have defined only one in the model. This constant interaction between various zones makes it a multi agent distributed environment.

Fig. 2(b) elaborates upon each zone and describes the various components and their respective functionality in detail. The Distributed Autonomous Agent Environment (DAAE, Zone) consists of various components such as buildings or nodes from where service requests are generated, drones or agents that serve requests that are generated and server. Each individual drone also comprises of it's internal server and a module that can interact with the simulation environment. The building or the nodes are responsible for generating a service request which is then passed on to the server. Along with the request, the X and Y coordinates of the building are also sent to the server, which will later be passed on to each individual drone to compute the linear distance. The server is responsible for validating the request which is then broadcasted to all the drones in the zone. Once the request is received by all the drones, they go through various checks such as verification of sensor values, battery level, authenticity of the request etc. After successful validation of the checks, all the available drones calculate their linear distance from the node that generates a request. After calculation the drones exchange their distance to the requested node, with all other drones. All drones mutually agree upon the drone that is closest to the requesting Node. This mutual agreement without the interference or involvement of any kind of central observer makes the whole model distributed and de-centralised where the decisions are taken by drones present in the distributed system. The drone module is responsible for all the communication with the server and is also responsible to carry out various checks and the linear distance calculation. After a drone has been mutually selected to serve the request, the other drones go to the start location and are available to serve any other request in the network. The described architecture of a zone is modeled in Uppaal, where each of the components are a template/process. The behavior of each component is modeled in the formal verification tool Uppaal [8] and the verification of the logic is carried out using Uppaal verifier. The tool also helps to

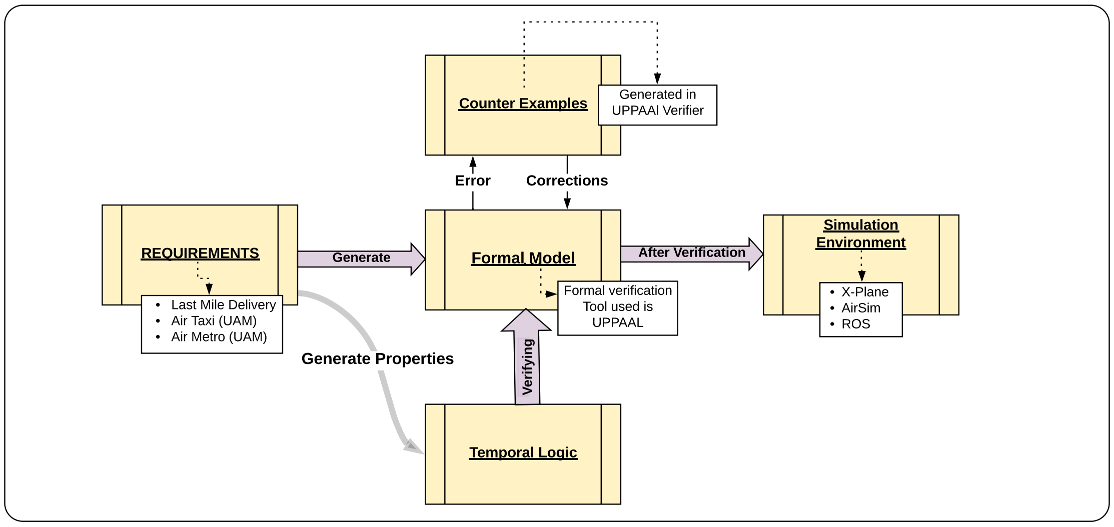

Fig. 1. Process Flow Diagram for Formal Verification of Autonomous Agents

generate simulation traces that help to identify the errors in logic which are then rectified.

This verified logic can be then translated and used to simulate this model in any real simulation environment such as AirSim [25], X-Plane [24] or ROS [26] that supports interaction of multiple autonomous agents. This real time simulation will help to generate real time data of the scenarios for a targeted service, which can be stored and later processed to improve the efficiency of the whole system. This data can also be used to develop distributed learning models for autonomous systems that will be more robust in nature and will be much more efficient. This discussed simulation is part of the future work while, this paper mainly focuses on proposing the architecture, generating formal models for the embedded components and finally, verification of the same using temporal logic to represent the requirements.

IV. Formal Modeling Paradigm

The modeling paradigm was selected after looking at several possible techniques of modeling, including Markov chains and architectural representations. We decided that the most appropriate method of representing user behavior is through the use of a Finite-State Automata (FSA) because it allows us to visualize the graphical diagram of the user's behavior easily. It enables the use of well-defined tools to perform automated analysis early in the design phase, which would empower us to reason about the logical representations of the user's behavior at the time and to evaluate alternative design options in case there were profound implications. We developed the models that are representing our knowledge base by following the principles of Finite-State automata (FSA) [27].

In order to choose the correct platform for the purpose of designing and verifying the formal model of user's behavior, several formalism such as NuSMV [28], Uppaal [8], PVS [29], and Z3 [30] were considered carefully. We chose Uppaal [8][31], due to its ability to model timing aspects that are critical for cybersecurity, as well as its ability to generate and visualize counterexamples. Uppaal represents models as timed automata, and Uppaal formalism enables compositionality supports model checking over networked timed automata using temporal logic. This modeling paradigm allows the execution of requirements as temporal logic queries to check the satisfaction of relevant safety properties exhaustively. We next describe the timed automata formalism used by Uppaal.

A. Modeling Paradigm for Timed Automata

The modeling paradigm is an extension of finite automaton with clocks, more popularly known as Timed Automata [32]. One of the tools implementing this formalism is Uppaal [8], which allows the modeling of network of Timed Automata. Clock or other relevant variable values used in guards on the transitions within the automaton. Based on the results of the guard evaluation, a transition may be enabled or disabled. Variables can be reset and implemented as invariants at a state. Modeling timed systems using a timed-automata approach is symbolic rather than explicit. It allows for the consideration of a finite subset of the infinite state space on-demand (i.e., using an equivalence relation that depends on the safety property and the timed automaton), which is referred to as the region automaton. There also exists a variety of tools to input and analyze timed automata and extensions, including the model checker Uppaal and Kronos.

