Formal Verification of IoT Protocol: In Design-Time and Run-Time Perspective

V. Geetha Lekshmy and Jinesh M. Kannimoola

Abstract IoT systems consist of smart devices ranging from a simple surveillance camera to pacemaker and mission-critical rockets. Though these systems are designed and developed systematically, it may malfunction due to hidden bugs that are uncovered only after deployment. Model checking and run-time verification are well-established procedures in formal methods to ensure the correctness of systems. We combine both these methods together to guarantee that IoT systems deployed in critical scenarios are fail-safe. This work aims at creating an end-to-end verification framework for IoT systems. Our system consists of (1) a design-time model for MQTT protocol based on the system specification, (2) a run-time model extracted from the execution trace of MQTT implementation and (3) the essential features of systems described in the temporal logic specification. The correctness of these models are checked against the specification using model checking and run-time verification approaches.

Keywords Model checking · Runtime verification · IoT systems

1 Introduction

Internet of things (IoT) systems with controllers and actuators have pervaded human lives. Seamless interconnection of heterogeneous devices with minimum computing power gained momentum and began to be used in wide range of applications in health care, flight control systems, security systems, etc. Many of these systems are so critical that they should be prone to few errors [9]. We must ensure the correctness of such systems from the initial stage of software engineering to the deployed

V. Geetha Lekshmy (✉) · J. M. Kannimoola
Department of Computer Science and Applications,
Amrita Vishwa Vidyapeetham, Amritapuri, India
e-mail: geethalekshmy@am.amrita.edu

J. M. Kannimoola
e-mail: jinesh@am.amrita.edu

© The Editor(s) (if applicable) and The Author(s), under exclusive license
to Springer Nature Singapore Pte Ltd. 2021
G. Ranganathan et al. (eds.), *Inventive Communication and Computational Technologies*, Lecture Notes in Networks and Systems 145,
https://doi.org/10.1007/978-981-15-7345-3_74

environment. Formal verification and specification are the building blocks to develop fail-proof critical systems. It ensures that essential features of the system specified in the unambiguous formal languages are holding in the entire life cycle of software development. In this work, we investigate how to combine two verification approaches to guarantee the correctness of IoT systems. We illustrate our approach using a widely used telemetry protocol in the IoT ecosystem.

Model checking [6] is an automated technique of verifying logical properties of a system that is modeled as a finite state machine. Model checking aids to uncover the errors at the design stage, thus preventing the errors to be propagated to the deployment stage. The system properties to be verified are expressed in a specification language and model checkers explore all possible paths of the execution to ensure that key properties are satisfied in all stages of the system. The counterexample generated from model checkers serves as a debugging method on models to understand the path which fails to satisfy the system properties and it helps to patch the design-time flaws in the system. However, model checking suffers from state-space explosion. To overcome this limitation symbolic model checking [4] is used, where states are represented symbolically and not explicitly. In [19], kripke structure (finite state machine) of modeled system is represented with a Boolean formula. SPIN [12] is an open-source explicit state model checker and NuSMV [5] is an open-source symbolic model checker. Run-time verification (RV) [10] analyzes program executions against the program specifications to uncover properties that are not satisfied by the system in run-time. Run-time verification also includes techniques for monitoring the execution of systems and detecting and correcting anomalies, preventing system failure. Unlike model checking, run-time verification focuses on a single run rather than all possible execution paths. IoT system comprises of heterogeneous devices working together and in this scenario proposal of strategies ensuring system correctness is challenging [18].

In this work, we are presenting a framework that combines both model checking and run-time verification for proving the correctness of Message Queue Telemetry Transport (MQTT)protocol. MQTT is one of the popular communication protocols in IoT systems designed for constrained environment. The MQTT protocol is modeled in the PROMELA language, and the key properties of protocol are specified in linear temporal logic (LTL). The SPIN model checker performs the symbolic model checking over the PROMELA model and validate the correctness of LTL specification. Both models and specifications are created from the OASIS standard of MQTT protocol [2]. The run-time model is generated from the execution trace of the implemented system. We are using execution logs of Mosquito MQTT broker to produce the state-space representation of MQTT system. We assure that the run-time model is also compliant with LTL properties.

