On CTL Model Checking of the MQTT IoT Protocol using the Sweep-Line Method

Alejandro Rodríguez, Lars Michael Kristensen and Adrian Rutle

Department of Computing, Mathematics, and Physics,
Western Norway University of Applied Sciences, Bergen, Norway
{arte,lmkr,aru}@hvl.no

Abstract. MQTT is a publish-subscribe communication protocol that is becoming increasingly used for 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. In this paper we 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 verification of the CTL properties, we use some property-specific algorithms that represent a first step towards a more generic 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) [11] is a formal modelling formalism convenient for specifying complex concurrent and distributed systems. CPN Tools [9, 13] 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 (set of reachable states) of a system in order to verify key behavioural properties. Temporal logics [19] 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 environments such as Machine-to-Machine communication (M2M) and IoT contexts designed with the aim of being light-weight and easy to implement. In earlier work [16], we have developed a formal and executable specification of MQTT [2] motivated by the fact that MQTT is until now only specified using an (ambiguous) natural language specification. MQTT contains relatively

CEUR-WS.org/Vol-2424/paper5.pdf

complex protocol logic for handling connections, subscriptions, and quality of service levels related to message delivery. Our initial verification was conducted using ordinary full state space as supported by CPN Tools and our initial experiments clearly highlighted the presence of the state explosion problem [8, 18] which is caused by the exponential growth in the number of reachable states of the system.

A large part of the model checking research has aimed at developing techniques for alleviating this inherent complexity problem. We can find several different families of reduction methods that have emerged during the last years (such as partial-order reduction methods [7] and hash compaction [17]) which try to reduce or provide more compact representations of the state space. However, 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 [10] which is based on the idea of exploiting a notion of progress exhibited by many systems. We focus on CTL since CPN Tools implements a CTL-like temporal logic called ASK-CTL [3]. ASK-CTL extends CTL in order to take into account both state and transition information.

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 [14]; and (2) the application of sweep-line based CTL model checking to our CPN model of the MQTT IoT protocol.

The rest of this paper is organised as follows. In Sect. 2 we introduce the sweep-line method and CTL model checking. Section 3 gives a brief review of the CPN model of the MQTT protocol. We describe the experiments carried out and the associated results in Sect. 4. Finally, in Sect. 5, 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.

2 Background

In this section we introduce the sweep-line method and the associated CTL model checking algorithms, and how we have implemented and integrated them into CPN Tools.

2.1 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. The progress measure is often specific for the system under consideration, but 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*. Defining 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 [10]. In conventional state space exploration, the states are kept in memory to recognise already visited states. However, states which have a progress value which 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.

In this paper, we consider the version of the sweep-line algorithm for *monotonic progress measures* (Ψ). A monotonic progress measure preserves the reachability relation, i.e., if a state s' is reachable from a state s, then $\Psi(s) \sqsubseteq \Psi(s')$. The progress exploited by the sweep-line method for a system is formalised by providing a monotonic progress measure as defined below, where S denotes the set of states and $s_0 \in S$ denotes 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 $\Psi : S \to O$ is a **progress mapping** such that $\forall s, s' \in reach(s_0) : s \to^* s' \Rightarrow \Psi(s) \sqsubseteq \Psi(s')$. □

A progress measure is non-monotonic when there is at least one *regress edge*, i.e., edges 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 measure and regress edges also exists [12], but is not the focus of our work.

2.2 CTL Model Checking with the Sweep-line Method

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 propagates information backwards from a state to its predecessors [6]. This follows the opposite work-flow than the forward progress-first exploration that the sweep-line method performs.

In this work, we consider a subset of CTL covering properties of the following forms, where Φ is a state formula:

Property - AGΦ "Invariantly", meaning that the property holds if Φ holds in all states that are reachable from the current state.

Property - EFΦ "Holds potentially" or "possibly", meaning that the property holds if it is possible to find a state reachable from the current state where Φ holds.

Property - AGEFΦ "Always possible", meaning that any state reachable from the current state, a state satisfying the predicate Φ can always be reached.

Property - AGAFΦ "Always eventually", meaning that from any state reachable from the current state, a state satisfying Φ will always eventually be reached.

We say that a formula (property) f holds in a given model M if f holds in the initial state s_0.

