Towards a Formal Verification of IoT System Models Designed in the Cisco Packet Tracer Tool

François Siewe
School of Computer Science and Informatics
De Montfort University
Leicester, United Kingdom
fsiewe@dmu.ac.uk

***Abstract*—The Cisco Packet Tracer (CPT) is a network and IoT systems design and simulation tool developed by Cisco Systems inc. The tool is widely used in academia and the industry to develop and simulate complex IoT systems for training purposes. Although the graphical notations of CPT are appealing to system developers, the tool does not support formal verification. Therefore, CPT is limited when it comes to the design and development of safety and secutiry critical IoT systems where faults can endanger people's lives or damage properties and businesses. This paper proposes a formal approach to the verification of IoT system models designed in the CPT tool, based on the Calculus of Context-aware Ambients (CCA). First, the graphical notations of CPT are mapped to processes in CCA that specify their behaviours. Then any desired property of a system can be specified as a Linear Temporal Logic (LTL) formula. Finally, the CCA Runtime Verification tool ccaRV is used to verify formally whether the behaviour of an IoT system model designed in CPT violates the system's requirements. The pragmatics of the proposed approach is illustrated using a case study.**

***Index Terms*—Internet of Things (IoT), Calculus of Context-aware Ambients (CCA), formal verification, simulation, runtime verification**

I. Introduction

The Internet of Things (IoT) technologies are growing fast and are increasingly more sophisticated [1]. It is estimated that over 29 billion IoT-enabled devices worldwide will be connected to the Internet by 2030 [2]. A typical IoT system is built from a multitude of devices of various types (e.g. sensors, actuators, controllers, servers, and network devices) often from different manufacturers, that must interact together through various protocols over the Internet to provide the functionalities of the IoT system. The design and verification of such a system is challenging, especially for safety and security critical IoT systems like smart homes and smart systems for managing national critical infrastructures (e.g. energy, water, and transport). As such, formal methods can play an important role in the design and verification of these complex systems.

Cisco Systems Inc developed a graphical tool, called Cisco Packet Tracer (CPT) [3], for the design, development and simulation of computer networks and IoT system models. This tool is widely used in academia and the industry for training purposes. Although the graphical notations of CPT are appealing to system developers, the tool does not support formal verification. Hence, CPT is limited when it comes to the design and development of safety and security critical IoT systems where faults can endanger people's lives or damage properties and businesses.

This paper proposes a formal approach to the verification of IoT system models designed in the CPT tool, based on the Calculus of Context-aware Ambients (CCA). CCA is a process calculus for the specification and verification of context-aware, mobile, and concurrent systems [4]. First, a formal semantics of the CPT graphical language is proposed in CCA. This allows for any IoT system model designed in CPT to be mapped to a CCA process that describes its behaviours. Then any desired property of a system can be specified as a formula in Linear Temporal Logic (LTL). Finally, the CCA Runtime Verification tool ccaRV is used to verify formally whether the behaviour of an IoT system model designed in CPT violates the system's requirements. The pragmatics of the proposed approach is illustrated using a case study. The contributions of this paper is threefold:

- A novel formal semantics of the graphical language of CPT in CCA is proposed (see Sect. V-A). Each icon in CPT is mapped to an CCA process that describes its behaviour.
- A novel methodology is proposed for the formal verification of IoT systems designed in CPT using CCA (see Sect. V-B). First, the IoT system model in CPT is translated into an CCA process, which is the parallel composition of all the processes corresponding to the devices that compose the IoT system. This CCA process can then be formally verified against any desired property using the CCA runtime verification tool ccaRV.
- A case study is used to illustrate the pragmatics of the proposed methodology (see Sect. VI).

The remaining of the paper is organised as follows. The related works are described in Sect. II. An overview of CPT is presented in Sect. III and that of CCA in Sect. IV. Section V presents the formal verification of the IoT system models developed in the CPT tool. A case study is given in Sect. VI and Sect. VII concludes the paper.

II. Related Work

CPT has been used in many projects to design and simulate IoT systems. Alfarsi et al. proposed the simulation of a smart

2836-3795/26/$31.00 ©2026 IEEE
DOI 10.1109/COMPSAC69091.2026.00309

home in CPT [5]. Finardi presents a practical approach to the simulation of IoT systems in CPT to support the teaching of IoT courses [6]. Sharma et al. use CPT to monitor or simulate network protocols and architectures [7]. Simulation is limited in guaranteeing the correctness of a system because it is often impossible to consider all possible inputs. However, formal verification techniques can help a great deal.