The remainder of this paper is organized as follows. Section 2 discusses the related literature in this field; Section 3 detailed our methodology; Section 4 presents verification of the system; and finally, Section 5 presents conclusions and areas of further work.

2 Related Work

Hinrichs et al. [11] in their positional paper convey the idea of combining model checking and verification as a single technique. They have illustrated examples of systems where algorithm correctness is checked with model checking and data structure correctness is checked at run-time. In this system, there is a clear separation of properties that could be verified using model checking and run-time verification. The work [15] extends a model checker DIVINE to support run-time verification. DIVINE model checker has two modes, run and verify. In run mode, a single execution of the program is explored, where assertions and all behaviors are checked. In verify mode along with the program, an environment that contains a stand-in operating system is given as input to the model checker.

Ankush Desai et al. proposed a framework [7] that combines both model checking and run-time verification for developing robotic systems. The key properties to be satisfied by the robotic software systems are specified in a high-level language P and the system is model checked. Here, model checking is done with assumptions about the correctness of interfaces of the verified software with the physical world and other software components. These assumptions that are to be satisfied by the interfaces are specified in signal temporal logic (STL) and it is monitored at run-time and the result is given as feedback to the software stack of a robotic system for decision making. Samir Ouchani in [22] formally models an IoT system using process algebra. This IoT system model is verified using PRISM (a model checker). Key system properties checked are written in probabilistic computational tree logic (PCTL). Torjusen et al. [24] proposed a technique where run-time verification enablers are included in adaptive security for smart IoT (ASSET) in e-health-based IoT systems. Adaptive security systems learn and adapt dynamically, by changing the parameters/structure of the system. It predicts threats of the system and makes security decisions to overcome the threats.

A monitor for run-time verification of IoT systems using Constraint Application Protocol (CoAP) is described in [14]. This non-intrusive, passive monitor captures CoAP messages in a particular network and generates simple events based on it. These events are articulated with event calculus and then processed by the complex event processing engine to find out abnormal behavior. İnçki and Ari [13] extend this approach where the activity/interaction of nodes in the IoT system is considered as events and modeled using message sequence charts. This message sequence chart is processed and behavior of the system is expressed as formulae of event calculus. From this, complex event patterns are generated as event processing language (EPL) which is given as input to Esper engine (Java-based complex event processing engine) for verification. Leotta et al. in [17] have used UML state machines to specify the expected system behavior. This specification is converted to trace expression manually. This is translated to prolog clauses, which is given as input to a run-time monitor. The monitor observes the system execution and records the events. Then, the trace is compared with the specification for detecting abnormal behavior.

Aktas et al. [1] proposed a system for run-time verification with self-healing capability in the IoT domain. There are two external services, self-healing service and predictive maintenance service, which the system under test interacts. Events in the IoT system are captured along with provenance information about IoT devices and sent to the run-time verification system for detection of faults. On detecting any abnormality, verification system triggers services for corrective actions by the self-healing system. Run-time verification of timed properties is proposed by Pinisetty et al. [23]. Given both expected and observed timed property of the system, a monitor is generated that takes the current execution of the system and predicts whether this execution satisfies or violates the given property in the future. In [16], Caroline Lemieux et al. proposed a mining tool, Texada that mines the traces and extracts properties that satisfy the LTL specification. The system takes logs and LTL formulae and outputs instances of LTL formulae, where the atomic propositions in the LTL formulae are replaced by events in the log. The property instances generated are checked for their validity on all traces.

3 Methodology

The proposed system consists of two components, (1) model checker and (2) run-time verifier. Figure 1 illustrates the overall architecture of our proposed system.

For the illustration, we include the subset of MQTT protocol [3] such as connection establishment, publish and subscribe. In the model checking component, a model is constructed from the specification and correctness properties are specified in linear temporal logic (LTL). In the run-time verification component, an existing stable implementation of MQTT protocol is executed and the events and client-server interactions are recorded in logs. We generate the state space from the log and the same set of correctness properties specified in design-time modeling are verified.