Algorithm 1 specifies the sweep-line algorithm. As mentioned in the introduction, we have based the implementation on the original sweep-line algorithm for monotonic progress measures. The algorithm starts with a hash table of visited states and a priority queue with 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 measure ψ_c is also initialized in line 4. Then, the algorithm executes a loop (lines 5-26) 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. We now check the behavioural property for the nodes in the layer, before we move to the next one (procedure checkProperty at line 8).

To model check the AGEF and AGAF properties, the algorithm exploits the *strongly connected components (SCCs)*. An SCC of a directed graph is a maximal subset of nodes that are mutually reachable. Because of the monotonicity of the progress measure, each SCC only contains nodes belonging to the same layer and hence each SCC is always contained in a single layer. Therefore, we can compute the SCCs 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:

Property - AGΦ 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 - EFΦ If at least one state 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 - AGEFΦ For this property, the procedure first computes 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 set of SCCs of Layer is terminal and Φ does not hold in any of the states contained in scc.

```
   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             checkProperty(Layer, Φ)
 9             forall s′ ∈ Layer do
10                 Nodes.delete(s′)
11             end
12             Layer ⟵ ∅
               /* Update progress measure for current layer        */
13             ψc = ψ(s)
14         end
15         Layer.insert(s)
           /* For every successor state of s                       */
16         forall (t, s′) such that s -t-> s′ do
17             if ¬(Nodes.contains(s′)) then
18                 Nodes.insert(s′)
19                 if (ψ(s) ⊐ ψ(s′)) then
20                     RaiseException(‘Regress edge found’)
21                 else
22                     Unprocessed.insert(s′)
23                 end
24             end
25         end
26     end
27 end
```

Algorithm 1: Sweep-line algorithm for monotonic progress measures

Property - AGAFΦ For this property, the procedure first computes the SCCs of the given Layer. For each scc, we first remove the states that satisfy the property. If the resulting set of nodes has a cycle, then the property is violated and therefore the execution finishes returning false.

Further details and the demonstrations of the properties AGEF and AGAF can be found in [14] (the procedure checkProperty is called CHECKSCC in that paper). After the property is checked, we remove the nodes in Layer, restart it and update the progress value ψ_c (lines 9-13). Then the state is added to the current layer (line 15) and its successor states are computed. Finally, if a successor state (node) has not yet been visited, it is added to the set of unprocessed nodes (line 18). If there is a regress edge the algorithm stops and an error message is returned (line 20).

As the continuation of the work presented in [14], 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.

3 The CPN MQTT Model

Our aim is to use our implementation of the CTL-based sweep-line model checking algorithms from the previous section to verify the key behavioral properties of the CPN model we have developed of the MQTT protocol [16].

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 [15], an MQTT client can operate as a publisher or subscribe, and we use the term client to generally refer to a publisher or a subscriber. The MQTT broker [15] 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. On one hand, the broker uses the topics to determine whether a subscribing client should receive the message or not. On the other hand, clients can subscribe to several topics depending on their interest.

3.1 Interaction Overview

MQTT defines five main operations: connect, subscribe, publish, unsubscribe, and disconnect. Such operations, except the connect which must be performed before the others by the clients, are independent of each other and can be triggered in parallel by either the clients or the broker. We have developed the CPN model following a modelling pattern that ensures modularity, and therefore encapsulation of both the protocol logic and the behaviour of such operations. The MQTT protocol delivers application messages according to the three Quality of Service (QoS) levels defined in [2], which are motivated by the different needs that IoT applications may have in terms of reliable delivery of messages.

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

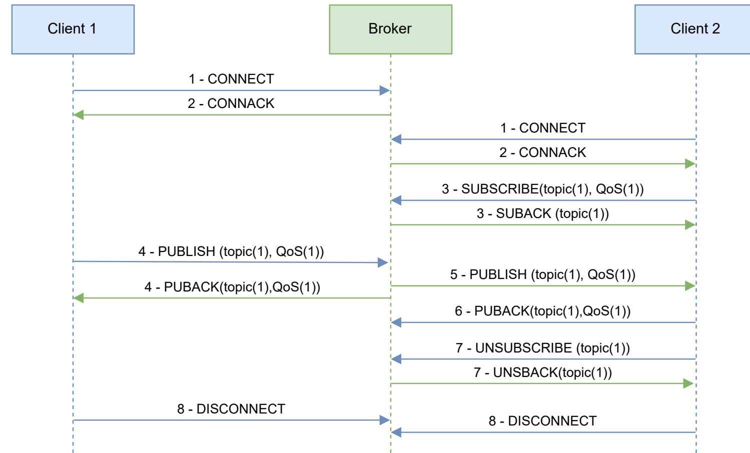

Fig. 1. Message sequence diagram illustrating the MQTT phases.

