Formal Modelling and Incremental Verification of the MQTT IoT Protocol

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. Machine to Machine (M2M) communication and Internet of Things (IoT) are becoming still more pervasive with the increase of communicating devices used in cyber-physical environments. A prominent approach to communication between distributed devices in highly dynamic IoT environments is the use of publish-subscribe protocols such as the Message Queuing Telemetry Transport (MQTT) protocol. MQTT is designed to be light-weight while still being resilient to connectivity loss and component failures. We have developed a Coloured Petri Net model of the MQTT protocol logic using CPN Tools. The model covers all three quality of service levels provided by MQTT (at most once, at least once, and exactly once). For the verification of the protocol model, we show how an incremental model checking approach can be used to reduce the effect of the state explosion problem. This is done by exploiting that the MQTT protocol operates in phases comprised of connect, subscribe, publish, unsubscribe, and disconnect.

1 Introduction

Publish-subscribe messaging systems support data-centric communication and have been widely used in enterprise networks and applications. A main reason for this is that a software system architecture based on publish-subscribe messaging provides better support for scalability and adaptability than the traditional client-server architecture used in distributed systems. The interaction and exchange of messages between clients based on the publish-subscribe paradigm are based on middleware usually referred to as a *broker* (or a bus) that manages *topics*. The broker provides space decoupling [9] allowing a client acting as a publisher on a given topic to send messages to other clients acting as subscribers to the topic without the need to know the identity of the receiving clients. The broker also provides synchronisation decoupling in that clients can exchange messages without being executing at the same time. Furthermore, the processing in the broker can be parallelized and handled using event-driven techniques.

The loose coupling and support for asynchronous point-to-multipoint messaging, make the publish-subscribe paradigm attractive also in the context of Internet of Things (IoT) which has experienced significant growth in applications and adoptability in recent years [17]. The IoT paradigm blends the virtual

and the physical worlds by bringing different concepts and technical components together: pervasive networks, miniaturisation of devices, mobile communication, and new ecosystems [6]. Moreover, the implementation of a connected product typically requires the combination of multiple software and hardware components distributed in a multi-layer stack of IoT technologies.

MQTT [3] is a publish-subscribe messaging protocol for IoT designed with the aim of being light-weight and and easy to implement. These characteristics make it a suitable candidate for constrained environments such as Machine-to-Machine communication (M2M) and IoT contexts where a small memory footprint is required and where network bandwidth is often a scarce resource. Even though MQTT has been designed to be easy to implement, it still contains relatively complex protocol logic for handling connections, subscriptions, and the various quality of service levels related to message delivery. Furthermore, MQTT is expected to play a key role in future IoT applications and will be implemented for a wide range of platforms and in a broad range of programming languages making interoperability a key issue. This, combined with the fact that MQTT is only backed by an (ambiguous) natural language specification, motivated us to develop a formal and executable specification of the MQTT protocol.

We have used Coloured Petri Nets (CPNs) [12] for the modelling and verification of the MQTT specification. The main reason is that CPNs have been successfully applied in earlier work to build formal specifications of communication protocols [8], data networks [5], and embedded systems [1]. To ensure the proper operations of the constructed CPN model, we have validated the CPN model using simulation and verified an elaborate set of behavioural properties of the constructed model using model checking and state space exploration. In the course of our work on the MQTT specification [3] and the development of the CPN model, we have identified a number of issues related in particular to the implementation of the quality of service levels. These issues are a potential source of interoperability problems between implementations. For the construction of the model we have applied some general modelling patterns for CPN models of publish-subscribe protocols. Compared to earlier work on modelling and verification of publish-subscribe protocols [18, 4, 10] (which we discuss in more details towards the end of this paper) our work specifically targets MQTT, and we consider a more extensive set of behavioural properties.

The rest of this paper is organised as follows. In Sect. 2 we present the MQTT protocol context and give a high-level overview of the constructed CPN model. Section 3 details selected parts of the CPN model of the MQTT protocol. In Sect. 4 we present our experimental results on using simulation and model checking to validate and verify central properties of MQTT and the CPN model. Finally, in Sect. 5 we sum up the conclusions, discuss related work, and outlines directions for future work. Due to space limitations, we cannot present the complete CPN model of the MQTT protocol. The constructed CPN model is available via [15]. The reader is assumed to be familiar with the basic concepts of Petri Nets and High-level Petri Nets [12].

2 MQTT Protocol and CPN Model Overview

MQTT [3] runs over the TCP/IP protocol or other transport protocols that provide ordered, lossless and bidirectional connections. 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 [14], 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 [14] 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. There are no restrictions in terms of hardware to run as an MQTT client, and any device equipped with an MQTT library and connected to an MQTT broker can operate as a client.

