Verifying Interoperability in Evolving IoT Systems

Hongming Zhang[1], Judy Bowen[1][a],Jessica Turner[1][b] and Jemma König [1][c]
[1]*University of Waikato, New Zealand*
hz226@students.waikato.ac.nz

Keywords: Model checking, IoT, Interoperability, Process mining, System upgrade.

Abstract: As IoT devices age, they must be replaced to maintain reliability and performance. During upgrades, ensuring interoperability among new and existing devices is critical for preserving designated system behaviours. Existing formal verification approaches rely on system documentation or source code to build formal models or extract flat models from system logs. We propose a novel log-driven verification framework that automatically discovers executable Hierarchical Colored Petri Nets (HCPNs) from raw IoT system logs. The framework integrates model checking to verify cross-layer interoperability during IoT system evolution and device replacement. We demonstrate the effectiveness of our approach using a street lighting system study.

1 INTRODUCTION

The Internet of Things (IoT) enables connection and data exchange among various devices. By collecting and analysing sensor data, IoT systems provide intelligent functionalities that support efficient living environments in areas such as smart cities (Krichen, 2023a). However, IoT devices have limited operational lifespans, and their performance and reliability tend to degrade over time, requiring appropriate upgrades and replacements. Replacing IoT devices and related system components is more practical than replacing the entire system, as it reduces costs while minimising system downtime. (Adepoju et al., 2024).

Interoperability between newly integrated devices and existing system components is crucial for maintaining overall system performance. Within IoT, interoperability can be classified into five levels (Noura et al., 2019): (1) Device interoperability, (2) Platform interoperability, (3) Semantic interoperability, (4) Syntax interoperability and (5) Networking interoperability. The rapid evolution of IoT technologies has altered the communication patterns and data formats in modern devices. As a result, effective interoperability verification methods have become essential when integrating new IoT devices or techniques.

Formal verification techniques are valuable for examining interoperability in IoT systems (Krichen, 2023b). Two primary approaches are model-based testing and model checking. Many existing approaches formalise system behaviours by assuming full access to system source code (Weigert et al., 2019) or detailed interface documentation (Giese et al., 2008). However, these methods are impractical for long-running, frequently updated systems, as keeping source code and documentation synchronised with evolving functionalities is costly and error-prone.

To reduce reliance on source code and documentation, process mining techniques can be applied to derive flat formal models from sensing, actuation, and device interaction logs, e.g. (De Fazio et al., 2023; Mangi et al., 2023). However, these models merge all workflows without clear component boundaries, making it difficult to verify individual components and their interactions.

In this paper, we propose a formal verification framework for IoT interoperability under device replacement scenarios. Our contributions are: A novel method to automatically derive HCPN models directly from raw IoT logs, enabling interoperability verification at both the device and system levels; A conformance checking approach that verifies conformance between HCPN models and event logs; Automatic HCPN discovery, hierarchical conformance validation, and interoperability property verification within a single formal verification workflow.

[a] https://orcid.org/0000-0003-2815-8267
[b] https://orcid.org/0000-0002-7623-882X
[c] https://orcid.org/0000-0002-1756-1168

2 BACKGROUND

Process mining is a technique for discovering formal models from event logs (Tajima et al., 2023), it involves three main components (Van Der Aalst et al., 2012). First, process discovery converts event logs into formal models. Second, conformance checking identifies differences between the discovered models and the actual system behaviour. Third, process enhancement improves the formal models by addressing deviation issues and performance bottlenecks. Petri nets are widely used in process mining, and tools such as PM4Py support their discovery and conformance checking. However, traditional Petri nets have a flat structure, which limits analysis of component-level behaviours in systems. Colored Petri Nets (CPNs) incorporate coloured tokens to represent data types and data-dependent behaviours. HCPNs further add substitution transitions to encapsulate component-level behaviours while maintaining the system-level structure (Benabdelhafid et al., 2025).

In this work, process mining is used to discover CPN models from IoT system logs, which are then transformed into HCPNs via our proposed algorithms to enable hierarchical modelling of system components. Conformance checking is applied to ensure model correctness.

3 RELATED WORK

In IoT contexts, system process model discovery through process mining requires structured event logs. However, raw IoT logs often lack crucial information, particularly case identifiers that link individual events into traces representing complete workflow executions. To address this limitation De Fazio et al. (2023) collaborated with domain experts to define clustering rules, such as fixed time windows and activity locations, to group related events into workflows and generate case identifiers. Similarly, Singh et al. (2020) leverages real-world observations that sensor events occurring at specific locations are likely to belong to the same workflow, extracting event patterns to determine the workflow associated with individual sensor logs.