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 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.
5. The Broker transmits the publish message to Client 2 which is subscribed to the topic.
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.

3.2 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 [16] for a detailed description of the MQTT protocol and the MQTT CPN model. The complete CPN model of the MQTT protocol consists of 24 modules organised into six hierarchical levels. The model is organized 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 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 logics without having any negative complexity impact on the model.

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 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.

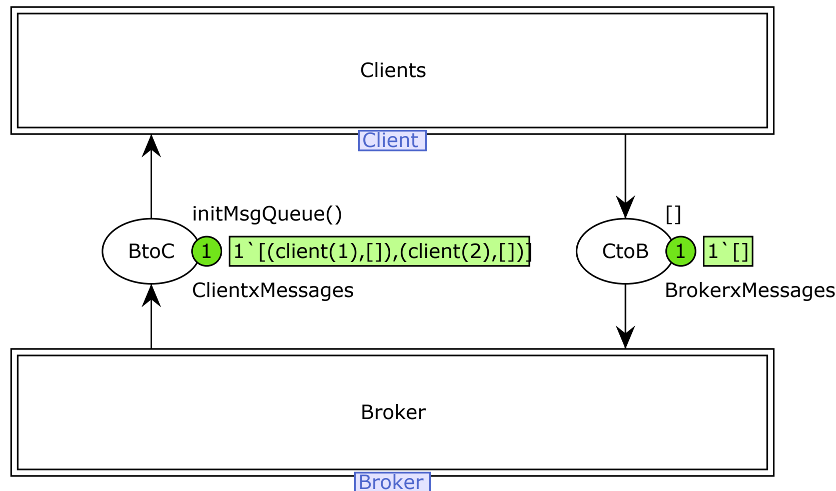

Fig. 2. The top-level module of the MQTT CPN model.

The two substitution transitions in Fig. 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.

3.3 Client and Broker State Modelling

The colour sets defined for modelling the client state are shown in Fig. 3. The place Clients (top-left place in Fig. 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

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

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

colset ClientState  = record topics : ListTopicxQoS *
                             state  : State *
                             pid    : PID;