- **Timed Automaton (TA)**
  A timed automaton is a tuple (L, l_0, C, A, E, I), where: L is a set of locations; $l_0 \in L$ is the initial location; C is the set of clocks; A is a set of actions, co-actions, and unobservable internal actions; $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset; and $I : L \to B(C)$ assigns invariants to locations.
  We define a clock valuation as a function $u : C \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. Let $\mathbb{R}^C$ be the set of all clock valuations. Let $u_0(x) =$

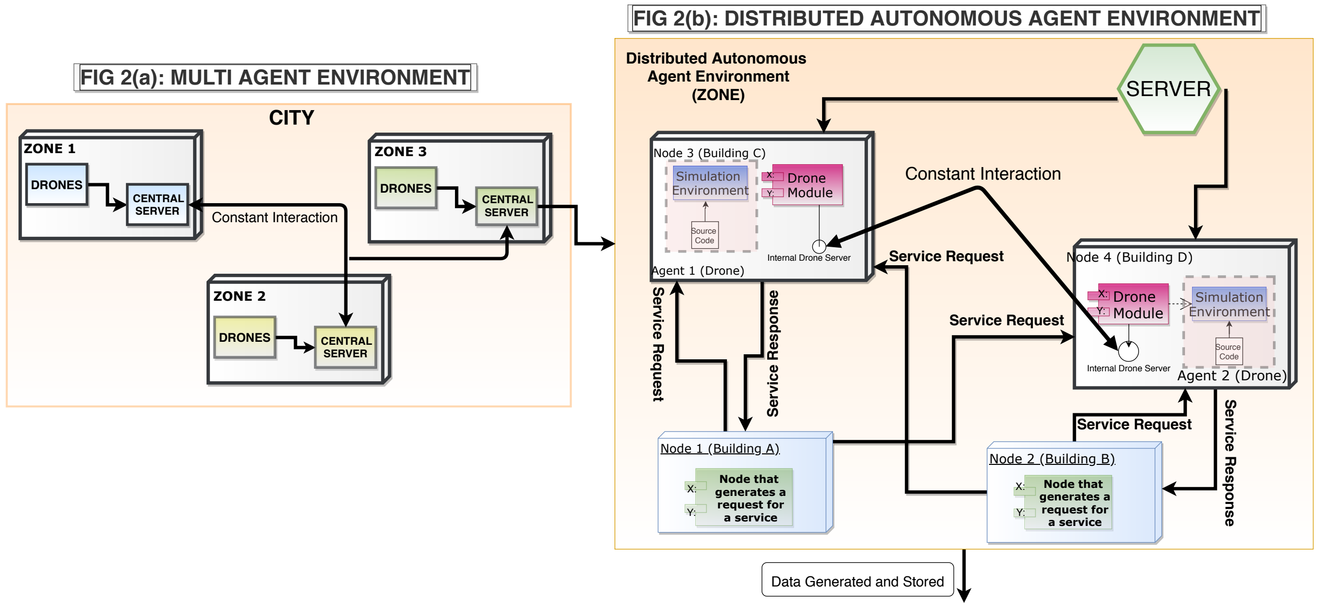

Fig. 2. Modeling Architecture of Distributed Autonomous Agents

0 for all $x \in C$. If we consider guards and invariants as the sets of clock valuations (with a slight relaxation of formalism), we can say $u \in I(l)$ means u satisfies $I(l)$.

- **Timed Automaton Semantics**
Let (L, l_0, C, A, E, I) be a timed automaton TA. The semantics of the TA is defined as a labelled transition system $\langle S, s_0, \rightarrow \rangle$, where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times \{\mathbb{R}_{\geq 0} \cup A\} \times S$ is the transition relation such that:
    1) $(l,\ u) \xrightarrow{d} (l,\ u{+}d)$ if $\forall\ d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$
    2) $(l,\ u) \xrightarrow{a} (l',\ u')$ if $\exists\ e = (l,\ a,\ g,\ r,\ l') \in E$ such that $u \in g$, $u = [r \mapsto 0]\ u$ and $u' \in I(l)$

  where for $d \in \mathbb{R}_{\geq 0}$, $u + d$ maps each clock x in C to the value $u(s) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C \setminus r$.
  Note that a guard g of a TA is a simple condition on the clocks that enable the transition (or, edge e) from one location to another; the enabled transition is not taken unless the corresponding action a occurs. Similarly, the set of reset clocks r for the edge e specifies the clocks whose values are set to zero when the transition on edge executes. Thus, a timed automaton is a finite directed graph annotated with resets of and conditions over, non-negative real-valued clocks. Timed automata can then be composed into a network of timed automata over a common set of clocks and actions, consisting of n timed automata $TA_i = (L_i, l_{i0}, C, A, E_i, I_i)$, $1 \leq i \leq n$. This enables us to check reachability, safety, and liveness properties, which are expressed in temporal logic expressions, over this network of timed automata. An execution of the TA, denoted by $exec(TA)$ is the sequence of consecutive transitions, while the set of execution traces of the TA is denoted by $traces(TA)$.

