Detection and Mitigation of Clock Deviation in the Verification & Validation of Drone-aided Lifting Operations

Abdelhakim Baouya[a,*], Brahim Hamid[a], Otmane Ait Mohamed[b], Saddek Bensalem[c]

[a]*IRIT, Université de Toulouse, CNRS, UT2, 118 Route de Narbonne, 31062 Toulouse Cedex 9, France*
[b]*Concordia University, Montréal, Canada*
[c]*Université Grenoble Alpes, VERIMAG, CNRS, Grenoble, France*

Abstract

Modern cyber-physical systems, relying on diverse computation logics, communication protocols, and technologies, are susceptible to environmental phenomena and production errors that can significantly impact system behavior. Ensuring resilience in these systems necessitates considering these factors during the high-level design stages to enable accurate functional forecasting. This paper presents an approach that models clock deviation's effects within physical and environmental conditions to perform verification & validation. We employ the OMNeT++ simulation framework to define the behavior of system in components-port-connectors fashion. The approach leverages Probabilistic Decision Tree rules, derived from the OMNeT++ simulation chart. The resulting rule-based model is then interpreted in the PRISM language for automated model verification. To validate our approach, we investigate how clock deviations influence the correctness of drone-aided lifting operations, serving as a representative application scenario. The research examines clock deviations from multiple sources, including standard specifications, product manufacturing variations, and operating temperature changes. Our examination explores the potential of validation through simulation and model checking, while also studying the approach's effectiveness in terms of scalability.

Keywords: System architecture, Clock drifts, Decision Trees, Formal methods, Drones.

Contents

*Corresponding author at : IRIT, Université de Toulouse, CNRS, UT2, 118 Route de Narbonne, 31062 Toulouse Cedex 9, France
Email addresses: abdelhakim.baouya@irit.fr (Abdelhakim Baouya), brahim.hamid@irit.fr (Brahim Hamid), otmane.aitmohamed@concordia.ca (Otmane Ait Mohamed), saddek.bensalem@univ-grenoble-alpes.fr (Saddek Bensalem)

1. Introduction

In contemporary software-intensive systems, the significance of environmental concerns has grown substantially. Early identification of these concerns is paramount, particularly during the architectural design phase when their semantics are most precisely defined [1, 2, 3]. Architecture modeling and analysis offer valuable ways to detect data inaccuracies during the initial stages of the system development process [4]. Modeling in component-based technology has emerged as a prominent research and development area due to its pivotal role in facilitating reuse and maintenance [5, 6, 7, 8].

Clock drift [9] is the gradual deviation of a clock's timekeeping from the reference time. Clocks are employed to synchronize various processes and maintain accurate data viability. However, factors such as manufacturing imperfections and temperature fluctuations [10, 11, 12] can cause degradation in a clock's accuracy over time, leading it to drift (or deviate) away from the reference time. Consider Aerial Wireless Sensor Networks (AWSNs) [13] as an example. Here, drones rely on sensors like cameras and LiDAR[1] for data collection, which requires precise timing. Clock drift can disrupt the synchronization between these sensors and the flight controller, leading to misaligned or misinterpreted data. To mitigate these effects, systems often employ synchronization protocols like Network Time Protocol (NTP) [14] for computer networks that regularly adjust their clocks with more accurate reference sources [15]. One synchronization algorithm employed to optimize battery life is the Firefly Gossip Protocol (FiGO) [10, 11, 16]. Its main purpose is node synchronization while disseminating data over WSNs [17]. The Firefly synchronization mechanism leverages listening windows. Upon startup, a node listens for broadcasts from neighboring agents (in our paper; it will be gateways). If a broadcast is received, the node adjusts its clock time to the average of the advertised times from its neighbors. This process iterates until all nodes synchronize their listening windows. The Gossip protocol, employed alongside Firefly, utilizes a "polite gossip" approach similar to Trickle as mentioned in [18].

Formal methods are widely used to mathematically verify properties related to the structure and behavior of physical systems [2, 7, 8, 19]. Their application in assessing clock deviation has been investigated in [10]. In this study, the authors verify whether the modeled system including communicating drones adheres to the standard protocols and behaves equivalently to the simulated system (observed through oscillators) using PRISM. It provides a key insight before implementing a swarm of drones. However, the state space explosion problem is addressed by removing some essential parts of the model losing its coherence. However, these methods often face limitations in terms of scalability. In response, several research works have explored abstraction techniques, such as partial-order reduction [20], [21], assume-guarantee reasoning [22, 23], SAT-based model checking [24, 25], and symmetry reduction [26]. While these approaches may lose some detail, an alternative approach involves capturing the system's behavior through a dedicated simulation platform by reasoning on the generated dataset when deriving a state-based model [27, 28]. This approach is acceptable by industrials [29, 30], which can enhance confidence in the original model's specifications [31].

Our work leverages formal methods to enable the specification and analysis of communicating cyber systems, taking into account the inherent challenges presented by variations in clock measurements. We achieve this by employing two complementary description languages that capture the system's structural and behavioral semantics, namely OMNeT++ [32] and PRISM [33]. This allows us to specify the system, detect, and reason for potential clocks inaccuracies. Specifically, we adopt a two-pronged approach: (1) *Modeling Component-based Architecture*: We utilize OMNeT++ to create a detailed high-fidelity model of the system's component-based architecture. This provides a concrete system implementation and serves as a reference point for our formal analysis. (2)*Formalization and Verification using PRISM*: We leverage PRISM's robust framework for expressing and analyzing probabilistic behavior. In particular, we employ Probabilistic Automata (PA) to capture the system's state transitions and Probabilistic Computation Tree Logic (PCTL) [34] to express and verify desired properties regarding system behavior. Additionally, we have developed a custom derivation flow within PRISM that interprets Probabilistic Decision Trees (PDT) generated from OMNeT++ simulation data, facilitating seamless integration and analysis. This approach is demonstrated through a use case scenario from the European research project CPS4EU [35]. The scenario involves two synchronized drones collaborating on a cooperative

[1] Light Detection And Ranging

lifting task. Resynchronization becomes necessary due to temperature variations, differing manufacturing standards, or production spread. Precise synchronization is crucial for maintaining a high-quality lifting service in these situations. However, it is important to note that drone systems are often highly optimized and may not be equipped with traditional clocks.

1.1. Contribution overview

Prior research has not investigated clock deviation modeling from a component-port-connector (CPC) perspective. To address this gap, we propose a novel approach for accurate and early assessment of communication system reliability at the software architecture level, considering the impact of clock deviation. Furthermore, we investigate the scalability of the proposed approach to understand its feasibility for the implemented clock deviation.

Based on research conducted by [10] and [12], we have identified three primary sources of clock deviation in systems compliant with the IEEE 802.15.4 standard: 1) variations during production, 2) implementation-specific deviations within the standard, and 3) differences in operating temperatures. Therefore, our contribution can be summarized as follows:

1. Formalizing clock deviation-related protocols aims to provide practitioners with mathematical insights, facilitating its application using formal methods.
2. Implementing clock deviation to observe variabilities based on simulation and model-checking.
3. Learning rule-based models from the simulation of clock deviation-related protocols to serve as a formal model for probabilistic assessment.
4. For broader applicability, we focus on the scalability of the learned model stemming from simulation. This ensures effective deployment on larger and more complex systems while maintaining accuracy and efficiency.
5. Evaluation of contributions in the context of crane orchestration via drones using PRISM for model-checking and OMNet++ for simulation.

1.2. Outline

The paper is structured as follows: Section 2 provides an overview and analysis of related work in this field. Section 3 provides the necessary background material to understand the proposed approach. Section 4 provides the workflow of the approach for investigating clock drift. Section 5 explains how manifestations of clock deviation are modeled within CPC models. To demonstrate the effectiveness of our approach, Section 6 presents a detailed use case involving crane orchestration through a drone system, where various analyses are performed. Section 7 discusses relevant outcomes related to the state space explosion and threats to validity. Finally, in Section 8, we conclude the paper by summarizing key findings and outlining potential directions for future research endeavors.

2. Literature review

This section lays the groundwork by exploring the fundamentals of clock drift, also known as clock deviation (both terms are interchangeable). We will delve into existing research activities focused on modeling and resolving clock deviation.

2.1. Clock drifts

Several research papers have explored the phenomenon of clock drift caused by variations within internal clocks. In this work, we build upon the research established in [10] and [12]. We categorize the different sources of clock variation into three main categories: Temperature Variations, Standard-Specific Drift, and Production Spread. [36].

2.1.1. Clock drift caused by temperature variations

According to Webster et al. [10] [11], differences in operating temperature can affect drone clock synchronization. If the clocks have different temperatures, the warmer clock will lag behind the colder clock by one tick. This means the colder clock will have ticked twice while the warmer clock has ticked only once, likely due to oscillator variations in [37] as Lenzen et al. observed similar phenomena on the Mica2 processor. Hence, on average, after every `230,468` ticks, the warmer clock will fall behind the colder one by exactly one tick. This was then translated into a probability of 1 in `230,468`, equivalent to `0.000004339`.

2.1.2. Standard-specific clock drift

According to the IEEE 802.15.4 standard and the work established by Elsts et al. [12] set a revision definition of maximum resynchronization period between wireless devices as it refers to the longest interval between synchronizations a network can tolerate while ensuring reliable communication. Three factors are identified in this standards, the guard time t_g (This is the time a receiver listens for a packet beyond the expected arrival time to account for clock drift), the preamble transmission time t_p (This is the time it takes to transmit the preamble, a signal preceding the actual data in a packet), and clock drift (δ). The following equation relates the maximum resynchronization period (Δt) to guard time and clock drift: $\Delta t \leq (t_g - 2 * t_p)/2 * \delta$. To illustrate these clock variations, a Python code is available in section artifacts that simulate drone resynchronization. This code extracts the average clock drift, which will be used for verification and validation purposes. The relevant Python code can be found in [38] under the model reference **P1**. The computed average clock drift is approximately `0.44` (parts per million).

2.1.3. Clock drift caused by production spread

Clock drift caused by production spread refers to the slight variations in clock frequency that occur between individual devices despite being manufactured identically. The manufacturer typically specifies the range (bounds) of these deviations. According to [12], when using oscillators, a tolerable drift in the worst-case scenario can be as high as `0.40` parts per million (ppm).

