Verification and Simulation: Detection and Mitigation of Clock Deviation

Abdelhakim Baouyaa,∗, Brahim Hamida, Otmane Ait Mohamedb, Saddek Bensalemc

*a University of Toulouse, IRIT, France*

*b Concordia University, Montréal, Canada*

*c Université Grenoble Alpes, VERIMAG, Grenoble, France*

## Abstract

Modern cyber-physical systems, relying on diverse computation logics, communication protocols, and technolo- gies, are susceptible to environmental phenomena that can significantly impact system behavior. Ensuring re- silience in these systems necessitates considering these environmental factors during the high-level design stages to enable accurate forecasting. This paper presents an approach that models and analyzes clock deviation’s ef- fects within physical and environmental conditions. We employ the OMNeT++ simulation framework to define the behavior of system components and connectors precisely. 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 devia- tions influence the orchestration of cranes in a drone system, serving as a representative application scenario. The research examines clock deviations from multiple sources, including the IEEE 802.15.4 specification, prod- uct 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:* Software architecture, Clock drifts, Decision Trees, Formal methods.

## Contents

## [Introduction](#_bookmark0) 3

* 1. [Contribution overview](#_bookmark1) 3
  2. [Outline](#_bookmark2) 4

## [Approach](#_bookmark3) 4

## [Background](#_bookmark5) 5

* 1. [A comprehensive analytical background on clock drifts](#_bookmark6) 5
     1. [Clock drift caused by temperature variations](#_bookmark7) 5
     2. [Standard-specific clock drift](#_bookmark8) 5
     3. [Clock drift caused by production spread](#_bookmark9) 5
  2. [Modeling formalism](#_bookmark10) 5
     1. [Probabilistic automata](#_bookmark11) 6
     2. [The PRISM language](#_bookmark13) 6

*∗*Corresponding author at :University of Toulouse, IRIT, France

*Email addresses:* [abdelhakim.baouya@irit.fr,](mailto:abdelhakim.baouya@irit.fr) [abaouya@acm.org](mailto:abaouya@acm.org) (Abdelhakim Baouya), [brahim.hamid@irit.fr](mailto:brahim.hamid@irit.fr) (Brahim Hamid),

[otmane.aitmohamed@concordia.ca](mailto:otmane.aitmohamed@concordia.ca) (Otmane Ait Mohamed), [saddek.bensalem@univ-grenoble-alpes.fr](mailto:saddek.bensalem@univ-grenoble-alpes.fr) (Saddek Bensalem)

*Preprint submitted to Elsevier April 20, 2024*

1. [Formalization](#_bookmark14) 6
   1. [Probabilistic Decision Trees](#_bookmark16) 7
   2. [OMNeT++](#_bookmark18) 8
2. [Communication in Challenging Physical and Environmental Conditions](#_bookmark23) 9
   1. [The Firefly-Gossip Protocol](#_bookmark24) 9
   2. [Clock Deviation](#_bookmark27) 11
   3. [The Synchronization Gateway](#_bookmark28) 12
3. [Experiments](#_bookmark32) 13
   1. [Case study](#_bookmark33) 13
   2. [Drones and InBox behavior modeling](#_bookmark35) 14
   3. [Property modeling and verification using PRISM](#_bookmark38) 16
   4. [Design and validation through simulation using OMNeT++](#_bookmark46) 20
   5. [From decision trees to probabilistic commands](#_bookmark51) 23
   6. [Computational aspects of models scalability](#_bookmark56) 26
4. [Discussion and threats to validity](#_bookmark58) 27
   1. [Beyond model checking](#_bookmark59) 27
   2. [Threats to validity](#_bookmark60) 27
5. [Related work](#_bookmark61) 27
6. [Conclusion](#_bookmark62) 28

## Introduction

In contemporary software-intensive systems, the significance of environmental concerns has grown substan- tially. Early identification of these concerns is paramount, particularly during the architectural design phase when their semantics are most precisely defined [[1](#_bookmark63), [2](#_bookmark64), [3](#_bookmark65)]. Architecture modeling and analysis offer valuable ways to detect data inaccuracies during the initial stages of the system development process [[4](#_bookmark66)]. Moreover, software component technology has emerged as a prominent research and development area within software engineering. Its notable success in the market and its pivotal role in facilitating reuse have contributed to its prominence [[5](#_bookmark67)].

Clock drift [[6](#_bookmark68)] 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 manufactur- ing imperfections and temperature fluctuations [[7](#_bookmark69), [8](#_bookmark70), [9](#_bookmark71)] can cause degradation in a clock’s accuracy over time, leading it to drift (or deviate) away from the reference time. To mitigate these effects, systems often employ synchronization protocols like Network Time Protocol (NTP) [[10](#_bookmark72)] for computer networks that regularly adjust their clocks with more accurate reference sources [[11](#_bookmark73)].

Formal methods are extensively used to mathematically verify properties pertaining to the structure and

behavior of physical systems. However, these methods often face limitations in terms of scalability. In response, several research works have explored abstraction techniques, such as partial-order reduction [[12](#_bookmark74)], [[13](#_bookmark75)], assume- guarantee reasoning [[14](#_bookmark76), [15](#_bookmark77)], SAT-based model checking [[16](#_bookmark78), [17](#_bookmark79)], and symmetry reduction [[18](#_bookmark80)]. While these approaches may sacrifice some level of 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 [[19](#_bookmark81), [20](#_bookmark82)]. This approach is acceptable by industrials [[21](#_bookmark83), [22](#_bookmark84)], which can enhance confidence in the original model’s specifications [[23](#_bookmark85)].

Our work leverages formal methods to enable precise specification and analysis of communicating cyber sys- tems, 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++ and PRISM [[24](#_bookmark86)]. This allows us to specify the system and detect and reason about

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 ar- chitecture. This provides a concrete implementation of the system 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) [[25](#_bookmark87)] to express and verify desired properties re- garding 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.

* 1. *Contribution overview*

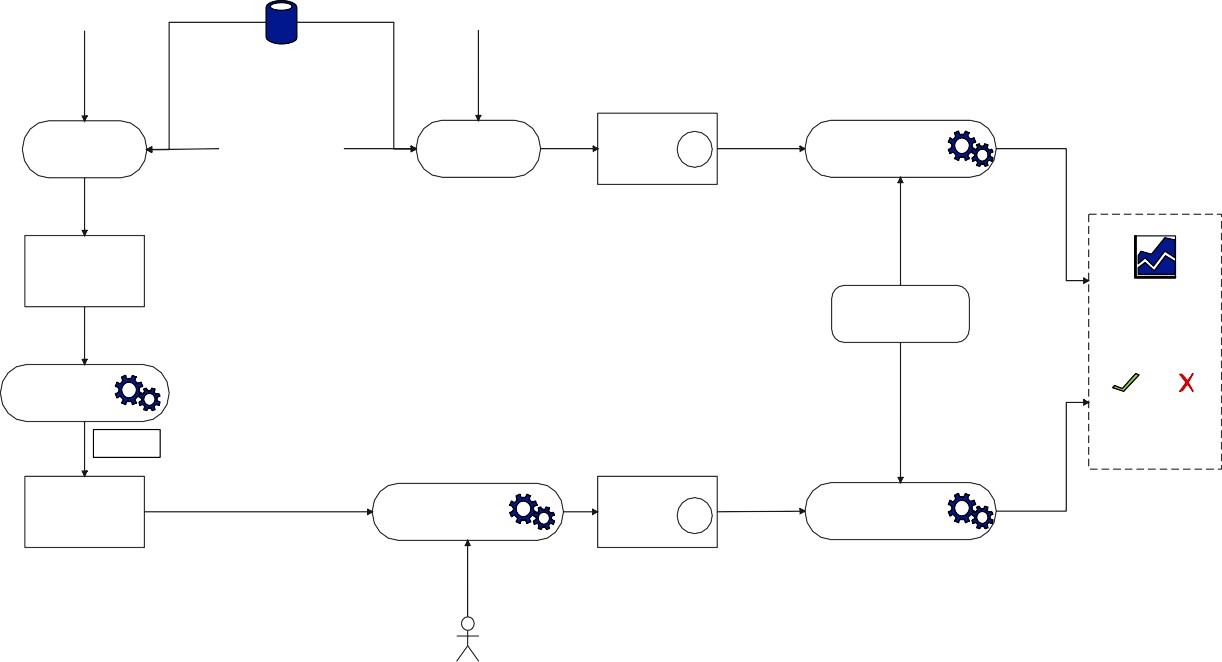
To our knowledge, the issue of clock deviation has not been previously addressed from different perspectives as in the context of component-port-connector formalism. Furthermore, the aspect of *scalability* has not been adequately addressed in existing research from a formal methods perspective. In light of these gaps, our proposed work aims to bridge this divide by developing a novel approach that enables an accurate and early assessment of software architecture with respect to clock deviation. The sources of clock deviation are determined based on the research conducted by [[7](#_bookmark69)] and [[9](#_bookmark71)]. The first source is attributed to the IEEE 802.15.4 standard, while the second source is associated with production variation. Additionally, differences in operating temperature contribute to the third source of clock deviation. By integrating probabilistic model-checking techniques into our methodology, we can conduct more comprehensive analyses to forecast the behavior of communication styles compared to previous studies. Therefore, our contribution can be summarized as follows threefold:

1. The formalization of modeling idiosyncrasies aims to provide practitioners with mathematical insights, facilitating the widespread application of proposed techniques using formal methods.
2. Detection of the presence of clock deviation using PCTL queries on the PA model in the case of synchronized and unsynchronized clocks,
3. We consider the crucial aspect of scalability in our approach, ensuring that it can be applied effectively to larger and more complex systems without compromising accuracy or efficiency.
4. Evaluation of the two previous contributions in the context of crane orchestration via drones using Proba- bilistic model-checking and validation through OMNet++.
   1. *Outline*

The paper is structured as follows: Section [2](#_bookmark3) provides the workflow of the approach for investigating clock drift. Section [3](#_bookmark5) provides the necessary background material to understand the proposed approach. Section [4](#_bookmark14) presents a formalization related to OMNeT++ and Probabilistic Decision Trees. Section [5](#_bookmark23) explains how mani- festations of clock deviation are modeled within CPC models. To demonstrate the effectiveness of our approach, Section [6](#_bookmark32) presents a detailed use case involving crane orchestration through a drone system, where various analyses are performed. Section [7](#_bookmark58) discusses relevant outcomes related to the state space explosion and threats to validity. Section [8](#_bookmark61) provides an overview and analysis of related work in this field. Finally, in Section [9](#_bookmark62), we conclude the paper by summarizing key findings and outlining potential directions for future research endeavors.

## Approach

We rely on the workflow depicted in Figure [1](#_bookmark4) to address the question raised in our contribution.



**Expert in OMNeT++ Modeling**

**Expert in Formal Modeling**

**Parameters of Devices Clock Drift**

**Modeling in OMNeT++**

**Specification in Natural Language**

**Modeling in PRISM**

**PRISM** P**1**

**Model**

**Probabilistic Model Checking**

**OMNeT++**

**Model**

**Synchronization Properties in PCTL**

**Quantitative results**

**Simulation**

**True**

**False**

***Generate***

**Qualitative results**

**Simulation Traces**

**Learning PDTs in**

**PRISM**

**PRISM**

**Model** P**2**

**Probabilistic Model Checking**

**Expert in Formal Modeling**

Figure 1: Proposed Flow for Verification and Validation.

*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 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 simula- tion generates a set of traces to evaluate the timing and synchronization behavior of the system.

*Learning.* Probabilistic Decision Tree (PDT) is generated by learning from the traces collected from the OM- NeT++ simulator. The learned PDT is then transformed into a new PRISM model 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 2 .

## Background

In this section, we will explore the comprehensive analytical background of clock drifts, also referred to as clock deviation (both terms are correct), and the formalism that captures the underlying semantics associated with clock drifts.

* 1. *A comprehensive analytical background on 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 [[7](#_bookmark69)] and [[9](#_bookmark71)]. We categorize the different sources of clock variation into three main categories: Temperature Variations, Standard-Specific Drift, and Production Spread.

* + 1. *Clock drift caused by temperature variations*

According to [Webster et al.](#_bookmark69) [[7](#_bookmark69)] [[8](#_bookmark70)], differences in operating temperature can affect drone clock synchroniza- tion. 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 [[26](#_bookmark88)] as [Lenzen et al.](#_bookmark88) 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.

* + 1. *Standard-specific clock drift*

According to the IEEE 802.15.4 standard and the work established by [Elsts et al.](#_bookmark71) [[9](#_bookmark71)] set a revision definition of maximum resynchronization period between wireless devices as it refers to the longest interval between syn- chronizations a network can tolerate while ensuring reliable communication. Three factors are identified in this standards, the guard time *tg* (This is the time a receiver listens for a packet beyond the expected arrival time to account for clock drift), the preamble transmission time *tp* (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 ( *∆t* ) to guard time and clock drift: *∆t* (*tg* 2 *tp*)*/*2 *δ* . To illustrate these clock variations, a Python code is available in section artifacts that simulates 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 [[27](#_bookmark89)] under the model reference **P1**. The computed average clock drift is approximately 0.44 (parts per million).

≤ − ∗ ∗

* + 1. *Clock drift caused by production spread*

Clock drift caused by production spread refers to the slight variations in clock frequency that occur between in- dividual devices despite being manufactured identically. The manufacturer typically specifies the range (bounds) of these deviations. According to [[9](#_bookmark71)], when using oscillators, a tolerable drift in the worst-case scenario can be as high as 0.40 parts per million (ppm).

* 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.

* + 1. *Probabilistic automata*

Probabilistic Automata (PA) [[28](#_bookmark90)] is a modeling formalism that exhibits probabilistic and nondeterministic features. Definition [1](#_bookmark12) 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 *µ*(*si*) = *pi* to the state *si* ∈ *S* .

**Definition 1.** A Probabilistic automata is a tuple  = ⟨*s*0*, S, Σ, AP, L, δ*⟩ :

* + - * *s*0 is an initial state, such that *s*0 ∈ *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* 2*AP* is a labeling function that assigns each state *s S* to a set of atomic propositions taken from the set of atomic propositions ( *AP* ), and

→ ∈

* + - * *δ* : *S Σ Dist*(*S*) is a probabilistic transition function assigning for each *s S* and *α Σ* a probabilistic distribution *µ Dist*(*S*) . The operational semantics rules in Figure [2](#_bookmark15) model all generic probabilistic transitions.

∈

× → ∈ ∈

For PA’s composition, this concept is modeled by the parallel composition [[28](#_bookmark90)]. During synchronization, each

*α*

*α*

PA resolves its probabilistic choice independently [[28](#_bookmark90)]. For transitions, *s*1 −→ *µ*1 and *s*2 −→ *µ*2 that synchronize

in *α* then the composed state ( *s*′1*, s*′2 ) is reached from the state ( *s*1*, s*2 ) with probability ( *µ*1(*s*′1) *µ*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 probabilistic automata (i.e., MDP) and properties expressed in Probabilistic Computation Tree Logic (PCTL) [[25](#_bookmark87)], [[28](#_bookmark90)], [[29](#_bookmark91)]. 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.

* + 1. *The PRISM language*

We consider the probabilistic automata (i.e., Markov Decision Process (MDP)) in PRISM language [[24](#_bookmark86)] as a modeling formalism to capture the semantics of stochastic automata and appropriate to express the non- determinism in wireless sensor networks.

The PRISM model 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 λ*1 : *u*1 + *. . .* + *λn* : *un* or, [*a*]*g u* , which means, for the action “ *a* ” if the guard “ *g* ” is true, then, an update “ *ui* ” is enabled with a probability “ *λi* ”. A guard is a condition that determines whether a command can be executed based on evaluations of variables using logic operators. The update “ *ui* ” is an evaluation of variables *vi* ; evaluated via expressions denoted by “ *θ* ” such that *θ* : *V* → lIJ , where lIJ is a domain of variables such that lIJ = N ∪ {*true, false*} .