2.1 Modelling Roles and Messages

Figure 1 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.

The complete CPN model of the MQTT protocol consists of 24 modules organised into six hierarchical levels. We have constructed a parametric CPN model which makes it easy to change the number of clients and topics without making changes to the net-structure. This makes it possible to investigate dif-

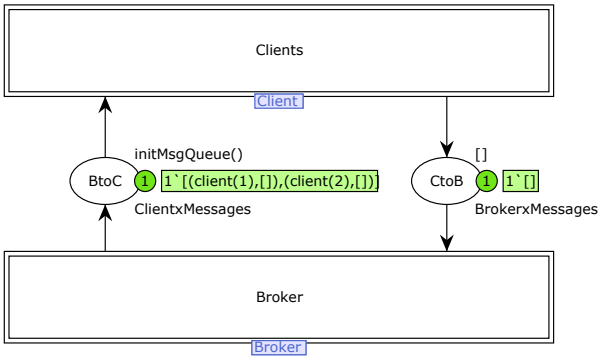

Fig. 1. The top-level module of the MQTT CPN model.

ferent configuration of MQTT and it is a main benefit provided by CPNs in comparison to ordinary Petri Nets.

The two substitution transitions in Fig. 1 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. Figure 2 shows the central data type definitions used for the colour sets of the places CtoB and BtoC and the modelling of clients and messages. The colour sets QoS is used for modelling the three quality of service levels supported by MQTT (see below), and the colour set PID is used for modelling the packet identifiers which plays a central role in the MQTT protocol logic. We have abstracted from the actual payload of the published messages. The reason for this is that the message transport structure and the protocol logic of MQTT is agnostic to the payload contained, i.e., the actual content that will be sent in the messages. For similar reasons, we also abstract from the hierarchical structuring of topics.

```
val T = 5;  (* number of topics  *)
val C = 2;  (* number of clients *)

colset Client = index client with 1..C;
colset Topic  = index topic with 1..T;
colset QoS    = index QoS with 0..2; (* quality of service *)
colset PID    = INT;                 (* packet identifiers *)

colset TopicxPID     = product Topic * PID;
colset TopicxQoSxPID = product Topic * QoS * PID;

colset Message = union CONNECT + CONNACK +
  SUBSCRIBE : TopicxQoSxPID + UNSUBSCRIBE : TopicxPID +
  SUBACK    : TopicxQoSxPID + UNSUBACK    : TopicxPID
  PUBLISH   : TopicxQoSxPID +
  PUBACK    : TopicxPID     + PUBREC      : TopicxPID +
  PUBREL    : TopicxPID     + PUBCOMP     : TopicxPID +
  DISCONNECT;

colset Messages = list Message;

colset ClientxMessage      = product Client * Message;
colset BrokerxMessages     = list ClientxMessage;

colset ClientxMessageQueue = product Client * Messages;
colset ClientxMessages     = list ClientxMessageQueue;
```

Fig. 2. Client and message colour set definitions

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. Even so, the CtoB and BtoC places are slightly different; while CtoB is modelled as a single queue that the broker manages to consume messages, BtoC is designed to maintain an incoming queue of messages for each client. This construction assures that all clients will have their own queue, individually respecting the ordered reception of messages. The function `initMsgQueue()` initialises the queues according to the number of clients specified by the symbolic constant `C`. The `BrokerxMessages` colour set for the CtoB place used at the bottom of Fig. 2 consists of a list of `ClientxMessage` which are pairs of `Client` and `Messages`.

We represent all the messages that the clients and the broker can use by means of the `Message` colour set. We use the terms packet and message indistinguishably when we refer to control packets. The control information used depends on the messages considered. As an example, a `Connect` message (packet) does not contain control information, but a `Publish` message requires a specific `Topic`, `QoS`, and `PID`. The `Topic` and `QoS` colour sets are both indexed types containing values (`topic(1)`, `topic(2)` ... `topic(T)` depending on the constant `T`, and `QoS(0)`, `QoS(1)` and `QoS(2)`, respectively. The `ClientxMessages` colour set for the BtoC place encapsulates all the queues (each one declared as a pair of `Client` and `Messages` in the `ClientxMessageQueue` colour set) in one single queue. This modelling pattern allows us to deal with the distribution of multiple messages in a single step in the broker side which in turn simplifies the modelling of the broker and reduces the number of reachable states of the model.