2.2. Reasoning on clock drifts

This literature review comprehensively surveys research on clock deviation or clock drift phenomena in networking. As discussed in Section 2.1.1, Elsts et al. [12] provide a comprehensive foundation for understanding clock drift related to temperature variations, standard-specific deviations, and production spread. These phenomena are all validated through oscillator observations. We rely on those parameters to build our models in OMNeT++ and PRISM. Moreover, the survey provided by Swain and Hansdah [39] to better understand clock synchronization protocols in wireless sensor networks (WSNs) from the perspective of how and when clock values are propagated within the network and how physical clocks are updated. Hauweele and Quoitin [40] study investigates clock drift in Wireless Sensor Networks (WSNs) and its impact on Radio Duty Cycle (RDC) protocols [41]. The authors address limitations in the COOJA simulator [42], which cannot accurately reproduce the clock drift. They propose a new mathematical model that allows them to simulate clock drift that causes periodic communication blackouts – a phenomenon previously impossible to replicate within COOJA. Several research works focus on achieving fast synchronization between physical clocks in Wireless Sensor Networks (WSNs) as in [43]; Xie et al. ensure a valid clock while it requires minimal computation communication from each node as it is suitable for large-scale WSNs due to its fast convergence. Zhang et al. [44] propose a robust maximum time synchronization (RMTS) scheme for clock synchronization in the presence of both noise and packet loss. This scheme leverages the concept of maximum consensus and is theoretically proven to be effective. The authors also validate its performance through simulation. Li et al. [45] focuses on the challenges of verifying timed security protocols when clock drift is present. It proposes an extension of the timed applied pi-calculus formalism to model clock drift and presents an automated verification approach for security properties [46]. Al-bayati et al.[36] proposes a probabilistic model checking approach to verify Clock Domain Crossing (CDC) interfaces. It models CDC interfaces as Markov Decision Processes (MDP) and uses PCTL properties to verify their correctness. The MDP incorporates a synchronizer with various states to mitigate metastability-related failures. Varalakshmi et al.[47] introduce a novel clock synchronization technique for clients and servers in the cloud. This synchronization

strengthens a "port hopping" method designed to prevent Denial-of-Service (DoS) attacks on open server ports. By knowing the open ports for a specific timeframe, it effectively thwarts DoS attacks on those ports. Tsurumi et al. [48] propose a clock drift estimation and compensation scheme for PLIM that eliminates any additional computational burden on end nodes (ENs). This is achieved by performing the entire operation at the gateway (GW) side. The authors validate their approach through both experimental testing and real-world implementation on a commercial LoRaWAN system. Lübken and Förster [49] address time synchronization in space WSNs. Harsh environments cause clock drift between sensor nodes, hindering low-latency data acquisition. The paper proposes a novel Adaptive Local Clock Control (ALCC) algorithm that significantly outperforms existing methods by minimizing clock deviation under these extreme conditions. The effectiveness of ALCC is validated through simulations and hardware testing. Kaburaki et al. [50] tackle data collection challenges in sensor networks using Low-Power Wide-Area Networks (LPWANs). The issue arises from collisions caused by both simple access schemes and clock drift in these networks. The authors propose a resource allocation scheme to avoid packet collisions under the effect of clock drift. Simulations show this method significantly improves packet delivery rates compared to existing approaches.

Traditionally, clock synchronization research relies heavily on simulation [39, 40, 43, 44, 47, 48, 49, 50, 51] for performance evaluation, model checking for the feasibility of operations under clock drifts [10, 36, 45], and estimation methods [52]. However, there is a lack of studies investigating the applicability of simulation and model checking (Validation & Verification) tools to assess the efficiency of synchronization algorithms compared to existing tools. While works like [10] provide a strong foundation for mathematically building synchronization protocols, their reliance on abstraction to manage state space explosion can be a disadvantage. Our approach explores synchronization through simulation and model checking but then learns models that accelerate reasoning. This is achieved not through abstraction, but by learning from streaming simulation data. The resulting models can then be verified using properties expressed in temporal logic.

3. Background

In this section, we explore the formalisms that capture the underlying semantics associated with clock drifts. Additionally, we will provide the formalization of Probabilistic Decision Trees (PDTs).

3.1. Modeling formalism

The modeling formalism presented in this paper primarily relies on PRISM. However, we will emphasize the mathematical foundations to acquaint readers with Probabilistic Automata (PA).

3.1.1. The PRISM language

We consider PA in PRISM language [33] as a modeling formalism to capture the semantics of stochastic automata and express the non-determinism in wireless sensor networks.

The PRISM model $\mathscr{P}$ is represented by a composition of modules on actions. Modules are characterized by variables and commands where the state of the module is the variable valuations. The behavior of each module is described by a set of commands (i.e., transitions). A command takes the form: $[a]g \rightarrow \lambda_1 : u_1 + \ldots + \lambda_n : u_n$ or, $[a]g \rightarrow u$, which means, for the action " a " if the guard " g " is true, then, an update " u_i " is enabled with a probability " λ_i ". A guard is a condition that determines whether a command can be executed based on evaluations of variables using logic operators. The update " u_i " is an evaluation of variables v_i ; evaluated via expressions denoted by " θ " such that $\theta : V \rightarrow \mathbb{D}$, where $\mathbb{D}$ is a domain of variables such that $\mathbb{D} = \mathbb{N} \cup \{true, false\}$.

Definition 1 ($\mathscr{P}$ state)**.** A state of $\mathscr{P}$ is the valuation of $\mathscr{P}$ modules variables $v_0, \ldots, v_n$ in the form $s = \langle x_0, \ldots, x_n, \theta \rangle$ such that $\theta \models g$ and g is the guard over states values., where x_i is the valuation of variable v_i .

3.1.2. Probabilistic automata

PA [53] is a modeling formalism that exhibits probabilistic and nondeterministic features implemented by PRISM. Definition 2 formally illustrates a PA where $Dist(S)$ denotes the set of convex distributions over the set of states S and μ is a distribution in $Dist(S)$ that assigns a probability $\mu(s_i) = p_i$ to the state $s_i \in S$.

Definition 2. A PA is a tuple $\mathscr{M} = \langle s_0, S, \Sigma, AP, L, \delta \rangle$:

- s_0 is an initial state, such that $s_0 \in S$,
- S is a set of states,
- Σ is a finite set of actions, communicating between automata is based on notation (!) for sending and (?) for receiving communicating data.
- $L : S \to 2^{AP}$ is a labeling function that assigns each state $s \in S$ to a set of atomic propositions taken from the set of atomic propositions (AP), and
- $\delta : S \times \Sigma \to Dist(S)$ is a probabilistic transition function assigning for each $s \in S$ and $\alpha \in \Sigma$ a probabilistic distribution $\mu \in Dist(S)$. The operational semantics rules in Figure 1 model all generic probabilistic transitions.

For PA's composition, this concept is modeled by the parallel composition [53]. During synchronization, each PA resolves its probabilistic choice independently [53]. For transitions, $s_1 \xrightarrow{\alpha} \mu_1$ and $s_2 \xrightarrow{\alpha} \mu_2$ that synchronize in α then the composed state (s_1', s_2') is reached from the state (s_1, s_2) with probability ($\mu_1(s_1') \times \mu_2(s_2')$). In the no synchronization case, a PA takes a transition where the other remains in its current state with probability one.

Performing probabilistic model checking relies on PA and properties expressed in Probabilistic Computation Tree Logic (PCTL) [34], [53], [54]. PCTL enables the expression of probabilistic queries such as "*What is the probability that the system will eventually reboot ?*", expressed as P =? [F reboot]. In this context, "reboot" is the label that refers to the states indicating a system reboot.

3.2. Probabilistic Decision Trees

PDTs [55] are a type of decision tree model that incorporates probabilistic information into the decision-making process. Unlike traditional decision trees that make deterministic decisions based on fixed rules, PDTs assign probabilities to each possible outcome at each decision node. Considering a dataset of a set of features as inputs I and a set of output features Q that are used mainly for reasoning, we can define PDTs as follows:

Definition 3. A PDT is represented as a tuple $\mathscr{DT} = \langle I, G, X, Rule \rangle$, where:

- I represents a set of input features
- G is a set of constraints applied over I and
- X represents a set of feature values. We define g_i as a constraint on each input feature I and $eval_v : I \to \mathbb{D}$ to evaluate the features of the PDT. Here, $v_0, \ldots, v_n$ denote the valuations of the output features.

When exploring the tree from the root to the leaf, a rule takes the form of $x_0 \wedge \ldots \wedge x_n \to_\lambda x_{n+1} \wedge \ldots \wedge x_{n+j}$, where λ represents the probability of occurrence of the rule in the dataset and $x_0 \models g_0 \ldots \wedge x_n \models g_n$. We use $[\![\mathscr{P}_{\mathscr{DT}}]\!]$ to identify a set of rules derived from the dataset.

The example in Figure 2 shows the structure of the PDT, where each node represents an input feature $I = \{\texttt{CO2}, \texttt{Humidity}\}$, except for the leaf node, which represents the output feature $I = I \cup \{\texttt{Ventilation}\}$. In this specific case, we can interpret the following rule: `CO2 >= High` $\wedge$ `humidity >= 80%` $\to_1$ `Ventilation level= 2`

The formalism of PDT can be encompassed within the formalism of Probabilistic Automata (PA), where the states capture the PDT features and transition the evolution of state features.

- **Update**. This axiom describes the probabilistic local updates for variable v_i :

$$\frac{s_i \xrightarrow{g:a}_{\lambda_i} s_i' \wedge \theta \models g}{\langle s_0, \ldots, s_i, \ldots, s_n, \theta \rangle \xrightarrow{a}_{\lambda_i} \langle s_0, \ldots, s_i', \ldots, s_n, \theta' \rangle}$$

  where $\theta' = \theta[v_i := effect(v_i)]$

- **Synchronization**. This axiom permits the synchronization between modules on a given action a:

$$\frac{s_i \xrightarrow{g_1:a}_{\lambda_i} s_i' \wedge \theta \models g_1 \wedge s_j \xrightarrow{g_2:a}_{\lambda_j} s_j' \wedge \theta \models g_2}{\langle s_0, \ldots, s_i, \ldots, s_j, \ldots, s_n, \theta \rangle \xrightarrow{a}_{\lambda_i \times \lambda_j} \langle s_0, \ldots, s_i', \ldots, s_j', \ldots, s_n, \theta' \rangle}$$

  where $\theta' = \theta[v_i := effect(v_i)]$

- **Send/receive**. This axiom describes synchronous data exchange:

$$\frac{s_i \xrightarrow{g_1:a!v_i}_{\lambda_i} s_i' \wedge \theta \models g_1 \wedge s_j \xrightarrow{g_2:a?v_j}_{\lambda_j} s_j' \wedge \theta \models g_2}{\langle s_0, \ldots, s_i, \ldots, s_j, \ldots, s_n, \theta \rangle \xrightarrow{a}_{\lambda_i \times \lambda_j} \langle s_0, \ldots, s_i', \ldots, s_j', \ldots, s_n, \theta' \rangle}$$

  where $\theta' = \theta[v_j := v_i]$

Figure 1: PRISM-PA Semantics Rules.