→

→



**Definition 2** ( state)**.** A state of is the valuation of modules variables *v*0*, . . . , vn* in the form *s* = *x*0*, . . . , xn, θ* such that *θ* = *g* and *g* is the guard over states values., where *xi* is the valuation of variable *vi* .

  

⟨ ⟩ |

## Formalization

This section presents the formalization of Probabilistic Decision Trees (PDTs). PDTs are crucial for constructing state-based models by leveraging rule sets. We will then formalize the architectural and behavioral language of OMNeT++ to serve as input for the rule learning process.

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

⟨*s*0*, . . . , si, . . . , sn, θ*⟩ −→*λi* ⟨*s*0*, . . . , s*′*i, . . . , sn, θ*′⟩

where *θ*′ = *θ*[*vi* := *effect*(*vi*)]

*si* −→*λi s*′*i* ∧ *θ* |= *g*

*g*:*a*

*a*

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

*g*1 :*a*

*g*2 :*a*

⟨*s*0*, . . . , si, . . . , sj, . . . , sn, θ*⟩ −→*λi*×*λj* ⟨*s*0*, . . . , s*′*i, . . . , s*′*j, . . . , sn, θ*′⟩

where *θ*′ = *θ*[*vi* := *effect*(*vi*)]

*si* −→*λi s*′*i* ∧ *θ* |= *g*1 ∧ *sj* −→*λj s*′*j* ∧ *θ* |= *g*2

*a*

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

*g*1 :*a*!*vi*

*g*2 :*a*?*vj*

⟨*s*0*, . . . , si, . . . , sj, . . . , sn, θ*⟩ −→*λi*×*λj* ⟨*s*0*, . . . , s*′*i, . . . , s*′*j, . . . , sn, θ*′⟩

where *θ*′ = *θ*[*vj* := *vi*]

*si* −→*λi s*′*i* ∧ *θ* |= *g*1 ∧ *sj* −→*λj s*′*j* ∧ *θ* |= *g*2

*a*

Figure 2: PRISM-PA Semantics Rules.

* 1. *Probabilistic Decision Trees*

Probabilistic Decision Trees (PDTs) [[30](#_bookmark92)] 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 Probabilistic Decision Tree is represented as a tuple  = ⟨*I, G, X, Rule*⟩ , 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 *gi* as a constraint on each input feature *I* and *evalv* :

*I* → lIJ to evaluate the features of the PDT. Here, *v*0*, . . . , vn* 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 ∧ *. . .* ∧ *xn* →*λ xn*+1 ∧ *. . .* ∧ *xn*+*j* , where *λ* represents the probability of occurrence of the rule in the dataset and *x*0 |= *g*0 *. . .* ∧ *xn* |= *gn.* We use **[** **]** to identify a set of rules derived from the dataset.

The example in Figure [3](#_bookmark17) shows the structure of the Probabilistic Decision Tree (PDT), where each node represents an input feature *I* = CO2*,* Humidity , except for the leaf node, which represents the output feature *I* = *I* Ventilation . In this specific case, we can interpret the following rule: CO2 >= High humidity >= 80% 1 Ventilation level= 2

→

∪ { } ∧

{ }

The formalism of Probabilistic Decision Trees (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:

*false*

CO2

High

*true*

Ventillation Level-0 Level-1

*false*

Humidity

50 %

*true*

p=0.25 p=0.75

Ventillation Ventillation

Level-0 Level-1 p=0.25 p=0.75

Level-2 p=1.0

Figure 3: Example of a Probabilistic Decision Tree (PDT)

* 1. *OMNeT++*

OMNeT++ [[31](#_bookmark93)] is a versatile C++ simulation library and framework specifically designed for building net- work 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 com- ponent 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* : *ϑ* lIJ . The NED Component is defined as follows:

→

**Definition 4.** A NED component “   ” is a tuple   = ⟨*ϑ, Σ, Bh*⟩ , where:

* *ϑ* = *ϑp* ∪ *ϑm* ∪ *ϑl* is a finite set of component variables
  + *ϑp* is a finite set of parameters associated with the component,
  + *ϑm* is a finite set of messages associated with the component. Also, we consider the *X* as valuation over *ϑm* and *X*′ as its new valuation.
  + *ϑl* 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 re- ceived/sent message.

To gain a deeper understanding of the OMNet++ component behavior, we establish semantics for the state based on component variables as follows:

 

**Definition 5** ( state)**.** A state “ *s* ” is the valuation *θ* of components variables such that:

     

*s* = ⟨*x*0*, . . . , xn, θ*⟩ such that *θ* |= *g* and *g* is the guard over states values.

Each component is capable of establishing communication with other 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 com- ponents. It can be represented as a tuple *Σa, da* , where: *Σa Σ* represents a set of gates that are associated with the interaction *a* , and *da* 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](#_bookmark20).

**Definition 7** (Composition)**.** The composition *γ* ( 1*, . . . , n* ) of *n* components is a network component resulting from composition on interaction *Σa* ⊆ *Σ* in Definition [6](#_bookmark19) represented by [*Internal Update*](#_bookmark21)and [*Send/receive*](#_bookmark22) operational semantics rules:

 

   

*s*0 *'*−→

*τ*

⟨*s*0*, . . . , sn*⟩ −→

*τ*

*s*′0

⟨*s*′0*, . . . , sn*⟩

(*Internal Update*)

where *Xϑ*′ = *X*[*vl* := *bh*(*l*)]

*l*

⟨*s*0*, . . . , s*1*, . . . , sn*⟩ −→ ⟨*s*′0*, . . . , s*′1*, . . . , sn*⟩

*s*0 *'*−−→ *s*′0 ∧ *s*1 *'*−−−→ *s*′1

*a*!*m a*?*m*

*a*

(*Send/receive*)

where *Xϑ*′ = *X*[*vm* := *m*]

*m*

## Communication in Challenging Physical and Environmental Conditions

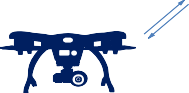
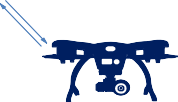
In this section, we will provide a detailed description of the modified FiGo protocol, which is responsible for synchronizing clocks and payload. The internal clock ticks deviation is formally described. Additionally, we will present the algorithm that handles clock computation and synchronization at the edge.

The purpose of establishing such a protocol is to ensure that liveness and safety requirements. In the context of the FiGo protocol, 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*”.

* 1. *The Firefly-Gossip Protocol*

The Firefly-Gossip Protocol (FiGo) algorithm has been specifically implemented for object synchronization, as demonstrated in [[7](#_bookmark69), [8](#_bookmark70), [32](#_bookmark94)]. 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 (Fig- ure [4](#_bookmark25)), 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 Algo- rithm [1](#_bookmark26). Input parameters include cycleLength, refractoryPeriod, the nextBroadcast, and sameThreshold. The refractoryPeriod is a duration during which no further transmissions are made after the initial transmis- sion 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.



**Sensor 4**



**Sensor 1**



**Edge Gateway**

**Sensor 2 Sensor 3**

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**

* 1. *Clock Deviation*

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](#_bookmark26), the clock progress is carried out by the sensor at line 32, 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. In the background section, we introduce 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](#_bookmark39) as follows:

*s*0 *'*−→

*τ*

*λ s*′0

⟨*s*0*, . . . , sn, X*⟩ −→*λ* ⟨*s*′0*, . . . , sn, X*′⟩

*τ*

(*Clock Drift*)

where *XH*′ = *X*[*Hl* := *Hl* + 2]

*l*

*s*0 *'*−→

*τ*

1−*λ s*′0

⟨*s*0*, . . . , sn, X*⟩ −→1−*λ* ⟨*s*′0*, . . . , sn, X*′⟩

*τ*

(*No Clock Drift*)

where *XH*′ = *X*[*Hl* := *Hl* + 1]

*l*

* 1. *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 [[8](#_bookmark70)]. 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](#_bookmark31).

