Verification of the MQTT IoT Protocol using Property-specific CTL Sweep-Line Algorithms

Alejandro Rodríguez, Lars Michael Kristensen and Adrian Rutle

Dept. of Computer Science, Electrical Engineering, and Mathematical Sciences,
Western Norway University of Applied Sciences, Bergen, Norway
{arte,lmkr,aru}@hvl.no

Abstract. MQTT is a publish-subscribe communication protocol being increasingly used for implementing internet-of-things (IoT) applications. In earlier work we have developed a formal and executable model of the MQTT protocol using Coloured Petri Nets (CPNs) and performed an initial verification of behavioural properties. The contribution of this paper is to investigate the use of the sweep-line method for verification of the MQTT CPN model in order to alleviate the effect of the state explosion problem. We formulate the behavioural properties using Computation Tree Logic (CTL) and show how to formulate a progress measure for the sweep-line method based on the main phases of the MQTT protocol. To perform the verification of properties, we provide some property-specific CTL model checking algorithms compatible with the sweep-line method.

1 Introduction

The development of distributed software systems is challenging, and one of the main approaches to tackle the challenges is to build an executable model of the system prior to implementation and deployment. Coloured Petri Nets (CPNs) [13] is a formal modelling formalism convenient for specifying complex concurrent and distributed systems. CPN Tools [9, 15] is a software tool that supports the construction, simulation (execution), state space analysis, and performance analysis of CPN models. One of the key functionalities of CPN Tools is the ability to perform model checking [1] of the modelled system. This means that one can generate the state space (the set of reachable states) of a system in order to verify key behavioural properties. Temporal logics [23] such as Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) are widely used to express behavioural properties of systems.

MQTT [2] is a publish-subscribe messaging protocol for IoT suited for constrained application domains such as Machine-to-Machine communication (M2M) and IoT contexts. MQTT is designed with the aim of being light-weight and easy to implement. In earlier work [19], we have developed a formal and executable specification of MQTT motivated by the fact that until now, the protocol has only been specified using an (ambiguous) natural language specification. MQTT contains relatively complex protocol logic for handling connections, subscriptions, and quality of service levels related to message delivery.

Our initial verification experiments was conducted using ordinary full state spaces and clearly highlighted the presence of the state explosion problem [8, 22]. This was caused by the exponential growth in the number of reachable states of the system with respect to the number of clients, packets, and topics. A large part of the model checking research has aimed at developing techniques for alleviating this inherent complexity problem. This includes several different families of reduction methods such as partial-order reduction methods [7] that reduce the number of interleaving execution considered, and hash compaction [21] which provides a compact representation of states with a small probability of not covering the complete state space. Since the amount of memory is often the limiting factor in model checking, we focus on the family of methods that combat state explosion by deleting states from memory during state space exploration. Specifically, we consider the sweep-line method [12] which is based on the idea of exploiting a notion of progress exhibited by many systems. We focus on CTL because CPN Tools implements a CTL-based temporal logic called ASK-CTL [3] which enables queries taking into account both state and event information. Furthermore, CTL is able to capture the behavioural properties of interest for the MQTT protocol.

The contribution of this paper is twofold: (1) the implementation of the sweep-line method using the Standard ML (SML) language together with the ability of performing model checking of certain behavioural properties specified using tailored CTL sweep-line model checking algorithms based on [17]; and (2) the application of sweep-line based CTL model checking to our CPN model of the MQTT IoT protocol. It should be noted that there already exists work on LTL model checking using the sweep-line method [10], but several of the behavioural properties that we aim to verify for MQTT are true CTL properties, i.e., not expressible in LTL [22, 24].

The rest of this paper is organised as follows. In Section 2 we introduce the sweep-line method and in Section 3 we provide the property-specific CTL model checking algorithm that we employ for the verification. Section 4 gives a brief review of the CPN model of the MQTT protocol. We describe the experiments carried out and the results obtained in Section 5. Finally, in Section 6, we sum up the conclusions and outline directions for future work. The reader is assumed to be familiar with the basic concepts of CPNs and CTL model checking techniques. This paper is based upon the workshop paper [20] and the conference paper [17].

2 The Sweep-Line State Space Exploration Method

The sweep-line method [4] is aimed at systems for which it is possible to define a measure of progress based on the states of the system. A progress measure maps each state of the system into a *progress value* and is in most cases specific for the system under consideration. In this paper, we consider the version of the sweep-line algorithm for *monotonic progress measures*. The key property of a monotonic progress measure is that for any given state s, all states reachable from s have a progress value which is greater than or equal to the progress value

of s. This means that a monotonic progress measure preserves the reachability relation. Having defined a progress measure of the system makes it possible to organise the state space into *layers* such that states that share the same progress value belong to the same layer.

The basic idea of the sweep-line method is to explore the state space in a least-progress-first order, one layer at a time, such that once all states in a given layer have been processed, they are removed from memory and the exploration proceeds to the next layer [12]. In conventional state space exploration, the states are kept in memory to recognise already visited states. However, a monotonic progress measure guarantees that states which have a progress value that is strictly less than the minimal progress value of those states for which successors have not yet been calculated can never be reached again. It is therefore safe to delete such states from memory which significantly reduces the memory usage during the state space exploration.

The progress exploited by the sweep-line method and formalised in the form of a progress measure is defined below in Definition 1 where S denotes the set of system states, $s_0 \in S$ denote the initial state, $s \rightarrow^* s'$ denotes that $s' \in S$ is reachable from $s \in S$ via some number of transitions, and $reach(s_0)$ the set of states reachable from the initial state.