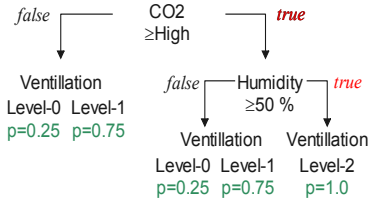

Figure 2: Example of a PDT

4. Approach

We rely on the workflow depicted in Figure 3 to address the question raised in our contribution. The key stages of the process involve *Modeling*, *Simulation*, *Learning*, and *Verification*.

Modeling. This specification uses natural language, making it easy for experts to model their existing system and identify clock frequencies from various sources. The modeled system is then translated into the PRISM language for verification using model-checking techniques (referred to as $\mathscr{P}_1$). An OMNeT++ model is also built based on a natural language specification using C/C++ constructs.

Simulation. The OMNeT ++ model was then simulated within the OMNeT++ graphical simulator. This simulation generates a set of traces to evaluate the timing and synchronization behavior of the system.

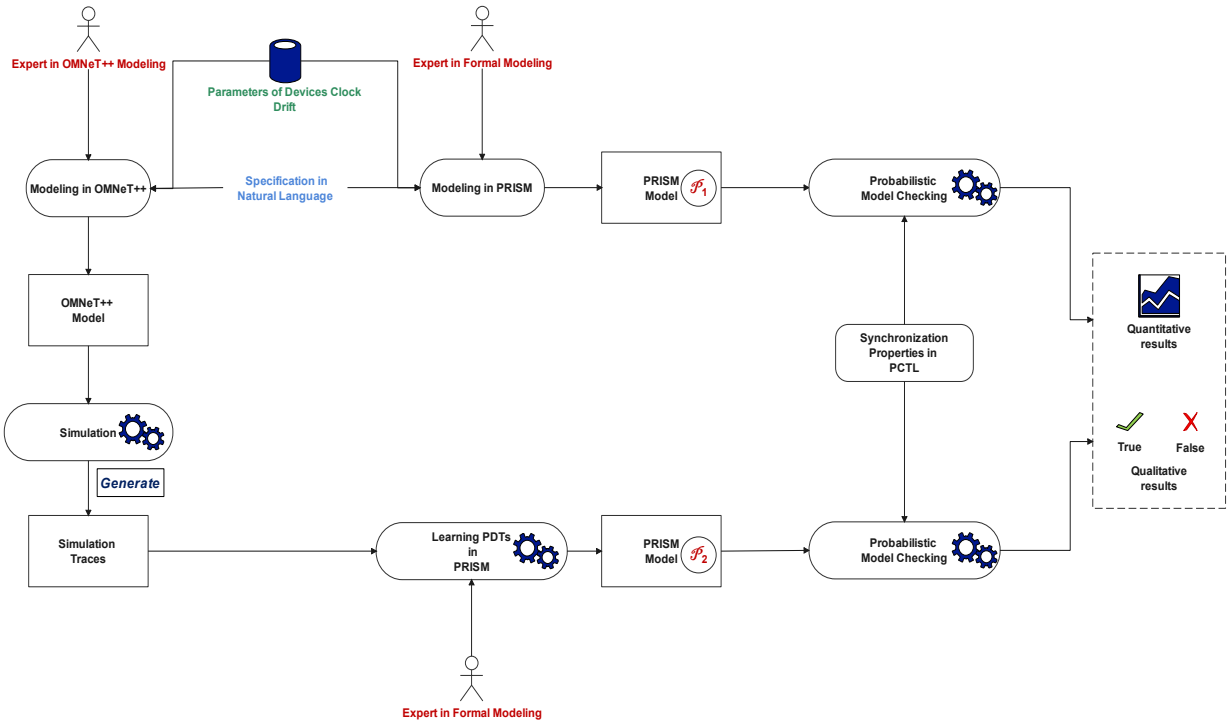

Figure 3: Proposed Flow for Verification and Validation.

Learning. PDT is generated by learning from the traces collected from the OMNeT++ simulator. The learned PDT is then transformed into a new PRISM model $\mathscr{P}_2$.

Verification. Finally, we evaluate the reliability of the synchronization planned by the PRISM models (M1 and M2) by checking them against system properties expressed in Probabilistic Computation Tree Logic (PCTL). We will also discuss some verification outcomes related to the scalability of the model $\mathscr{P}_2$.

5. Communication in Challenging Physical and Environmental Conditions

This section dives into the details of the modified Firefly-Gossip Protocol (FiGo) protocol, responsible for synchronizing both clocks and payload data. We will formally describe the deviation of internal clock ticks. Additionally, we will present the algorithm used for clock computation and synchronization at the network edge. To lay the groundwork, we first formalize the architectural and behavioral language of OMNeT++, allowing us to capture the system's architecture accurately.

5.1. Formalization of OMNeT++

OMNeT++ [32] is a versatile C++ simulation library and framework specifically designed for building network and Cyber-Physical simulators. It offers extensibility, modularity, and a component-based approach. The modeling framework comprises two essential components: the NED topology description language and the component code written in C/C++. The NED modeling language is composed of a set of components and connections. We reuse the function $eval$ defined previously such that $eval : \vartheta \rightarrow \mathbb{D}$. The NED Component is defined as follows:

Definition 4. A NED component " $\mathscr{ND}$ " is a tuple $\mathscr{ND} = \langle \vartheta, \Sigma, Bh \rangle$, where:

- $\vartheta = \vartheta_p \cup \vartheta_m \cup \vartheta_l$ is a finite set of component variables

- ϑ_p is a finite set of parameters associated with the component,
- ϑ_m is a finite set of messages associated with the component. Also, we consider the X as valuation over ϑ_m and X' as its new valuation.
- ϑ_l is a finite set of local variables associated with the component.

- Σ is a finite set of gates that can take the form of $input$ (formally ?) and $output$ (formally !). We identify by τ as internal behavior related to the OMNeT component.
- Bh function is implemented in C/C++ and is responsible for performing specific operations on the received/sent message.

To gain a deeper understanding of the OMNet++ component behavior, we establish semantics for the $\mathcal{ND}$ state based on component variables as follows:

Definition 5 ($\mathcal{ND}$ state)**.** A $\mathcal{ND}$ state " s " is the valuation θ of $\mathcal{ND}$ components variables such that: $s = \langle x_0, \ldots, x_n, \theta \rangle$ such that $\theta \models g$ and g is the guard over states values.

Each $\mathcal{ND}$ component is capable of establishing communication with other $\mathcal{ND}$ components by utilizing its gates to create a composite component. The definition is outlined as follows:

Definition 6 (Interaction)**.** An interaction in OMNeT++ refers to a realized connectivity activity between components. It can be represented as a tuple $\langle \Sigma_a, d_a \rangle$, where: $\Sigma_a \subseteq \Sigma$ represents a set of gates that are associated with the interaction a , and d_a is a delay associated with the interaction.

The result of the interaction is a composition of synchronized components obtained by using the component composition operator γ presented in Definition 7.

Definition 7 (Composition)**.** The composition γ ($\mathcal{ND}_1, \ldots, \mathcal{ND}_n$) of n components is a network component resulting from $\mathcal{ND}$ composition on interaction $\Sigma_a \subseteq \Sigma$ in Definition 6 represented by *Internal Update* and *Send/receive* operational semantics rules:

$$\frac{s_0 \xhookrightarrow{\tau} s'_0}{\langle s_0, \ldots, s_n \rangle \xrightarrow{\tau} \langle s'_0, \ldots, s_n \rangle} \quad \textit{(Internal Update)}$$

where $X'_{\vartheta_l} = X[v_l := bh(l)]$

$$\frac{s_0 \xhookrightarrow{a!m} s'_0 \wedge s_1 \xhookrightarrow{a?m} s'_1}{\langle s_0, \ldots, s_1, \ldots, s_n \rangle \xrightarrow{a} \langle s'_0, \ldots, s'_1, \ldots, s_n \rangle} \quad \textit{(Send/receive)}$$

where $X'_{\vartheta_m} = X[v_m := m]$

The purpose of verifying FiGo protocol is to ensure that liveness and safety requirements. In the context of the FiGo protocol, the liveness requirement is expressed as follows: "*When the clock ticks of the sensors are not synchronized, they should eventually synchronize*". In the payload case, the liveness requirement is expressed as follows: "*When the sensor payloads are not equivalent, they should eventually become equivalent.*" However, the safety requirement ensures that "*the sensor command is received at the same instant of time H*".

5.2. The Firefly-Gossip Protocol

The FiGo algorithm has been specifically implemented for object synchronization, as demonstrated in [10, 11, 16]. However, researchers have made modifications to fine-tune the algorithm to meet their specific requirements. Initially, the algorithm was designed to synchronize clocks among sensor nodes (Figure 4), facilitating a unified time measurement across the network. Our modifications to the algorithm enable the edge gateway to calculate the average clock of the sensor nodes. The modified algorithm is presented in Algorithm 1. Input parameters

include cycleLength, refractoryPeriod, the nextBroadcast, and sameThreshold. The refractoryPeriod is a duration during which no further transmissions are made after the initial transmission to conserve energy usage. The cycleLength is the total period in which the sensor node is performing computation, transmission, and sleeping (i.e., refractoryPeriod). The nextBroadcast represents the precise moment in time when the sensor node transmits its local clocks and payload.

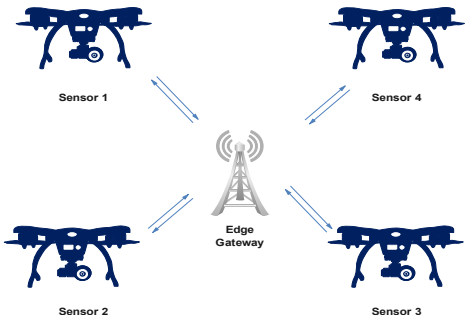

Figure 4: Sensor Nodes Orchestration using Edge Gateway.

Each sensor node is uniquely identified by its local clock tick H, payload Pld, and the collected message msg, which is initially empty. Additionally, the sameCount flag is used to indicate the number of times the message has been broadcasted out of the sensor nodes zone. These sensor information are portrayed in lines 1-4.