colset ClientxState = product Client * ClientState;
```

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

by the `ClientxState` colour set which is a product of `Client` and `ClientState`. The colour set `ClientState` used to represent the state of a client which consists of a list of `TopicxQoS`, a `State`, and a `PID`. Using this, a client stores the topics it is subscribed to, and the quality of service level of each subscription.

The ClientProcessing submodule in Fig. 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` has been constructed to capture both behaviours. We have structured the broker similarly as we have done for clients. This can be seen from Fig. 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.

4 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.

4.1 Progress Measure

The first thing we need to consider is how we 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), we have defined the progress measure over the different states the clients can go through. In the experiments, we consider two clients, so the initial state is made up with the two clients in the `READY` state and the final state is reached when both clients are in a `DISC` state.

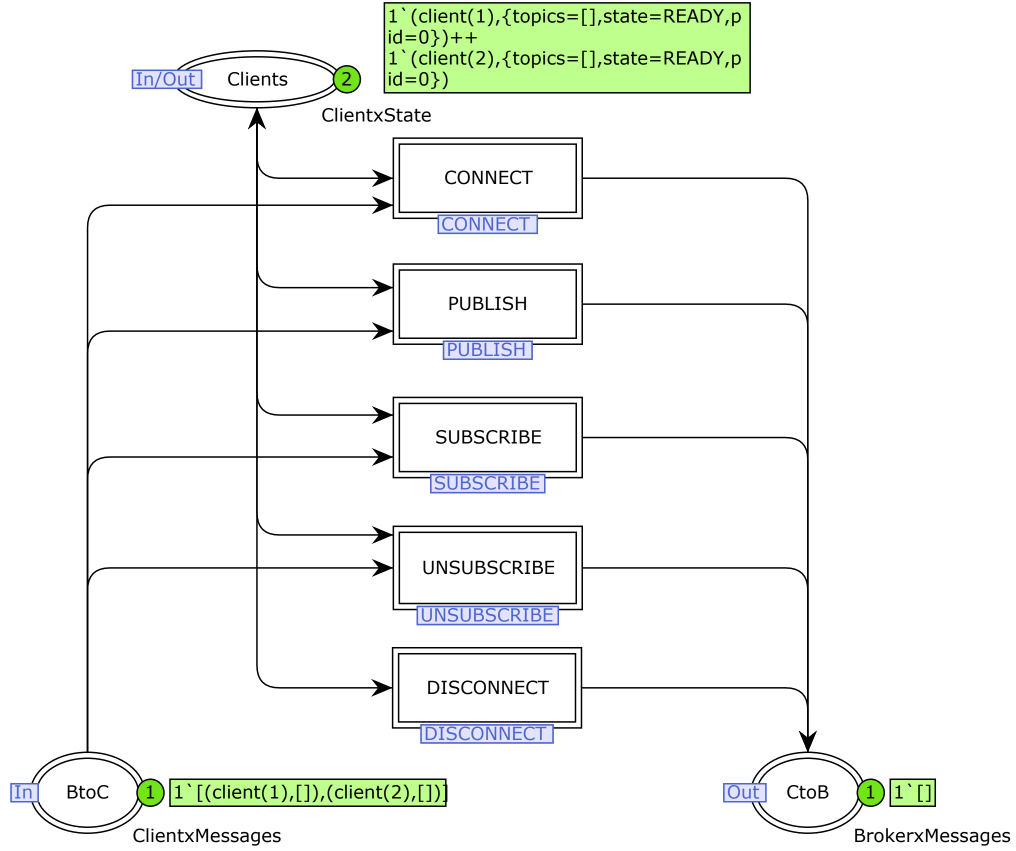

Fig. 4. ClientProcessing submodule.

As we have two clients handling four different states, our definition of this progress measure over the possible combinations will split our state space into 16 layers. Since the progress measure is defined such that the progress values are integers, we have assigned 1 for `READY`, 2 for `WAIT`, 3 for `CON` and 4 for `DISC`. It is important to note that the clients cannot go back to a previous state, 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 measure, we compute it using the following equation with c_1 and c_2 being client 1 and client 2, respectively:

$$\psi_c = B * state(c_1) + state(c_2)$$

and where B is a base. Essentially, we interpret the states of the two client as a base B 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.

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 (connect, subscribe, etc.), and the size of the

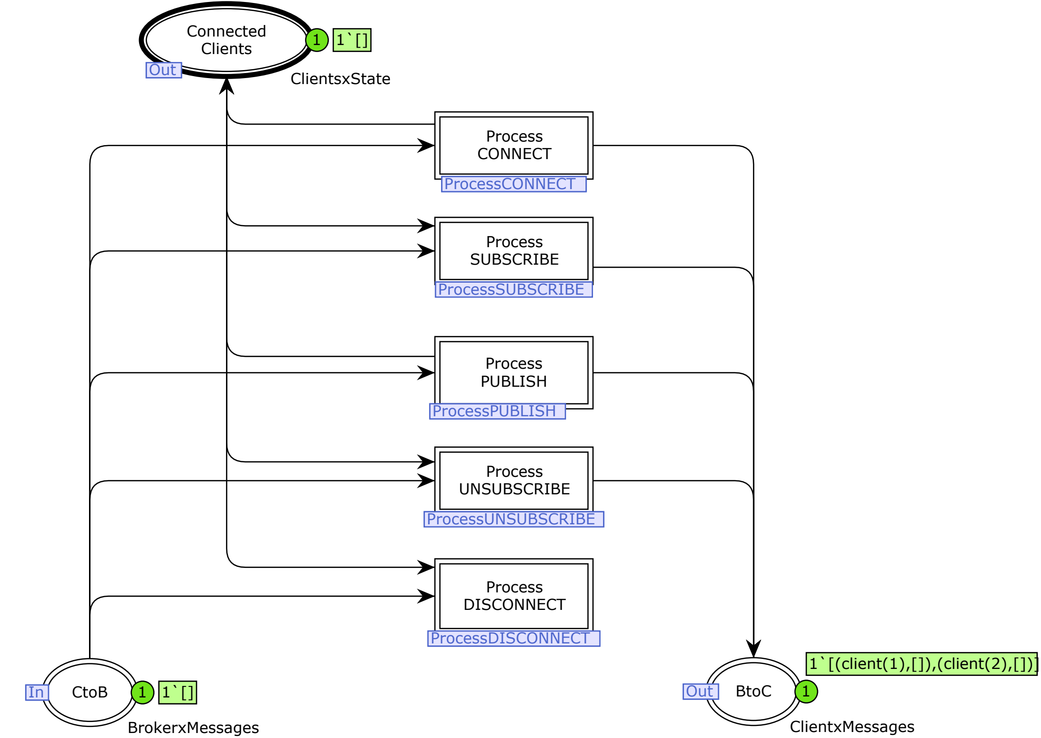

Fig. 5. The BrokerProcessing module.

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 (connect, subscribe, data exchange, unsubscribe, and disconnect). 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 will highlight the results of first, running the state space using the sweep-line algorithm, and second, verifying certain behavioural properties.

4.2 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 verify each individual client. In other words, the properties make sure that every client involved verifies them.

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 and 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-EventualSubscribed 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-EventualUnsubscribed If the client sends an unsubscribe message, then eventually both the clients and the broker sides consider the client to be unsubscribed.

Step 3. Publish and QoS levels. In this step, we add the ability for the clients to subscribe and unsubscribe 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 executions in which each client publishes a message.
S3-P3-CanReceive For each client there exists executions in which the 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 need to have one of the forms described in Sect. 2.2. In Table 2 we have marked 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 $AG(\Phi) \Rightarrow AF(\Psi)$ will be part of our future work.

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.