A brief description of tools/languages used to develop this system is given below. **SPIN Model checker**: SPIN [12] is a tool for analyzing the consistency of concurrent systems. Promela (process meta-language) is a modeling language used in the SPIN that facilitates modeling of concurrent processes and its communication. The model described in the Promela consists of proctype definitions that correspond to the behavior of processes in the system. The communication between these processes is realized using channels. Each proctype definition in Promela corresponds with a transition system. The global behavior of system is represented as a single transition system, obtained by interleaving individual transition for each proctype. SPIN verifies the correctness of a concurrent system by executing the process definitions and validating the system against correctness claims specified in linear temporal logic (LTL) and assert statements. SPIN generates counterexample for the failed claims and it helps to improve the model. LTL is a model temporal logic to specify how the behavior of the system changes over time. LTL claims are converted to Buchi automaton in SPIN model checker [8].

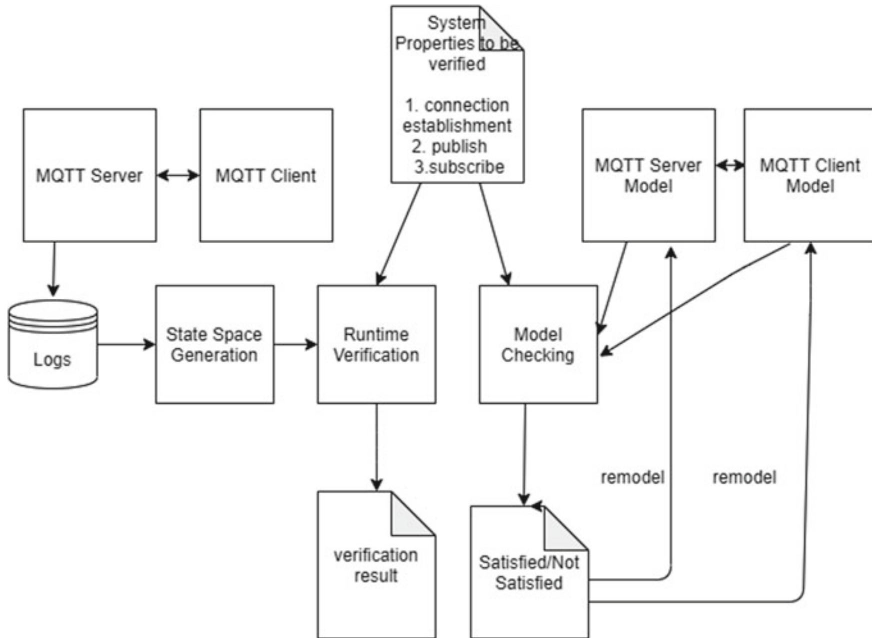

Fig. 1 System architecture

MQTT Protocol: Message Queue Telemetry Transport (MQTT) protocol [3] is a publish-subscribe protocol used for communication between nodes in an IoT system. This protocol requires less memory footprint, thus suitable for resource-constrained devices. MQTT protocol operates in a pre-defined way. Every node (client) communicates with a broker (server) by exchanging control packets. In this work, we deal with the following properties of MQTT.

1. Connection Establishment: This process requires exchange of CONNECT and CONNACK packets between broker and node. For every connect request from the node, the broker should respond with CONNACK with appropriate response code.
2. Subscribe Message: A node can subscribe to a topic by sending SUBSCRIBE packet to the broker, and the broker must respond with a SUBACK with appropriate reason code. The message delivered by the broker depends on the quality of service (QoS) in subscribe and QoS associated with the published message.
3. Publish Message: A node can publish a topic by sending PUBLISH packet containing information including data, QoS, etc. Broker should respond appropriately depending on QoS. This protocol defines three delivery semantics based on quality of service (QoS) expected out of the protocol.

   a. QoS = 0, "at most once" where the message is sent once and there is no guarantee that the receiver will receive it.

b. QoS = 1, "at least once" where the message is guaranteed to be delivered at least once. However, node may receive duplicates of the message.
c. QoS = 2, "exactly once" ensures that the message will be received by the client exactly once, without duplication.

MQTT protocol is lightweight and is suited for systems with component nodes communicating asynchronously. However, it supports only small payload and is not appropriate for systems which require large data sequences.
Mosquitto: Mosquitto [21] is an open-source MQTT broker. A broker acts as an intermediate entity between two communicating nodes. Nodes send data pertaining to a particular topic to the broker. The broker forwards the message to those nodes who have subscribed the topic. The performance of different publicly deployed MQTT brokers is evaluated in [20]. The brokers are tested for their message load handling for 1000 real-time messages subscribed over a topic, which takes least time to deliver messages from server to client at QoS level 0 and 2.