Once event logs are constructed, formal behavioural models can be discovered using process mining techniques. Seiger et al. (2024) apply process mining to discover Business Process Model and Notation 2.0 (BPMN) models that represent operational behaviours in an IoT-enabled smart factory. Furthermore, discovered models are often transformed into alternative formal representations to meet specific behaviour representation or verification requirements. For instance, Mangi et al. (2023) transforms discovered Petri nets to a Discrete-Time Markov Chain, a type of probabilistic model used to represent stochastic interactive behaviour.

To evaluate the correctness of discovered formal models, conformance checking algorithms are applied to assess the consistency between model behaviour and system execution logs. Yamaguchi and Ahmadon (2022) propose a token-based replay algorithm to calculate the overall fitness of data flow models compared with system execution logs. Zheng et al. (2022) extends traditional deterministic alignment-based conformance checking to validate deviations between process models and probabilistic sensor logs.

Overall, process mining approaches have been employed to extract system behavioural models from raw IoT logs. However, existing modelling and conformance checking methods used in IoT primarily focus on the discovery and verification of flat formal models, which lack explicit component boundaries and structural layering. To address this limitation, this paper proposes a method for discovering hierarchical formal models that capture both component-level and system-level behavioural structures.

4 PROPOSED METHODOLOGY

In this section, we introduce our proposed framework, which has been implemented in Python and is hosted at: https://github.com/hz226/hcpn_generation.

4.1 Preliminaries

We first provide the definitions that are fundamental to our proposed verification framework.

Definition 1 (IoT device). An IoT device is a physical entity equipped with sensors or actuators within an IoT system.

Definition 2 (Sensor log). A sensor log $l_s \in \mathcal{L}_S$ records a sensing action performed by a sensor; each sensor reading log includes the following elements W3C (2017):

$$l_s = (sid, m, val, t, d)$$

where $sid \in \mathcal{SID}$ is the sensor identifier, $m \in \mathcal{M}$ refers to the measurement type, $val \in \mathcal{V}$ is the sensed value, $t \in \mathcal{T}$ is the timestamp, and $d \in \mathcal{D}$ indicates the hosting IoT device.

Definition 3 (Actuation log). An actuation log $l_a \in \mathcal{L}_A$ captures an actuation event executed by an actuator after receiving a command. An actuation record includes the following information W3C

(2017):

$$l_a = (id, m, t, s_{pre}, s_{post}, d)$$

where $id \in I\mathcal{D}$ is the actuator identifier, $m \in \mathcal{M}$ refers to the procedure or command that trigger the actuation event, $t \in \mathcal{T}$ is the timestamp when the actuation occurs, s_{pre} represents the actuator's state before actuation, and s_{post} is the actuator's state after executing the command, and $d \in \mathcal{D}$ denotes the hosting IoT device.

Definition 4 (Interaction log). An interaction log $l_i \in \mathcal{L}_I$ records a communication event between two IoT devices via a specific communication protocol and is represented as W3C (2017):

$$l_i = (d_s, d_t, i_c, cmd, m_i, t, d_p)$$

where $d_s, d_t \in \mathcal{D}$ denote the source and target devices, $i_c \in I_C$ refers to the protocol binding used for interaction, $cmd \in C$ refers to the interaction affordance name, corresponding to an action, or event subscription, $m_i \in M_i$ denotes the exchanged message, $t \in T$ refers to the timestamp, $d_p \in D$ indicates the device that generates the log, either the source device d_s or the target device d_t.

Definition 5 (Event log).

An event log records the execution information of a process and is defined as Tajima et al. (2023):

$$e = (c, a, t)$$

where the case ID identifier c identifies a unique sequence of activities (i.e., a trace), the activity name a refers to the name of the executed operation, and the timestamp t indicates the time at which the activity occurs

Definition 6 (Sensor event log). A sensor event $e_s \in \mathcal{E}_S$ is a type of event log (Definition 5), which represents a sensing event. It is derived from a sensor log l_s (Definition 2) and is represented as

$$e_s = (c_s, a_s, t, val, d, sid)$$

where case identifier $c_s \in \mathcal{C}_S$ identifies a trace of sensing activities performed by a sensor sid installed on its hosting device d, activity label $a_s \in \mathcal{A}_S$ represents the sensing action corresponding to log entry $m \in l_s$, the remaining elements are identical to those in l_s.