B. Uppaal

Uppaal [8], an acronym based on a combination of UPPsala and AALborg universities, is an integrated tool environment for modeling, simulation and verification of real-time systems as networks of timed automata, extended with data types (bounded integers, arrays, etc.). It is used to model the logic of real time systems. For our work we have used to model the behavioral of the components for an UAM architecture [7]. We further use Uppaal [8] to verify our modeled logic in timed automata and then propose meaningful insights and results. The tool consists of three main features. First is the editor window where we model the behavioral logic for each of the modules described in detail below. Next is the simulator window, where we run a step by step simulation of the modeled logic. This helps to understand the real time functioning of the behavior of each module and further helps to refine our logic. The last and the most important part is the verifier. The verifier, utilizes a model-checker to perform an exhaustive exploration of the dynamic behavior of the system for proving safety and bounded liveliness properties. Properties are written in temporal logic to verify the logic developed. The verifier helps verify important aspects of the model and gives a deep understanding of the functioning of the model in real time scenario. It also helps to find flaws in the model that can rectified in the editor window. As a result, we are able to model a logic that has been verified and can be deployed in real time scenarios. The implemented model for UAM services as case study is described in detail in the next subsection along with detailed description and functionality of each module.

C. Model in Uppaal

In this subsection we elaborate on our approach to address distributed modeling and analysis for DAAE in UAM. For now, we consider that, there are three drones represented by a Drone module (Section 3) serving in a zone which is inside a city that has many such similar zones along with a Server module (Section 2), Sensor module (Section 4) and an Input module (Section 1). All these specific modules have specific roles and functions in the UAM architecture whose behavioral logic has been modeled in Uppaal. Three instances have been created for the drones in the system declaration since, all the three drones are assumed to have similar behavior for now. Algorithm 1, maps the step by step behavioral logic for drone module in Uppaal. The request is generated by a random function by the input module. The request is sent as a synchronisation event by the input module to the server module. Along with the synchronisation action, coordinates of the requesting node or building are also sent to the server module. The server module then processes the request and broadcasts it to all the individual drones available to server a request. The drone before receiving a request, checks for all sensor values using the help of sensor module (Section 4). After all checks have been performed, they then process the request received and mutually elect a drone that will serve the request without the interference of the Server module. This process of mutual selection makes the whole UAM architecture distributed and decentralised in nature. The design and functionality of each individual module along with their role in the whole behavioral model is described below:

1) **Input Module**
The instances of request are generated by the Input module. It is the one responsible for generating a random request which then goes as a synchronisation event to the server where it is processed and broadcasted to all the drones in the environment. As seen in Figure 3, the Input module makes a random transition from the $Start$ state to $Generate\_Request$ state. This transition generates a random integer less than 100 and based on the integer generated it further makes a transition to one of the buildings in the environment i.e Building A, B or C. Through this process, we have tried to depict a random request generator which sends a request for service synchronisation command to the Server module. Along with the $request\_from\_building$ synchronisation, the Input module also sends the coordinates of the building from where the request is generated. These coordinates are further sent to each individual drone by the Server module. These coordinates are used in distance calculation of each drone from the building. After generating a random request, the Input module makes a transition back to the $start$ state to generate a new request for service. This process continues and random requests are generated which are then served by the drone.

2) **Server Module**
The Server module describes the behavioral logic for Server which is responsible for routing the request generated by the Input module. As seen in Fig. 4, the Server module transitions from $Start$ state to $Wait\_For\_Request$ state when it receives a request for service from the Input module. It immediately sends a synchronisation request to the Drone module. This request goes as a synchronisation event and is received by each drone

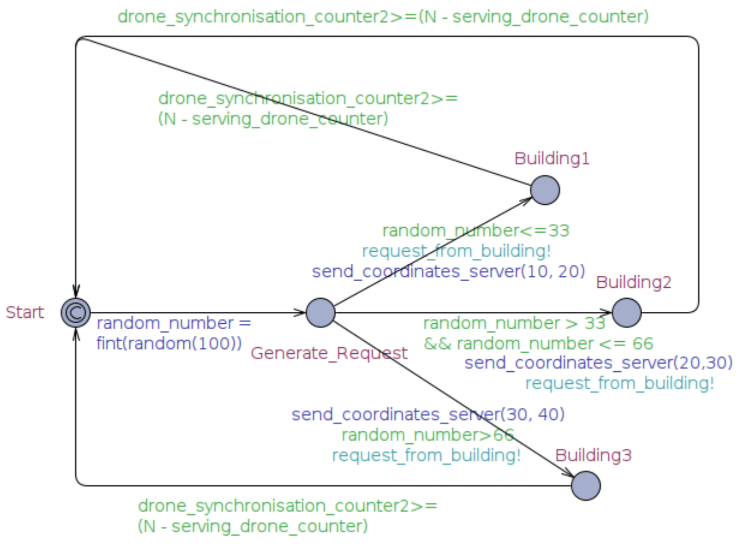
drone_synchronisation_counter2>=(N - serving_drone_counter)
drone_synchronisation_counter2>=
(N - serving_drone_counter)
Building1
random_number<=33
request_from_building!
send_coordinates_server(10, 20)
Building2
Start
random_number =
fint(random(100))
Generate_Request
random_number > 33
&& random_number <= 66
send_coordinates_server(20,30)
request_from_building!
send_coordinates_server(30, 40)
random_number>66
request_from_building!
Building3
drone_synchronisation_counter2>=
(N - serving_drone_counter)

Fig. 3. Behavioral Model in UPPAAL for Input Module

that is available to serve a request. The request is broadcasted to all the available drones along with the location coordinates of the building from where the request is generated. Only after the drones have received an authentic request from the Server module, they proceed further to calculate linear distance in-order to mutually elect the nearest drone to serve the request. At this point, the server module waits until it receives a synchronisation serve! event from the drone that is chosen to serve the request. The drone which is chosen to serve the request sends a synchronisation action to the server module indicating that, the request generated is being served by one of the drones present in the environment. Only after receiving the serve! synchronisation, it transitions from $Wait\_For\_Communication$ state to $Repeat\_Request$ state. During this transition, the time taken from the moment a request is sent and until it is accepted by the nearest drone is stored in a variable called $time\_server$. After this state, the server modules makes a transition back to the $Start$ state to process and send any other request if available to the Drone module. This process continues repeatedly until there are no more service requests.