3.1 Design-Time Modeling

We have modeled node as well as broker as proctype in Promela. Communication between these processes occurs through two synchronous Promela channels and defined as:

```
chan ctob=[0] of {int,mtype};
chan btoc=[0] of {int,mtype};
```

The client process sends a connect request (CONNECT) over channel *ctob* along with its identification (clientid). This clientid is used to uniquely identify a client in the server side. The server responds with an acknowledgement (CONNACK) with appropriate response code over channel *btoc*. Similar to CONNECT request, it sends a publish (PUBLISH)/subscribe (SUBSCRIBE) request to the server and receives an acknowledgement (PUBACK/SUBACK) back. The following code snippet gives client proctype implementation.

```
/*Type of messages defined*/
mtype{CONNECT,CONNACK,ERROR,PUBLISH,SUBSCRIBE,
DISCONNECT,SUBACK,PUBACK};
/*Defining channels for communication
from client to server and vice-versa*/
chan ctob=[0] of {int,mtype};
chan btoc=[0] of {int,mtype};
proctype client(int client_id){
 int c_id;
 mtype m1,m2;
```

```
 bool con=0; bool cack=0;
 bool sub=0; bool pub=0;
 bool suback=0; bool puback=0;
 do
  ::con==0 ->
  atomic{m1=CONNECT;ctob!client_id,m1; con=1;}
  ::con==1 ->btoc??eval(client_id),m2;
  if
   :: m2==CONNACK ->cack=1; m2=ERROR;
   :: m2==SUBACK -> printf("suback received");suback=1;
   :: m2==PUBACK ->printf("pub ack received");puback=1;
   :: else skip;
  fi;
  ::(con==1)-> ctob!client_id,PUBLISH;pub=1;
  ::(con==1)->  ctob!client_id,SUBSCRIBE;sub=1;
 od;
}
```

Following is the model of MQTT broker that responds to the CONNECT, SUBSCRIBE and PUBLISH. Server identifies each client using a clientid and keeps track of its CONNECT request. All clients share a common channel with the server for communication, and a particular client reads a message addressed with its own clientid.

```
connected con_clients[5];
proctype broker(){
 mtype m1,m2;
 int client_id;
 do
  :: ctob?client_id,m1  ->
  if
   :: m1==CONNECT ->
      con_clients[client_id].con_request=1 ;
   m2=CONNACK; btoc!client_id,m2;
   con_clients[client_id].con_ack=1;
  ::else
    printf("process subscribe/publish");
   if
    :: m1==SUBSCRIBE -> m2=SUBACK;btoc!client_id,m2;
    :: m1==PUBLISH -> m2=PUBACK;btoc!client_id,m2;
   fi;
  fi;
 od;
}
```

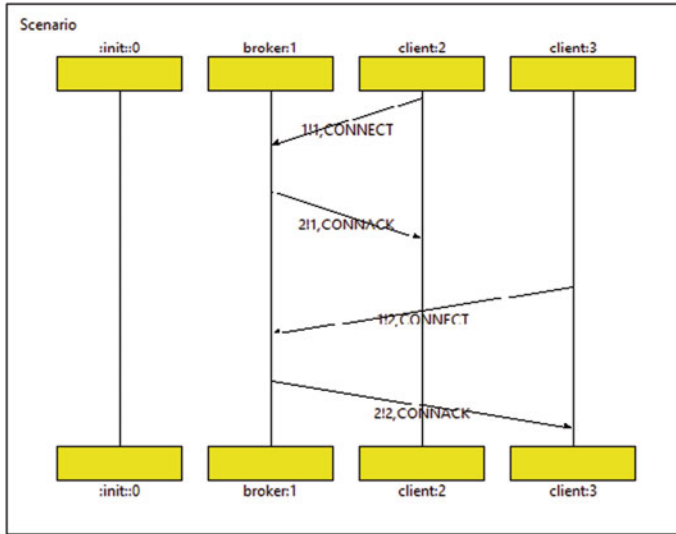

Fig. 2 Sequence of MQTT client-broker interaction in SPIN

This model can accept up to five clients. The following *init* process creates two clients and a broker. The broker waits for the request from the clients and send appropriate response to client which is read by corresponding client.

```
init{
  run broker();
  run client(0);
  run client(1);
}
```

Figure 2 illustrates the working of MQTT client-broker model developed with two clients sending connect requests and receiving appropriate responses in SPIN.