Definition 7 (Actuation event log). An actuation event $e_a \in \mathcal{E}_A$ is a type of event log (Definition 5), which represents an actuation event recorded in an actuation log l_a (Definition 3) and is defined as

$$e_a = (c_a, a_{cmd}, t, s_{pre}, s_{post}, d, id)$$

where case identifier $c_a \in \mathcal{C}_A$ identifies a trace of actuations performed by an actuator on its hosting device d, activity label $a_{cmd} \in \mathcal{C}$ represents the command triggering the actuation event related to the actuation log entry $m \in l_a$, the remaining elements are the same as those in l_a.

Definition 8 (Interaction event log). An interaction event $e_i \in \mathcal{E}_I$ is a type of event log (Definition 5), which represents an interaction event recorded in an interaction log l_i (Definition 4) is represented as

$$e_i = (c_i, a_i, t, m_i, d_s, d_t, d_p)$$

where case identifier $c_i \in \mathcal{C}_I$ identifies a trace of related interactions, activity label a_i denotes the communication endpoint or operation corresponding to the interaction log entry $cmd \in l_i$, and other elements are the same as those in l_i.

Definition 9 (Event log tree). An event log tree is a directed acyclic graph (DAG) that organises IoT event logs in a hierarchical structure:

$$\mathcal{T} = (\mathcal{V}, \mathcal{E})$$

where the node set $\mathcal{V} \subseteq \mathcal{V}_E \cup \mathcal{V}_A \cup \mathcal{V}_D \cup \mathcal{V}_I$, and $\mathcal{V}_E$ denotes sensor event nodes, $\mathcal{V}_A$ actuation event nodes, $\mathcal{V}_D$ device component nodes, and $\mathcal{V}_I$ interaction nodes.

The edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ represents directed relationships among the nodes, including **interaction edges** that connect an interaction node $v_i \in \mathcal{V}_I$ to its corresponding source and target device component nodes $v_d \in \mathcal{V}_D$, and **component edges** that connect sensor $v_e \in \mathcal{V}_E$ and actuation event nodes $v_a \in \mathcal{V}_A$ within a device component.

Definition 10 (Sensor event node). A sensor event node $V_d^E \in \mathcal{V}_E$ corresponds to a sensor event log $e_s \in \mathcal{E}_S$ (Definition 6) via the mapping $\mapsto e_s$, where $\mapsto$ denotes "maps to":

$$V_d^E \mapsto e_s, e_s = (c_s, a_s, t, val, d, sid)$$

Definition 11 (Actuation event node). An actuation event node $\mathcal{V}_d^A \in \mathcal{V}_A$ maps to an actuation event log $e_a \in \mathcal{E}_A$ (Definition 7) using the mapping $\mapsto e_a$ where $\mapsto$ denotes "maps to":

$$V_d^A \mapsto e_a, e_a = (c_a, a_{cmd}, t, s_{pre}, s_{post}, d, id)$$

Definition 12 (Device component node). A device component node $v_d \in \mathcal{V}_D$ comprises associated sensor event nodes $\mathcal{V}_d^E \subseteq \mathcal{V}_E$ and actuation event nodes $\mathcal{V}_d^A \subseteq \mathcal{V}_A$ corresponding to the same device d:

$$\mathcal{V}_d = \mathcal{V}_d^E \cup \mathcal{V}_d^A$$

These event nodes are ordered by timestamp t:

$$\forall e_i, e_j \in V_d, \quad e_i \prec e_j \iff t(e_i) \leq t(e_j),$$

Definition 13 (Interaction node). An interaction node $\mathcal{V}_d^I \in \mathcal{V}_I$ corresponds to an interaction event log $e_i \in \mathcal{E}_I$ (Definition 8) via the mapping:

$$\mathcal{V}_d^{\mathcal{A}} \mapsto e_i, e_i = (c_i, a_i, t, v_{d_s}, v_{d_t}, d_p)$$

where $v_{d_s}, v_{d_t} \in \mathcal{V}_D$ are the device component nodes related to the source device d_s and target device d_t, $a_i \in \mathcal{A}_I$ is the activity label associated with the interaction event e_i.

We use existing definitions for Colored Petri Nets (Jensen, 1987), CPN modules (An et al., 2018) and Hierarchical Colored Petri Nets ()ali2024modeling.

4.2 Data Collection

The first step of our verification framework uses Node-RED and ThingsBoard to monitor and collect sensing, actuation, and device interaction data from IoT devices. All sensors must be registered in ThingsBoard as device entities for data storage.