3) **Drone Module**
The Drone Module defines the behavioral logic of drone architecture in the UAM model. There are many instances of the drone module that can be defined in the system declaration of UPPAAL editor environment. Fig. 5 below graphically shows the drone module in the UPPAAL editor window. Initially every drone is in the $Start$ state. Once the drones are ready, they go to $Ready$ state. While making the transition from $Start$ to $Ready$,certain counters are initialized. The variable i in the module represents the identification number of the drone that is being referenced. Whilst in the $Ready$ state,

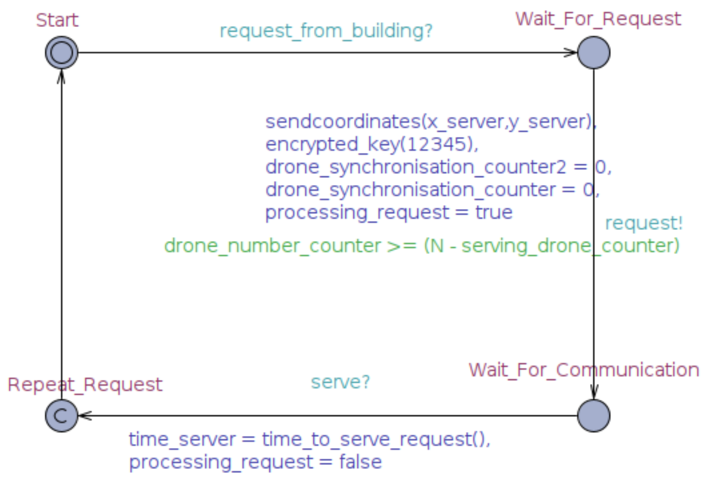
Start
request_from_building?
Wait_For_Request
sendcoordinates(x_server,y_server),
encrypted_key(12345),
drone_synchronisation_counter2 = 0,
drone_synchronisation_counter = 0,
processing_request = true
request!
drone_number_counter >= (N - serving_drone_counter)
Repeat_Request
serve?
Wait_For_Communication
time_server = time_to_serve_request(),
processing_request = false

Fig. 4. Behavioral Model in UPPAAL for Server Module

each drone waits for every other available drone and also waits for the Server to generate a request. Once the request has been generated, it is decrypted by each drone to check if the request is coming form authentic server or not and if proved, the drones make transition from $Ready$ to $Sensor\_Check$ state. The $Sensor\_Check$ state is where each individual drones will check if the various parameters are working normally and if the drone is in good condition to fly and serve a request. If the sensors are normal and the condition of the drone is healthy to fly it will make a transition to $Availability\_Check$ state. If any of the instrument or parameter is not working properly the drone will exit the loop by making a transition to $Report\_Error$ state. In the next state, each drone performs a linear distance calculation to calculate it's distance from the Node where request is generated. After calculating the distance, all drones will update their respective distances to a global list along with their specific identification number. After updating the distance, all drones mutually agree upon the drone which is closest to the requesting Node and select the drone nearest to the Node, to serve the request. Here the drones also perform a check for principle of quasi-synchrony [15] [17] i.e no drone should serve more than twice while others have not served once. This way all the drones get a chance to serve the requests if they are not the closest to the requesting node. This process where the drones mutually agree upon the one to serve the request without the interference of the server module or any other central module, makes the architecture distributed [17] and de-centralized in nature. After completing all these steps and mutually selecting the drone to serve a request, all drones wait at the state $Make\_Decision$ where the decision is made by each individual drone according to the mutual agreement. The drone chosen to serve the request makes a transition to the state $Serving\_Request$ while others make a transition to $Ready\_To\_Serve$. All the other drones are available again to serve any new request generated by the server module. The drone serving the request, updates certain variables and makes itself unavailable for any new request. It also sends a serve synchronisation command to the server to indicate that the request generated by the server is being served. After serving the request, the drone calculates the total time taken to serve the request and makes itself available again to serve any new request. This process continues for each and every request generated at the server side. Every time a new request is generated all the available drones perform sensor checks, authenticity check and shortest distance calculation. Always, the drone closest to the requesting node is chosen to serve the request keeping in mind the principle of quasi synchrony is satisfied [15] [17].

4) **Sensor Module**
The Sensor Module as shown in Figure 6 consists of the $Start$ and the Get_Sensor states respectively. Every time a drone transitions from $Start$ state to $Ready$ state a synchronisation event $start$ is sent to the Sensor module. The sensor module then synchronises and makes a transition from the $Start$ state to Get_Sensor state. While transitioning, it gets the latest real-time sensor values such as altitude, fuel, temperature etc. of the respective drone that sends the synchronisation and returns it to the Drone module. These values are later used by the Drone to check if it is healthy to serve the request and if all parameters are above the safe threshold limit.

D. Formal Verification Requirements

Uppaal allows for verifying requirements modeled as properties, that are useful for ensuring correctness, detecting inconsistencies, as well as flaws in the design according to the proposed modeling and analysis framework for UAM model. For example, Uppaal is capable of detecting whether there is a deadlock in the model, the results of which can further be used to find out logical flaws in the behavior of the developed model. In this subsection, we present various requirements modeled as properties, that one may want to verify with respect to UAM model, and also present meaningful insights into them along with brief description of each. The verification helps to check for any inconsistencies or flaws that may be present in the behavioral logic. After identifying the flaws, they are corrected and a consistent and a robust UAM model is presented through this work.