Definition 1 (Monotonic Progress Measure). A **monotonic progress measure** is a tuple $\mathcal{P} = (O, \sqsubseteq, \Psi)$ such that O is a set of **progress values**, $\sqsubseteq$ is a total order on O, and $\Psi : S \rightarrow O$ is a **progress mapping** such that $\forall s, s' \in reach(s_0) : s \rightarrow^* s' \Rightarrow \Psi(s) \sqsubseteq \Psi(s')$. □

A progress measure is non-monotonic when there is at least one *regress edge*, i.e., an edge where the source state has a larger progress value than the destination state. A generalised version of the sweep-line method that can handle non-monotonic progress measures and regress edges also exists [14], but is not the focus of our work. It was already proved [12] that the sweep-line method guarantees full coverage of the state space, and in the case of a monotonic progress measure it terminates after having explored each reachable state once. In the case of a non-monotonic progress measures, termination is still guaranteed but some states may be explored multiple times.

Algorithm 1 based on [12] specifies the sweep-line algorithm for monotonic progress measures. The algorithm starts with a hash table of visited states and a priority queue on progress values containing the states that are still to be processed. Both are initialized at the beginning with the initial state s_0 (lines 2-3). The progress value for the current (initial) layer ψ_c is also initialized in line 4. Then, the algorithm executes a loop (lines 5-28) which ends when all the reachable states have been processed. For each iteration, we select one of the states with the lowest progress value among the unprocessed states (line 6). The condition in line 7 checks if the progress value of the layer is strictly less than the progress value of the selected state; if so, we are about to move into the next layer. This is the point where we invoke the property-specific CTL model checking algorithm for the property Φ using the CHECKPROPERTY procedure at

```
   Data:
   Nodes ▷ Hash table of visited states currently stored.
   Unprocessed ▷ Priority queue of unprocessed states.
   Layer ▷ List of states processed in the current layer.
   ψc ▷ Progress value for current layer.
   Φ ▷ Property to be verified.
   Result: True if the property is satisfied, false otherwise.
 1 begin
 2     Nodes.insert(s0)
 3     Unprocessed.insert(s0)
 4     ψc ⟵ ψ(s0)
 5     while ¬(Unprocessed.isEmpty()) do
           /* node with lowest progress measure                    */
 6         s ⟵ Unprocessed.getMinElement()
 7         if ψc ⊏ ψ(s) then
 8             if ¬ (checkProperty(Layer, Φ)) then
 9                 return false
10             end
11             forall s′ ∈ Layer do
12                 Nodes.delete(s′)
13             end
14             Layer ⟵ ∅
               /* Update progress measure for current layer        */
15             ψc ⟵ ψ(s)
16         end
17         Layer.insert(s)
           /* For every successor state of s                       */
18         forall (t, s′) such that s -t-> s′ do
19             if ¬(Nodes.contains(s′)) then
20                 Nodes.insert(s′)
21                 if (ψ(s) ⊐ ψ(s′)) then
22                     RaiseException('Regress edge found')
23                 else
24                     Unprocessed.insert(s′)
25                 end
26             end
27         end
28     end
29     return true
30 end
```

Algorithm 1: Sweep-line algorithm for monotonic progress measures

line 8. If the CHECKPROPERTY determines that the property is violated, then we return false and the algorithm stops. The implementation of CHECKPROPERTY is the subject of the next section. In line 18, we use $s \xrightarrow{t} s'$ to denote that the transition t is enabled in state s, and that the occurrence of t in s leads to the state s'. If the property is never violated the algorithm returns true at the end of the execution (line 29).

3 CTL Property Checking Algorithms

CTL [5] is an important branching temporal logic that is sufficiently expressive for the formulation of an important set of behavioural system properties. Even though a large set of properties can be specified using the semantics of CTL, there are some restrictions when applying them with the sweep-line method algorithm. The challenge of combining CTL model checking with the sweep-line method is that conventional algorithms for CTL model checking propagate information backwards from a state to its predecessors [6]. This follows the opposite workflow than the forward progress-first exploration that the sweep-line method performs.

In this paper, we do not consider the full CTL, but only formulas of the $AG\{EF, AF\}$-fragment that can be obtained from the following grammar, where p as an atomic state proposition and ϕ is called a *state predicate*:

$$\begin{aligned}\Phi &::= \mathbf{AG}\,\psi \mid \psi \\ \psi &::= \mathbf{EF}\,\phi \mid \mathbf{AF}\,\phi \mid \phi \\ \phi &::= p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi\end{aligned}$$

The formulas expressing behavioural properties to be verified are interpreted over the paths of the state space as informally explained below:

Property - AG ψ "Invariantly", which holds if ψ holds in all states that are reachable from the current state.

Property - EF ϕ "Holds potentially" or "possibly", which holds if it is possible to find a state reachable from the current state where ϕ holds.

Property - AF ϕ "Holds eventually" which holds if from the current state, a state satisfying ϕ is always eventually reached.

Property - AG EF ϕ "Always possible", which holds if from any state reachable from the current state, a state satisfying ϕ can always be reached.

Property - AG AF ϕ "Always eventually", which holds if from any state reachable from the current state, a state satisfying ϕ is always eventually reached.

We say that a formula (property) Φ holds if Φ holds in the initial state s_0. To model check the **AF EF** and **AG AF** properties, we exploit the set of *strongly*

connected components (SCC). A strongly connected component of a directed graph is a maximal subgraph determined by nodes that are mutually reachable. A strongly connected component is *terminal* if no states in the component has outgoing edges to states in other components. It should be noted that when checking the **AG AF** and **AF** properties we implicitly add a self-loop to any terminal states, i.e., (deadlocked) states without enabled transitions.

Because of the monotonicity of the progress measure, each strongly connected component only contains nodes belonging to the same layer and is hence always contained in a single layer. This is formally stated in the proposition below.