Krichen presents a survey on formal verification and validation techniques for IoT systems [8]. It follows that the Model-Driven Software Engineering (MDSE) approach is widely used in the design and development of IoT systems. Typically, the component diagrams of the Unified Modelling Language (UML) are used to model the architecture of an IoT system, then an Event-B formal specification of the system is derived from the system architecture and verified within the Rodin platform [9]. In our case, a model of an IoT system is designed in CPT tool, then the model is mapped to a process in CCA and verified using the CCA runtime verification tool ccaRV [10], [11].

III. Designing IoT System Models in the CPT Tool

The CPT tool provides a graphical interface for designing models of IoT systems. CPT distinguishes five types of objects that can be used to build IoT systems. These are: (i) network devices (like routers, and switches); (ii) end devices (like home appliances, and user devices); (iii) components (like control boards, sensors, and actuators); (iv) connections, which are lines that represent various types of cables; and (v) environment variables (like temperature, and light). The environment variables are used to model the behaviours of the physical world phenomena, like how the temperature or the light changes over time at a specific place. Each of these objects is represented in the graphical interface by an icon that can be dragged onto the drawing panel. An IoT system is built from a set of devices and components connected through wired or wireless connections. An example of an IoT system model in CPT is given in Fig. 1. This IoT system is a simple intrusion detection system that uses a trip wire sensor to detect motion. When motion is detected the microcontroller must raise an alarm, turn on the light and the webcam to record the scene. Once an IoT system model is built, it can be executed within CPT to simulate the behaviours of the system under various environment contexts. This is of paramount importance as it helps the system developers to test the system before it is physically built. It is even more important if an IoT system model's behaviours could be verified formally against the system's requirements. Section V explains how this can be done using the Calculus of Context-aware Ambients (CCA).

IV. Overview of CCA

CCA is a process calculus for specifying and reasoning about the behaviour of context-aware, mobile, and concurrent systems. The full syntax and the formal semantics of CCA can be found in [4], [10]. CCA is based on a single notion of *ambient*, which conceptualises a place where a computation can happen. An ambient is represented in the form $n[P]$, where

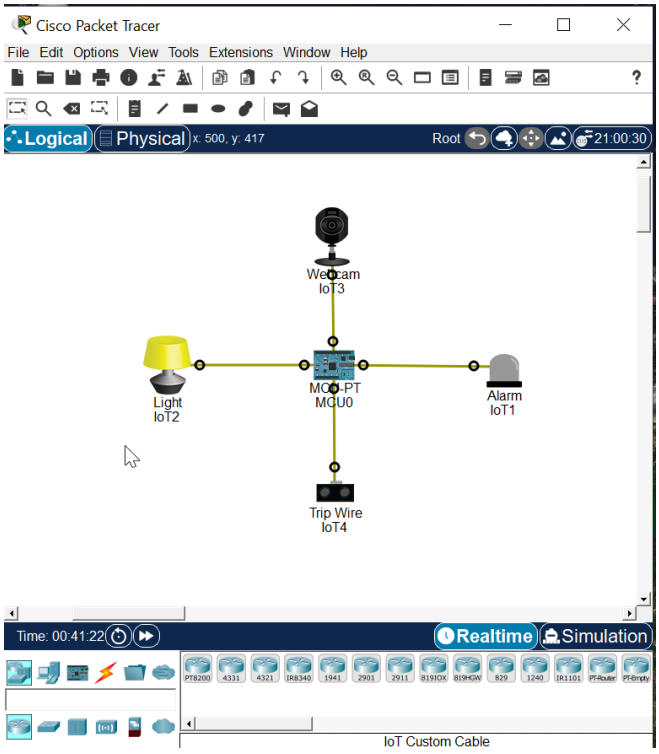


Fig. 1. An example of an IoT system model in CPT

n is the name of the ambient and P is a process describing the behaviour of the ambient. A process is built from a set of elementary actions called capabilities.