Once ThingsBoard is configured, Node-RED is deployed as an external monitoring service, subscribing to sensing, actuation, and interaction events via protocol-specific nodes and forwarding collected data to ThingsBoard. The IoT system is then executed in either simulated or real-world environments, where triggered events are continuously collected. ThingsBoard stores all records with timestamps in its integrated PostgreSQL database. A script is applied to extract and transform the data into sensor, actuation, and interaction logs, according to the formats defined in Definitions 2, 3, and 4 in Section 4.1. The resulting timestamp-ordered dataset is exported as a Comma-Separated Values (CSV) file, which is the input for the subsequent step of generating event logs.

4.3 Event Log Generation

After collecting the raw device logs, the IoTLog2EventLog algorithm (Algorithm 1) is applied to generate event logs and split them into training and testing sets for subsequent HCPN discovery and conformance checking. The algorithm consists of the following seven steps:
Step 1 (line 4): Grouping device logs.
Step 2 (line 5): Determine the optimal inter-event time gap threshold Δ_t.
Step 3 (lines 6–10): Case identifier assignment.
Step 4 (lines 12–16): Activity extraction.
Step 5 (lines 17–18): Event log generation.
Step 6 (line 19): Identify unique event traces of each device using the TopK function.
Step 7 (line 20): Generate training and testing sets of event logs using the Transform_Split function.

1 **Input:** Device logs $\mathcal{L} = \mathcal{L}_S \cup \mathcal{L}_A \cup \mathcal{L}_I$;
2 **Output:** Device event logs $\mathcal{E} = \mathcal{E}_S \cup \mathcal{E}_A \cup \mathcal{E}_I$;
3 Initialize $\mathcal{E}_S, \mathcal{E}_A, \mathcal{E}_I, \mathcal{V} \leftarrow \emptyset$, $caseID \leftarrow 1$;
4 **foreach** *cluster G in* $\mathcal{L}$ *grouped by device* **do**
5 $\quad \Delta_t \leftarrow$ `calc_timegap`(G);
6 $\quad prev \leftarrow$ `null`;
7 $\quad$ **foreach** *log* $l \in G$ *ordered by timestamp* **do**
8 $\quad\quad t \leftarrow \pi_t(l)$;
9 $\quad\quad$ **if** $prev \neq$ `null` ***and*** $(t - prev > \Delta_t)$ **then**
10 $\quad\quad\quad caseID \leftarrow caseID + 1$;
11 $\quad\quad$ **if** $l \in \mathcal{L}_S$ **then**
12 $\quad\quad\quad activity \leftarrow \pi_m(l)$;
13 $\quad\quad$ **else if** $l \in \mathcal{L}_A$ **then**
14 $\quad\quad\quad activity \leftarrow \pi_{a_{cmd}}(l)$;
15 $\quad\quad$ **else if** $l \in \mathcal{L}_I$ **then**
16 $\quad\quad\quad activity \leftarrow \pi_{cmd}(l)$;
17 $\quad\quad (\mathcal{E}_S, \mathcal{E}_A, \mathcal{E}_I) \leftarrow f(l, caseID, activity)$;
18 $\quad\quad prev \leftarrow t$;
19 $\quad\quad \mathcal{V} \leftarrow \mathcal{V} \cup \text{TopK}(\mathcal{E})$;
20 $(\mathcal{E}_S, \mathcal{E}_A, \mathcal{E}_I) \leftarrow$ Transform_Split$(\mathcal{V})$;
21 **return** $(\mathcal{E}_S, \mathcal{E}_A, \mathcal{E}_I)$;

Algorithm 1: IoTLog2EventLog

4.4 Process Mining

Once the training event logs are generated, we integrate process mining with log clustering to derive HCPN models from the logs. The process mining process is illustrated by Algorithm 2.
Step 1 (lines 4–7): Device-specific CPN modules are discovered for each device.
Step 2 (lines 8–13): Interaction CPN modules are created to represent interactions between devices. Step 3 (lines 14–15): Substitution transitions in each interaction module are linked to corresponding device CPN modules from Step 1.

Previous research approaches create flat Peri nets, which represent all device workflows in a single structure; in contrast, our HCPNs decompose workflows into layered CPN modules. Compared with the flat Petri net, the HCPN is more modular and can be verified at both device-level and system-level behaviours.