In the initial stage of the protocol, in line 6, the sensor node actively listens for incoming messages. If the message is stated listening (line 7), the protocol is triggered to execute the instructions outlined in lines 8-26. if the sensor node is in sleeping mode, then it calibrates its clocks to the average value of the sensor nodes of its zones gathered from the zone gateway (lines 8-11). If the collected payload and the sensor payload are equivalent, the sensor node sends an order to the activator and increments the sameCount value to 1, indicating that one order has been executed during the cycleLength. If the sensor payload exceeds the calculated average, it needs to be calibrated. Similarly, if the payload differs from the message payload, the sensor should transmit its information to the gateway to recalculate the average time value. This operation is executed when the sensor nodes disconnect from the sensor zone. Furthermore, if the received message is not in the listening mode, a broadcast is sent to the gateway, and the sameCount is reset.

When a sensor's cycleLength is reached, both the local clock tick and the sameCount are reset in lines 29-30. When the sensor clock tick does not meet the sensor's cycleLength, the sensor clock tick continues to increase.

Algorithm 1: FiGo Communication and Synchronization Protocol

Data: cycleLength, refractoryPeriod, nextBroadcast, sameThreshold.

```
1  H ← Init ;                                          /* The initial value of drone tick. */
2  Pld ← 1 ;                                           /* The initial payload value. */
3  msg ← empty() ;                                     /* The message status. */
4  sameCount ← 0 ;                                     /* The same count variable for lifting. */
5  while True do
6      msg ← heard();
7      if msg.listen() then
8          if (H > refractoryPeriod) then
9              H ← msg.H;        /* Update the local clock tick with the synchronized tick. */
10             Pld ← msg.Pld;
11         end
12         if (Pld == msg.Pld) then
13             sameCount ← sameCount + 1;
14             sendCommand(H, Pld);                    /* Send lift message to the crane. */
15             Pld ← sensing();                        /* Sensing a new payload. */
16         else
17             if (Pld > msg.Pld) then
18                 Pld ← msg.Pld
19             else
20                 transmit(H, Pld)
21             end
22         end
23     else
24         if (H==nextBroadcast) || ( (H==(nextBroadcast+refractoryPeriod) % cycleLenght) &&
            sameCount<sameThreshold) then
25             transmit(H, Pld); /* transmits the local clock tick and payload to the gateway.
                */
26             sameCount ← 0;
27         end
28     end
29     if (H==cycleLenght) then
30         H ← 0 ;                                     /* reset the clock tick and sameCount. */
31         sameCount ← 0;
32     else
33         H ← H + 1 ;                    /* tick progress until cycle length is reached. */
34     end
35 end
```

5.3. Clock Deviation in FiGo

In the absence of synchronization, device clocks operate autonomously without coordination. Each sensor sends its payload at its own clock frequency without acknowledging a reference clock. Taking into account the implementation of the FiGo protocol outlined in Algorithm 1, the clock progress is carried out by the sensor at line 33, utilizing a one-time unit. However, in the presence of environmental conditions, the internal clock ticks may progress differently. It is assumed that in the presence of drifting, the clock progress occurs with an interval of two-time units [10]. In the background section, we have introduced three distinct approaches to illustrate progress with different rates. Assuming that the rate of drift is denoted as λ , we model two rules inherited from OMNeT++ 1 as follows:

$$\frac{s_0 \overset{\tau}{\hookrightarrow}_{\lambda} s_0'}{\langle s_0, \ldots, s_n, X\rangle \overset{\tau}{\rightarrow}_{\lambda} \langle s_0', \ldots, s_n, X'\rangle} \quad \textit{(Clock Drift)}$$

where $X'_{H_l} = X[H_l := H_l + 2]$

$$\frac{s_0 \overset{\tau}{\hookrightarrow}_{1-\lambda} s_0'}{\langle s_0, \ldots, s_n, X\rangle \overset{\tau}{\rightarrow}_{1-\lambda} \langle s_0', \ldots, s_n, X'\rangle} \quad \textit{(No Clock Drift)}$$

where $X'_{H_l} = X[H_l := H_l + 1]$

5.4. The Synchronization Gateway

Clock drift mitigation involves the utilization of an intermediary component or coordinator to reset the internal clock. This reset is based on the average of the clocks received from various devices, as emphasized by [11]. In our proposed approach, the coordinator is responsible for gathering the internal clocks from all the sensors. Once the clocks are collected, an averaging calculation is performed to determine the new clock value. Subsequently, this new value is communicated back to the respective devices.

The classical FiGo algorithm handles synchronization among the sensor nodes. However, in this modified version, the responsibility for synchronization is shifted to the gateway. When a sensor transmits its local clock and payloads, the gateway performs the synchronization process. For this purpose, we refer to the gateway as "InBox". The algorithm depicting this synchronization process can be found in Algorithm 2.

Algorithm 2 is designed to operate with the maximum sensor value, denoted as `max`, within its designated area. Each gateway is uniquely identified in lines 1-6 based on its calculated clock tick `H`, payload `Pld`, and the received/sent message `msg`. The variable `Length` represents the count of received transmit messages and is initialized to 0. The payload and clock ticks are stored separately in two distinct stacks, namely `ListOfH` and `ListOfPld`.

The "InBox" gateway initiates the process by listening to incoming messages, as indicated in line 8 of the algorithm. If the message is flagged as being in transmit mode, the corresponding clock tick and payload are pushed into their respective stacks, namely `ListOfH` and `ListOfPld` in lines 10-11. In this algorithm, the length is incremented to keep track of the received messages. When the length of received messages reaches the maximum number of incoming messages (i.e., sensor nodes), the gateway calculates the average clock ticks and payload in lines 15-16. Subsequently, it broadcasts these messages to its neighboring nodes in line 17. Finally, the stack length is reset to 0, and the stacks are emptied. Formally, we model the operational semantics of the messages received (including clocks and payloads) by the inbox originating from sensor nodes using Rule *Receive*. On the other hand, Rule *Broadcast* describes how the averaged clock is broadcasted to the sensor nodes. The communication channel selected for data transfer is denoted as "p".

$$\frac{s_0 \overset{p?\langle H,Pld\rangle}{\hookrightarrow} s_0'}{\langle s_0, \ldots, s_n, X\rangle \overset{p}{\rightarrow} \langle s_0', \ldots, s_n, X'\rangle} \quad \textit{(Receive)}$$

where $X' = X[length := length + 1]$

$$\frac{[\![Inbox]\!] = s_i \overset{p!\langle H,Pld\rangle}{\hookrightarrow} s_i' \bigwedge_{j\ldots n} [\![\mathscr{ND}_j]\!] = s_j \overset{p?\langle H,Pld\rangle}{\hookrightarrow} s_j'}{\langle s_i, s_j, \ldots, s_n, X\rangle \overset{p}{\rightarrow} \langle s_i', s_j', \ldots, s_n, X'\rangle} \quad \textit{(Broadcast)}$$

where $X' = X[length := 0]$

Algorithm 2: InBox implementation at the Edge Gateway

Data: max.

$H \leftarrow Init$; /* The initial value of drone tick. */
$Pld \leftarrow 1$; /* The payload message for crane lifting. */
$msg \leftarrow empty()$; /* The message status. */
$Length \leftarrow 0$; /* The drones length. */
$ListOfH \leftarrow stack()$; /* The list of clock ticks. */
$ListOfPld \leftarrow stack()$; /* The list of payload. */
while $True$ **do**
  $msg \leftarrow heard()$;
  if $(msg.transmit())$ **then**
    $ListOfH.push(msg.H)$; /* push the received clock ticks to the stack. */
    $ListOfPld.push(msg.Pld)$; /* push the received payloads to the stack. */
    $Length \leftarrow Length + 1$; /* increments the stack length. */
  end
  if $(Length == max)$ **then**
    $H \leftarrow \text{AVG}(ListOfH)$;
    $Pld \leftarrow \text{AVG}(ListOfPld)$;
    $broadcast(H, Pld)$; /* broadcast the average value of clock ticks and payloads. */
    $Length \leftarrow 0$
    $ListOfH \leftarrow empty()$
    $ListOfPld \leftarrow empty()$
  end
end

6. Experiments

6.1. Case study

Description: The use case scenario (Figure 5) [35] involves the process of lifting a platform using drones without human intervention. The cranes use sophisticated automation to perform lifting tasks with minimal human intervention. The project utilizes drones to synchronize the lifting operation between cranes. At each time unit H , the drone sends a lifting measurement to one crane in order to stabilize the platform. The platform is equipped with two markers scanned by the drone to calculate their distance from the ground. The resulting lifting distance is then sent as a payload message to one crane until an optimal position for the platform is reached. The operation necessitates the involvement of an edge gateway responsible for clock synchronization. Additionally, one of the drones is particularly sensitive to environmental conditions, which can result in the discussed clock drift, as mentioned in the previous sections.

As outlined in [56], the process flow is described as follows:

Actor: Drone Operator, Crane Operator.

Preconditions:

- A stable landing pad for drones has been set up.
- One drone is available for operation.
- Cranes have been positioned at strategic locations around the site.

Basic Flow:

- (1) The drone operator initializes communication messaging between drones and the gateway.
- At each time unit H , the drone sends its internal clock ticks value to the edge and then receives the average values (2).

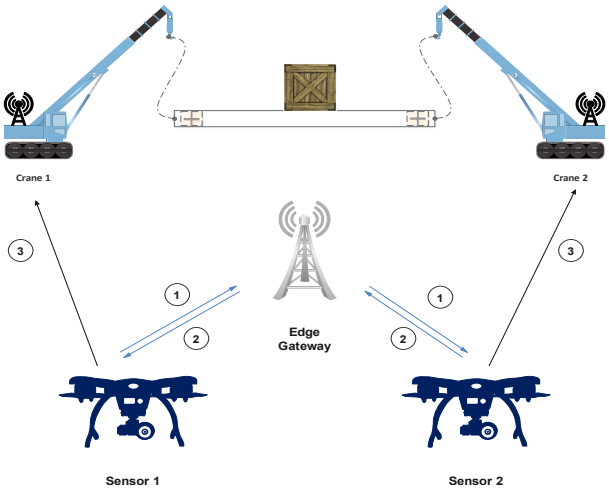

Figure 5: Cranes Lifting Orchestration via a Drone [35].

- The drone scans two markers on the lift system's platform and calculates their distance from the ground level.
- Using this information, (3) they send a payload message containing calculated lift measurements required by different cranes involved in stabilizing operations.
- Crane operators receive these messages and adjust their equipment accordingly.
- The process repeats iteratively until optimal positioning has been achieved through the refinement of lift measurements communicated between these two systems.

Post-Conditions:

- The lift operation was completed successfully without any damages or accidents.
- The cargo was safely lifted onto its designated location.