A. Capabilities

The syntax of a capability M is given by the grammar in Table I. The capability `skip` represents one transition, i.e. one execution step. An ambient can move into a sibling ambient

TABLE I
Syntax of Capabilities

M	::=	$\texttt{skip} \mid \texttt{in}\ n \mid \texttt{out} \mid \texttt{del}\ n \mid \alpha\ \texttt{recv}(y_1, \ldots, y_\ell)$
		$\mid \alpha\ \texttt{send}(z_1, \ldots, z_\ell) \mid \alpha\ x(z_1, \ldots, z_\ell)$
α	::=	$@ \mid n@ \mid \# \mid n\# \mid :: \mid n :: \mid \epsilon$

n by performing the capability `in` n; and move out of its parent ambient by executing the capability `out`. An ambient can exchange messages with another ambient using the output capability α `send`$(z_1, \ldots, z_\ell)$ to send a list of names $z_1, \ldots, z_\ell$ to a location α, and the input capability α `recv`$(y_1, \ldots, y_\ell)$ to receive a list of names from a location α into the variables $y_1, \ldots, y_\ell$, for some $\ell \geq 0$. The location α can be '@' to mean any parent, 'n@' to mean a specific parent n, '#' to mean any child ambient, 'n#' to mean a specific child n, '::' to mean any sibling, 'n ::' to mean a specific sibling n, or ϵ (empty string) to mean the executing ambient itself. A capability `del` n deletes an empty child ambient n (i.e. $n[\mathbf{0}]$). A capability of the form $\alpha\ x(z_1, \ldots, z_\ell)$ calls the

process abstraction x defined at the location α and the names $z_1, \ldots, z_\ell$ are the actual parameters, $\ell \geq 0$. Capabilities can be composed in a dot-separated sequence to form a process.

B. Processes

The syntax of a process P is given by the grammar in Table II. The process $\mathbf{0}$, aka inactivity process, does nothing

TABLE II
SYNTAX OF PROCESSES

P, Q	$::=$	$\mathbf{0} \mid P\|Q \mid !P \mid n[P] \mid < \kappa > M.P \mid \{P\}$ $\mid$ `if` $< \kappa_1 > M_1.P_1 \ \ldots \ < \kappa_\ell > M_\ell.P_\ell$ `fi` $\mid$ `if` $< \kappa_1 > M_1.P_1 \ \ldots \ < \kappa_\ell > M_\ell.P_\ell$ `else` P `fi` $\mid$ `find` $x_1, \ldots, x_\ell$: κ `for` $P \mid$ `proc` $x(y_1, \ldots, y_\ell)\ P$

and terminates immediately. The process $P|Q$ denotes the parallel composition of the processes P and Q. The replication $!P$ denotes a process which can always create a new parallel copy of P, i.e. $!P$ is equivalent to $P|!P$. The process $n[P]$ denotes an ambient named n whose behaviour is described by the process P. A context expression κ is a logical formula that specifies a property upon the state of the environment. A context-guarded prefix $< \kappa > M.P$ is a process that waits until the environment satisfies the context expression κ, then performs the capability M and continues like the process P. A process of the form $\{P\}$ behaves just like P. An if-then process `if` $< \kappa_1 > M_1.P_1 \ \ldots \ < \kappa_\ell > M_\ell.P_\ell$ `fi` waits until at least one of the context-expressions $(\kappa_i)_{1 \leq i \leq \ell}$ holds; then proceeds non-deterministically like one of the processes $< \kappa_j > M_j.P_j$ for which κ_j holds. An if-then-else process `if` $< \kappa_1 > M_1.P_1 \ \ldots \ < \kappa_\ell > M_\ell.P_\ell$ `else` P `fi` behaves like an if-then process, but does not wait and continues like the process P if none of the branches can be executed.

A search process `find` $x_1, \ldots, x_\ell$: κ `for` P looks for a list of names $n_1, \ldots, n_\ell$ in the context such that the context-expression κ in which each occurrence of x_i is replaced by n_i holds, and continues like the process P in which each occurrence of x_i is replaced by n_i, $1 \leq i \leq \ell$ and $\ell \geq 1$. A process of the form `proc` $x(y_1, \ldots, y_\ell)\ P$ defines a process abstraction named x, whose behaviour is described by the process P. The names $y_1, \ldots, y_\ell$ are the formal parameters of the process abstraction, $\ell \geq 0$.