Requirement 1 : The existence of deadlock within the system should be verified

$$A[\,]\ Not\ deadlock$$

This requirement is stated to check if there exists any deadlock in the system. The requirement is modeled as a property in UPPAAL verifier, that proves and presents a simulation trace of the state where a deadlock exists. After examining the particular case and scenario we find out that

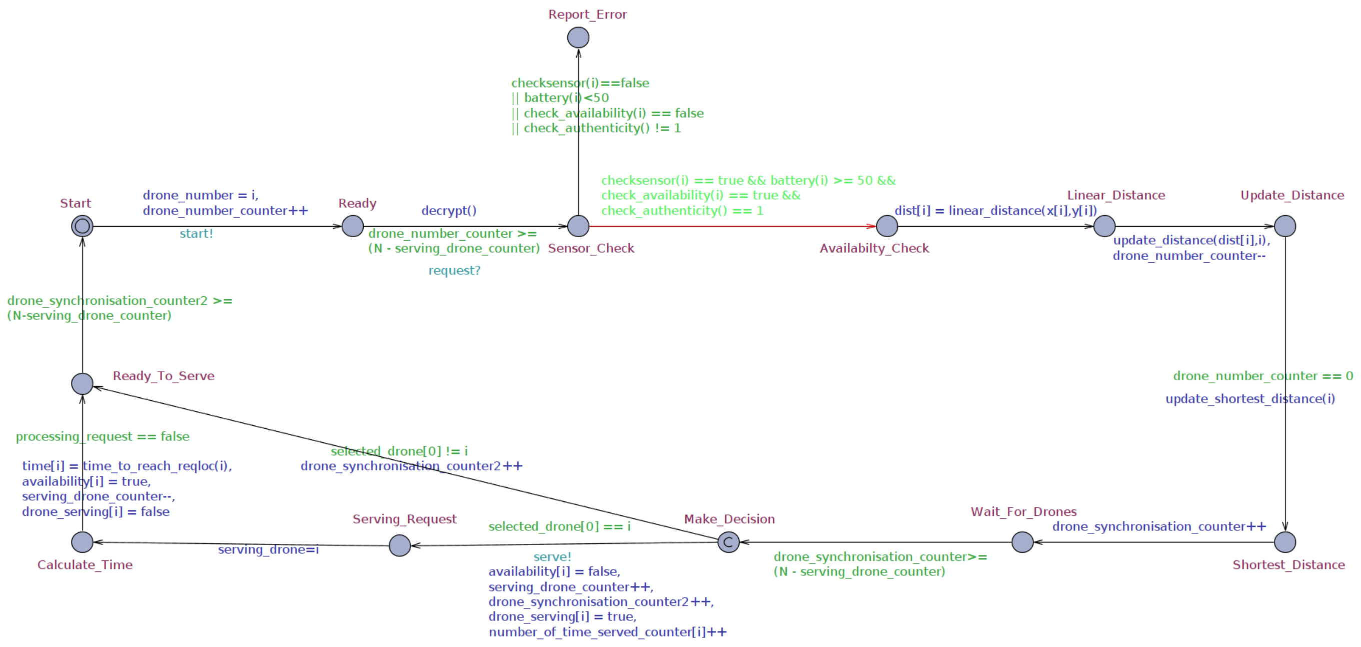
Report_Error
checksensor(i)==false
|| battery(i)<50
|| check_availability(i) == false
|| check_authenticity() != 1
Start
drone_number = i,
drone_number_counter++
start!
Ready
decrypt()
drone_number_counter >=
(N - serving_drone_counter)
request?
Sensor_Check
checksensor(i) == true && battery(i) >= 50 &&
check_availability(i) == true &&
check_authenticity() == 1
Availabilty_Check
dist[i] = linear_distance(x[i],y[i])
Linear_Distance
update_distance(dist[i],i),
drone_number_counter--
Update_Distance
drone_number_counter == 0
update_shortest_distance(i)
Shortest_Distance
drone_synchronisation_counter++
Wait_For_Drones
drone_synchronisation_counter>=
(N - serving_drone_counter)
Make_Decision
selected_drone[0] == i
serve!
availability[i] = false,
serving_drone_counter++,
drone_synchronisation_counter2++,
drone_serving[i] = true,
number_of_time_served_counter[i]++
Serving_Request
serving_drone=i
Calculate_Time
time[i] = time_to_reach_reqloc(i),
availability[i] = true,
serving_drone_counter--,
drone_serving[i] = false
processing_request == false
Ready_To_Serve
selected_drone[0] != i
drone_synchronisation_counter2++
drone_synchronisation_counter2 >=
(N-serving_drone_counter)

Fig. 5. Behavioral Model in UPPAAL for Drone Module

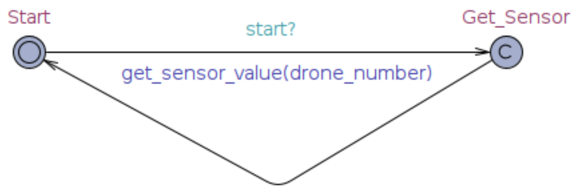

Fig. 6. Behavioral Model in UPPAAL for Sensor Module

the deadlock does not indicate any flaw or inconsistency in the logic. Similarly, for every specific system, this scenario will have to be examined to figure out if the deadlock is necessary or it is a reflection of faults and inconsistencies in the system. For example, if the service provider does not want to provide any service during night time, then a deadlock at night will indicate correct and consistent logic. Therefore, specific to the model, deadlocks have to be examined to see if it's needed or the logic has to be changed in order to remove them. **Requirement 2 : All the drones present in the system shall be able to provide service at the same time**

$E <> Drone1.Calculate\_Time$ && Drone2.Calculate_Time && Drone3.Calculate_Time

The requirement checks if all the drones present in the model can be busy at the same time to provide service to different requests i.e, all of them are servicing three individual requests simultaneously. For our model, this requirement proves indicating, that there exists a path where eventually, all three drones can be serving at the same time which shows that each drone functions independent of the other drones, but the decision are taken with mutual agreement. The following requirement also helps to justify the distributed nature of each autonomous agent in the environment that functions independent of the other agents.