4.5 Conformance Checking

We assess the correctness of the HCPN by detecting deviations between the HCPN and an event log tree using alignment-based conformance checking. As input for the conformance checking, an event log tree

1 **Input:** $\mathcal{E} = \mathcal{E}_S \cup \mathcal{E}_A \cup \mathcal{E}_I$;
2 **Output:** $\Sigma_h = (S, SM, PS, FS)$;
3 $S, SM, PS \leftarrow \emptyset$, $\Sigma_D \leftarrow \emptyset$;
4 **foreach** $d \in \{\pi_d(e) \mid e \in \mathcal{E}_S \cup \mathcal{E}_A\}$ **do**
5 $\quad \Sigma_d \leftarrow$ `discover_device_cpn`$(\{e \in \mathcal{E}_S \cup \mathcal{E}_A \mid \pi_d(e) = d\})$;
6 $\quad S \leftarrow S \cup \{\Sigma_d\}$;
7 $\quad \Sigma_D[d] \leftarrow \Sigma_d$;
8 **foreach** (d_s, d_p) *such that* $\exists e_i \in \mathcal{E}_I$ **do**
9 $\quad E_{(d_s,d_p)} \leftarrow \{e_i \in \mathcal{E}_I \mid \pi_{d_s}(e_i) = \pi_{d_p}(e_i)\}$;
10 $\quad \Sigma_{(d_s,d_p)} \leftarrow$ `discover_interaction_cpn`$(E_{(d_s,d_p)})$;
11 $\quad t^{d_s}_{sub}, t^{d_t}_{sub} \leftarrow$ `SubTrans`(d_s, d_t);
12 $\quad SM(t^{d_s}_{sub}) = \Sigma_D[d_s]$;
13 $\quad SM(t^{d_t}_{sub}) = \Sigma_D[d_t]$;
14 $\quad PS \leftarrow PS \cup \{(p^{\Sigma_D[d_s]}_{\text{out}}, p^{\Sigma_i}_{\text{in}}), (p^{\Sigma_i}_{\text{out}}, p^{\Sigma_D[d_t]}_{\text{in}})\}$;
15 $\quad S \leftarrow S \cup \{\Sigma_i\}$;
16 **return** $\Sigma_h = (S, SM, PS, FS)$;

Algorithm 2: EventLog2HCPN

must be generated. Device component nodes and their sensor and actuation event nodes are created and then interaction nodes are constructed. Finally, the event log tree $\mathcal{T}$ is generated.

4.5.1 Conformance Checking Algorithm

Once the event log tree has been created, we apply checking (Algorithm 3) to detect deviations between the HCPN and its corresponding event log tree, and visualise deviations.
Step 1 (Lines 7–9): The align_device_node function is assigned to perform conformance checking on CPN modules against device component nodes.
Step 2 (Lines 10–11): The align_interaction_node is assigned to perform conformance checking between CPN modules and interaction nodes.
Step 3 (Lines 13–19): We handle cases where the corresponding module is not found.
Step 4 (Line 20): All deviation results are collected to compute the overall fitness (f) of the HCPN module compared to the event log tree.

Finally, deviations are visualised. We developed HCPNCytoscapeVisualizer, an interactive tool that displays CPN modules of the HCPN hierarchically and highlights detected deviations (>>) on transitions. It was implemented based on the open-source Python library, Dash Cytoscape. The visualiser tool annotates transitions and substitution transitions with conformance checking deviations, marking model and log moves with >> and summarising deviations for each CPN module. This visualisation enables rapid identification and analysis of deviations between the HCPN and event logs.

1 **Input:** Event Log Tree $\mathcal{T} = (\mathcal{V}_E, \mathcal{V}_A, \mathcal{V}_D, \mathcal{V}_I)$;
2 **Input:** Hierarchical CPN $\Sigma_h = (S, SM, PS, FS)$;
3 **Output:** Alignment $\mathcal{A}$, Fitness f;
4 **Constants:** δ_D, δ_I;
5 $maxCost \leftarrow 0$, $totalCost \leftarrow 0$, $\mathcal{V}_{deviation} \leftarrow \emptyset$;
6 **foreach** $v \in \mathcal{V}_D \cup \mathcal{V}_I$ **do**
7 $\quad module \leftarrow \phi_D(v)$;
8 $\quad$ **if** $v \in \mathcal{V}_D$ **then**
9 $\quad\quad alignFunc \leftarrow$ align_device_node(module, δ_D);
10 $\quad$ **else**
11 $\quad\quad alignFunc \leftarrow$ align_interaction_node(module, δ_I);
12 $\quad$ **if** $module =$ `SKIP_MODULE` **then**
13 $\quad\quad m \leftarrow \begin{cases} (>>, \text{skipCost}, \{>>\}, \delta_D) & \text{if } v \in \mathcal{V}_D \\ (>>, \text{skipCost}, \{>>\}, \delta_I) & \text{if } v \in \mathcal{V}_I \end{cases}$;
14 $\quad$ **else**
15 $\quad\quad m \leftarrow alignFunc$;
16 $\quad (align, cost, violations, cost_{max}) \leftarrow m$;
17 $\quad maxCost \leftarrow maxCost + cost_{max}$;
18 $\quad totalCost \leftarrow totalCost + cost$;
19 $\quad \mathcal{V}_{deviation} \leftarrow \mathcal{V}_{deviation} \cup \{(violations, module)\}$;
20 $f \leftarrow 1 - \dfrac{totalCost}{maxCost}$;
21 **return** $(\mathcal{A}, f)$;