C. Context-expressions

In CCA, a context is a process with possibly a single hole in it. A context-expression (CE) is a logical formula over the set of all contexts. The syntax of a context-expression κ is given by the grammar in Table III.

TABLE III
SYNTAX OF CONTEXT-EXPRESSIONS

κ	$::=$	$\mathbf{0} \mid$ `true` $\mid$ `false` $\mid n = m \mid$ `this` $\mid n[\kappa] \mid \kappa_1 \mid \kappa_2$ $\mid \kappa_1$ `and` $\kappa_2 \mid \kappa_1$ `or` $\kappa_2 \mid$ `not` $\kappa \mid$ `next` $\kappa \mid$ `somewhere` κ

The CE `true` holds for all context while the CE `false` holds for no context. A CE $n = m$ holds if the names n and m are identical. The CE $\mathbf{0}$ holds for the empty context $\mathbf{0}$. The CE `this` holds solely for the hole context, i.e. the position of the process evaluating that context-expression. Propositional operators such as `not`, `and` and `or` expand their usual semantics to context-expressions. A CE $\kappa_1|\kappa_2$ holds for a context if that context is a parallel composition of two contexts such that κ_1 holds for one and κ_2 holds for the other. A CE $n[\kappa]$ holds for a context if that context is an ambient named n such that κ holds inside that ambient. A CE `next` κ holds for a context if that context has a child context for which κ holds. A CE `somewhere` κ holds for a context if there exists somewhere in that context a sub-context for which κ holds. Some examples of context-expressions that are used later in this paper are given in Table IV.

V. FORMAL VERIFICATION OF IOT SYSTEM MODELS

This section presents a semantic mapping of the CPT graphical notations onto CCA processes, and proposes a methodology for the formal verification of IoT system models designed in CPT.

A. Mapping CPT Icons onto CCA Processes

A semantics of the CPT graphical language is defined as a mapping of CPT icons onto CCA processes (see Table V and Table VI). It follows that an environment variable is mapped onto an ambient that continuously measures (using the processes P1 to PN) and stores (i.e. sends) the current state of the corresponding physical environment phenomenon (see row 1 in Table V). Similarly, a physical environment sensor is represented in CCA as an ambient that reads (i.e. receives) the current state of the physical environment phenomenon from the corresponding environment variable ambient. The sensor then sends this value to any device it is connected to, e.g. a controller (see row 2 in Table V). A controller is a device that receives inputs (namely from sensors) and issues command signals to the devices (namely actuators) it controls. The behaviour of a controller is described by an ambient that iteratively executes a loop function, which receives a series of values from the input pins (i.e. input0 to inputN), performs some computation (i.e. P0 to PK), and then sends the results through the output pins (i.e. out0 to outK), (see row 3 in Table V).

An actuator in CPT is a device that receives a command signal from a controller and changes its state accordingly (e.g. from on to off). This includes devices like a light, a coffee machine, and an alarm. In CCA, an actuator is represented as an ambient that receives a command from a controller ambient and changes its state if necessary. The current state s of an actuator is represented as a child ambient of the form s[0] (see row 4 in Table V). In a client-server architecture, a server is a computer or computer program which manages access to a centralized resource or service in a network. Typically, a server receives a request from a client, performs some action and sends a response back to the client. Thus in CCA, a server can be modelled as an ambient that provides some services (i.e. service0 to serviceN). Each service can be defined as a

TABLE IV
EXAMPLES OF CONTEXT EXPRESSIONS

Context-expression			Description
$has(n)$	=	somewhere(this | n[true] | true)	n is located at self.
$detected()$	=	somewhere(this | next(true) | true)	self has a child ambient.
$detected(n)$	=	somewhere(n[next(true) | true] | true)	n has a child ambient.
$state(s)$	=	somewhere(this | s[0] | true)	self current state is s.
$state(d, s)$	=	somewhere(d[s[0] | true] | true)	s is the current state of the device d.

process abstraction. A service request is a tuple (client, service, params), where params represents a coma-separated list of parameters. When the server receives a service request, it calls the corresponding process abstraction to fulfill the request (see row 5 in Table V).