3.2 Run-Time Modeling

Run-time data is extracted from the execution trace of the MQTT implementation. As we discussed, we are collecting log of Mosquitto MQTT broker. It includes all the activities like connection establishment, message subscription, etc. Figure 3 shows the example of Mosquitto log. This log is created by running an instance of Mosquitto MQTT broker and instances of MQTT clients. The clients include instance of MQTT spy (an open-source utility to monitor MQTT activity) and MQTT client implemented in Java using Paho Java client (an MQTT client library in Java).

```
1580272791: Sending CONNACK to Geethalekshmi080225260 (0, 0)
1580272793: New connection from 127.0.0.1 on port 1883.
1580272793: New client connected from 127.0.0.1 as Geethalekshmi095217161 (p2, c1, k60).
1580272793: No will message specified.
1580272793: Sending CONNACK to Geethalekshmi095217161 (0, 0)
1580272821:   PUBLISH from Geethalekshmi080225260 (d0, q2, r1, m1, 'data', ... (3 bytes))
1580272821: Sending PUBREC to Geethalekshmi080225260 (m1, rc0)
1580272821: Received PUBREL from Geethalekshmi080225260 (Mid: 1)
1580272822: Sending PUBCOMP to Geethalekshmi080225260 (m1)
1580272839: Received SUBSCRIBE from Geethalekshmi095217161
1580272839:    data (QoS 2)
1580272839: Geethalekshmi095217161 2 data
1580272839: Sending SUBACK to Geethalekshmi095217161
1580272839: Sending PUBLISH to Geethalekshmi095217161 (d0, q2, r1, m1, 'data', ... (3 bytes))
1580272839: Received PUBREC from Geethalekshmi095217161 (Mid: 1)
1580272839: Sending PUBREL to Geethalekshmi095217161 (m1)
1580272839: Received PUBCOMP from Geethalekshmi095217161 (Mid: 1, RC:0)
```

Fig. 3 Snap shot of log file of MQTT server

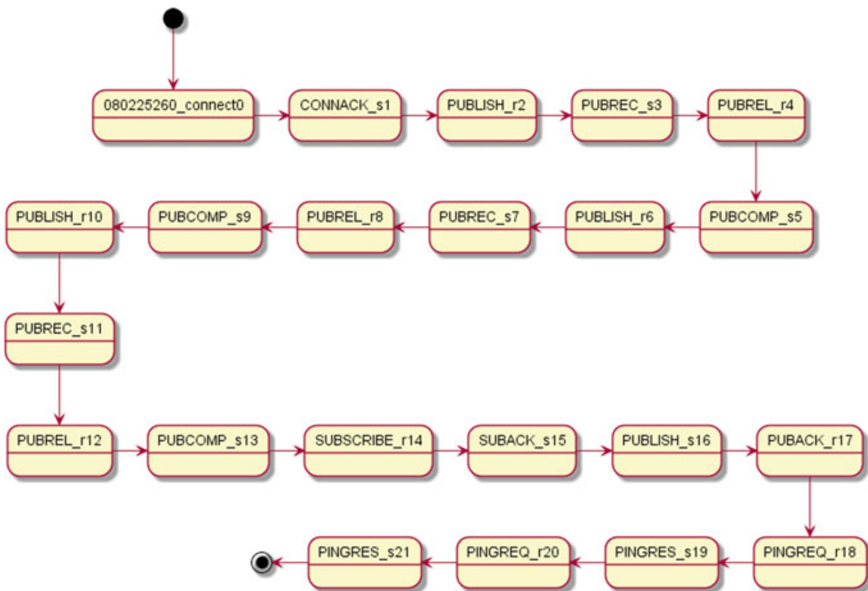

Fig. 4 State diagram of MQTT broker

We have used a Python parser to obtain the run-time model, it is essentially state space over the linear time. Figure 4 is a example of such state space generated from log.

4 Verification

We have verified the following properties in our design-time and run-time model.