Proposition 1. *Let $\mathcal{P} = (O, \sqsubseteq, \psi)$ a monotonic progress measure, SCC be the set of strongly connected components, and let $scc \in SCC$ be a strongly connected component. Then: $\forall s, s' \in scc : \psi(s) = \psi(s')$.*

Proof. Assume that there exists an $scc \in SCC$ and states $s, s' \in scc$ such that $\psi(s) \neq \psi(s')$. Hence either $\psi(s) \not\sqsubseteq \psi(s')$ or $\psi(s') \not\sqsubseteq \psi(s)$. Since s and s' are in the same *scc*, then they are mutually reachable and there must therefore exists a pair of states (s_i, s_j) on the path from either s to s' or s' to s such that $\psi(s_i) \not\sqsubseteq \psi(s_j)$. This contradicts the fact that the progress measure is monotonic.

Based on this, we can compute the strongly connected components for a given layer immediately before we delete the nodes in the current layer and move to the next one. The algorithm checks the property depending on the form of the property as outlined below.

Property - AG ϕ We check that every node within the layer satisfies ϕ. If ϕ does not hold in one of them, we return false and abort the exploration.

Property - EF ϕ If at least one state is encountered that satisfies ϕ, then true is returned and the execution finishes. Thus, false will be returned if at the end of the exploration not a single state satisfying ϕ has been found.

Property - AG EF ϕ For this property, we first compute the SCCs of the given Layer. The property will not be satisfied and therefore the procedure will finish the execution returning false, if any *scc* among the *SCC* of Layer is terminal and ϕ does not hold in any of the states contained in *scc*.

Property - AG AF ϕ For this property, we first compute the *SCC* of the given Layer. We then remove the states that satisfy ϕ. If the resulting set of nodes has a cycle, then the property is violated and therefore the execution immediately finishes returning false.

Property - AF ϕ This property can be checked in a similar fashion as **AG AF** ϕ with the modification that we can truncate the search at *SCC* where all cycles include a state satisfying ϕ.

The two first properties can easily be checked by just inspecting each state encountered during the sweep-line state space exploration. For verification of the two other properties, we invoke the procedure CHECKPROPERTY at the moment where the algorithm is about the leave the current layer and move into the next

ones. We do not detail the checking of $\mathbf{AF}\,\phi$ as it is very similar to $\mathbf{AG}\,\mathbf{AF}\,\phi$ as explained above.

A consequence of Proposition 1 is that SCC can be computed by considering one layer at a time. Furthermore, Theorem 1 ensures that the sweep-line method covers all reachable states which means that we will encounter all strongly connected components at some stage. The remaining step is therefore to link the inspection of SCC to the model checking of the $\mathbf{AG}\,\mathbf{EF}$ and $\mathbf{AG}\,\mathbf{AF}$ properties. This is done in the proposition below which formalises the requirements informally introduced above.

Proposition 2. *Let SCC be the set of strongly connected components of M, $SCC_T \subseteq SCC$ the set of terminal strongly connected components, and let ϕ be a state predicate. Then:*

1. $\mathbf{AG}\,\mathbf{EF}\,\phi$ *is satisfied* $\Leftrightarrow \forall scc \in SCC_T\ \exists s \in scc : \phi(s)$
2. $\mathbf{AG}\,\mathbf{AF}\,\phi$ *is satisfied* $\Leftrightarrow \forall scc \in SCC : scc \setminus \{s \in scc : \phi(s)\}$ *is acyclic*

Proof. First we prove 1. Assume that $\mathbf{AG}\,\mathbf{EF}\,\phi$ holds and there exists a terminal scc named scc_t such that no states in scc_t satisfy ϕ. Since all states belong to some scc, then we can find a path from the initial state to a state s in scc_t. Since scc_t is terminal and do not contain states satisfying ϕ, then we can no longer reach states that satisfies ϕ from s. Hence, $\mathbf{AG}\,\mathbf{EF}\,\phi$ cannot hold. Assume that each terminal scc contains a state satisfying ϕ and let s be any reachable state. Since we cannot have cycles that spans multiple SCC and all states belong to some scc, there must exists a path from the scc to which s belongs to a state s' in some terminal scc. Within this terminal scc, all states are mutually reachable and by our assumption at least one state in there satisfies ϕ. Hence, $\mathbf{AG}\,\mathbf{EF}\,\phi$ holds.

Next we prove 2. Assume that $\mathbf{AG}\,\mathbf{AF}\,\phi$ holds and there exists a scc such that when all states satisfying ϕ are removed from scc we still have a cycle consisting of states in scc. In that case we can find a path $s_0, s_1 \ldots s$ leading to a state s on this cycle, and we can then extend this to an infinite path by repeating the states on the cycle to which s belong. Since no state on the cycle satisfy ϕ, then $\mathbf{AG}\,\mathbf{AF}\,\phi$ cannot hold. Hence, we cannot have such cycles. Assume now that each strongly connected component becomes acyclic when removing states satisfying ϕ. Since all cycles belongs to some strongly connected component, then we cannot have cycles where no states satisfy ϕ. Hence, from any states on an infinite path we must eventually encounter a state satisfying ϕ which means that $\mathbf{AG}\,\mathbf{AF}\,\phi$ holds.

Based on Proposition 2 we can now specify the checkProperty procedure which is given in Algorithm 2. The procedure first computes the SCC of the given layer $\mathcal{L}$. Here any algorithm for computing SCC can be used, and we do not specify this further. Based on the SCC and Proposition 2, the procedure then checks whether the property being investigated is violated in which case false is returned and the entire algorithm terminates. At the end of the algorithm (line 18), true is returned in case the property was never violated.

```
1  begin
2  |  SCC ← ComputeSCC(Layer)
3  |  if Φ ≡ AG EF φ then
4  |  |  forall scc ∈ SCC do
5  |  |  |  if isTerminal(scc) ∧ ∀s ∈ scc : ¬φ(s) then
6  |  |  |  |  return false
7  |  |  |  end
8  |  |  end
9  |  end
10 |  if Φ ≡ AG AF φ then
11 |  |  forall scc ∈ SCC do
12 |  |  |  V ← scc \ {s ∈ scc | φ(s)}
13 |  |  |  if hasCycle(V) then
14 |  |  |  |  return false
15 |  |  |  end
16 |  |  end
17 |  end
18 |  return true
19 end
```

Algorithm 2: Checking strongly connected components of current layer

We have not specified the details of the isTerminal and hasCycle procedures. The isTerminal procedure can be implemented by checking that all successors of nodes in the *scc* are contained in the *scc*. The hasCycle procedure can be implemented by, e.g., a depth-first search of the nodes in V.