Experimental setup. We used PRISM v4.7 for our analysis. The experiments were conducted on an Ubuntu-I7 system with 32GB of RAM. The system models are available online [38], with references `M1` to `M6`. The engine is set to "Explicit"; PRISM supports other computation engines (MTBDD, sparse, and hybrid) that offer different performances regarding model and properties structure (Please refer to documentation in [57]). Rewards properties are not supported by the "Explicit" engine. A Python code was developed to learn PDT based on the generated OMNeT++ dataset and mapped it to the PRISM code under reference `P2`. We describe the desired properties of our models in the Modeling section.

Artefacts. The experiments detailed in this section are both publicly available and fully reproducible. The source code in PRISM and OMNeT++ can be accessed from the public GitHub repository [38]. The website includes instructions for setting up and running the experiments and how to use the OMNeT++ execution trace data.

6.2. Drones and InBox behavior modeling

The modeled system for drone behavior is depicted in Algorithm 1. The algorithm is structured with if-then-else statements that must be translated into PRISM commands. This translation is based on personal knowledge rather than an automated procedure. The first instruction in Algorithm 1 line 8 is to check if the drone is in a

Listing 1: PRISM Code for Drone Module

```
1  module D1
2  /*Variable definitions */
3  ...
4  D1H : [0..MAXH] init H1;
5  ...
6  /* phase 1*/
7   [D1_STATUS_LISTENING] s=-1 & listen1 -> (s'=0);
8
9  [D1_CHECK_PARAMETER] s=0 & D1H > D1refractoryPeriod -> (s'=1) & (D1H'=
       MSG_H)& (D1Pld'=MSG_PLD);
10 [D1_CHECK_PARAMETER_NOT] s=0 & !(D1H > D1refractoryPeriod) -> (s'=1);
11
12 /*Phase 2 */
13 [D1_SAME_COUNT] s=1 & D1Pld=MSG_PLD & D1sameCount<2 -> (s'=2) & (
       D1sameCount'=D1sameCount+1);
14
15 [D1_ELSE_SAME_COUNT] s=1 & D1Pld>MSG_PLD  -> (s'=5) & (D1Pld'=MSG_PLD);
16
17 [D1_ELSE_ELSE_SAME_COUNT] s=1 & D1Pld<MSG_PLD  -> (s'=5);
18
19 /*Phase 3 */
20 [D1_TRANSMIT_TO_INBOX_NOT_LISTENING] s=-1 & !listen1 &  ((D1H=
       D1nextBroadcast)  | ( (D1H=mod(D1nextBroadcast+D1refractoryPeriod,
       D1cycleLength)) & D1sameCount<sameThreshold)) -> (s'=5) & (
       D1sameCount'=0) ;
21
22 [D1_COUNTING] s=-1 & !listen1 &  !((D1H=D1nextBroadcast)  | ( (D1H=mod(
       D1nextBroadcast+D1refractoryPeriod,D1cycleLength)) & D1sameCount<
       sameThreshold)) -> (s'=7)  ;
23
24 /*Phase 4 */
25 [D1_TRANSMIT_TO_INBOX] s=5  -> (s'=7);
26
27 /*Phase 5 */
28 [D1_CHECK_CYCLE_LENGTH] (s=7) & D1H=D1cycleLength -> (s'=-1) & (D1H'=0) &
       (D1sameCount'=0);
29
30 [D1_NO_CHECK_CYCLE_LENGTH] (s=7) & D1H!=D1cycleLength & D1H<MAXH-1-> (s
       '=-1) &(D1H'=D1H+1);
31 endmodule
```

listening mode based on the Inbox gateway. The if-then block in lines 8-11 is mapped to PRISM commands in Listing 1 lines 6-7.

The if-then-else block of Algorithm 1 in lines 12-22 is translated into PRISM commands in lines 6-7 of Listing 1. The corresponding PLDs are checked on If-Block aligns with PRISM commands in line 10. An additional guard (`D1sameCount<2`) is included to reinforce the PRISM commands to increment `D1sameCount` (See PRISM documentation). In cases where the current `Pld` exceeds the heard `Pld`, the drone's `Pld` is updated using the

PRISM command in line 12. If neither of the guards in the previous conditions are met as in Algorithm 1 lines 19-21, the PLD and local clock are transmitted to the gateway `InBox`.

If the drone is not in listening mode, i.e., broadcasting mode as shown in Algorithm 1 lines 23-28, it is translated into two PRISM commands in Listing 1 lines 17-19. The first command in line 17 checks, if the drone is not in listening mode, where the current drone clock corresponds to the next broadcast or the cycle length, has not yet been reached, then the drone samecount is reset. When this condition is met, the command in line 22 is executed to transmit the current message features. However, if the condition in line 19 is not satisfied (the else block is not specified in the algorithm), then the clock is reset or incremented according to Algorithm 1 lines 29-34.

The PRSIM commands in the Listing 1 (lines 25-27) correspond to the translation of Algorithm 1 (lines 29-34). The first command handles cases where the local clock equals the drone's cycle length. In this scenario, both the current clock `H` and `samecount` are reset. The second command corresponds to the incrementing of the local clock.

Algorithm 2 implements a stack-based approach: receiving drone clocks and pushing Plds onto a stack (lines 9-13). Once the stack reaches its maximum length, the average drone clock and PLD are broadcasted to other drones. However, PRISM language doesn't support complex data structures due to limitations in its exploration engine. Therefore, the algorithm's functionality is mapped to a set of commands in Listing 2.

The first PRISM command (line 5) collects the first drone's local clock and PLD, and activates listening mode (by setting the listen variable to true). The second command (line 6) receives the second drone's data and also activates listening mode. Command in ine 9 calculates the average Pld and local clock using PRISM's built-in "`ceil`" function. Once the calculation is complete, line 11 synchronizes with the first drone (corresponding to line 7 in Listing 2). Similarly, line 13 synchronizes with the second drone.

Listing 2: PRISM Code for InBox Gateway Module

```
1  module D1
2  /*Variable definition*/
3  listen1: bool init false;
4  ...
5  [D1_TRANSMIT_TO_INBOX] listen1=false -> (listen1'=true) & (MSG_H1'=D1H) &
       (MSG_PLD1'=D1Pld);
6
7  [D2_TRANSMIT_TO_INBOX] listen2=false -> (listen2'=true) & (MSG_H2'=D2H) &
       (MSG_PLD2'=D2Pld);
8
9  [MID] !cpt & listen1 & listen2  & MSG_H<MAXH & MSG_PLD<height -> (cpt'=
      true) & (MSG_H'=ceil((MSG_H2+MSG_H1)/MAX_DRONES)) & (MSG_PLD'=ceil((
      MSG_PLD2+MSG_PLD1)/MAX_DRONES));
10
11 [D1_STATUS_LISTENING]  cpt & listen1 ->  (listen1'=false)  ;
12
13 [D2_STATUS_LISTENING]  cpt & listen2 ->  (listen2'=false);
14 endmodule
```

6.3. Property modeling and verification using PRISM

This use case focuses on key properties that ensure the quality of the payload delivered at transmission time H (denoted as instant D1H for the first drone and D2H for the second drone).

Property1: Once the drones transmit their clock ticks to the gateway, it is expected that the internal clock ticks of the drones will synchronize.

Property1

$$\texttt{P} =?[\ \texttt{G}(\ !(\texttt{D1H} == \texttt{D2H}) \implies \texttt{F}\ (\texttt{D1H} == \texttt{D2H})\ )] \tag{1}$$

Property 1 relies on the concept of liveness in formal verification. Liveness guarantees that a desirable state will eventually occur, even if it might take a while. In this case, the property ensures that desynchronization eventually leads to synchronization. The property uses the drone variables `D1H` and `D2H`. `D1H` represents the local clock of the first drone, and `D2H` represents the local clock of the second drone. Both drones' local clocks are initialized to different values. The probabilistic evaluation shows a certainty `100%` of the gateway's ability to calculate the average correct value.

As detailed in section 2.1, monitoring the drones' local clock updates is essential in the presence of environmental noise. We can model this clock drift in the drone's local clock computations on line 30 of the Listing 1. As in [10], the noise can be represented as a probabilistic PRISM command in Listing 3. Both modeled drone modules `D1` and `D2` are characterized by probabilistic commands, where the drift is modeled by a clock update with two units, and a correct update is modeled by one unit. The probabilistic variables are defined in Listing 3 (lines 2-3) as `double` types: `d1proba` and `d2proba`. This allows users to set their values before PRISM model construction, making the model parameterizable.

Listing 3: PRISM Code for Clock Drift Observation

```
1  /* Probabilistic variable definition
2  const double d1proba;
3  const double d2proba;
4  ...
5  module D1
6  ...
7  [D1_NO_CHECK_CYCLE_LENGTH] (s=7) & D1H!=D1cycleLength & D1H<MAXH-1-> (1-
       d1proba):(s'=-1) &(D1H'=D1H+1)+d1proba:(s'=-1) &(D1H'=D1H+2);
8  [driftD1] s=8 -> (s'=-1);
9  endmodule
10
11 module D2
12 ...
13 [D2_NO_CHECK_CYCLE_LENGTH] (s=7) & D2H!=D1cycleLength & D2H<MAXH-1-> (1-
       d2proba):(s'=-1) &(D2H'=D2H+1)+d2proba:(s'=-1) &(D2H'=D2H+2);
14 [driftD2] s=8 -> (s'=-1);
15 endmodule
16 rewards "Desynchronized" \\
17         // Reward for being in the Desynchronized state (driftD1 or
              driftD2)\\
18         [driftD1] true :  1;\\
19         [driftD2] true :  1;\\
20 endrewards
```

Without synchronization in the referenced PRISM model "`M2`" in [38], clock drift can lead to inaccurate cargo lifting values. In this version, the gateway will not perform synchronization. For example, let's consider verifying

the model against the following property :

> ***Property2***: What are the cumulative impacts on lifting operations when multiple clocks are unsynchronized and exhibit varying degrees of drift?

Property2

$$\mathrm{R}\{\text{"Desynchronized"}\} =?[\, \mathrm{C}^{\leq \mathrm{T}}\,] \tag{2}$$

This property uses a reward operator to quantify the cost of drifting commands. Reward properties associate a cost with model paths. This reward is accumulated unit the property is reached. This reward is synchronized with the drifting commands. The model denoted as `M2`, can be found in [38]. The model checking results are shown in Figure 6. Both drone models need to be updated to include a specification for drift monitoring (lines 16-20). The reward function (lines 16-20 of Listing 3) captures transitions where drifting occurs and assigns a penalty 1 for each observation.