Requirement 3 : An available drone shall always provide service to a generated service request

$E <> Server1.Wait\_For\_Communication$
&& (Drone1.Serving_Request or Drone2.Serving_Request or Drone3.Serving_Request)

The requirement checks if there exists a path where when a request is sent by server, it is always served by the available drones. This helps us to know that there are no neglected requests and that whenever a request is sent, it is always served and not ignored. This requirement helps to verify if any service requested is left unattended in the environment.

Requirement 4 : All the drones shall mutually agree upon who should be the service provider

$A <> (selected\_drone[0] == selected\_drone[1])$ && $(selected\_drone[2] == selected\_drone[0])$ && $(selected\_drone[0] == selected\_drone[2])$

This requirement modeled as a property proves indicating for all paths eventually, all the drones mutually agree on the drone that will provide the service. The global list $selected\_drone$ contains same elements which tells us that the service provider has been chosen with mutual consent without the interference of any external or central server. This specific requirement helps to verify that even though the proposed framework is distributed in nature but the drones take certain decisions with mutual consent without the interference of any central server or agent.

Requirement 5 : All the drones shall mutual agree upon, who is the closest to the location of reqeust

$A <> (shortest\_distance[0] == \ shortest\_distance[1]$
&& shortest_distance[1] == shortest_distance[2]
&& shortest_distance[0]== shortest_distance[2])

The above stated requirement is modeled as a property in the Uppaal verifier. We get a simulation trace indicating that for all paths eventually, all drones are able to decide upon the drone that is closest to the requesting node or building. All drones individually update the global list $shortest\_distance$ indicating the distance of the drone that is closest to the requesting building and is available to serve the request.

Requirement 6 : No drone shall provide services more than two times while other drones are idle

$E <> (number\_of\_time\_served\_counter[0] > 2\ OR$
number_of_time_served_counter[1]>2 OR
number_of_time_served_counter[2]>2)

Using the above stated requirement, we implemented the principle of quasi synchrony. As a result, we try to check, if there exists a path where either of the drones have served more than two times, while others have not served even once. This requirement modeled as a property keeps processing and does not indicate a yes or no since it is an unbounded system. This implies it is a liveliness property and hence it does not find a state where the following condition holds true. We run the execution for almost 11,700 states till we get server connection lost error and until that time it does not hold true. In a way this implies that there isn't any path where this property holds true (i.e principle of quasi synchrony holds till the time we don't lose connection with the server) but we cannot say that for sure.

Requirement 7 : Authenticity of the incoming request from the server shall be verified by all drones

$A <> (Drone1.key == Server1.local\_key)$ &&
(Drone2.key==Server1.local_key) &&
(Drone3.key==Server1.local_key)

Yes, for all paths eventually, all drones check if the request is coming from the authorised server or not. The server while sending a synchronisation service request, sends an encrypted key along with it. Each drone individually decrypts the key and compares it with the existing shared key. Only if the request is authentic, it will be served by the available drones otherwise it will be ignored.

Requirement 8 : An unavailable drone shall not process another request until it becomes available again.

$E <> ((Drone1.availability[0] == false$ &&
Drone1.Make_Decision) OR (Drone2.availability[1]==false &&
Drone2.Make_Decision) OR (Drone3.availability[2]==false
&& Drone3.Make_Decision))

The above requirement tries to find if there exists a path eventually where a drone is already in the process of serving a request, goes to serve a request again i.e an unavailable drones serves a new incoming request. The requirement modeled as property keeps on running for approximately 12,065 states until connection to the server is lost. This indicates that till 12,065 states, there is no state where the above condition holds true. To prove or disprove the property we need to consider a bounded automata. Therefore, for now we cannot say for sure if the above property is true since it keeps on running in search for a simulation trace without generating a counterexample.

Property 9 : A drone with poor health shall not be chosen to serve an incoming request
The above requirement tries to find if there exists a path eventually where the battery of a drone is less than 50% and it is chosen to serve the request.

$E <> ((bat[0] < 50$ && $Drone1.Make\_Decision)$
OR (bat[1]<50 && Drone2.Make_Decision) OR (bat[2]<50 &&
Drone3.Make_Decision))

We assume a threshold of 50% and do not want any drone with a battery value of less than the threshold to serve a request. This threshold value can be changed if needed. The property keeps on running to find a path until the server connection is lost. We need to make the model bounded in order to prove the following liveliness property. We can say that for at-least 12,458 states there doesn't exist any such path, but cannot guarantee for the whole model since it keeps on running without generating a counterexample.

Property 10 : A drone with a malfunctioned sensor should not be chosen to serve a request

$E <> ((technical\_sensor[0] == false$ &&
Drone1.Make_Decision) OR (technical_sensor[1]
==false && Drone2.Make_Decision) OR
(technical_sensor[2]==false && Drone3.Make_Decision))

Through this requirement we try to investigate if there exists a path eventually, where a drone whose sensor has been malfunctioned or is not working properly, is chosen to serve the request. The requirement stated as property keeps on running until server connection is lost indicating it is unable to find such path for the number of states it runs. We need to make the model bounded to accurately indicate if it holds true or not. As of now, we cannot say for sure that the property holds true for the whole model since it keeps on running infinitely without generating a counterexample.

V. RESULTS

This section evaluates the results of the various properties that are mentioned above. In general, we are able to verify that the distributed drones in the autonomous environment mutually agree and take decisions without the interference of any central server or module. Table 1 below evaluates the experimental results for each property. The first two columns of the Table 1 show the time taken (in seconds) by each property to execute and the total run-time memory (in megabytes) consumed. The next column indicates if the property proves or not. As we can see, some of the properties prove and some do not. Few properties keep running in loop until we get a server connection error. For these properties, we can't say for sure if they hold true or not since it is an unbounded system. The next column describes the number of states each property iterates through. Some properties that prove, iterate through all reachable states. If the verifier finds a counterexample for a particular property, it gives a simulation trace and indicates that the property does not hold true. These properties also iterate through all possible reachable states to look for a counterexample. The properties that keep on running without proving, iterate through many states as listed in the table until we get server connection error. The next column indicates if a simulation trace is generated while verifying a property. It is noteworthy that in Uppaal a simulation trace is generated when a property does not hold true i.e. the model checker finds a counterexample. Some properties which keep on running, do not generate any simulation trace and we get server connection error. An automated simulation trace is also generated when the "There exists (E<>)" property proves. Through this verification process, we are able to verify the formal behavioral logic and develop a model which is consistent and free of errors.