Algorithm [2](#_bookmark31) 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*](#_bookmark29). On the other hand, Rule [*Broadcast*](#_bookmark30)describes how the averaged clock is broadcasted to the sensor nodes. The communication channel selected for data transfer is denoted as “p”.

*s*0 *'*

*p*?⟨*H,Pld*⟩

⟨*s*0*, . . . , sn, X*⟩ −→ ⟨*s*′0*, . . . , sn, X*′⟩

where *X*′ = *X*[*length* := *length* + 1]

*p*

−→

*s*

0

(*Receive*)

**[***Inbox* **]** = *si '*

*p*!⟨*H,Pld*⟩

−→

*s*

*i j...n*

**[**  **]** = *s '*

*j j*

*p*?⟨*H,Pld*⟩

−→

*s*

*j*

⟨*si, sj, . . . , sn, X*⟩ −→ ⟨*s*′*i, s*′*j, . . . , sn, X*′⟩

where *X*′ = *X*[*length* := 0]

*p*

(*Broadcast*)

**Algorithm 2:** InBox implementation at the Edge Gateway

**Data:** max.

**1** *H Init* ; /\* The initial value of drone tick. \*/

←

# **2** *Pld* 1 ; /\* The payload message for crane lifting. \*/

←

**3** *msg empty*() ; /\* The message status. \*/

←

**4** *Length* 0 ; /\* The drones length. \*/

←

**5** *ListOf H stack*() ; /\* The list of clock ticks. \*/

←

**6** *ListOf Pld stack*() ; /\* The list of payload. \*/

←

**7 while** *True* **do**

**8** *msg heard*();

←

**9 if** *(msg.transmit*()*)* **then**

**10** *ListOf H.push*(*msg.H*) ; /\* push the received clock ticks to the stack. \*/

**11** *ListOf Pld.push*(*msg.Pld*) ; /\* push the received payloads to the stack. \*/

**12** *Length Length* + 1 ; /\* increments the stack length. \*/

←

**13 end**

**14 if** *(Length* == *max)* **then**

**15** *H* AVG(*ListOf H*) ;

←

**16** *Pld* AVG(*ListOf Pld*) ;

←

**17** *broadcast*(*H, Pld*) ; /\* broadcast the average value of clock ticks and payloads. \*/

**18** *Length* 0

←

**19** *ListOf H empty*()

←

**20** *ListOf Pld empty*()

←

**21 end**

**22 end**

## Experiments

* 1. *Case study*

*Description*: The use case scenario (Figure [5](#_bookmark34)) [[33](#_bookmark95)] involves the process of lifting a platform using drones. 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.

*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*:

* + - The drone operator initializes communication messaging between drones and the gateway.



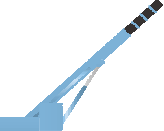
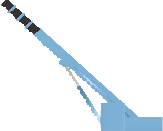
1

* + - At each time unit *H* , the drone sends its internal clock ticks value to the edge and then receives the average



2

values .



**Crane 1 Crane 2**



**Edge Gateway**



**1**

**2**



**1**

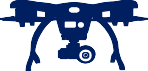
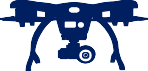
**2**



**3**



**3**



**Sensor 1 Sensor 2**

Figure 5: Cranes Lifting Orchestration via a Drone [[33](#_bookmark95)].

* + - 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 [[27](#_bookmark89)], with references **M1** to **M6**. The engine is set to “Explicit”; PRISM supports other computation engines (MTBDD, sparse, and hybrid) that offer differ- ent performances regarding model and properties structure (Please refer to documentation in [[34](#_bookmark96)]). 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 [[27](#_bookmark89)]. The website includes instructions for setting up and running the experiments and how to use the OMNeT++ execution trace data.

* 1. *Drones and InBox behavior modeling*

The modeled system for drone behavior is depicted in Algorithm [1](#_bookmark26). 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](#_bookmark26) line 8 is to check if the drone is in a

**Listing 1: PRISM Code for Drone Module**

1. **module** D1

# /\* Variable definitions \*/

3 ...

4 D1H : [0.. MAXH ] **init** H1 ;

5 ...

# /\* phase 1\*/

1. [ D 1 \_STATUS\_LISTENING ] s=-1 & listen1 -> (s ’=0) ;

8

# [ D 1 \_CHECK\_PARAMETER ] s=0 & D1H > D 1 refractory Period -> (s ’=1) & ( D1H ’= MSG\_H )& ( D1Pld ’= MSG\_PLD );

1. [ D 1 \_CHECK\_PARAMETER\_NOT ] s=0 & !( D1H > D 1 refractory Period ) -> (s ’=1) ;

11

# /\* Phase 2 \*/

1. [ D 1 \_SAME\_COUNT ] s=1 & D1Pld = MSG\_PLD & D1sameCount <2 -> (s ’=2) & ( D1sameCount ’= D 1 same Count +1) ;

14

# 15 [ D 1 \_ELSE\_SAME\_COUNT ] s=1 & D1Pld > MSG\_PLD -> (s ’=5) & ( D1Pld ’= MSG\_PLD );

16

# 17 [ D 1 \_ELSE\_ELSE\_SAME\_COUNT ] s=1 & D1Pld < MSG\_PLD -> (s ’=5) ;

18

# /\* Phase 3 \*/

1. [ D 1 \_TRANSMIT\_TO\_INBOX\_NOT\_LISTENING ] s=-1 & ! listen1 & (( D1H = D 1 next Broadcast ) | ( ( D1H = mod ( D 1 next Broadcast + D 1 refractory Period , D 1 cycle Length )) & D1sameCount < same Threshold )) -> (s ’=5) & ( D1sameCount ’=0) ;

21

# 22 [ D 1 \_COUNTING ] s=-1 & ! listen1 & !(( D1H = D 1 next Broadcast ) | ( ( D1H = mod ( D 1 next Broadcast + D 1 refractory Period , D 1 cycle Length )) & D1sameCount < same Threshold )) -> (s ’=7) ;

23

# /\* Phase 4 \*/

1. [ D 1 \_TRANSMIT\_TO\_INBOX ] s=5 -> (s ’=7) ;

26

# /\* Phase 5 \*/

1. [ D 1 \_CHECK\_CYCLE\_LENGTH ] ( s =7) & D1H = D 1 cycle Length -> (s ’= -1) & ( D1H ’=0) & ( D1sameCount ’=0) ;

29

# [ D 1 \_NO\_CHECK\_CYCLE\_LENGTH ] ( s =7) & D1H != D 1 cycle Length & D1H <MAXH -1 - > (s ’= -1) &( D1H ’= D1H +1) ;

1. **endmodule**

listening mode based on the Inbox gateway. The if-then block in lines 8-11 is mapped to PRISM commands in Listing [1](#_bookmark36) lines 6-7.

The if-then-else block of Algorithm [1](#_bookmark26) in lines 12-22 is translated into PRISM commands in lines 6-7 of Listing

[1](#_bookmark36). 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](#_bookmark26) 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](#_bookmark26) lines 23-28, it is translated into two PRISM commands in Listing [1](#_bookmark36) 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](#_bookmark26) lines 29-34.

The PRSIM commands in the Listing [1](#_bookmark36) (lines 25-27) correspond to the translation of Algorithm [1](#_bookmark26) (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](#_bookmark31) 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](#_bookmark37).

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](#_bookmark37)). Similarly, line 13 synchronizes with the second drone.

**Listing 2: PRISM Code for InBox Gateway Module**

1

2

3

4

5

**module** D1

/\* Variable definition \*/ listen1 : **bool init** false ;

...