2.2 Quality of Service

The MQTT protocol delivers application messages according to the three Quality of Service (QoS) levels defined in [3]. The QoS levels are motivated by the different needs that IoT applications may have in terms of reliable delivery of messages. It should be noted that even if MQTT has been designed to operate over a transport service with lossless and ordered delivery, then message reliability still must be addressed as logical transport connections may be lost.

The delivery protocol is symmetric, and the clients and the broker can each take the role of either a sender or a receiver. The delivery protocol is concerned solely with the delivery of an application message from a single sender to a single receiver. When the broker is delivering an application message to more than one client, each client is treated independently. The QoS level used to deliver an outbound message from the broker could differ from the QoS level designated in the inbound message. Therefore, we need to distinguish two different parts of delivering a message: a client that publishes to the broker and the broker that forwards the message to the subscribing clients. The three MQTT QoS levels for message delivery are:

At most once: (QoS level 0): The message is delivered according to the capabilities of the underlying network. No response is sent by the receiver and

no retry is performed by the sender. The message arrives at the receiver either once or not at all. An application of this QoS level is in environments where sensors are constantly sending data and it does not matter if an individual reading is lost as the next one will be published soon after.

At least once (QoS level 1): Where messages are assured to arrive, but duplicates can occur. It fits adequately for situations where delivery assurance is required but duplication will not cause inconsistencies. An application of this are idempotent operations on actuators, such as closing a valve or turning on a motor.

Exactly once (QoS level 2): Where messages are assured to arrive exactly once. This is for use when neither loss nor duplication of messages are acceptable. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.

When a client subscribes to a specific topic with a certain QoS level it means that the client is determining the maximum QoS that can be expected for that topic. When the broker transfers the message to a subscribing client it uses the QoS of the subscription made by the client. Hence QoS guarantees can get downgraded for a particular receiving client if subscribed with a lower QoS. This means that if a receiver is subscribed to a topic with a QoS level 0, no matter if a sender publishes in this topic with a QoS level 2, then the receiver will proceed with its QoS level 0.

3 Modelling the Protocol Roles and their Interaction

We now consider the different phases and the client-broker interaction in the MQTT protocol, and show how we have modelled the MQTT protocol logic using CPNs. MQTT defines five main operations: connect, subscribe, publish, unsubscribe, and disconnect. Such operations, except the connect which must be the first one for the clients, are independent of each other and can be triggered in parallel by either the clients or the broker. 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.

3.1 Interaction Overview

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 3 shows a sequence diagram for a scenario where two clients connect, perform subscribe, publish and unsubscribe, and finally disconnect from the broker. The numbers on each step of the communication define the interaction of the protocol as follows:

1. Client 1 and Client 2 request a connection to the Broker.

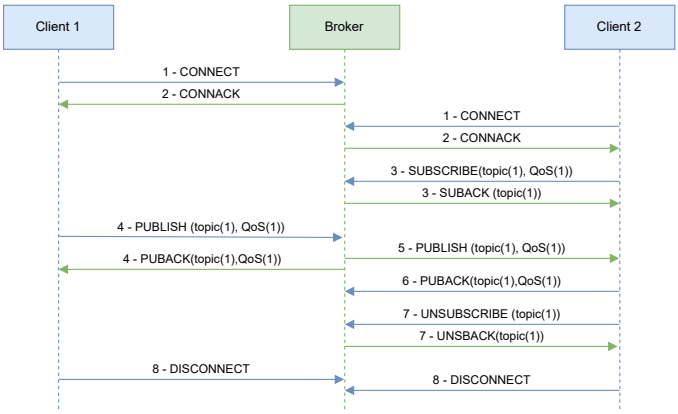

Fig. 3. Message sequence diagram illustrating the MQTT phases.

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 Client and Broker State Modelling