As simulation time increases, the variation in internal clocks also increases. This high drift is primarily caused by production variations and the IEEE 802.15.4 standards. Temperature, however, has little impact on clock deviation. This suggests that manufacturing processes, potentially due to automation or human intervention, have a larger effect on clock drift compared to temperature.

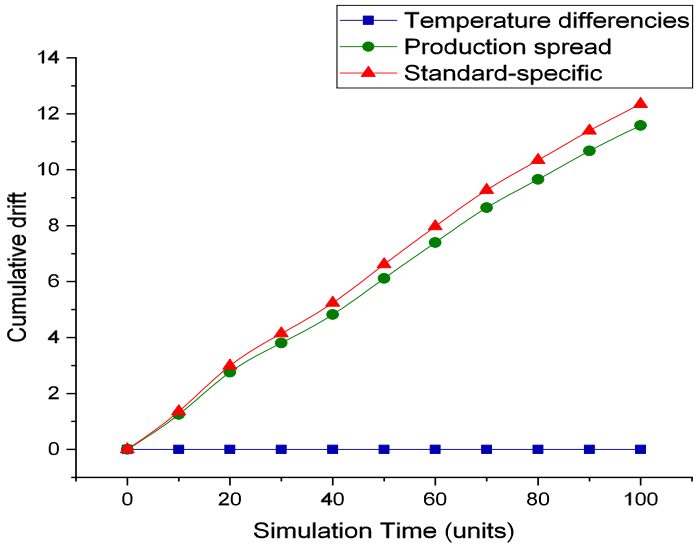

Figure 6: Cummulative Effects od desynchronization.

Considering the updated model "`M3`" incorporating drifting monitoring through probabilistic commands and gateway synchronization, we will verify property 1 while varying `d1proba` and `d2proba` within the range `[0,1]` with increments of `0.1`. The resulting model checking results are portrayed in Figure 7 and Figure 8 during variation of the probabilistic variable `d1proba` and probabilistic variable `d2proba`, respectively.

The graphs in Figure 7 and Figure 8 demonstrate that if the drones' local clocks drift with different values, the property is not satisfied with a probability greater than `0%`. This implies that the property cannot be guaranteed using the global operator alone. To ensure synchronization through the gateway, we define the following property:

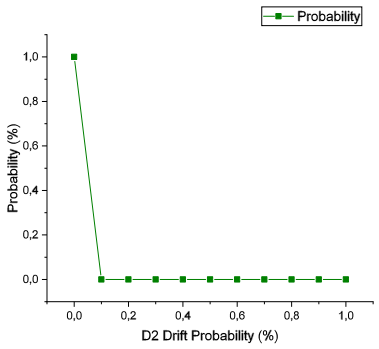

Figure 7: Verification of Property 1 with Drift variation on D2.

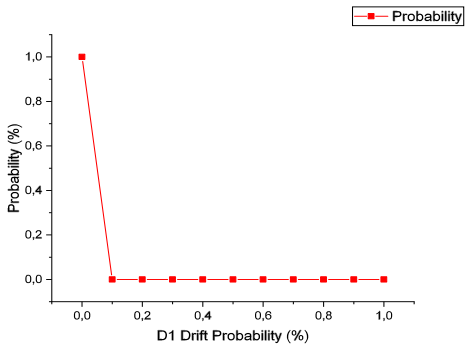

Figure 8: Verification of Property 1 with Drift variation on D1.

Property3: The system eventually reaches a state where both drones (D1 and D2) are synchronized after desynchronization.

Property3

$$\mathtt{P} =?[\ !(\mathtt{D1H} == \mathtt{D2H}) \implies \mathtt{F}\ (\mathtt{D1H} == \mathtt{D2H})\ ] \tag{3}$$

Checking the property 3 while varying `d1proba` and `d2proba` within the range `[0,1]` with increments of `0.1` under synchronization. The resulting model checking results are portrayed in Figure 10 and Figure 9 during variation of the probabilistic variable `d1proba` and probabilistic variable `d2proba`, respectively. The results show that during system execution, in spite, of the local clock time drifts, the gateway performs resynchronization with a probability of `100%`.

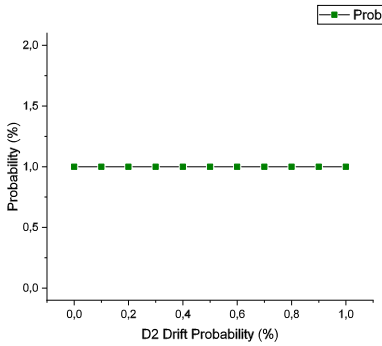

Figure 9: Verification of Property 3 with Drift variation on D2.

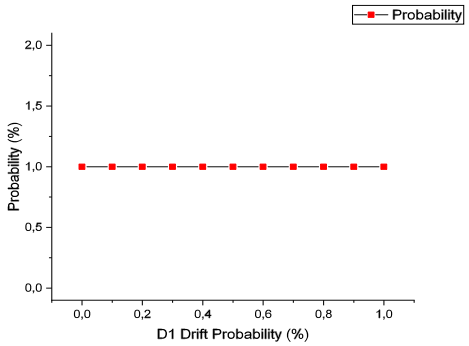

Figure 10: Verification of Property 3 with Drift variation on D1.

Limitations. While the modeled system using PRISM can handle up to 10^9 states, this capacity is still a limitation. Despite setting boundaries on model variables, such as those defined in the first drone module (lines 4-8 of the

Listing 4: PRISM Code for Clock Drift Observation

```
1  ...
2  module D1
3  ...
4  D1cycleLength : [0..MAXH] init 13;
5
6  D1refractoryPeriod : [0..MAXH] init 6;
7
8  D1nextBroadcast : [0..MAXH] init NEXT_BROADCAST;
9  ...
10 endmodule
```

code snippet Listing 4), the model encounters state space explosion. For example, the variable `D1cycleLength` is restricted by the global variable `MAXH`, which is set to 15 units. This approach effectively reduces the state space compared to an open-boundary model, which would exacerbate the challenge faced by many model checkers. Another challenge arises not from model construction but from the property being verified. This property is based on liveness, as defined by Alur [58]. This is a common experience in model checking, where a specific (and potentially complex) algorithm is required to verify the model against the property. One of the solutions that is provided in this paper falls in the scope of model checking by validation.

6.4. Design and validation through simulation using OMNeT++

To address the problems encountered in the previous section, designers can model systems using OMNeT++, which leverages C/C++ but requires knowledge of component-based programming. Algorithms 1 and 2 are implemented in C/C++ code. However, the model architecture components denoted as $\mathcal{ND}$ (See section 5.1), are constructed using OMNeT++ language constructs. The code snippet in Listing 5 portrays the architecture of the use case. The model without and with synchronization using inbox gateway is available on [38] referenced as "`C1`" and "`C2`", respectively.

The OMNeT++ architecture is characterized by three simple modules (i.e. $\mathcal{ND}$ components) (other complex components can be deployed depending on the complex architecture). The modules are characterized by parameters and gates. Parameters are initial variables defined by the OMNeT++ user. In our case `driftRate` refers to the drifting value collected from the hardware and environmental characteristic (see Section 2.1) to make our model flexible and parametrizable. Gates using the keyword `gates` refers to the $\mathcal{ND}$ component port. Data can be exchanged by broadcasting (using `output` port) or receiving through this port (using `input` port) on lines 6-8, lines 14-17, and line 23.

The OMNeT++ system composition is defined by the network configuration in lines 26-43. The $\mathcal{ND}$ components are instantiated as submodules (using the keyword submodules) in lines 29-33. During this instantiation, additional configuration parameters like icons can be specified.

Communication between components is established through connections configured using the keyword `connections`. For example, the `DRONE_TRANSMIT` output gate of the Drone module is connected to the `DRONE0_TRANSMIT` input gate of the `InBox` module. Lines 35-42 define other connections between component gates. Communication can be delayed by specifying a value for the delay parameter. For instance, a 100-millisecond delay can be set for connections.

The graphical representation of the network architecture is shown in Figure 11. On the right side, you'll find various modules available for instantiating additional $\mathcal{ND}$ components. This tool promotes reusability by allowing the definition of reusable modules, providing a flexible configuration option common in most simulation tools. Additionally, designers can enhance the architecture's clarity for non-expert designers by adding icons to different components.

Listing 5: Architecture of Cranes Lifting Orchestration via Drones

```
simple Drone //Drone component
{
    parameters:
        double driftRate;
    gates:
        input DRONE_LISTEN;
        output DRONE_TRANSMIT;
        output CRANE_LIFT;
}

simple InBox // gateway component
{
    gates:
        input DRONE0_TRANSMIT;
        input DRONE1_TRANSMIT;
        output DRONE0_LISTEN;
        output DRONE1_LISTEN;
}

simple Crane //crane component
{
    gates:
        input CRANE_LIFT;
}

network CranesOrchestration //architecture composition
{
  submodules:
      Drone0: Drone { }
      Drone1: Drone { }
      Gateway: InBox {  }
      Crane0: Crane { }
      Crane1: Crane { }
 connections:
  Drone0.DRONE_TRANSMIT --> {  delay = 100ms; } --> Gateway.
      DRONE0_TRANSMIT;
  Drone1.DRONE_TRANSMIT --> {  delay = 100ms; } --> Gateway.
      DRONE1_TRANSMIT;

  Gateway.DRONE0_LISTEN --> {  delay = 100ms; } --> Drone0.DRONE_LISTEN;
  Gateway.DRONE1_LISTEN --> {  delay = 100ms; } --> Drone1.DRONE_LISTEN;

  Drone0.CRANE_LIFT --> {  delay = 100ms;} --> Crane0.CRANE_LIFT;
  Drone1.CRANE_LIFT --> {  delay = 100ms; } --> Crane1.CRANE_LIFT;
}
```

The internal structure of the drone component is defined using a class in C/C++. This class encapsulates the

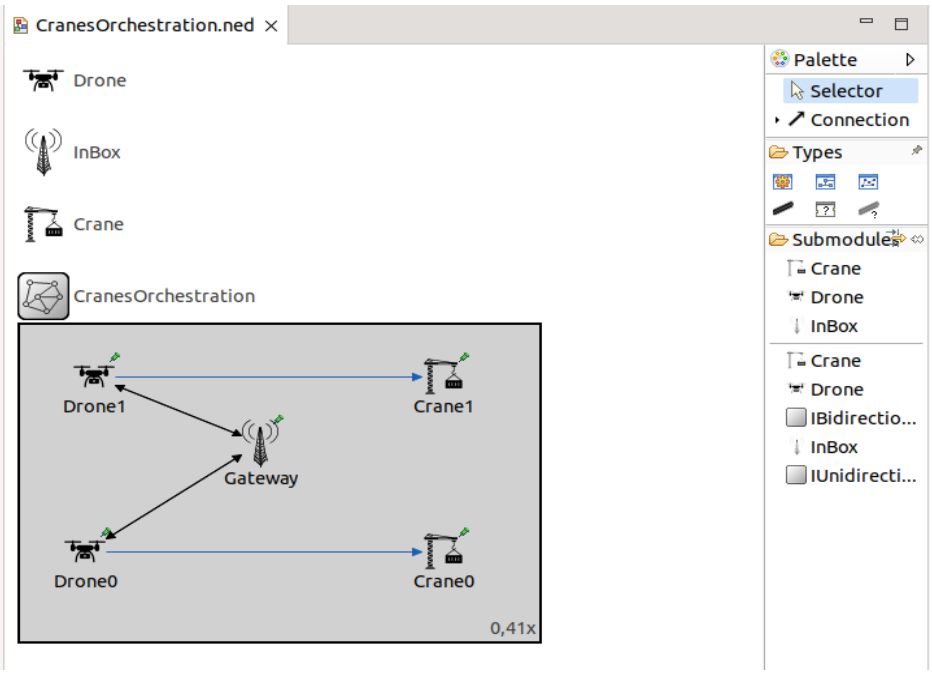

Figure 11: Cranes Lifting Orchestration via a Drone using OMNeT++ Graphical Tool.