1. Connection establishment—A broker that receives a CONNECT control packet should return a CONNACK with appropriate response code. A node should sent only one CONNECT request to the broker after network connection is established.
2. a. A publish message with QoS = 1 should be responded with PUBACK message
   b. A publish message with QoS = 2 should follow a sequence of control packets as given in Fig. 5.
3. A subscribe message received by the broker with QoS $\geq$ 1 should be responded with SUBACK message with appropriate response code.

We have expressed this properties as the following LTL statement.

$$\Box(\text{con_request_client1} \Rightarrow \Diamond\text{con_response_client1})$$

$$\Box(\text{con_request_client2} \Rightarrow \Diamond\text{con_response_client2})$$

This specification ensures that a connection request received by the broker at any time should be followed by a response at any time in the future. For example, con_request_client1 is the variable that keeps track of request from client1 and con_response_client1 is a variable for tracking response to client1.

$$\Box(\text{pub_client1} \ \& \ \text{pub_client1_qos}) \Rightarrow \Diamond\text{puback_client1})$$

$$\Box(\text{pub_client2} \ \& \ \text{pub_client2_qos}) \Rightarrow \Diamond\text{puback_client2})$$

In this specification variable, pub_client1 keeps track of publish request from client1 and variable pub_client1_qos stores the quality of service associated with the publish request. If a publish request arrives and its QoS value is 1, then variable puback_client1 should be set to true, where puback_client1 keeps track of PUBACK message that is sent to client1. The similarly we can define the second statement for client2

$$\Box((\text{publish_r} \Rightarrow \Diamond\text{pubrec_s}) \Rightarrow \text{pubrel_r} \Rightarrow \Diamond\text{pubcomp_s})$$

This specification ensures if a PUBLISH message is received by the broker from a client, it will respond with a PUBREC message, which will in-turn be followed by receiving of PUBREL and sending of PUBCOMP messages.

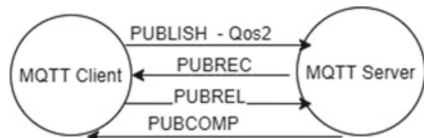

Fig. 5 Publish message with QoS 2

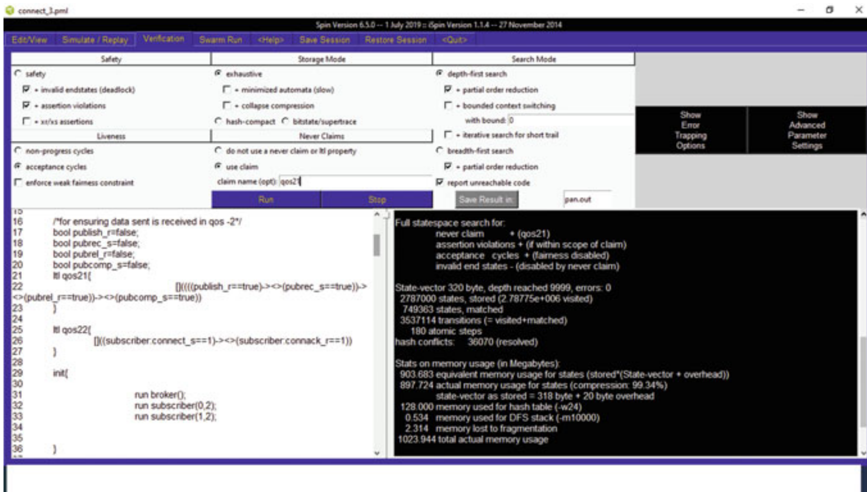

Fig. 6 Verification in iSpin

The verification process ensure that both design-time model and run-time model are satisfied the above-mentioned LTL properties.

Design-time Verification: We verify satisfiability of LTL statement in Promela model. SPIN model checker performs the verification process over the model using symbolic model checking. If the property is not satisfied by model, SPIN generates the counterexample and it gives execution path leads to the failure. Figure 6 gives an example of SPIN verification.

Run-time verification: It checks whether all the essential features designed are actually implemented and it is behaving as expected when the system is deployed. LTL properties are checked for compliance with the state space of a running MQTT implementation. The broker-client implementation we selected satisfies all the specified LTL properties.