[ D 1 \_TRANSMIT\_TO\_INBOX ]

( MSG\_PLD1 ’= D1Pld );

listen1 = false -> ( listen1 ’= true ) & ( MSG\_H1 ’= D1H ) &

6

7

[ D 2 \_TRANSMIT\_TO\_INBOX ]

( MSG\_PLD2 ’= D2Pld );

listen2 = false -> ( listen2 ’= true ) & ( MSG\_H2 ’= D2H ) &

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

12

13

14

[ D 1 \_STATUS\_LISTENING ]

cpt & listen1 -> ( listen1 ’= false ) ;

[ D 2 \_STATUS\_LISTENING ]

**endmodule**

cpt & listen2 ->

( listen2 ’= false );

* 1. *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***

P =?[ G( !(D1H == D2H) =⇒ F (D1H == D2H) )]

(1)

Property [1](#_bookmark39) 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 [3.1](#_bookmark6), monitoring the drones’ local clock updates is essential in the presence of environ- mental noise. We can model this clock drift in the drone’s local clock computations on line 30 of the Listing [1](#_bookmark36). As in [[7](#_bookmark69)], the noise can be represented as a probabilistic PRISM command in Listing [3](#_bookmark40). 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](#_bookmark40) (lines 2-3) as double types: d1proba and d2proba. This allows users to set their values before PRISM model construction, making the model parameterizable.

**module** D2

...

[ D 2 \_NO\_CHECK\_CYCLE\_LENGTH ] ( s =7) & D2H != D 1 cycle Length & D2H <MAXH -1 - > (1 - d2proba ):( s ’= -1) &( D2H ’= D2H +1) + d2proba :( s ’= -1) &( D2H ’= D2H +2) ;

**endmodule**

14

1. /\* Probabilistic variable definition
2. **const double** d1proba ;
3. **const double** d2proba ;

4 ...

5 **module** D1

6 ...

1. [ D 1 \_NO\_CHECK\_CYCLE\_LENGTH ] ( s =7) & D1H != D 1 cycle Length & D1H <MAXH -1 - > (1 - d1proba ):( s ’= -1) &( D1H ’= D1H +1) + d1proba :( s ’= -1) &( D1H ’= D1H +2) ;
2. **endmodule**

9

10

11

12

13

**Listing 3: PRISM Code for Clock Drift Observation**

Without synchronization in the referenced PRISM model “**M2**” in [[27](#_bookmark89)], 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 unsynchro- nized and exhibit varying degrees of drift?

***Property2***

R{”Desynchronized”} =?[ C≤T ]

(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 [[27](#_bookmark89)]. The model checking results are

shown in Figure [6](#_bookmark41). 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.



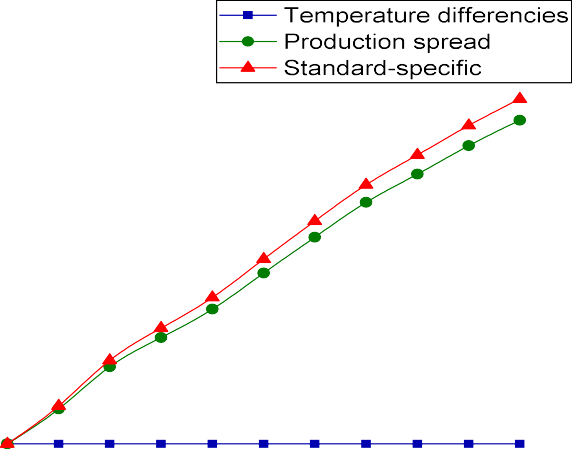


Figure 6: Cummulative Effects od desynchronization.

Considering the updated model “**M3**” incorporating drifting monitoring through probabilistic commands and gateway synchronization, we will verify property [1](#_bookmark39) 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](#_bookmark42) and Figure [8](#_bookmark42) during variation of the probabilistic variable d1proba and probabilistic variable d2proba, respectively.

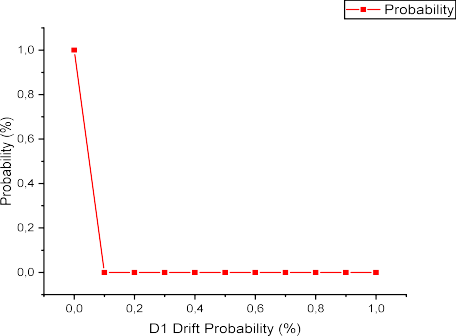
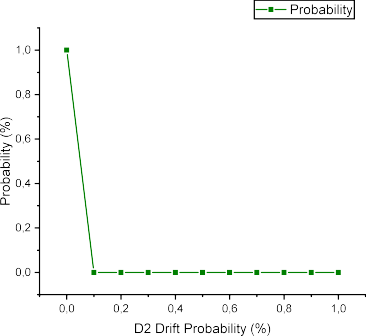


Figure 7: Verification of Property [1](#_bookmark39) with Drift variation on D2. Figure 8: Verification of Property [1](#_bookmark39) with Drift variation on D1.

The graphs in Figure [7](#_bookmark42) and Figure [8](#_bookmark42) 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 guaran- teed using the global operator alone. To ensure synchronization through the gateway, we define the following property:

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

***Property3***

P =?[ !(D1H == D2H) =⇒ F (D1H == D2H) ]

(3)

Checking the property [3](#_bookmark43) 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](#_bookmark44) and Figure [9](#_bookmark44) 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%.

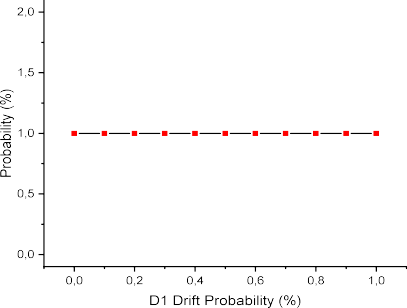
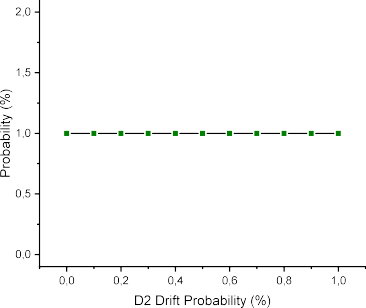


Figure 9: Verification of Property [3](#_bookmark43) with Drift variation on D2. Figure 10: Verification of Property [3](#_bookmark43) with Drift variation on D1.

D 1 refractory Period : [0.. MAXH ] **init** 6;

D 1 next Broadcast : [0.. MAXH ] **init** NEXT\_BROADCAST ;

...

**endmodule**

...

**module** D1

...

D 1 cycle Length : [0.. MAXH ] **init** 13;

1

2

3

4

5

6

7

8

9

10

**Listing 4: PRISM Code for Clock Drift Observation**

*Limitations.* While the modeled system using PRISM can handle up to 109 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 code snippet Listing [4](#_bookmark45)), 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](#_bookmark97) [[35](#_bookmark97)]. 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.

* 1. *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](#_bookmark26) and [2](#_bookmark31) are implemented in C/C++ code. However, the model architecture components denoted as (See section [4.2](#_bookmark18) ), are constructed using OMNeT++ language constructs. The code snippet in Listing [5](#_bookmark47) portrays the architecture of the use case. The model without and with synchronization using inbox gateway is available on [[27](#_bookmark89)] referenced as “**C1**” and “**C2**”, respectively.

 

The OMNeT++ architecture is characterized by three simple modules (i.e. components) (other com- plex 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 [3.1](#_bookmark6)) to make our model flexible and parametrizable. Gates using the keyword gates refers to the 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 compo- nents 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](#_bookmark48). On the right side, you’ll find various modules available for instantiating additional 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.

 

The internal structure of the drone component is defined using a class in C/C++. This class encapsulates the drone’s behavior through a set of attributes (data) and functions (methods). An example code snippet for the drone component is provided in the Listing [6](#_bookmark49). 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](#_bookmark50).