Listing 6: Drone Component Functions

```
class Drone : public cSimpleModule
{
public:
    Drone();
private:
    cMessage*  TRANSMIT;
    double driftRate; // drift rate in seconds
    int H;
    int PLD;
    int nextBroadcast;
    int cycleLenght;
    int refractoryPeriod;
    int sameCount;
    const int  sameThreshold=2;
    std::exponential_distribution<double> distribution;
    void transmit(int H, int pld);
protected:
    // The following redefined virtual function holds the algorithm.
    virtual void initialize() override;
    virtual void handleMessage(cMessage *msg) override;
};
```

drone's behavior through a set of attributes (data) and functions (methods). An example code snippet for the

Listing 7: C/C++ Code Internal Component Structure

```
1  void Drone::initialize()
2  {
3      driftRate = par("driftRate");
4  }
5  void Drone::handleMessage(cMessage *msg)
6  {
7  ...
8      if(H==cycleLenght){
9          H=0;
10         sameCount=0;
11     }else{
12         double interval = distribution(generator,std::
               exponential_distribution<double>::param_type(driftRate));
13         if (interval <= driftRate) {
14             H=H+2;
15         } else {
16             H=H+1;
17         }
18     }
19 }
20 void Drone::transmit(int H, int pld)
21 {
22     ...
23     TRANSMIT = new cMessage(result.c_str());
24     send(TRANSMIT, "DRONE_TRANSMIT"); // send out the message
25   }
```

drone component is provided in the Listing 6. The parameters referring to the drift `driftRate` are defined as a double value. The `driftRate` is assigned its initial value during simulation setup using the par function within the `initialize` function (lines 1-4) of the Listing 7.

The `handleMessage` function (lines 5-19) in Listing 7 is responsible for processing incoming messages. For example, it can collect complex messages for internal computations based on previously defined algorithms. However, drift is handled differently in our implementation (line 12). We use the `distribution` function to calculate an interval at which the simulation generates a drift value based on the `driftRate` parameter. If the interval is greater than the `driftRate`, the local clock is incremented by 2 units. Otherwise, it is incremented by 1 unit. It is important to note that OMNeT++ offers specific libraries for managing drift. However, these libraries might limit designer control over the implementation.

The `transmit` function handles the transmission of messages in `string` format, adhering to the OMNeT++ message structure declared on line 6 of Listing 6. It achieves this by invoking a predefined function, `send`, which is responsible for communicating the message through the designated port, `DRONE_TRANSMIT`. The designers have to configure the structure of the string message with the relevant data during the processing.

Validation is achieved through simulation within the OMNeT++ environment. The simulator depicts packets originating from drones and their transmission to the gateway clock synchronization. Finally, the system instructs cranes to initiate lifting operations. The source code is publicly available, enabling academic and industry readers to download and interact with the simulator. Users can manipulate the animation speed by accelerating or decelerating it for a customized experience. However, this paper focused on collecting specific values from the gateway, including the input drone clock values and the generated clock values used for synchronization. Com-

munication lag is a critical factor in synchronization, the clock H and payload Pld messages are sent first to the gateway before to the drones. In our current work, estimating the non-exploitable time during model checking is performed within the simulation scenario. This estimation focuses on the number of messages sent by drones and collected by the gateway. Using the simulator, we assess how synchronization impacts communication delays, resulting in a delay range of [0.3; 21.3] seconds. The developed OMNeT++ code incorporates the computation of communication lag.

The generated CSV file containing a collection of monitored drones' internal clocks is available online (see [38]). Users can access and consult this data for further analysis. The file is produced by custom C/C++ code that appends data entries to the CSV file. Each entry follows a specific header structure, including simulation time, the first drone's clock value, the second drone's clock value, and the synchronized value. Figure 12 depicts a snapshot of the different internal clock values (lines blue and green). By analyzing this figure, we can observe that the synchronization process (red line) occurs periodically, coinciding with each captured instance of desynchronization. This validation proves the accuracy of the algorithm in running synchronization.

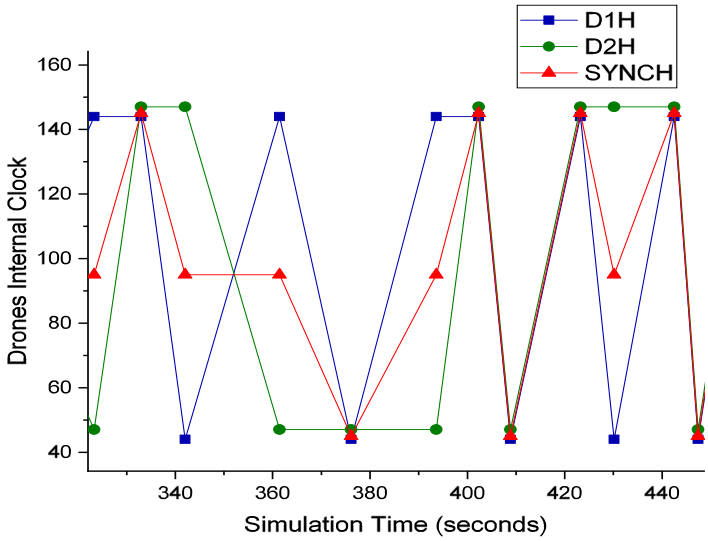

Figure 12: Plotting Internal Clock Synchronization for Model under Temperature Variations.

To assess the validation accuracy, we employed a PDT. First, we trained the decision tree on the previously generated CSV file. This tree learns relationships between features (e.g., D1H and D2H) and synchronized drone clock constraint outcomes in the form of classes. For example, when considering the clock drift caused by temperature variations as discussed in [10] (see Figure 13 for the decision tree visualization), the tree would classify data points based on clocks values and predict the corresponding synchronized outcome. The decision tree has three synchronization classes, labeled from 0 to 2. The root node splits data based on a constraint on the feature `D2H`. The second-level nodes further refine the classification using constraints on `D1H` values. Finally, the leaf nodes assign data points to the corresponding synchronization classes. The relevant Python code can be found in [38] under the model reference **P2**.

6.5. From decision trees to probabilistic commands

The decision tree generated in Figure 13 is translated into a set of PRISM commands shown in Listing 8. The PRISM model uses three constant global variables: `NOTDEFINED`, `D1H`, and `D2H`. The first variable, initialized to -1 in line 4, represents an undefined class state. The other variables (D1H and D2H), defined in lines 5-6, are

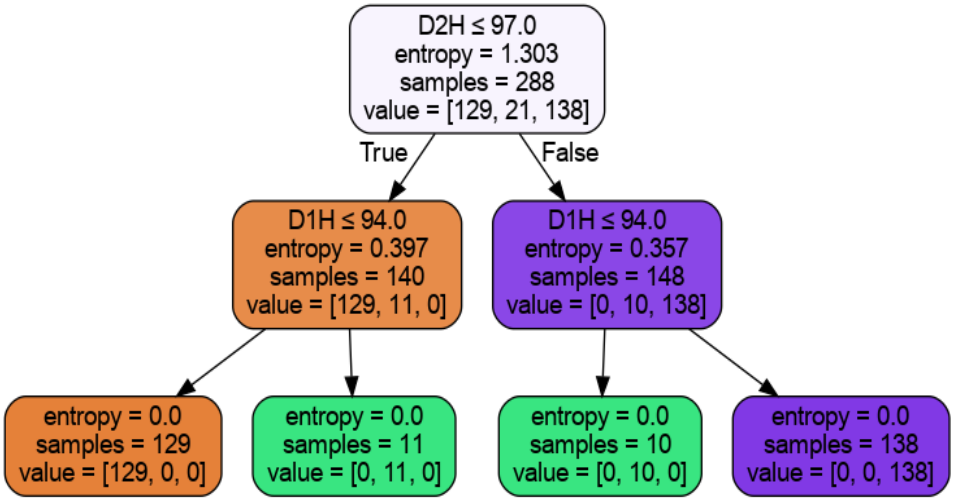

Figure 13: Generated Decision Tree from CSV File .

initialized by the user during model checking. The generated classes are identified by the state variable C (line 12). This variable can take values from -1 to 3. Since the decision tree generates three classes, class 0 corresponds to a value of 0, and class 2 corresponds to a value of 2. By default, the variable is initialized to NOTDEFINED (-1).

The first rule (line 15) defines a constraint on `D1H` and `D2H`. If `D2H` is less than 97 time units and `D1H` is less than 94 time units, then the state variable CL is assigned class 0 with a probability of 0.007. Otherwise, CL is assigned the `NOTDEFINED` class with a probability of `0.99`. This constraint pattern is repeated in subsequent commands (lines 17-25).

Now consider the definition of the stochastic model generated by mapping the decision trees to PRISM commands. The properties expressed here will differ from those previously discussed. The following property is expressed in natural language as:

> ***Property4***: After the drones transmit their clock ticks to the gateway, the generated synchronization class is expected to fall within the interval [0, 2].