The completeness of the basic sweep-line algorithm and Proposition. 1 ensures that all strongly connected components will eventually have been computed and inspected in Algorithm 2. Furthermore, Algorithm 2 is a direct implementation of the two properties stated in Proposition 2. We therefore have the following theorem concerning the correctness of our algorithm:

Theorem 1. *Let $\mathcal{P} = (O, \sqsubseteq, \psi)$ be a monotonic progress measure, and let $\Phi \equiv$ **AG EF** ϕ or $\Phi \equiv$ **AG AF** ϕ. Then Algorithm 1 terminates and Φ is satisfied if and only if the algorithm returns true.*

In Algorithm 2 we have separated the computation of SCC from the checking of the SCC. As an optimisation it is possible to integrate the checking of the properties of a *scc* into the *scc* computation algorithm. This could make it possible to check the SCC as they are encountered by the *scc*-algorithm. As a further optimisation it is also possible to compute the SCC as the layer is being explored and not at the end of exploring a layer. However, for reason of clarity, we have decided to separate the two steps in the formulation of the algorithm.

As the continuation of the work presented in [17], we have implemented Algorithm 1 using the Standard ML language, and integrated it into CPN Tools. This allows us not only to analyse states spaces of models constructed using CPN Tools taking advantage of the sweep-line method, but also to verify the

aforementioned behavioural properties. We have also optimised the algorithm, so every time a property is violated or we know that it cannot be further satisfied, the execution stops to save time.

4 The CPN MQTT Model

Our aim is to use the property-specific sweep-line model checking algorithms for CTL from the previous section to verify the key behavioural properties of the CPN model we have developed of the MQTT protocol [19].

MQTT applies topic-based filtering of messages with a topic being part of each published message. An MQTT client can subscribe to a topic to receive messages, publish on a topic, and clients can subscribe to as many topics as they are interested in. As described in [18], an MQTT client can operate as a publisher or as a subscriber, and we use the term client to generally refer to a publisher or a subscriber. The broker [18] is the core of any publish/subscribe protocol and is responsible for keeping track of subscriptions, receiving and filtering messages, deciding to which clients they will be dispatched, and sending them to all subscribed clients. The MQTT protocol delivers application messages according to the three Quality of Service (QoS) levels defined in [2], which are motivated by the typically needs that IoT applications may have in terms of reliable delivery of messages.

4.1 Interaction Overview

MQTT defines five main operations: connect, subscribe, publish, unsubscribe and disconnect. Such operations, except the connect which must be performed a priori by each of the clients who want to participate in the communication, are mutually independent and can be triggered in parallel by the clients and processed by the broker. We have developed the CPN model following modelling patterns that ensure modularity, and thereby encapsulation of both the protocol logic and the behaviour of such operations.

In order to show how the clients and the broker interact, we describe the different actions that clients may carry out by considering an example. Figure 1 shows a sequence diagram for a scenario where two clients connect, perform subscribe, publish and unsubscribe, and finally disconnect from the broker. The protocol interaction is as follows:

1. Client 1 and Client 2 request a connection to the Broker.
2. The Broker sends back a connection acknowledgement (CONNACK) to confirm the establishment of the connection.
3. Client 2 subscribes to topic 1 with a QoS level 1, and the Broker confirms the subscription with a subscribe acknowledgement message.
4. Client 1 publishes on topic 1 with a QoS level 1. The Broker responds with a corresponding publish acknowledgement (PUBACK).
5. The Broker transmits the publish message to Client 2 which is subscribed to the topic.

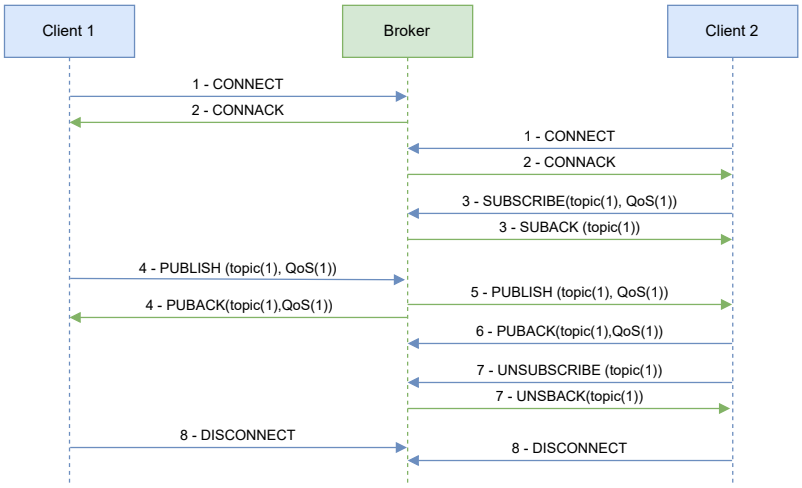

Fig. 1. Message sequence diagram illustrating the MQTT phases.

6. Client 2 gets the published message, and sends a publish acknowledgement back as a confirmation to the Broker that it has received the message.
7. Client 2 unsubscribes to topic 1, and the Broker responds with an unsubscribe acknowledgement.
8. Client 1 and Client 2 disconnect.

4.2 CPN Model Overview

We now briefly show and discuss the model and its main elements that are important for the understanding of the work carried out. We refer the reader to [19] for a detailed description of the MQTT protocol and the MQTT CPN model. The complete CPN model of the MQTT protocol consists of twenty four modules organised into six hierarchical levels.