The handleMessage function (lines 5-19) in Listing [7](#_bookmark50) 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](#_bookmark49). 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

|  |  |
| --- | --- |
| **Listing 5: Architecture of Cranes Lifting Orchestration via Drones** | |
| 1 | **simple** Drone // Drone component |
| 2 | { |
| 3 | **parameters** : |
| 4 | **double** drift Rate ; |
| 5 | **gates** : |
| 6 | **input** DRONE\_LISTEN ; |
| 7 | **output** DRONE\_TRANSMIT ; |
| 8 | **output** CRANE\_LIFT ; |
| 9 | } |
| 10 |  |
| 11 | **simple** InBox // gateway component |
| 12 | { |
| 13 | **gates** : |
| 14 | **input** DRONE 0 \_TRANSMIT ; |
| 15 | **input** DRONE 1 \_TRANSMIT ; |
| 16 | **output** DRONE 0 \_LISTEN ; |
| 17 | **output** DRONE 1 \_LISTEN ; |
| 18 | } |
| 19 |  |
| 20 | **simple** Crane // crane component |
| 21 | { |
| 22 | **gates** : |
| 23 | **input** CRANE\_LIFT ; |
| 24 | } |
| 25 |  |
| 26 | **network** Cranes Orchestration // architecture composition |
| 27 | { |
| 28 | **submodules** : |
| 29 | Drone0 : Drone { } |
| 30 | Drone1 : Drone { } |
| 31 | Gateway : InBox { } |
| 32 | Crane0 : Crane { } |
| 33 | Crane1 : Crane { } |
| 34 | **connections** : |
| 35 | Drone0 . DRONE\_TRANSMIT -- > { delay = 100 ms; } -- > Gateway . |
|  | DRONE 0 \_TRANSMIT ; |

# Drone0 . CRANE\_LIFT -- > { delay = 100 ms ;} -- > Crane0 . CRANE\_LIFT ;

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 36 | Drone1 . DRONE\_TRANSMIT  DRONE 1 \_TRANSMIT ; | -- > | { | delay | = | 100 ms; | } | -- > | Gateway . |
| 37 |  |  |  |  |  |  |  |  |  |
| 38 | Gateway . DRONE 0 \_LISTEN | -- > | { | delay | = | 100 ms; | } | -- > | Drone0 . DRONE\_LISTEN ; |
| 39 | Gateway . DRONE 1 \_LISTEN | -- > | { | delay | = | 100 ms; | } | -- > | Drone1 . DRONE\_LISTEN ; |
| 40 |  |  |  |  |  |  |  |  |  |

1. Drone1 . CRANE\_LIFT -- > { delay = 100 ms; } -- > Crane1 . CRANE\_LIFT ;

43 }

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.

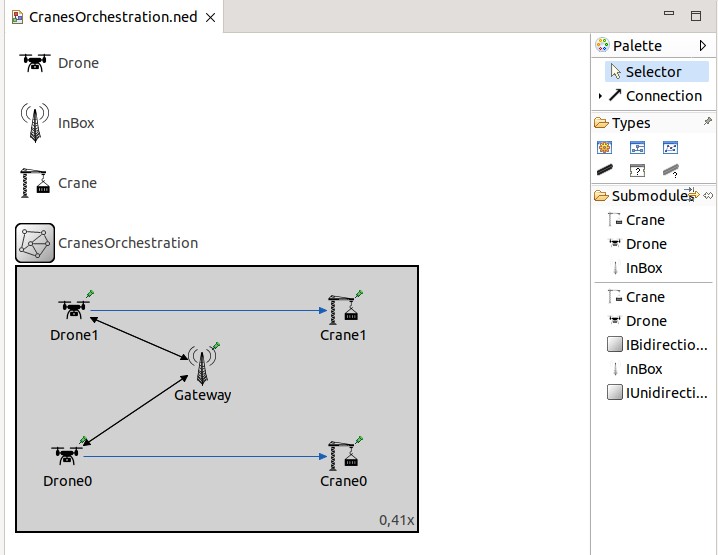


Figure 11: Cranes Lifting Orchestration via a Drone using OMNeT++ Graphical Tool.

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** handle Message ( c Message \* msg ) override ;

};

same Threshold =2;

**const int**

H;

PLD ;

next Broadcast ; cycle Lenght ; refractory Period ; same Count ;

**int**

**int int int int int**

**double** drift Rate ; // drift rate in seconds

TRANSMIT ;

c Message \*

**class** Drone : **public** c Simple Module

{

**public** :

Drone ();

**private** :

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

**Listing 6: Drone Component Functions**

The generated CSV file containing a collection of monitored drones’ internal clocks is available online (see

[[27](#_bookmark89)]). Users can access and consult this data for further analysis. The file is produced by custom C/C++ code

|  |  |  |  |
| --- | --- | --- | --- |
| 13 |  | | **if** ( interval <= drift Rate ) { |
| 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 c Message ( result . c\_str ()); | |
| 24 |  | send ( TRANSMIT , " DRONE\_TRANSMIT "); // send out the message | |
| 25 | } |  | |

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](#_bookmark52) de- picts 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.

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

1 **void** Drone :: initialize ()

2 {

3 drift Rate = par (" drift Rate ");

4 }

5 **void** Drone :: handle Message ( c Message \* msg )

6 {

7 ...

8 **if** ( H== cycle Lenght ){

9 H =0;

1. same Count =0;
2. } **else** {
3. **double** interval = distribution ( generator , std :: exponential\_distribution < **double** >:: param\_type ( drift Rate ));

To assess the validation accuracy, we employed a probabilistic decision tree. 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 [[7](#_bookmark69)] (see Figure [13](#_bookmark53) 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 [[27](#_bookmark89)] under the model reference **P2**.

* 1. *From decision trees to probabilistic commands*

The decision tree generated in Figure [13](#_bookmark53) is translated into a set of PRISM commands shown in Listing [8](#_bookmark54). 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 12: Plotting Internal Clock Synchronization for Model under Temperature Variations.

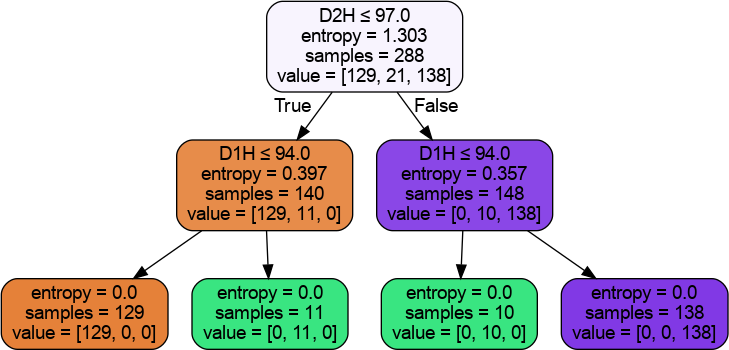


Figure 13: Generated Decision Tree from CSV File .

|  |  |
| --- | --- |
| **Listing 8: Mapping to Probabilistic Commands for Model under Temperature Variations** | |
| 1 | dtmc // This line declares the model as a Markov Decision Process ( MDP ) |
| 2 |  |
| 3 | // Constant definitions |
| 4 | **const int** NOTDEFINED = -1; // Defines a constant value for " not defined " |
| 5 | **const double** D1H ; // Declares a constant double variable named D1H |
| 6 | **const double** D2H ; // Declares a constant double variable named D2H |
| 7 |  |
| 8 | // Module definition |
| 9 | **module** Generated Decisiontree // Defines a module named |
|  | Generated Decisiontree |
| 10 |  |
| 11 | // State variable |
| 12 | 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 |
| 13 |  |
| 14 | // PRISM rules |
| 15 | [ rule1 ] D2H <= 97.0 & D1H <= 94.0 -> ( 0 . 007751937984496124 ) : ( CL ’ = |
|  | 0) + ( 0 . 9922480620155039 ) : ( CL ’ = NOTDEFINED ); // Probability |
|  | ( 0 . 9922 ...) for setting C’ to NOTDEFINED |
| 16 |  |
| 17 | [ rule2 ] D2H <= 97.0 & D1H > 94.0 -> |
| 18 | ( 0 . 09090909090909091 ) : ( CL ’ = 1) + |
| 19 | ( 0 . 9090909090909091 ) : ( CL ’ = NOTDEFINED ); |
| 20 |  |
| 21 | [ rule3 ] D2H > 97.0 & D1H <= 94.0 -> |
| 22 | (0.1) : ( CL ’ = 1) + |
| 23 | (0.9) : ( CL ’ = NOTDEFINED ); |
| 24 |  |
| 25 | [ rule4 ] D2H > 97.0 & D1H > 94.0 -> |
| 26 | ( 0 . 007246376811594203 ) : ( CL ’ = 2) + |
| 27 | ( 0 . 9927536231884058 ) : ( CL ’ = NOTDEFINED ); |
| 28 |  |
| 29 | **endmodule** // End of module definition |

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 com- mands. 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***

P =?[ G( !(D1H == D2H) =⇒ F (CL ≥ 0&CL ≤ 2) )]

(4)

Verifying Property [4](#_bookmark55) shows a 100% probability that the synchronization will be achieved based on drifting cat- egories in section [3.1](#_bookmark6): 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.