Table VI gives the mapping for tagging devices, motion sensors, wireless access points, and end devices. Due to the space limit, the mapping for the other network devices such as routers, switches and hubs are not given here, but can be defined in a similar manner using the concept of ambient. Tagging is the annotation of the physical world (places, objects and people) with digital information. Examples of tags include barcodes, QR codes, RFID tags, Bluetooth tags, and Wi-Fi tags. Indeed, when a tag is in the range of a reader, the digital information stored in the tag is passed on to the reader, which in turn can send it to a server for processing. In CCA, a tag and a tag reader can be represented as ambients. The boundary of the reader ambient delimits the range of the reader, i.e. a tag is in the range of a reader if that tag is a child ambient to the reader. Thus, a tag is mapped to an ambient (see row 1 in Table VI) that stores some identification data that it can share with a parent ambient (i.e. a tag reader). A tag reader is represented as an ambient that receives an id number from a tag in its range and sends a service request to a server, then waits for a response from the server (see row 2 in Table VI).

A motion sensor is a device that detects movement in its field of view. In CCA, a motion sensor can be represented as an ambient (see row 3 in Table VI). The boundary of the ambient represents the field of view of the sensor. A movement is detected if a child ambient is present. The context expression $detected()$ defined in Table IV holds if a movement is detected. A wireless access point (WAP) is a network device that allows wireless end devices to connect to a (wired) network. A WAP has a range that is a radius above which wireless connection is not possible. The behaviour of a WAP can be described as follows. A WAP can receive traffics from a wireless end devices and forward them either to the destination wireless end device if connected, or to the wired network interface otherwise. A WAP can also receive traffics from the wired network interface and send them to the destination wireless end device if connected, or simply drop them otherwise. In CCA, a WAP can be modelled as an ambient whose boundary represents the range of the WAP (see row 4 in Table VI). Therefore, a device is said to be in the range of a WAP if that device is a child ambient to the WAP. Wireless communication is modelled as message passing between a parent ambient (i.e. a WAP) and a child ambient (i.e. a wireless device).

An end device is a source or a destination device of data transmitted over a network system. A computer, a phone, a printer, and any other user devices connected to a network are examples of end devices. An end device is connected to the network via a network cable or a wireless interface. In CCA, an end device can be specified as an ambient. A pattern for an end device is given in row 5 in Table VI, where 'network' represents the wired network interface, and P1, P2, Q1, and Q2 are continuation processes. Thus, an end device can receive or send messages through the network.

B. Verification of IoT System Models

An IoT system model in CPT is a collection of devices connected through network cables or wireless connections. This IoT system model can be mapped onto a CCA process, which is the parallel composition of all the ambients corresponding to its devices. The network cables are represented as sibling-to-sibling communications, while wireless connections are described as child-to-parent and parent-to-child communications. The CCA process corresponding to an IoT system model can now be formally verified using the runtime verification tool ccaRV. Readers are referred to [10] for more details about the ccaRV tool. As depicted in Fig. 2, the tool takes in input a CCA process and an LTL formula and produces a verdict on whether the execution of the process violates the formula. Typically, the formula specifies some desired property of the IoT system. The following section illustrates how to use the ccaRV tool to verify an IoT system model designed in CPT.

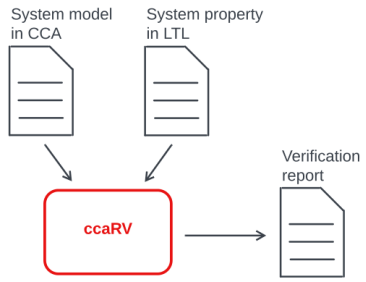


Fig. 2. The CCA runtime verification tool ccaRV [10]

VI. CASE STUDY

Let us consider the simple intrusion detection system depicted in Fig. 1. It is composed of a microcontroller unit