Algorithm 3: Hierarchical Alignment-Based Conformance Checking

4.6 Interoperability Properties

After refining the HCPNs models, interoperability properties are defined to ensure that the original HCPNs satisfy all existing core functionalities, whereas upgraded HCPNs must satisfy both these original functionalities and any new features through device replacement. These properties are formally specified using ASK-CTL, a Computation Tree Logic (CTL) formal logic, and classified into four types: reachability, liveness, safety, and deadlock freedom.

4.7 Model Checking Interoperability

Once the interoperability properties have been defined, the CPN Tools model checker is applied to ver-

ify interoperability in both the original HCPNs and the upgraded HCPNs after device replacement. This step is crucial for detecting potential interaction issues between newly added devices and the remaining devices in IoT systems, ensuring that device replacement does not impact interoperability in IoT systems prior to deployment. The verification process involves four steps: (1) Importing the HCPNs into the CPN Tools; (2) Developing standard ML (SML) scripts to verify the defined ASK-CTL properties; (3) Generating the full state space of the HCPNs using the State Space Tool in the CPN Tools and executing the SML scripts via the Evaluate ML tool. The execution results indicate whether each property holds (True) or not (False); (4) Analysing violated properties.

5 EXPERIMENTAL STUDY

In this section, we demonstrate the effectiveness of our verification framework using a simulated street lighting system for a road bridge, inspired by a real-world Adaptive Lighting System (SCALS) Gagliardi et al. (2020). The simulated system consists of an entrance controller light pole, an exit controller light pole, and seven worker light poles. The controller poles have motion sensors, cameras, light actuators and microcontrollers, while the worker poles only have light actuators and microcontrollers.

When the entrance pole detects a car entering, or the exit controller pole detects a car leaving, the controller poles then notify each other and the worker poles, which adjust their lighting according to their predefined strategies. For example, if a car is detected on the bridge, the pole lights turn on; otherwise, they dim to a lower level. In our experiment, the motion sensors of the controller poles are replaced with new ones that have different communication patterns with the cameras than the original ones.

5.1 Experiment Setup

We used two separate Node-RED workflows integrated with MQTT protocol modules to simulate and monitor the system behaviours. The Node-Red workflow simulates the sensing, actuation, and interaction behaviours of the entrance pole, the exit pole and the worker poles. The workflows are triggered to simulate car entering and leaving under two scenarios: a busy scenario, where 50 cars queue to pass through the 1 km-long bridge, traffic moves slowly, each car requires 6 minutes to cross; and an idle scenario observing 50 cars, where only one car crosses the bridge every hour, and each crossing takes 2 minutes.

To store the collected data, we imported the motion sensors, cameras, light actuators, and microcontrollers of the light poles into ThingsBoard as device entities. We then obtained the MQTT telemetry upload commands for each device's data from the ThingsBoard device connectivity interface. These commands were configured in the Node-RED monitor flows to enable data transmission.

5.2 Data Collection

After configuring the Node-RED workflows and ThingsBoard, we ran the lighting system by triggering the entrance and exit light pole workflows at set intervals and configuring the car-leaving function block to simulate the designed busy and idle scenarios. We then extracted and transformed the collected sensor, actuation, and interaction data into the predefined log formats (Defns 2, 3, 4).

5.3 Event Log Generation

We ran the Python implementation of Algorithm 1 to determine the time gap threshold for the collected logs within the same case, and then converted them into event logs for each light pole. Using the identified time gap threshold (four seconds) for case ID generation, we converted the collected logs into event logs grouped by device. We then executed the Variant_Detection algorithm to identify the top 10 high-frequency event traces for each device. Each variant trace was then mapped back to its corresponding event logs. The event logs were split into a training set that comprises the first 40% of the logs, and a testing set that comprises the rest of the logs.