4.3 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 of 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). The TV-Time column amounts the time 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. This is also dependent on how we specify the progress measure. As part of future work, we are investigating how to define a more optimised version taking advantage of the bounded packet identifiers together with the already defined progress using the state of the clients.

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	26.32%	0.024 s	0.032 s	33,33%
2. 1 + Subscribe	507	1054	208	41.03%	0.156 s	0.218 s	39.75%
3. 2 + Unsubscribe	1,849	4,120	867	46.89%	0.908 s	1.125 s	23.90%
4. 3 + Pub QoS 0	4,282	8,840	1,788	41.76%	3.406 s	4.063 s	19,29%
5. 3 + Pub QoS 1	11,462	23,934	4,125	35.99%	14.795 s	21.596 s	45,97%
6. 3 + Pub QoS 2	43,791	85,682	15,932	36.38%	134.446 s	298.512 s	122,03%

5 Conclusions and Future work

We have presented an implementation of the sweep-line method and the ability to verify on-the-fly four CTL behavioural properties in CPN Tools. We have performed multiple executions, first using the traditional way that CPN Tools uses for computing the state space and check the specified behavioural properties, and second using the implemented algorithm. Our results show that even though using the sweep-line approach takes more time, it still reduces the amount of memory usage significantly. Furthermore, checking the properties with the novel implementation revealed the same results than checking them with CPN Tools. This provides us an experimental guarantee that both the sweep-line method theoretical background to check the described subset of CTL properties, and also the actual implementation we have performed are sound (since verification of models with CPN Tools has been widely used and reflected in the literature).

We see several directions for future work based on the experiments presented in this paper. We plan to simulate a more complete set of scenarios where different configurations are investigated. For instance, this might include number of clients, different progress measures, distinct queue sizes or the possibility of retransmitting packages. We are working on being able to check a larger set of CTL properties. An example of it, is the example of the *S2-P3-EventualSubscribed* property, already explained in Sect. 4. Such properties 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.

We also see potential improvements in being capable of including non-monotonic progress measures. This would significantly increase the amount of models that can be analysed, for instance, we could also run the algorithm in the cyclic version of the CPN MQTT model described in Sect. 3. As explained at the end of Sect. 4, the way of defining the progress measure for a model will have a direct impact on its layering, and as part of this we are also investigating how to improve the progress measure defined in the course of this work.

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. K. Jensen, L. M. Kristensen, and T. Mailund. The sweep-line state space exploration method. *Theoretical Computer Science*, 429:169–179, 2012.
11. 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.

12. 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.
13. 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.
14. 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.
15. MQTT essentials part 3: Client, broker and connection establishment. `https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe`.
16. A. Rodriguez, L. M. Kristensen, and A. Rutle. On modelling and validation of the mqtt iot protocol for m2m communication. *39th International Conference on Application and Theory of Petri Nets and Concurrency*, 2138:99–120, 2018.
17. 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.
18. A. Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.
19. J. Van Leeuwen and J. Leeuwen. *Handbook of theoretical computer science*, volume 1. Elsevier, 1990.