The colour sets defined for modelling the client state are shown in Fig. 4. The place Clients (top-left place in Fig. 5) uses a token for each client to store their respective state during the communication. This is a modelling pattern that allows not only to parameterize the model so we can change the number of clients without modifying the structure, but also to maintain all the clients independently in only one place and with a proper data structure that encapsulates all the information required. The states of the clients are represented by the `ClientxState` colour set which is a product of `Client` and `ClientState`. The record colour set `ClientState` is 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 `State` colour set is an enumeration type containing the values `READY` (for

```
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. 4. Colour set definitions used for modelling client state.

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).

Below we present selected parts of the model by first presenting a high-level view of the clients and broker sides, and then illustrating how the model captures the execution scenario described in Section 3.1 where two clients connects, one subscribes to a topic, and the other client publishes on this topic. The unsubscribe and the disconnection phases are not detailed due to space limitations.

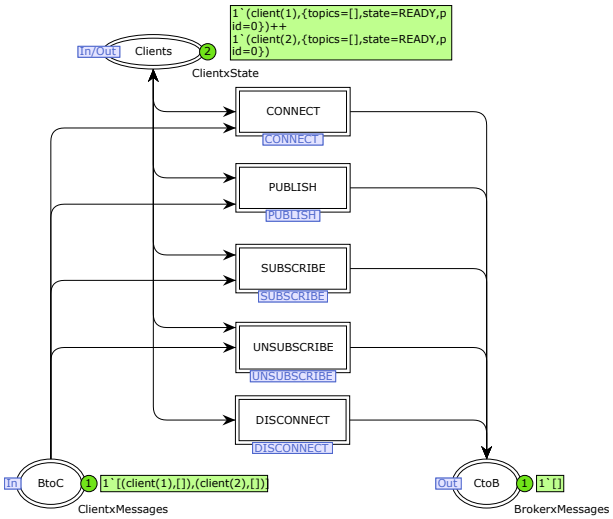

Fig. 5. ClientProcessing submodule.

3.3 Client Modelling

The ClientProcessing submodule in Fig. 5 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.

The socket place Clients stores the information of all the clients that are created at the beginning of the execution of the model. In this scenario there are two clients, and the value of the tokens representing the state of the two clients is provided in the green rectangle (the marking of the place) next to the Clients place. The BtoC and CtoB port places are associated with the socket places already shown in Fig. 2.

3.4 Broker Modelling

We have modelled the broker similarly as we have done for clients. This can be seen from Fig. 6 which shows the BrokerProcessing submodule. The Connected-Clients 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.

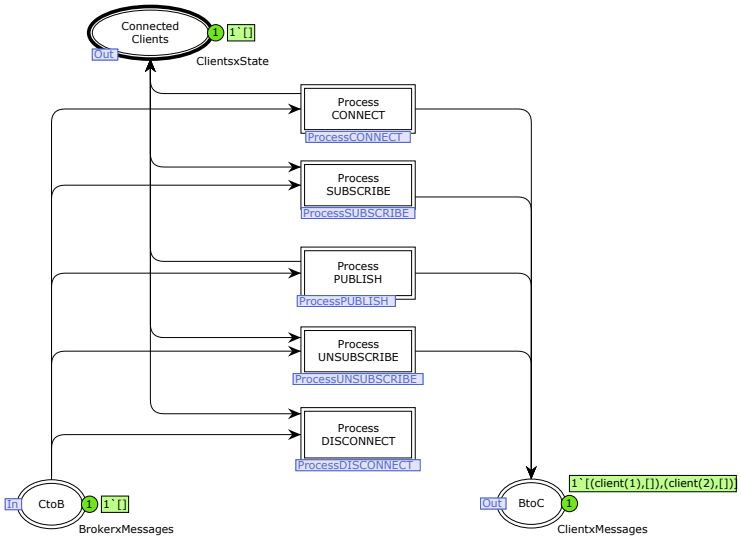

Fig. 6. The BrokerProcessing module.

3.5 Connection Phase

The first step for a client to be part of the message exchange is to connect to the broker. A client will send a `CONNECT` request, and the broker will respond with a `CONNACK` message to complete the connection establishment. Figure 7 shows the CONNECT submodule in a marking where `client(1)` has sent a `CONNECT` request and it is waiting (`state = WAIT`) for the broker acknowledgement processing to finish such that the connection state can be set to `CON`.