The model is organised following a modelling pattern that ensures modularity and therefore, encapsulation of the protocol logic and behaviour of such operations. This offers advantages both for readability and understandability of the model and also, for making it easier to detect and fix errors during the incremental verification. For instance, this has allowed us to make a clear separation of the different QoS functional logic without having any negative complexity impact on the model. Note that the verification is incremental in the sense that we start with a core functionality of the protocol, and then we incrementally add more operations until we have the complete functionality included. This implies that we incrementally verify properties associated to each set of the operations.

Figure 2 shows the top-level module of the CPN MQTT model which consists of two *substitution transitions* (drawn as rectangles with double-lined borders) representing the Clients and the Broker roles of MQTT. Substitution transitions

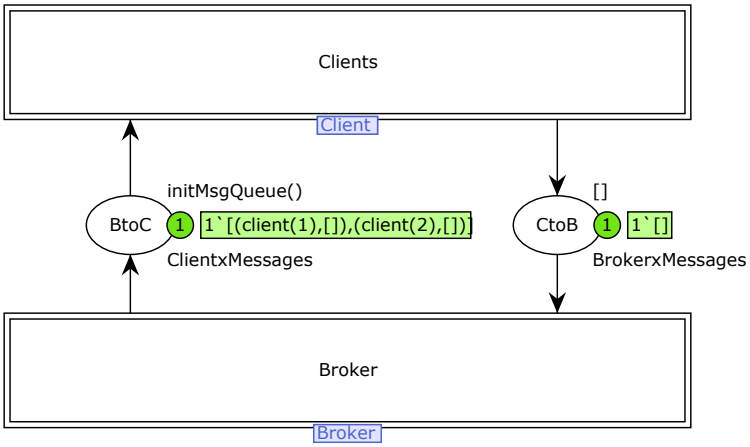

Fig. 2. The top-level module of the MQTT CPN model.

constitute the basic syntactical structuring mechanism of CPNs and each of the substitution transitions has an associated *module* that models the detailed behaviour of the clients and the broker, respectively. The name of the (sub)module associated with a substitution transition is written in the rectangular tag positioned next to the transition.

The two substitution transitions in Figure 2 are connected via directed arcs to the two places CtoB and BtoC. The clients and the broker interact by producing and consuming tokens on the places. The places CtoB and BtoC are designed to behave as queues. The queue mechanism offers some advantages that the MQTT specification implicitly indicates. The purpose of this is to ensure the ordered message distribution as assumed from the transport service on top of which MQTT operates.

4.3 Client and Broker State Modelling

The colour sets defined for modelling the client state are shown in Figure 3. The ClientProcessing submodule in Figure 4 models all the operations that a client can carry out. Clients can behave as senders and receivers, and the five substitution transitions `CONNECT`, `PUBLISH`, `SUBSCRIBE`, `UNSUBSCRIBE` and `DISCONNECT` have been constructed to capture both behaviours.

The place Clients (top-left place in Figure 4) uses a token for each client to store its respective state during the communication. The `State` colour set is an enumeration type containing the values `READY` (for the initial state), `WAIT` (when the client is waiting to be connected), `CON` (when the client is connected), and `DISC` (for when the client has disconnected). The states of the clients are represented by the `ClientxState` colour set which is a product of `Client` and `ClientState`. The colour set `ClientState` is used to represent the state of a client and consists of a list of `TopicxQoS`, a `State`, and a `PID`. Using this, a

```
colset State = with READY | DISC | CON | WAIT;

colset TopicxQoS     = product Topic * QoS;
colset ListTopicxQoS = list TopicxQoS;

colset Client = index client with 1..C;
colset ClientState  = record topics : ListTopicxQoS *
state  : State *
pid    : PID;

colset ClientxState = product Client * ClientState;
```

Fig. 3. Colour set definitions used for modelling client state.

client stores the topics it is subscribed to, and the quality of service level of each subscription. The colour set `PID` is used for modelling the packet identifiers which play a central role in the MQTT protocol logic.

We have structured the broker similarly as we have done for clients. This can be seen from Figure 5 which shows the BrokerProcessing submodule. The ConnectedClients place keeps the information of all clients as perceived by the broker. This place is designed as a central storage, and it is used by the broker to distribute the messages over the network. The broker behaviour is different from that of the clients, since it will have to manage all the requests and generate responses for several clients at the same time.

5 Model Checking and Experimental Results

In this section we show how we have performed sweep-line based model checking of the CPN MQTT model and present the results from the experiments.

5.1 Progress Measure

The first aspect to consider is how to define the progress measure of the model. Since the model runs in an acyclic configuration there is a final state where all the clients are disconnected and we take advantage of the `PID` as a way to keep track of the evolution of the message interchange. We have therefore defined the progress measure as a combination of the different states the clients can go through in conjunction with the `PIDs`. In the experiments, we consider two clients, so the initial state is made up of two clients in the `READY` state and `PID = 0` and the final state is reached when both clients are in a `DISC` state and the `PID = 3`.

Our definition of this progress measure over the possible combinations splits our state space into 100 layers. We have also experimented with other progress