Property4

$$\mathtt{P} =?[\, \mathtt{G}(\, !(\mathtt{D1H} == \mathtt{D2H}) \implies \mathtt{F}\,(\mathtt{CL} \geq 0 \& \mathtt{CL} \leq 2)\,)] \tag{4}$$

Verifying Property 4 shows a `100%` probability that the synchronization will be achieved based on drifting categories in section 2.1: Temperature Variations, Standard-Specific Drift, and Production Spread, starting from an initial state with unsynchronized internal clocks. Such a result confirms the efficiency of the algorithm regarding the different parameter variations.

6.6. Computational aspects of models scalability

The PRISM models referenced as "`M1`" to "`M6`" in [38] suffer from the problem of state space explosion, as discussed previously in the context of model checking limitations. This issue stems from the liveness properties and the way they are verified by the supported engines. Our solution is to model the system in OMNeT++ as "`C2`" in [38]. We then generate the corresponding dataset to train decision tree rules, which are then used to build a more tractable model. In this context, we present a model architecture that scales from two drones to six drones. We then compare the time it takes to construct and verify these models regarding property 4.

Listing 8: Mapping to Probabilistic Commands for Model under Temperature Variations

```
dtmc  // This line declares the model as a Markov Decision Process (MDP)

// Constant definitions
const int NOTDEFINED = -1; // Defines a constant value for "not defined"
const double D1H;       // Declares a constant double variable named D1H
const double D2H;       // Declares a constant double variable named D2H

// Module definition
module GeneratedDecisiontree  // Defines a module named
    GeneratedDecisiontree

  // State variable
  CL : [NOTDEFINED..3] init NOTDEFINED;  // Declares a state variable
      named C with a range of [NOTDEFINED, 3] and initial value 0 to
      identify a set of classes

  // PRISM rules
  [rule1] D2H <= 97.0 & D1H <= 94.0 ->  (0.007751937984496124) : (CL' =
      0) + (0.9922480620155039) : (CL' = NOTDEFINED);  // Probability
      (0.9922...) for setting C' to NOTDEFINED

  [rule2] D2H <= 97.0 & D1H > 94.0 ->
    (0.09090909090909091) : (CL' = 1) +
    (0.9090909090909091) : (CL' = NOTDEFINED);

  [rule3] D2H > 97.0 & D1H <= 94.0 ->
    (0.1) : (CL' = 1) +
    (0.9) : (CL' = NOTDEFINED);

  [rule4] D2H > 97.0 & D1H > 94.0 ->
    (0.007246376811594203) : (CL' = 2) +
    (0.9927536231884058) : (CL' = NOTDEFINED);

endmodule  // End of module definition
```

This comparative analysis will provide insights into the limitations of model checking for validation, particularly regarding scalability.

We constructed a model based on the decision trees generated from the data. This model was then replicated for three clusters. Each cluster represents the structure of two drones and three cranes. We verified the model against Property 4. Finally, we measured the time to construct the model and the time to perform verification, considering both the computed states and transitions. In Table 1 we portray the evaluation of the developed model "M6" and the scaled models "M7", "M8", and "M9" in [38]. These models exhibit different characteristics regarding the number of states, transitions, model construction time, and model verification time. The reference model, for example, exhibits a high number of states and transitions. This captures the complexity of the algorithms' structure and the use of variable numbers. However, models "M7", "M8", and "M9" demonstrate fewer states and transitions, making them more suitable for timely verification. Verifying the reference model on two clusters took a very long time without producing any results, despite the computational power of the execution machine.

Model ID	NB Cluster	States	Transitions	Model Construction	Model Checking
M3	Reference Model	149533	387244	2.245 seconds	1.643 seconds
M7	1	2	4	0.001 seconds	0.003 seconds
M8	2	4	12	0.004 seconds	0.039 seconds
M9	3	8	32	0.031 seconds	0.002 seconds

Table 1: Computational Aspects of Models Scalability.

According to the reference paper [59], the verification cost is defined as 1 - ($\frac{Tv(\mathscr{P}_2)}{Tv(\mathscr{P}_1)}$) and abstraction efficiency is defined as 1 - ($\frac{Tc(\mathscr{P}_2)}{Tc(\mathscr{P}_1)}$). We refer by $\mathscr{P}_1)$ to the reference model ("M3") while we refer by $\mathscr{P}_2)$ to models in "M7", "M8", and "M9". The function Tv computes the model verification time and the function Tc computes the model construction time. Computation times are collected from the PRISM model checker verification engine.

In this case, the model construction efficiency is very high at `0.999`. This indicates that constructing the abstract model takes significantly less time compared to the reference model. Similarly, the verification cost of `0.998` suggests that verifying the abstract model is also faster. However, the reference model with multiple clusters cannot be verified and doesn't provide any results. To address this limitation, we introduce the concept of a limit function. This concept allows the verification and construction times to approach 1, indicating very efficient processing in model construction and verification.

6.7. Threats to validity

In the following section, we identify and highlight the potential threats associated with our experimentation that may arise during the infrastructure deployment.

Since the dataset was generated by an OMNeT++ simulator rather than a real-world infrastructure, the collection process might not fully capture parameters relevant to physical deployment. This raises the possibility of outliers and potentially meaningless data, requiring careful handling during analysis.

The dataset was collected from the OMNeT++ gateway component, where we specifically captured data at that level. Consequently, experiments with similar models that include gateways are likely. However, the chosen component for data collection might limit the experiment's monitoring to gateways and not other network components.

7. Discussion

In this work, we demonstrate the application of formal methods, specifically model checking with PRISM models, for verifying clock synchronization in a drone communication system. In Section 5, we formalize semantic rules for managing clock deviation, which can be implemented in various formalisms like PRISM, UPPAAL [60], and BIP [61]. We implemented the protocol properties using PCTL, ensuring messages are transmitted from drones to the gateway and received even under conditions of clock deviation.

The verification process becomes hard due to the number of variables and their configuration, as portrayed in section 6. This is attributed to algorithms explicitly designed for verifying a kind of liveness properties [58]. In scenarios like this, abstraction can prove advantageous in reducing computational overhead, yet it may result in a loss of model meaning. To overcome the limitations of model-checking engines, we employ OMNet++ to model the system accurately and subsequently construct PRISM models that are learned using PDT. These models incorporate statistical observations from the streaming simulation dataset, closely resembling the initial PRISM model regarding verification results.

The experimental results indicate thanks to the learning process, model checking can effectively verify even large-scale early synchronization detection. Using simulation (OMNeT++) and verification during the initial wireless engineering phases can identify potential side effects much earlier in the design process. This not only allows for timely detection but also leads to a considerable reduction in project costs and duration. Also, Due to the automation and absence of manual abstractions, this transformation could be integrated into robust model-based design frameworks such as Eclipse Papyrus [62]. The artifact's compatibility with such frameworks allows for effortless incorporation, enhancing the overall efficiency and effectiveness of the modeling process.

The OMNeT++ model for the orchestration of lifting tasks is designed with flexibility in mind. Unlike traditional approaches, designers are not required to build the entire model from scratch. Instead, they can leverage pre-designed in OMNeT++ components editor like cranes, drones, and gateways, allowing them to focus on the specific lifting orchestration logic. This is made possible by the robust C/C++ programming environment, which ensures efficient message marshaling and unmarshalling between the physical components.

As the literature discusses, our proposed operational semantics rules provide a unified approach to capturing observed resynchronization. This reusability translates to their applicability across diverse communication protocols susceptible to clock drift. Indeed, our study demonstrates their successful generalization to two distinct formalisms: OMNeT++ and PRISM. OMNeT++ operates at the low level of communication protocols using C/C++ constructs. In contrast, PRISM is dedicated to modeling and analysis at a high level.

The paper proposes a verification approach applicable to various synchronization algorithms. This approach can assist engineers in certifying their protocols and ensuring their correctness. Scalability, a major challenge in modeling communication protocols as networks grow, was also investigated. To address scalability, we employed learning algorithms to more effectively capture communication traces and build formal models. This design allows collaboration between users from diverse backgrounds due to the flexibility of approaching the algorithm from different perspectives. In contrast, automation offers the advantage of requiring only the designer's input to gain modeling insights.

8. Conclusion

This paper investigates the impact of clock deviation within a component-port-connector architecture model. The OMNeT++ framework is used to precisely define the behavior of system components and connectors. The resulting model is then translated into the PRISM language, utilizing Probabilistic Decision Tree rules derived from the OMNeT++ simulation chart. Requirements related to synchronized clocks, expressed in Probabilistic Computation Tree Logic (PCTL), are verified against two models constructed in PRISM PA: a reference model and one generated by learning decision trees. A use case is then implemented in both formalisms to demonstrate the effectiveness of the proposed approach for validation.

The models incorporate parameters inspired by real-world phenomena observed in the IEEE 802.15.4 specification, including product manufacturing variations and operating temperature changes. This parameterized construction allows practitioners to perform customized validation and verification. Our results demonstrate that building stochastic models from simulated models leads to more efficient models with fewer states and transitions compared to PRISM models.

In future research endeavors, we plan to formally capture the bi-simulation between the constructed models and the learned abstract model. This task presents a significant challenge due to the potential differences in the state spaces of the input and abstract models. Additionally, we aim to develop an automated approach for feature selection during the learning process. The current approach, while promising, requires human intervention in choosing the features used for model construction. Finally, to enhance realism, we plan to incorporate more precise communication delays in future work. We aim to achieve this by utilizing electronic tools for delay measurement or more advanced simulation techniques.

References

[1] C. Baier, J.-P. Katoen, Principles of model checking, The MIT Press, 2008.

[2] A. Nouri, B. L. Mediouni, M. Bozga, J. Combaz, S. Bensalem, A. Legay, Performance Evaluation of Stochastic Real-Time Systems with the SBIP Framework, International Journal of Critical Computer-Based Systems (2018) 1–33.

[3] A. Baouya, S. Ouchani, S. Bensalem, Formal modelling and security analysis of inter-operable systems, in: Advances and Trends in Artificial Intelligence. Theory and Practices in Artificial Intelligence, volume 13343, Springer International Publishing, 2022, pp. 555–567. URL: https://link.springer.com/10.1007/978-3-031-08530-7_47. doi:10.1007/978-3-031-08530-7_47.

[4] Q. Rouland, B. Hamid, J. Jaskolka, Formal specification and verification of reusable communication models for distributed systems architecture, Future Generation Computer Systems 108 (2020) 178–197. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19311926. doi:https://doi.org/10.1016/j.future.2020.02.033.

[5] Q. Rouland, B. Hamid, J. Jaskolka, Reusable formal models for threat specification, detection, and treatment, in: S. Ben Sassi, S. Ducasse, H. Mili (Eds.), Reuse in Emerging Software Engineering Practices, Springer International Publishing, Cham, 2020, pp. 52–68.