TABLE V
SEMANTICS OF CPT ICONS

	CPT icon	CCA process
1	Environment variables	`env_var_name[ !recv(v). if < true > ::send(v).send(v).0 < cond1 > P1.send(v1).0 ... < condN > PN.send(vN).0 fi | send(v0).0 //initial value v0 ]`
2	Physical environment sensors	`sensor_name[ !recv().env_var_name::recv(t). if < true > ::send(t).send().0 else send().0 // drop the value fi | send().0 ]`
3	Controllers	`controller_name[ proc loop() { // definition !recv(). input0::recv(v0). input1::recv(v1). ... inputN::recv(vN). if < cond0 > P0.out0::send(w0).0 < cond1 > P1.out1::send(w1).0 ... < condK > PK.outK::send(wK).0 else skip.0 fi.send().0 | send().0 } | loop() // call ]`
4	Actuators	`actuator_name[ !recv().controller::recv(x). if < not state(x) > skip.find y: state(y) for del y.send().x[0] else send().0 fi | send().0 | s0[0] // initial state is s0. ]`
5	Servers	`server_name[ proc service0(client, params) { P0.send().0 } | proc service1(client, params) { P1.send().0 } ... | proc serviceN(client, params) { PN.send().0 } | !recv().::recv(client, service, params).service(client, params).0 | send().0 ]`

TABLE VI
SEMANTICS OF CPT ICON (CONT'D)

	CPT icon	CCA process
1	Tags	`tag_name[ !@send(id).0 ]`
2	Tag readers	`reader_name[ !recv().#recv(id).server:: send(reader_name, service, id). server::recv(response).send().0 | send().0 ]`
3	Motion sensors	`motion_sensor_name[ !recv(). if < detected() > controller:: send(1).send().0 < not detected() > controller:: send(0).send().0 else send().0 fi | send().0 ]`
4	Wireless access point	`wap_name[ !#recv(src, dest, params). if < has(dest) > dest#send(src, dest, params).0 else network::send(src, dest, params).0 fi | !network::recv(src,dest,params). if < has(dest) > dest#send(src, dest, params).0 else skip.0 // drop the message fi ]`
5	End devices	`end_device_name[ // Ethernet interface !network::recv(src, dest, params).P1 | network::send(end_device_name, dest, params).Q1 // Wireless interface | !@recv(src, dest, params).P2 | @send(end_device_name, dest, params).Q2 ]`

(MCU0) connected to a trip wire (IoT4), a light (IoT2), a webcam (IoT3) and an alarm (IoT1). Each of these devices can be specified as an CCA process using the semantic mappings given in Table V and Table VI. Then the whole IoT system model corresponds in CCA to the parallel composition of all these processes. A trip wire is a type of motion sensor that can detect movement across a laser beam. Therefore the device IoT4 in Fig. 1 is mapped to the following CCA ambient.

```
IoT4[
   !recv().
   if
    < detected() > MCU0::send(1).send().0
    < not detected() > MCU0::send(0).send().0
   else
    send().0
   fi
   | send().0
   //| user[0]
]
```

To trigger the sensor, one just has to add a child ambient to it, e.g. the ambient `user[0]`.

A light is an actuator device that has three possible states: off, dim, and on. Thus it is mapped to the following ambient. Initially, the light is off.

```
IoT2[
   !recv().MCU0::recv(x).
   if
     < not state(x) > skip.find y:state(y)
       for del y.send().x[0]
   else
     send().0
   fi
  | send().0
  | off[0] // initial state is off.
]
```

Similarly, the webcam IoT3 and the alarm IoT1 are actuator devices with two states: on and off; and so are mapped to ambients identical to the ambient IoT2 above. The microcontroller unit MCU0 is connected in input to one device (i.e. IoT4) and in output to 3 devices (i.e. IoT1, IoT2, and IoT3); therefore it is mapped to the following ambient.

```
MCU0[
   proc loop() { // definition
     !recv().
     IoT4::recv(v).
     if
      < v=1 > IoT1::send(on). IoT2::send(on).
               IoT3::send(on).0
      < v=0 > IoT1::send(off). IoT2::send(off).
               IoT3::send(off).0
     else
      skip.0
     fi.send().0
     | send().0
   }
   |
   loop() // call
]
```

The behaviour of the IoT system model in Fig. 1 is described by the following CCA process, which is the parallel composition of the ambients corresponding to the devices that composed the IoT system.

```
MCU0[...] | IoT1[...] | IoT2[...]
| IoT3[...] | IoT4[...]
```

where '...' denotes the process specifying the behaviour of the corresponding ambient. This CCA process can now be used to formally verify the behaviour of the IoT system against a given property. The experiment environment comprised a 64-bit Toshiba laptop with an Intel Core i7 processor and 16 GB of RAM running a Windows 10 Pro operating system. For example, one may want to check that when a motion is detected, the alarm (i.e. IoT1) will eventually turn on and then off when no motion is detected. This property is specified by the LTL formula:

```
[](detected(IoT4) -> <>state(IoT1,on))
and
[](not detected(IoT4) -> <>state(IoT1,off))
```

The verification report produced by the ccaRV tool for this property is given in Fig. 3. The CCA process is checked against the formula over 2000 runs in order to cover a large number of execution paths. The result shows that no violations were found. The verification results for other formulas are given in Table VII.