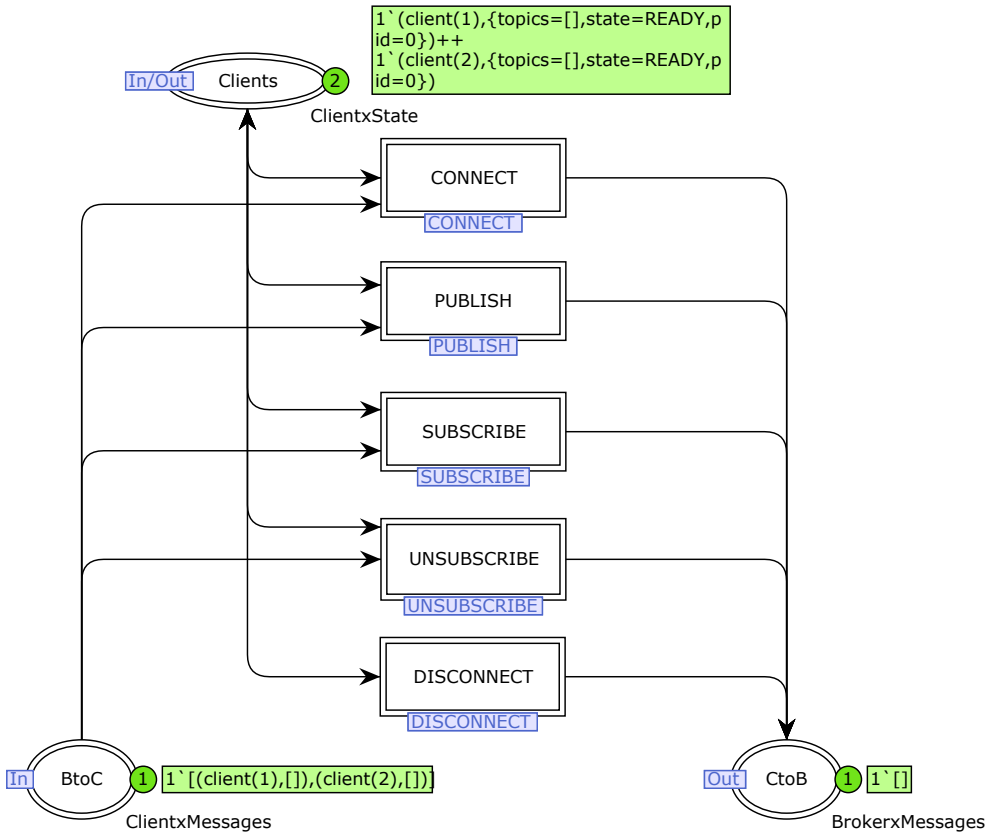

Fig. 4. ClientProcessing submodule.

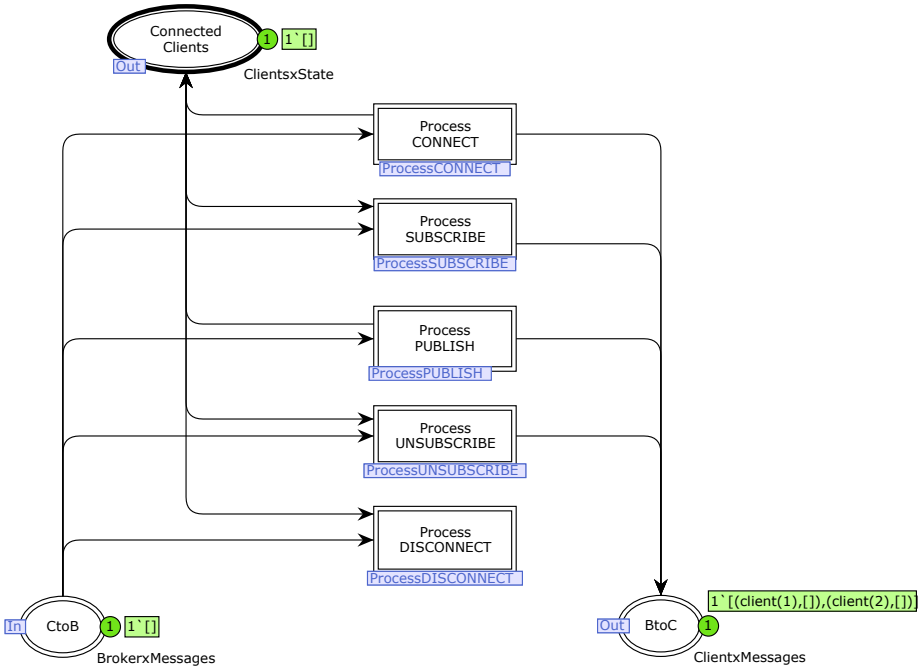

Fig. 5. The BrokerProcessing module.

measures specifications, for instance, just taking into account the states or only the PIDs which for each such separated choice produces a total of 16 layers. In our experience, there is a trade-off between the granularity and the size of each layer, and it is up to the analyst to decide depending on the concrete resources. Since the progress measure is defined such that the progress values are integers, we have for the states assigned 1 for `READY`, 2 for `WAIT`, 3 for `CON` and 4 for `DISC`, and 1 for `PID = 0`, 2 for `PID = 1`, 3 for `PID = 2` and 4 for `PID = 4`. It is important to note that the clients cannot backtrack to a previous state nor to a lower `PID`. For instance, if client 1 reaches the `CON` state, it can never be again in the `WAIT` state. As we need to keep a global notion of progress, we compute it using the following equation with c_1 and c_2 being client 1 and client 2, respectively and where B is a base:

$$\psi_c = B^3 * state(c_1) + B^2 * pid(c_1) + B^1 * state(c_2) + B^0 * pid(c_2)$$

Essentially, we interpret the states and the `PIDs` of the two clients as a number where B is required to be larger than the number of states of each client. In our experiments, we have used $B = 10$, i.e., the decimal numbering system. With this, we can obtain a progress value for each possibility (between 1111 and 4444) and respecting the monotonic ordering of non-regress.

As we have implemented the model in a modular and parameterized fashion, we are able to control several elements, for instance, the number of clients, the operations those clients can perform (e.g., connect and subscribe), and the size of the queues for handling messages. Note that, in order to obtain a finite state space, we have to limit the number of clients and topics, and also bound the packet identifiers. The packet identifiers are incremented throughout the execution of the different phases of the protocol, i.e., the connect, subscribe, data exchange, unsubscribe, and disconnect phases. This means that we cannot use a single global bound on the packet identifiers as a client could reach this bound, e.g., already during the publish phase and hence the global bound would prevent (block) a subsequent unsubscribe to take place. We therefore introduce a local upper bound on packet identifiers for each phase. This local bound expresses that the given phase may use packet identifiers up to this local bound. In the next subsection, we present the results of, first, running the state space using the sweep-line algorithm, and second, verifying certain behavioural properties.