* 1. *Computational aspects of models scalability*

The PRISM models referenced as “**M1**” to “**M6**” in [[27](#_bookmark89)] 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 [[27](#_bookmark89)]. 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](#_bookmark55). 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](#_bookmark55). 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](#_bookmark57) we portray the evaluation of the developed model “**M6**” and the scaled models “**M7**”, “**M8**”, and “**M9**” in [[27](#_bookmark89)]. 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 [[36](#_bookmark98)], the verification cost is defined as 1 - ( *T v*(2 ) ) and abstraction efficiency

*Tv*(1 )

is defined as 1 - ( *T c*(2 ) ). We refer by 1) to the reference model (“**M3**”) while we refer by 2) to models in

“**M7**”, “**M8**”, and “

*Tc*(1 )

computes the model verification time and the function

computes the

**M9**”. The function *Tv Tc*

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.

## Discussion and threats to validity

In this section, we discuss the validity of our research results and the potential applications of the proposed approach.

* 1. *Beyond model checking*

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](#_bookmark23), we formalize semantic rules for managing clock deviation, which can be implemented in various formalisms like PRISM, UPPAAL [[37](#_bookmark99)], and BIP [[38](#_bookmark100)]. 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](#_bookmark32). This is attributed to algorithms explicitly designed for verifying a kind of liveness properties [[35](#_bookmark97)]. 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 [[39](#_bookmark101)]. The artifact’s compatibility with such frameworks allows for effortless incorporation, enhancing the overall efficiency and effectiveness of the modeling process.

* 1. *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 a 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.

## Related work

This literature review comprehensively surveys research on clock deviation or clock drift phenomena in net- working. As discussed in Section [3.1.1](#_bookmark7), [Elsts et al.](#_bookmark71) [[9](#_bookmark71)] provide a comprehensive foundation for understanding clock drift related to temperature variations, standard-specific deviations, and production spread. These phe- nomena 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](#_bookmark102) [[40](#_bookmark102)] on clock synchronization protocols to better understand clock synchronization protocols in wireless sensor networks (WSNs) from the per- spective of how and when clock values are propagated within the network and how physical clocks are updated. [Hauweele and Quoitin](#_bookmark103) [[41](#_bookmark103)] study investigates clock drift in Wireless Sensor Networks (WSNs) and its impact on Radio Duty Cycle (RDC) protocols [[42](#_bookmark104)]. The authors address limitations in the COOJA simulator [[43](#_bookmark105)], which cannot accurately reproduce the clock drift. They propose a new mathematical model that allows them to simu- late 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 Wire- less Sensor Networks (WSNs) as in [[44](#_bookmark106)]; [Xie et al.](#_bookmark106) 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.](#_bookmark107)

[[45](#_bookmark107)] 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.

Traditionally, clock synchronization research relies heavily on simulation for performance evaluation. How- ever, there’s a lack of studies investigating the applicability of simulation and model checking tools to assess the efficiency of synchronization algorithms compared to existing tools. While works like [[7](#_bookmark69)] provide a strong foun- dation 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.

## 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: 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 spec- ification, 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.

*Acknowledgement*

The research leading to the presented results was conducted within the research profile of s**A**fety and se**C**ur**I**ty as**S**urrance for critical **IoT** systems (ACIS-IOT) supported by the Centre National de la Recherche Scientifique (CNRS) and **H**uman-C**E**ntric Collabo**R**ative Architectural Decision-**M**aking for S**E**cure **S**ystem Design (HERMES- DESIGN) supported by Institut de Cybersécurité d’Occitanie (ICO).

## References

1. C. Baier, J.-P. Katoen, Principles of model checking, The MIT Press, ???? OCLC: ocn171152628.
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, ????, 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](http://dx.doi.org/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/](https://www.sciencedirect.com/science/article/pii/S0167739X19311926) [pii/S0167739X19311926](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.