During the verification process, a counterexample was generated along with a simulation trace which showed that the above stated property was not satisfied. As seen in Fig. 7 the two available drones ($Drone1$ and $Drone2$) are not initialized yet since they are at $S0\_Start$ state. The Server ($Server1$) receives $request\_from\_building!$ synchronisation event from the Input Module ($Input\_Request$) indicating that a request for service has been generated, which needs to be sent to all the available drones.

The server module then broadcasts the request to all the available drones available by sending a $request!$ synchronisation. As observed, the broadcasted $request!$ synchronisation is not received by the drones since they are still at $S0\_Start$ state and hence, the service request goes unattended. The property verification process helped to identify the flaw in the logic design that the request generated by the server would sometimes go unattended and will never be served. This identification of flaw led to redesign of the logic and later we were able to rectify the logic and were able to verify the above stated property.

This specific counterexample shows how formal verification and formal model checking helps in identifying and removing flaws and inconsistencies in proposed logic during design time of complex automated systems. Fig. 7 depicts one among several counterexamples which we encountered during model checking process. The property stated during model checking intends to verify if all available drones in the system are ready to receive a request when a server is sending it.

TABLE 1. EXPERIMENTAL RESULTS OF PROPERTY VERIFICATION AND MODEL CHECKING.

Property	Time (sec)	Virtual Memory (mb)	Does it Prove?	No. of states iterated	Simulation Trace
1	0.09	10.456	No	All reachable states	Yes
2	0.017	10.452	Yes	All reachable states	No
3	0.001	10.712	Yes	All reachable states	No
4	0.001	10.712	Yes	All reachable states	No
5	0.001	10.632	Yes	All reachable states	No
6	60	1202	No, keeps running	11700	No, server connection error
7	64	1161	No, keeps running	14543	No, server connection error
8	68	1162	No, keeps running	12065	No, server connection error
9	68	1167	No, keeps running	12458	No, server connection error
10	63	1177	No, keeps running	13230	No, server connection error

A. Discussion

Our method provides artifacts such as models, logic, verification results as evidence that indicate satisfaction of requirements for a system, that is being designed. The evidence is obtained by, performing model checking at design time. This design time analysis also helps in clearly identifying the requirements that need to be implemented to achieve the functionality, by the creation of the model or the system, while abiding by the constraints provided by regulatory bodies.

Our approach, is agnostic to the technology that is finally used for implementation, thereby focused on identifying and representing the requirement for the problem to be solved, without getting into the complexities of implementation. It helps in generating and evaluating all the possible test cases that need to be tested for actual drones/agents to successfully execute the desired mission.

Once this is verified during design it can be implemented in any simulation environment and finally deployed on drones or autonomous agents. This method thus ensures through formal verification, the correctness of the logic designed. In our case, it verifies the logic developed in providing services by autonomous agents in a distributed environment.

B. Limitations

There are a few limitations with the study that has been conducted. Firstly, the Uppaal model built to represent the distributed protocol environment, needs to be evaluated for Scalability. Currently, it only represents three instances of the Drone Module. We need to evaluate and verify the behavioral logic for at-least more than ten drone modules since the Urban Air Mobility environment will consist of numerous drones. Secondly, the logic developed in the formal verification environment has not been mapped to an actual simulation environment. As part of next step of this research, we plan to map it to a simulation environment such as ROS and evaluate the performance of the distributed protocol architecture with multiple agents carrying out a specialized task within the environment. Lastly, some of the properties do not prove and keep on running, trying to find out a counterexample. This is because, currently, the model is unbounded. We need to bound the model and also come up with a more abstract representation in-order to figure out how the properties can be proved.

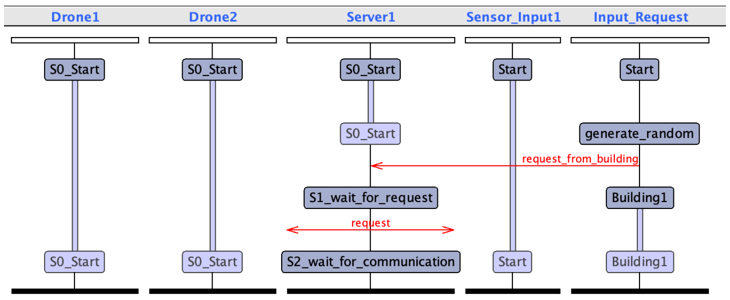

Fig. 7. Counterexample Generated during Property Verification

VI. Conclusion

Through this study, we proposed a formally verifiable framework to represent logically the behavioral that should be satisfied by the components in the infrastructure, required for distributed autonomous agents to successfully provide services. Through the property verification, we were able to prove that the distributed autonomous agents mutually agree without the interference of any central server or module. The autonomous agents are able to take decisions independently and also in synchronisation when needed. The representation is formally verified and is free of any flaws and inconsistencies.

We plan to further extend this work by incorporating scalability and heterogeneity analysis to the present study and see how heterogeneous autonomous systems behave in a distributed environment. We also plan to model and formally verify similar distributed models with other model checking tools such as, nuXmv, PRISM and see how the results vary and further try to generalise our model. Finally, we envision to map the logic from the formal model to a simulation environment.