5.2 Incremental Verification and Properties

We have designed a system to run six incremental executions which gives us more control to detect errors during the validation of the model and the verification of the properties. The six different scenarios are wrapped within three different steps. In the first step we include only the parts related to clients connecting and disconnecting. In the second step we add subscribe and unsubscribe, and finally in the third step we add data exchange considering the three quality of service levels in turn. At each step, we include verification of additional properties. Below we briefly discuss the three steps and the properties verified at each step. Note that properties that reason about clients are verified for each individual client. In other words, the properties make sure that every client involved satisfies the property being verified.

Step 1. Connect and Disconnect. In this first step we consider only the part of the model related to clients connecting and disconnecting to the broker.

S1-P1-ConsistentConnect The clients and the broker have a consistent view of the connection state.

S1-P2-ClientsCanConnect There exists a reachable state in which each client is connected to the broker.

S1-P3-ConsistentTermination Each terminal state (dead marking) has a consistent and desired behaviour.

S1-P4-PossibleTermination The protocol can always be terminated, i.e., a terminal state (dead marking) can always be reached.

Step 2. Subscribe and Unsubscribe. In this step, we add the ability for the clients to subscribe and unsubscribe (in addition to connect/disconnect from step 1).

S2-P1-CanSubscribe There exists states in which both the clients and the broker sides consider each client to be subscribed.

S2-P2-ConsistentSubscription In every state there is a consistent subscription in both clients and broker sides.
S2-P3-PossiblySubscribed If the client sends a subscribe message, then eventually both the clients and the broker sides will consider the client to be subscribed.
S2-P4-CanUnsubscribe For each client there exists executions in which the client sends an unsubscribe message.
S2-P5-EventuallyUnsubscribed If the client sends an unsubscribe message, then eventually that both the clients and the broker sides consider the client to be unsubscribed.

Step 3. Publish and QoS levels. We add the ability for the clients to publish and receive messages in addition to the rest of the properties of Steps 1 and 2.

S3-P1-PublishConnect Each client can publish if it is in a connected state.
S3-P2-CanPublish There exists an execution in which each client publishes a message.
S3-P3-CanReceive For each client there exists an execution in which each client receives a message.
S3-P4-ReceiveSubscribed A client only receives data if it is subscribed to the topic, i.e., the client side considers the client to be subscribed.

Table 1 shows the representation of the properties in CTL. Note that the verified properties have the forms described in Section 3. We have marked in Table 1 some properties with "*". The property S2-P3 has been computed as if it were an *EF* property (the same applies to S2-P5). However, this does not completely verify the property since it only checks that it is possible to find a state where the client is subscribed. What we really want to check is that we can reach a state where the client sends a subscribe message, and eventually after that the client is subscribed in the broker side. The implementation of such properties of the form $AG(\Phi \Rightarrow AF(\Psi))$ is part of our future work.

5.3 Experimental Results

Table 2 summarises the statistics as a result of running the six scenarios, using both approaches, the traditional CPN state space exploration and the sweep-line method approach, and verifying the properties aforementioned. The States and Arcs columns give the number of states and edges, respectively, in the state space. The Peak column lists the peak number of states stored in memory (i.e., the number of states in the largest layer). The Rel. Mem. Reduction column indicates the reduction of memory as the result of using the sweep-line method, compared to the total number of states (stored in memory by the tradition approach). For instance, in row number 5 in Table 2, we have a reduction in memory consumed of 84.17%, which means that the number of states we have in memory corresponds to the 15.83% of the total amount of states we would store using the traditional approach. The TV-Time column amounts the time

Table 1. CTL properties verified.

Property	**CTL formula**	**Description**
S1-P1	AGΦ	Φ: Consistent connection.
S1-P2	EFΦ	Φ: Each client is connected to the broker.
S1-P3	AG(¬ DM ∨ Φ)	DM: Dead marking | Φ: desired dead marking.
S1-P4	AGEF DM	DM: Dead marking (checked in S1-P3 that it is desired).
S2-P1	EFΦ	Φ: Each client can subscribe
S2-P2	AGΦ	Φ: Each client is consistently subscribed .
S2-P3*	EFΦ	Explanation above.
S2-P4	EFΦ	Φ: Each client can unsubscribe.
S2-P5*	EFΦ	Explanation above.
S3-P1	AG (Φ ⇒ Ψ)	Φ: Client connected | Ψ: Client can publish.
S3-P2	EFΦ	Φ: Each client can send a publish.
S3-P3	EFΦ	Φ: Each client can receive a publish.
S3-P4	AG (Φ ⇒ Ψ)	Φ: Client receives a publish | Ψ: Client is subscribed.

that took for the traditional procedure to verify the properties. The SLV-Time column details the time needed to verify the properties using the sweep-line approach. Finally, the column Rel. Time Increment gives the relative additional time that was necessary for the sweep-line method to proceed, compared to the traditional approach.

The two approaches provided the same results during the evaluation of the properties, keeping the consistency of the verification process. Even though the sweep-line is more time consuming, the memory usage was successfully reduced even in the worst case scenario. The highest relative time consumption is located in the third row with an increase of 63.48%. However, this should not been taken completely as reference since the calculation with such a low number of states and arcs is very sensitive to also the time that takes to compute the state space and the SCCs.

6 Conclusions and Future Work

We have presented the application of the sweep-line method for verifying an elaborate set of behavioral properties of the MQTT protocol. The application of the sweep-line method relied on a set of on-the-fly algorithms for model checking selected CTL behavioral properties. We have compared the application of the sweep-line method with the application of standard CTL model checking in CPN

Table 2. Results on the six incremental executions using both approaches.