5.4 Process Mining

We discovered an HCPN from the training event log dataset using our implementation of Algorithm 2. Fig. 1 shows an example of an interaction CPN module from the generated HCPN. This CPN module represents the interaction workflow in which the entrance light pole E1 sends notifications to the exit light pole E2 and worker light poles W1 to turn on their lights. The substitution transitions E1 and E2 link to the CPN modules representing the internal behaviours of the light poles E1 and E2. The transitions E1_turn_on_E2 and E1_turn_on_W1 represent E1 sending a notification to the light poles E2 and W1.

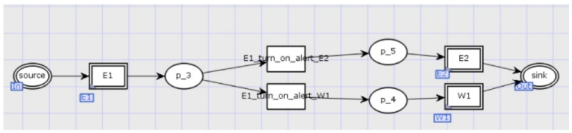

Figure 1: Generated HCPN

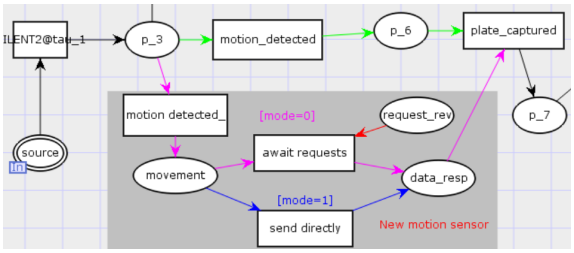

Figure 2: Pre- and post-upgrade CPN model of the entrance light pole. Green represents the data flow of the existing motion sensor, pink represents the new motion sensor in mode 0, and blue represents the new sensor in mode 1.

5.5 Conformance Checking

For conformance checking, we first generated an event log tree from the testing set of event logs. Next, we performed conformance checking of the HCPN against the event log tree (Algorithm 3). The result reveals a deviation "dim <<model move>>" within the CPN module for the entrance pole E1 (the green block in the middle). This deviation indicates that the testing event logs of E1 do not contain the "dim" log entry, whereas it is included in the CPN module.

For deriving a new HCPN, we updated the workflow of the motion sensors on the entrance and exit light poles according to the functionality description of the new motion sensors in Section 5.

5.6 Model Checking Interoperability

To verify whether the new HCPN satisfies the core interoperability requirements of the lighting system, we defined three properties: reachability, liveness, and deadlock. The reachability property evaluates data exchange between motion sensors and cameras of the entrance and exit lighting poles. The liveness property ensures that the worker light poles' lights are turned on when they are notified by the entrance pole. The deadlock check confirms that all workflows in the HCPN never become stuck.

Following the model checking steps described in Section 4.7, the verification results indicated that the HCPN failed all defined properties. By running the simulation function of the CPN Tools model checker, we found the simulation stopped at the sensor block of the entrance pole. As illustrated in Fig 2, the issue was that in the default mode (value = 0), the new motion sensor only sends motion data after receiving a request (red arrow in the diagram), while the camera was waiting for the sensor to actively trigger image capture. This caused an interdependency because the default behaviour of the new sensor was incompatible with the existing capture logic. To resolve this, we changed the new motion sensor to mode 1 (shown in blue), which allows it to actively send detected motion signals and trigger the camera immediately. After this adjustment, we repeated the verification on the updated HCPN, confirming all the properties were satisfied. This demonstrates how real-world interoperability issues can be identified and solutions explored.

6 DISCUSSION

In our approach, case identifiers for event logs are generated using the maximum inlier inter-event time gap. Compared with fixed time windows derived from rule-based approaches or domain knowledge, our method more effectively captures event sequences within a workflow while reducing the influence of extreme outliers.

For automatic HCPN generation, our method leverages log clustering and process mining to construct both device-level and interaction-level CPN modules. The generated HCPN models can be executed directly in CPN Tools for interoperability verification rather than just visualisation. Compared with flat Petri nets, HCPNs enable efficient component upgrades and hierarchical verification at both device and system levels.

For conformance checking on HCPN, we combine modular and integration checking. Modular checking verifies each CPN module against its corresponding event node in the event log tree, while integration checking evaluates sequences of interaction events from both the source and target devices. This combined approach enables the detection of deviations across the HCPN hierarchy, from individual devices to system-level workflows. Our visualisation tool shows deviations throughout the hierarchy, supporting analysis at both model and system scale. This provides a detailed view of process compliance in IoT. In the lighting system study, our framework successfully detected device interoperability issues when motion sensors were replaced with new ones. Beyond this scenario, the framework can be applied to a wide range of interoperability challenges, including mismatched data units, identical data fields with different meanings, outdated APIs, and incompatible communication protocols. Detection of these issues can be performed using the same interoperability properties in the study.