5 Conclusion and Future Scope

This work is a step toward the integration of model checking and run-time verification in IoT systems. We have developed design-time and run-time model for MQTT protocol and verify the correctness. We used LTL language to specify the correctness properties and it helps to eliminate the ambiguity in the specification. The challenge involved in this approach is to make the process work seamlessly from model checking to run-time verification. This work can be extended to generate a monitor automatically from LTL specification so that run-time verification will be fully automated.

References

1. Aktas MS, Astekin M (2019) Provenance aware run-time verification of things for self-healing internet of things applications. Concurr Comput: Pract Exp 31(3):e4263
2. Andrew Banks KB, Briggs Ed, Gupta R (2019) MQTT Version 5.0
3. Banks A, Briggs E, Borgendale K, Gupta R (2019) MQTT Version 5.0. OASIS Standard
4. Burch JR, Clarke EM, McMillan KL, Dill DL, Hwang L-J (1992) Symbolic model checking: 1020 states and beyond. Inf Comput 98(2):142–170
5. Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) Nusmv 2: an opensource tool for symbolic model checking. In: International conference on computer aided verification. Springer, pp 359–364
6. Clarke EM (1997) Model checking. In: International conference on foundations of software technology and theoretical computer science. Springer, pp 54–56
7. Desai A, Dreossi T, Seshia SA (2017) Combining model checking and runtime verification for safe robotics. In: 17th international conference on runtime verification (RV), pp 172–189
8. Gastin P, Oddoux D (2001) Fast LTL to Büchi automata translation. In: International conference on computer aided verification. Springer, pp 53–65
9. Hassan WH et al (2019) Current research on internet of things (IoT) security: a survey. Comput Netw 148:283–294
10. Havelund K, Roşu G (2018) Runtime verification-17 years later. In: International conference on runtime verification. Springer, pp 3–17
11. Hinrichs TL, Sistla AP, Zuck LD (2014) Model check what you can, runtime verify the rest. In: HOWARD-60, vol 42, pp 234–244
12. Holzmann GJ (1997) The model checker SPIN. IEEE Trans Softw Eng 23(5):279–295
13. İnçki K, Ari I (2018) A novel runtime verification solution for IoT systems. IEEE Access 6:13501–13512
14. İnçki K, Arı İ, Sözer H (2017) Runtime verification of IoT systems using complex event processing. In: 2017 IEEE 14th international conference on networking, sensing and control (ICNSC). IEEE, pp 625–630
15. Kejstová K, Ročkai P, Barnat J (2017) From model checking to run-time verification and back. In: International conference on runtime verification. Springer, pp 225–240
16. Lemieux C, Park D, Beschastnikh I (2015) General LTL specification mining. In: 2015 30th IEEE/ACM international conference on automated software engineering (ASE). IEEE, pp 81–92
17. Leotta M, Ancona D, Franceschini L, Olianas D, Ribaudo M, Ricca F (2018) Towards a runtime verification approach for internet of things systems. In: International conference on web engineering. Springer, pp 83–96
18. Leotta M, Clerissi D, Franceschini L, Olianas D, Ancona D, Ricca F, Ribaudo M (2019) Comparing testing and runtime verification of IoT systems: a preliminary evaluation based on a case study. In: Proceedings of the 14th international conference on evaluation of novel approaches to software engineering. SCITEPRESS—Science and Technology Publications, Lda, pp 434–441
19. McMillan KL (1993) Symbolic model checking. Springer, Boston, MA, pp 25–60
20. Mishra B (2018) Performance evaluation of MQTT broker servers. In: International conference on computational science and its applications. Springer, pp 599–609
21. Mosquitto E (2018) An open source MQTT broker. Eclipse Mosquitto ™[cit. 2018-04-23]. Dostupné z: Mosquitto.org
22. Ouchani S (2018) Ensuring the functional correctness of IoT through formal modeling and verification. In: International conference on model and data engineering. Springer, pp 401–417
23. Pinisetty S, Jéron T, Tripakis S, Falcone Y, Marchand H, Preoteasa V (2017) Predictive runtime verification of timed properties. J Syst Softw 132:353–365
24. Torjusen AB, Abie H, Paintsil E, Trcek D, Skomedal Å (2014) Towards run-time verification of adaptive security for IoT in eHealth. In: Proceedings of the 2014 European conference on software architecture workshops. ACM, p 4