Configuration	States	Arcs	Peak	Rel. Mem. Reduction	TV-Time	SLV-Time	Rel. Time Increment
1. Conn-Disconn	35	48	9	74.29%	0.00 s	0.00 s	0%
2. 1 + Subscribe	507	1,054	180	64.50%	0.156 s	0.219 s	79%
3. 2 + Unsubscribe	1,849	4,120	300	83.78%	1.328 s	2.171 s	63.48%
4. 3 + Pub QoS 0	4,282	8,840	711	83.4%	4.453 s	4.983 s	11.9%
5. 3 + Pub QoS 1	11,462	23,934	1,815	84.17%	20.172 s	28.531 s	41.44%
6. 3 + Pub QoS 2	43,791	85,682	7,037	83.93%	168.113 s	250.708 s	49.13%

Tools demonstrating a substantial reduction in memory usage at the expense of a modest increase in execution time. The consistency between the results obtained using conventional CTL model checking and the results obtained with the implementation of our property-specific CTL model checking algorithms for the sweep-line method serves as a validation of our new approach.

We see several possible directions for future work based on the results and experiments presented in this paper. We plan to investigate a more complete set of scenarios where different configurations are considered. This includes the number of clients, different progress measures, distinct queue sizes, and the possibility of retransmitting packets. This is going to be relevant to make other analysis and study, first, how the number and size of the strongly connected components affects the sweep-line method and second, how the the reduction factor grows with the value of the parameter. Related to this, there are also several possibilities for improving the implementation of the property-specific CTL model checking algorithms that we employ.

CTL model checking with the sweep-line method has until now been an open research problem, and the algorithms presented represents a first step towards addressing this. The extension of our approach to cover a larger subset of CTL properties is an important direction of future work. An example is the *S2-P3-EventualSubscribed* property discussed in Sect. 5. Properties on this form can be explored in a two-steps fashion way, where first the property in the left-hand side of the implication is accomplished, and then a second instance of the state space is explored, checking whether the property in the right-hand side is satisfied or not. The work presented in [16] on using tailored model checking algorithms for different CTL properties could serve as a starting point. A key challenge is to identity a subset of CTL compatible with the least-progress-first exploration order of the sweep-line method. In the context of symbolic model checking using binary-decision diagrams (BDDs), forward CTL model checking algorithms have been developed [11]. However, the sweep-line method is not compatible with the use of BDDs. The reason is that deleting states from a BDD (as required by

the sweep-line method) may cause the memory usage for storing the BDD to increase. This counteracts the idea of how the sweep-line method alleviates the state explosion problem.

A more open direction of future work is to develop CTL model checking techniques that can be used for non-monotonic progress measures - and not only monotonic progress measures as presented in this paper. We see potential improvements in being capable of including non-monotonic progress measures. It would significantly expand the class of models that can be analysed, for instance, we could also run the algorithm in the cyclic version of the CPN MQTT model.

References

1. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT press, 2008.
2. A. Banks and R. Gupta. MQTT Version 3.1.1. *OASIS standard*, 29, 2014. `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html`.
3. A. Cheng, S. Christensen, and K. H. Mortensen. Model Checking Coloured Petri Nets - Exploiting Strongly Connected Components. *DAIMI report series*, 26(519), 1997.
4. S. Christensen, L. M. Kristensen, and T. Mailund. A Sweep-line Method for State Space Exploration. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–464. Springer, 2001.
5. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.
6. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-state Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.
7. E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State Space Reduction using Partial Order Techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.
8. E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the State Explosion Problem. In *Tools for Practical Software Verification*, pages 1–30. Springer, 2012.
9. CPN tools. `http://cpntools.org/`.
10. S. Evangelista and L. M. Kristensen. Hybrid On-the-Fly LTL Model Checking with the Sweep-Line Method. In *Proc. of ICATPN'12*, pages 248–267, 2012.
11. H. Iwashita, T. Nakata, and F. Hirose. CTL Model Checking Based on Forward State Traversal. In *Proc. of Computer-Aided Design*, pages 82–87. IEEE Computer Society, 1996.
12. K. Jensen, L. Kristensen, and T. Mailund. The Sweep-line State Space Exploration Method. *Theoretical Computer Science*, 429:169–179, 2012.
13. K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254, Jun 2007.
14. L. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proc. of Formal Methods 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 549–567, 2002.

15. L. M. Kristensen and S. Christensen. Implementing Coloured Petri Nets using a Functional Programming Language. *Higher-order and symbolic computation*, 17(3):207–243, 2004.
16. T. Liebke and K. Wolf. Taking Some Burden Off an Explicit CTL Model Checker. In *Prof. of ICATPN'19*, volume 11522 of *LNCS*, pages 321–340. Springer-Verlag, 2019.
17. A. Lilleskare, L. M. Kristensen, and S.-O. Høyland. CTL Model Checking with the Sweep-line State Space Exploration Method. *Proc. of Norwegian Informatics Conference (NIK)*, 2017.
18. MQTT essentials part 3: Client, broker and connection establishment. `https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe`.
19. A. Rodríguez, L. M. Kristensen, and A. Rutle. Formal Modelling and Incremental Verification of the MQTT IoT protocol. *Transactions on Petri Nets and Other Models of Concurrency*, 2019. [Forthcoming]. Available: https://bit.ly/2piJoK9.
20. A. Rodríguez, L. M. Kristensen, and A. Rutle. On CTL Model Checking of the MQTT IoT Protocol using the Sweep-Line Method. In *Petri Nets and Software Engineering. International Workshop, PNSE'19, Aachen, Germany, June 24, 2019*, volume 2424 of *CEUR Workshop Proceedings*, pages 57–72, 2019.
21. U. Stern and D. L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 206–224. Springer, 1995.
22. A. Valmari. The State Explosion Problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.
23. J. Van Leeuwen and J. Leeuwen. *Handbook of Theoretical Computer Science*, volume 1. Elsevier, 1990.
24. M. Y. Vardi. Branching vs. linear time: Final showdown. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–22. Springer, 2001.