References

[1] S. Li, Y. Wan, S. Fu, M. Liu, and H. F. Wu, “Design and implementation of a remote uav-based mobile health monitoring system,” in *Nondestructive Characterization and Monitoring of Advanced Materials, Aerospace, and Civil Infrastructure 2017*, vol. 10169. International Society for Optics and Photonics, 2017, p. 101690A.

[2] C. He, Y. Wan, and J. Xie, “Spatiotemporal scenario data-driven decision for the path planning of multiple uass,” in *Proceedings of the Fourth Workshop on International Science of Smart City Operations and Platforms Engineering*, 2019, pp. 7–12.

[3] Y. Qi, D. Wang, J. Xie, K. Lu, Y. Wan, and S. Fu, “Birdseyeview: Aerial view dataset for object classification and detection,” in *Proceedings of IEEE GLOBECOM 2019 Workshop on Computing-Centric Drone Networks*, 2019.

[4] K. Cesare, R. Skeele, Soo-Hyun Yoo, Yawei Zhang and G. Hollinger, “Multi-uav exploration with limited communication and battery,” in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 2230–2235.

[5] I. O. A. Ten Harmsel and E. Atkins, “Emergency flight planning for an energy-constrained multicopter,” in *Journal of Intelligent and Robotic Systems*, 2017, pp. 145–165.

[6] D. Jourdan, M. Piedmont, V. Gavrilets, D. Vos, and J. McCormick, “Enhancing UAV Survivability through Damage Tolerant Control,” in *AIAA Guidance, Navigation and Control Conference*, 2010.

[7] D. P. Thipphavong, R. Apaza, B. Barmore, V. Battiste, B. Burian, Q. Dao, M. Feary, S. Go, K. H. Goodrich, J. Homola *et al.*, “Urban air mobility airspace integration concepts and considerations,” in *2018 Aviation Technology, Integration, and Operations Conference*, 2018, p. 3676.

[8] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, “Uppaal: A tool suite for automatic verification of real-time systems,” *Theoretical Computer Science*, pp. RS–96–58, 1996.

[9] D. Cofer, I. Amundson, R. Sattigeri, A. Passi, C. Boggs, E. Smith, L. Gilham, T. Byun, and S. Rayadurgam, “Run-time assurance for learning-enabled systems.” in *NASA Formal Methods Symposium,*, 2020.

[10] S. Bhattacharyya, N. Neogi, T. Eskridge, M. Carvalho, and M. Stafford, “Formal assurance for cooperative intelligent agent,” in *NASA Formal Methods Symposium LNCS*, vol. 10811, 2018.

[11] J. Rushby and R. Whitehurst, “Formal verification of al software,” SRI NASA, Tech. Rep., 1989.

[12] J. Davis, D. Kingston, and L. Humphrey, “When human intuition fails: Using formal methods to find an error in the ’proof’ of a multi-agent protocol.” in *CAV*, 2019.

[13] C. Kern and M. R. Greenstreet, “Formal verification in hardware design: a survey,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 2, pp. 123–193, 1999.

[14] M. Devillers, D. Griffioen, J. Romijn, and F. Vaandrager, “Verification of a leader election protocol: Formal methods applied to ieee 1394,” *Formal methods in system design*, vol. 16, no. 3, pp. 307–320, 2000.

[15] S. Phillips, “Distributed systems and their protocols,” *Computer Communications*, vol. 7, no. 1, pp. 12 – 16, 1984. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0140366484900689

[16] E. M. Clarke and J. M. Wing, “Formal methods: State of the art and future directions,” *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[17] S. Bhattacharyya, S. Miller, J. Yang, S. Smolka, B. Meng, C. Sticksel, and C. Tinelli, “Verification of quasi-synchronous systems with uppaal,” in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, 2014, pp. 8A4–1–8A4–12.

[18] S. P. Miller, S. Bhattacharyya, C. Tinelli, S. Smolka, C. Sticksel, B. Meng, and J. Yang, “Formal verification of quasi-synchronous systems,” Rockwell Collins, Tech. Rep., 2015.

[19] M. Thomas Ball, “Formal methods and tools for distributed systems,” 2019. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2019/01/NUS2019.pdf

[20] P. Küfner, U. Nestmann, and C. Rickmann, “Formal verification of distributed algorithms,” in *IFIP International Conference on Theoretical Computer Science*. Springer, 2012, pp. 209–224.

[21] F. Fakhfakh, M. Tounsi, M. Mosbah, and A. H. Kacem, “Formal verification approaches for distributed algorithms: A systematic literature review,” *Procedia Computer Science*, vol. 126, pp. 1551 – 1560,

2018, Knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 22nd International Conference, KES-2018, Belgrade, Serbia. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050918314066

[22] S. Gritzalis, D. Spinellis, and P. Georgiadis, "Security protocols over open networks and distributed systems: Formal methods for their analysis, design, and verification," *Computer Communications*, vol. 22, no. 8, pp. 697–709, 1999.

[23] H. D. Yoo and S. M. Chankov, "Drone-delivery using autonomous mobility: An innovative approach to future last-mile delivery problems," in *2018 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 2018, pp. 1216–1220.

[24] R. Garcia and L. Barnes, "Multi-uav simulator utilizing x-plane," in *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*. Springer, 2009, pp. 393–406.

[25] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and service robotics*. Springer, 2018, pp. 621–635.

[26] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[27] C. P. P., *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Hoboken, NJ: Wiley, 2006, ch. Finite State Machine: Principle and Practice, pp. 313–371, doi: https://doi.org/10.1002/0471786411.ch10.

[28] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 359–364.

[29] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas, *PVS: Combining specification, proof checking, and model checking*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 411–414.

[30] L. D. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[31] "Uppaal :A toolbox for modeling, simulation and verification of real time systems." [Online]. Available: www.uppaal.com

[32] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Inf. Comput.*, vol. 104, no. 1, p. 2–34, May 1993.