7 CONCLUSION

In this work, we address interoperability verification in IoT systems undergoing device upgrades. Our framework provides a novel method to automatically discover executable HCPNs from raw IoT logs without event correlation identifiers. Event correlations are determined using proposed learning algorithms that identify the optimal time interval between consecutive IoT logs. We also introduce a layer-based conformance checking approach that integrates modular and integration checking to verify the correctness of the HCPNs. Our visualisation tool enables rapid identification of deviations across hierarchical layers. An experiment on a lighting system that requires motion sensor replacement demonstrates the applicability of the framework on IoT interoperability verification. For future work, we plan to investigate methods for transforming natural-language interoperability requirements into formal properties, reducing dependence on domain knowledge.

REFERENCES

Adepoju, A. H., Eweje, A., Collins, A., and Austin-Gabriel, B. (2024). Framework for migrating legacy systems to next-gen data architectures while ensuring seamless integration and scalability. *Intl Journal of Multidisciplinary Research and Growth Evaluation*, 5(6):1462–1474.

An, Y., Wu, N., Zhao, X., Li, X., and Chen, P. (2018). Hierarchical colored petri nets for modeling and analysis of transit signal priority control systems. *Applied Sciences*, 8(1):141.

Benabdelhafid, M., Adi, K., Timo, O. L. N., and Logrippo, L. (2025). Hierarchical colored petri nets for vulnerability detection in software architectures.

De Fazio, R., Balzanella, A., Marrone, S., et al. (2023). CaseID detection for process mining: a heuristic-based methodology. In *International Conference on Process Mining*, pages 45–57. Springer.

Gagliardi, G., Lupia, M., Cario, G., et al. (2020). Advanced adaptive street lighting systems for smart cities. *Smart Cities*, 3(4):1495–1512.

Giese, H., Henkler, S., and Hirsch, M. (2008). Combining formal verification and testing for correct legacy component integration in mechatronic UML. *Lecture Notes in Computer Science*, 5135:248.

Jensen, K. (1987). Coloured petri nets. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *Petri Nets: Central Models and Their Properties*, pages 248–299, Berlin, Heidelberg. Springer Berlin Heidelberg.

Krichen, M. (2023a). Improving formal verification and testing techniques for internet of things and smart cities. *Mobile networks and applications*, 28(2):732–743.

Krichen, M. (2023b). A survey on formal verification and validation techniques for internet of things. *Applied Sciences*, 13(14):8122.

Mangi, F. A., Su, G., and Zhang, M. (2023). PM2PMC: A probabilistic model checking approach in process mining. In *IAS Global Conference on Emerging Technologies (GlobConET)*, pages 1–6. IEEE.

Noura, M., Atiquzzaman, M., and Gaedke, M. (2019). Interoperability in internet of things: Taxonomies and open challenges. *Mobile networks and applications*, 24:796–809.

Seiger, R., Franceschetti, M., and Abbad-Andaloussi, A. (2024). A process to non-invasively augment legacy IoT systems using business processes and microservices. In *Procs of the 14th International Conference on the Internet of Things*, pages 1–9.

Singh, P., Flammini, F., and Caporuscio, M. e. (2020). Towards self-healing in the internet of things by log analytics and process mining. In *30th European Safety and Reliability Conference,(ESREL2020 PSAM15), Italy*, pages 4644–4651.

Tajima, K., Du, B., Narusue, Y., et al. (2023). Step-by-step case id identification based on activity connection for cross-organizational process mining. *IEEE Access*, 11:60578–60589.

Van Der Aalst, W., Adriansyah, A., De Medeiros, A. K. A., et al. (2012). Process mining manifesto. In *BPM 2011 International Workshops,Revised Selected Papers, Part I*, pages 169–194. Springer.

W3C (2017). Semantic sensor network ontology (w3c recommendation). https://www.w3.org/TR/vocab-ssn/. Accessed: 2026-01-26.

Weigert, T., Kolchin, A., Potiyenko, et al. (2019). Generating test suites to validate legacy systems. In *System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0: 11th International Conference, Germany, Proceedings 11*, pages 3–23. Springer.

Yamaguchi, S. and Ahmadon, M. A. B. (2022). A token-replay-based conformance checking method for dataflow in iot system. In *37th Intl Technical Conference on Circuits/Systems, Computers and Communications*, pages 1027–1030. IEEE.

Zheng, J., Papapanagiotou, P., and Fleuriot, J. D. (2022). Alignment-based conformance checking over probabilistic events. *arXiv preprint arXiv:2209.04309*.