VII. Conclusion

In this paper, we have presented a novel approach to the formal verification of IoT system models designed in the CPT tool. A formal semantics of the CPT graphical language is proposed as a semantic mapping of CPT icons onto CCA processes. That is, each device in CPT can be mapped onto an CCA process. The semantics of a given IoT system model in CPT is merely the parallel composition of the processes that correspond to the devices the IoT system model is built from. This process can then be verified formally against any property using the CCA runtime verification tool, ccaRV. A case study is used to illustrate the pragmatics of the proposed approach. In future work, we will extend the approach to include the semantics of network devices like routers, switches and hubs.

```
*****************
  ltl p { ( [] ((detected(IoT4) ->  <> (state(IoT1,on)))) and
[] ((not (detected(IoT4)) ->  <> (state(IoT1,off))))) }
*****************
Total number of failed runs: 0
Total number of passed runs: 2,000
Total number of runs: 2,000
Total number of reduction steps: 200,000
Total number of congruence steps: 211,352
Total verification time: 130,794 milliseconds
Average time per run: 65.40 milliseconds
Standard deviation per run: 8.16 milliseconds
*****************
```

Fig. 3. An example ccaRV verification report

TABLE VII
VERIFICATION RESULTS OF SAMPLE FORMULAS FOR 2000 RUNS

Formula	Verification result
`<>[]state(IoT2,on)`	violations
`[]<>state(IoT2,on)`	violations
`[](state(IoT2,on) -> <> state(IoT3,on))`	no violations
`[](detected(IoT4) -> <> state(IoT3, on))`	no violations

References

[1] T. Sutikno and D. Thalmann, "Insights on the internet of things: past, present, and future directions," *Telkomnika (Telecommunication Computing Electronics and Control)*, vol. 20, pp. 1399–1420, 12 2022.

[2] J. Howarth. (2023) 80+ amazing iot statistics (2024-2030). [Online]. Available: https://explodingtopics.com/blog/number-of-iot-devices

[3] C. P. Tracer, "Cisco packet tracer," *URL: http://www. cisco. com/web/learning/netacad/coursecatalog/PacketTracer. html*, 2013.

[4] F. Siewe, H. Zedan, and A. Cau, "The Calculus of Context-aware Ambients," *Journal of Computer and System Sciences*, vol. 77, no. 4, pp. 597–620, July 2011.

[5] G. Alfarsi, J. Jabbar, R. M. Tawafak, S. I. Malik, A. Alsidiri, and M. Alsinani, "Using cisco packet tracer to simulate smart home," *International Journal of Engineering Research & Technology (IJERT)*, vol. 8, no. 12, pp. 670–674, 2019.

[6] A. Finardi, "Iot simulations with cisco packet tracer," Master's thesis, Helsinki Metropolia University of Applied Sciences, June 2018.

[7] A. Sharma, A. Kumar, and M. Kumar, "Comparison of packet tracer and eve-ng tools for efficient network design," in *2024 11th International Conference on Computing for Sustainable Global Development (INDIACom)*, 2024, pp. 917–919.

[8] M. Krichen, "A survey on formal verification and validation techniques for internet of things," *Applied Sciences*, vol. 13, p. 8122, 07 2023.

[9] A. Saidi, M. Hadj Kacem, I. Tounsi, and A. Hadj Kacem, "A formal approach to specify and verify internet of things architecture," *Internet of Things*, vol. 24, p. 100972, 2023.

[10] F. Siewe, "Runtime verification tool for the calculus of context-aware ambients," *Mathematics*, vol. 13, no. 22, 2025. [Online]. Available: https://www.mdpi.com/2227-7390/13/22/3606

[11] F. Siewe and G. M. Ngounou, "On the execution and runtime verification of UML activity diagrams," *Software*, vol. 4, no. 1, 2025. [Online]. Available: https://www.mdpi.com/2674-113X/4/1/4