1. J. Schiffer, C. A. Hans, T. Kral, R. Ortega, J. Raisch, Modeling, analysis, and experimental validation of clock drift effects in low-inertia power systems, IEEE Transactions on Industrial Electronics 64 (2017) 5942–5951. doi:[10.1109/TIE.2016.2638805](http://dx.doi.org/10.1109/TIE.2016.2638805).
2. M. Webster, M. Breza, C. Dixon, M. Fisher, J. McCann, Exploring the effects of environmental conditions and design choices on iot systems using formal methods, Journal of Computational Science 45 (2020) 101183. URL: [https://www.sciencedirect.com/](https://www.sciencedirect.com/science/article/pii/S1877750320304841) [science/article/pii/S1877750320304841](https://www.sciencedirect.com/science/article/pii/S1877750320304841). doi:https://doi.org/10.1016/j.jocs.2020.101183.
3. M. Webster, M. Breza, C. Dixon, M. Fisher, J. McCann, Formal verification of synchronisation, gossip and environmental effects for wireless sensor networks, Electronic Communications of the EASST (2018).
4. A. Elsts, X. Fafoutis, S. Duquennoy, G. Oikonomou, R. Piechocki, I. Craddock, Temperature-resilient time synchronization for the internet of things, IEEE Transactions on Industrial Informatics 14 (2018) 2241–2250. doi:[10.1109/TII.2017.2778746](http://dx.doi.org/10.1109/TII.2017.2778746).
5. D. Chefrour, Evolution of network time synchronization towards nanoseconds accuracy: A survey, Computer Communications 191 (2022) 26–35. URL: <https://www.sciencedirect.com/science/article/pii/S0140366422001359>. doi:https://doi.org/10. 1016/j.comcom.2022.04.023.
6. J. Wåhslén, I. Orhan, D. Sturm, T. Lindh, Performance evaluation of time synchronization and clock drift compensation in wireless personal area networks, in: Proceedings of the 7th International Conference on Body Area Networks, BodyNets ’12, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, 2012, p. 153–158.
7. A. Valmari, A stubborn attack on state explosion, Formal Methods in System Design 1 (1992) 297–322. URL: [https://doi.org/10.](https://doi.org/10.1007/BF00709154) [1007/BF00709154](https://doi.org/10.1007/BF00709154). doi:[10.1007/BF00709154](http://dx.doi.org/10.1007/BF00709154).
8. D. Peled, All from one, one for all: on model checking using representatives, in: C. Courcoubetis (Ed.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 1993, pp. 409–423.
9. A. Pnueli, In transition from global to modular temporal reasoning about programs, in: K. R. Apt (Ed.), Logics and Models of Concurrent Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 1985, pp. 123–144.
10. T. A. Henzinger, S. Qadeer, S. K. Rajamani, You assume, we guarantee: Methodology and case studies, in: A. J. Hu, M. Y. Vardi (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 440–451.
11. A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without bdds, in: W. R. Cleaveland (Ed.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 193–207.
12. P. F. Williams, A. Biere, E. M. Clarke, A. Gupta, Combining decision diagrams and sat procedures for efficient symbolic model checking, in: E. A. Emerson, A. P. Sistla (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 124–138.
13. E. M. Clarke, R. Enders, T. Filkorn, S. Jha, Exploiting symmetry in temporal logic model checking, Formal Methods in System Design 9 (1996) 77–104. URL: <https://doi.org/10.1007/BF00625969>. doi:[10.1007/BF00625969](http://dx.doi.org/10.1007/BF00625969).
14. Y. Falcone, S. Krsti´c, G. Reger, D. Traytel, A taxonomy for classifying runtime verification tools, International Journal on Software Tools for Technology Transfer 23 (2021) 255–284. URL: <https://doi.org/10.1007/s10009-021-00609-z>. doi:[10.1007/s10009-021-](http://dx.doi.org/10.1007/s10009-021-00609-z) [00609-z](http://dx.doi.org/10.1007/s10009-021-00609-z).
15. A. Baouya, S. Chehida, S. Ouchani, S. Bensalem, M. Bozga, Generation and verification of learned stochastic automata using k-nn and statistical model checking, Applied Intelligence 52 (2022) 8874–8894. URL: <https://doi.org/10.1007/s10489-021-02884-4>. doi:[10.1007/s10489-021-02884-4](http://dx.doi.org/10.1007/s10489-021-02884-4).
16. A. Baouya, O. Ait Mohamed, D. Bennouar, S. Ouchani, Safety analysis of train control system based on model-driven design method- ology, Computers in Industry 105 (2019) 1–16. URL: <https://www.sciencedirect.com/science/article/pii/S0166361518302276>. doi:https://doi.org/10.1016/j.compind.2018.10.007.
17. J. A. Davis, M. Clark, D. Cofer, A. Fifarek, J. Hinchman, J. Hoffman, B. Hulbert, S. P. Miller, L. Wagner, Study on the barriers to the industrial adoption of formal methods, in: C. Pecheur, M. Dierkes (Eds.), Formal Methods for Industrial Critical Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 63–77.
18. S. A. Chien, Formal methods for trusted space autonomy: Boon or bane?, in: J. V. Deshmukh, K. Havelund, I. Perez (Eds.), NASA Formal Methods, Springer International Publishing, Cham, 2022, pp. 3–13.
19. M. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proc. 23rd International Conference on Computer Aided Verification (CAV’11), volume 6806 of *LNCS*, Springer, 2011, pp. 585– 591.
20. H. Hansson, B. Jonsson, A logic for reasoning about time and reliability 6 (????) 512–535. URL: [https://dl.acm.org/doi/10.1007/](https://dl.acm.org/doi/10.1007/BF01211866) [BF01211866](https://dl.acm.org/doi/10.1007/BF01211866). doi:[10.1007/BF01211866](http://dx.doi.org/10.1007/BF01211866).
21. C. Lenzen, P. Sommer, R. Wattenhofer, Optimal clock synchronization in networks, in: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, SenSys ’09, Association for Computing Machinery, New York, NY, USA, 2009, p. 225–238. URL: <https://doi.org/10.1145/1644038.1644061>. doi:[10.1145/1644038.1644061](http://dx.doi.org/10.1145/1644038.1644061).
22. Abdelhakim Baouya, Paper Artefacts Sources, <https://acis-iot.github.io/csi2024.html>, 2024.
23. V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, Automated verification techniques for probabilistic systems, in: M. Bernardo, V. Issarny (Eds.), Formal Methods for Eternal Networked Software Systems (SFM’11), volume 6659 of *LNCS*, Springer, 2011, pp. 53–113.
24. A. Aziz, V. Singhal, F. Balarin, R. K. Brayton, A. L. Sangiovanni-Vincentelli, It usually works: The temporal logic of stochastic systems, in: P. Wolper (Ed.), Computer Aided Verification, volume 939, Springer Berlin Heidelberg, ????, pp. 155–165. URL: [http://link.](http://link.springer.com/10.1007/3-540-60045-0_48) [springer.com/10.1007/3-540-60045-0\_48](http://link.springer.com/10.1007/3-540-60045-0_48).
25. M. L. Wei Zhou, Yi Lu, K. Zhang, Machine learning methods based on probabilistic decision tree under the multi-valued preference environment, Economic Research-Ekonomska Istraživanja 35 (2022) 38–59. doi:[10.1080/1331677X.2021.1875866](http://dx.doi.org/10.1080/1331677X.2021.1875866).
26. OMNeT++, Discrete Event Simulator, <https://omnetpp.org/>, Since 2008. [Accessed: January-2023].
27. M. Oxford, D. Parker, M. Ryan, Quantitative verification of certificate transparency gossip protocols, in: 2020 IEEE Conference on Communications and Network Security (CNS), 2020, pp. 1–9. doi:[10.1109/CNS48642.2020.9162197](http://dx.doi.org/10.1109/CNS48642.2020.9162197).
28. CPS4EU Project, CPS Tool Evaluation. D5.6, [https://cps4eu.eu/wp-content/uploads/2022/11/CPS4EU\_D5.6\_CPS-Tool\_](https://cps4eu.eu/wp-content/uploads/2022/11/CPS4EU_D5.6_CPS-Tool_Evaluation_V1.0.pdf) [Evaluation\_V1.0.pdf](https://cps4eu.eu/wp-content/uploads/2022/11/CPS4EU_D5.6_CPS-Tool_Evaluation_V1.0.pdf), 2022. [Online; accessed 19-March-2023].
29. Prism computation engines, <https://www.prismmodelchecker.org/manual/ConfiguringPRISM/ComputationEngines>, Accessed

2024.

1. R. Alur, Principles of Cyber-Physical Systems, The MIT Press, 2015.
2. C. Wang, G. D. Hachtel, F. Somenzi, Abstraction Refinement for Large Scale Model Checking, Springer Publishing Company, Incorpo- rated, 2014.
3. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, M. Hendriks, Uppaal 4.0 (2006).
4. A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, J. Sifakis, Rigorous component-based system design using the BIP framework 28 (????) 41–48. URL: <http://ieeexplore.ieee.org/document/5719588/>. doi:[10.1109/MS.2011.27](http://dx.doi.org/10.1109/MS.2011.27).
5. E. Foundation, Eclipse papyrus, ???? URL: <https://www.eclipse.org/papyrus/>, accessed April 20, 2024.
6. A. R. Swain, R. Hansdah, A model for the classification and survey of clock synchronization protocols in wsns, Ad Hoc Networks 27 (2015) 219–241. URL: <https://www.sciencedirect.com/science/article/pii/S1570870514002960>. doi:https://doi.org/10. 1016/j.adhoc.2014.11.021.
7. D. Hauweele, B. Quoitin, Toward accurate clock drift modeling in wireless sensor networks simulation, Computer Communications 163 (2020) 1–11. doi:https://doi.org/10.1016/j.comcom.2020.08.025.
8. M. Bezunartea, B. Sartori, J. Tiberghien, K. Steenhaut, Tackling malfunctions caused by radio duty cycling protocols that do not appear in simulation studies, in: Proceedings of the First ACM International Workshop on the Engineering of Reliable, Robust, and Secure Embedded Wireless Sensing Systems, FAILSAFE’17, Association for Computing Machinery, New York, NY, USA, 2017, p. 10–15. URL: <https://doi.org/10.1145/3143337.3143340>. doi:[10.1145/3143337.3143340](http://dx.doi.org/10.1145/3143337.3143340).
9. Contiki, Cooja simulator, 2024. URL: [https://docs.contiki-ng.org/en/develop/doc/tutorials/Running-Contiki-NG-in-](https://docs.contiki-ng.org/en/develop/doc/tutorials/Running-Contiki-NG-in-Cooja.html) [Cooja.html](https://docs.contiki-ng.org/en/develop/doc/tutorials/Running-Contiki-NG-in-Cooja.html).
10. K. Xie, Q. Cai, M. Fu, A fast clock synchronization algorithm for wireless sensor networks, Automatica 92 (2018) 133–142. doi:https:

//doi.org/10.1016/j.automatica.2018.03.004.

1. X. Zhang, H. Chen, K. Lin, Z. Wang, J. Yu, L. Shi, Rmts: A robust clock synchronization scheme for wireless sensor networks, Journal of Network and Computer Applications 135 (2019) 1–10. doi:https://doi.org/10.1016/j.jnca.2019.02.028.