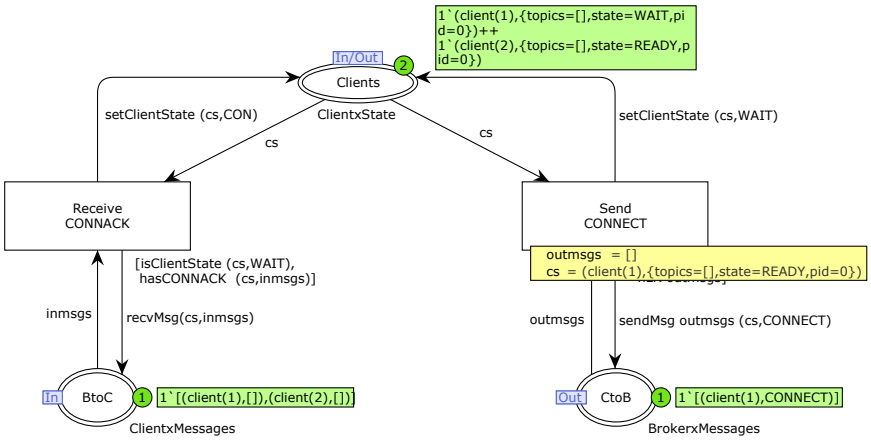

Fig. 7. CONNECT module after the sendCONNECT occurrence.

The broker will receive the `CONNECT` request. The broker will register the client in the place ConnectedClients and send back the acknowledgement. Figure 8 shows the situation where `client(1)` is connected in the broker side and the `CONNACK` response has been sent back to the client. The function `connectClient()` used on the arc from the ProcessCONNECT transition to the ConnectedClients place will record the connected client on the broker side. The last step of the connection establishment will occur again in the clients side, where the transition `ReceiveCONNACK` (in Fig. 7) will be enabled, meaning that the confirmation for the connection of `client(1)` can proceed.

3.6 Subscription Phase

Starting from the point where both clients are connected (i.e., for both clients, the state is `CON` as shown at the top of Fig. 9), `client(2)` will send a `SUBSCRIBE` request to `topic(1)` with `QoS(1)`. The place PendingAcks represents a queue that each client maintains to store the `PIDs` that are waiting to be acknowledged. In this example, the message has assigned a `PID = 0`, and `client(2)` is waiting for an acknowledgement to this subscription with a `PID = 0`. When a client receives a `SUBACK` (subscribe acknowledgement) it will check that the packet identifier (0

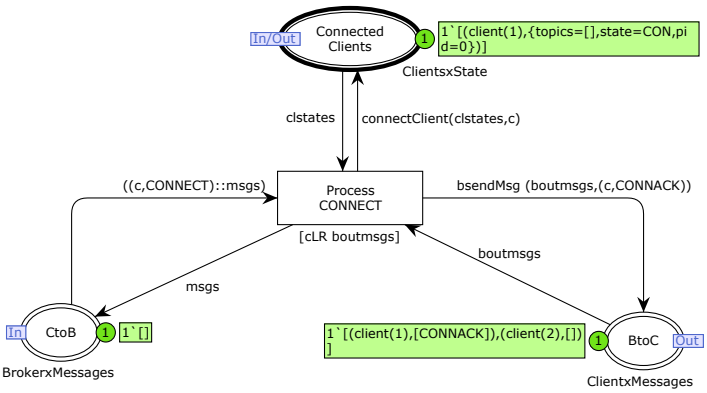

Fig. 8. ProcessCONNECT module after the ProcessCONNECT occurrence.

in this case) is the same to ensure that the correct packet is being received. At the bottom right side of the Fig. 9, the message has been sent to the broker.

We show now the situation where the `SUBSCRIBE` request has been processed by the broker as represented in Fig. 10. The function `brokerSubscribeUpdate()` manages the subscription process, so if the client is subscribing to a new topic, it will be added to the client state stored in the broker. If the client is already subscribed to this topic it will update it. In the example, one can see that `client(1)` keeps the same state, but `client(2)` has appended this new topic to its list. The corresponding `SUBACK` message has been sent to `client(2)` (with the `PID` set to 0) to confirm the subscription. Next, `client(2)` will detect that the response has arrived and it will check that the packet identifiers correspond to each other.

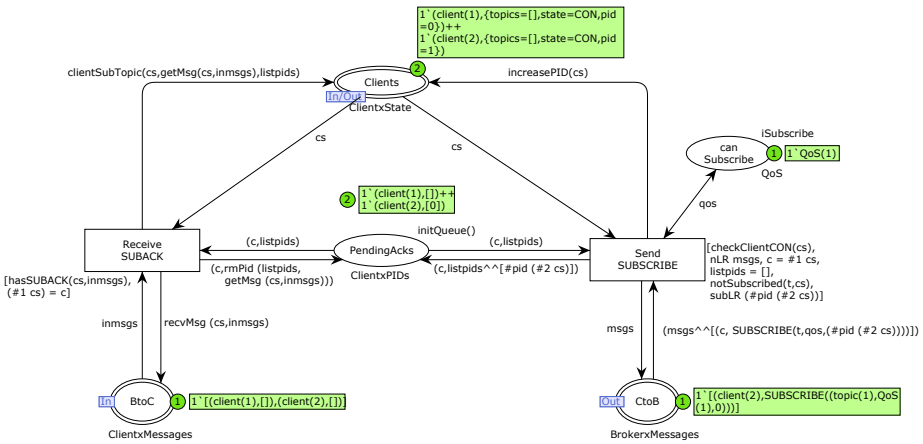

Fig. 9. SUBSCRIBE module after the SUBSCRIBE occurrence.

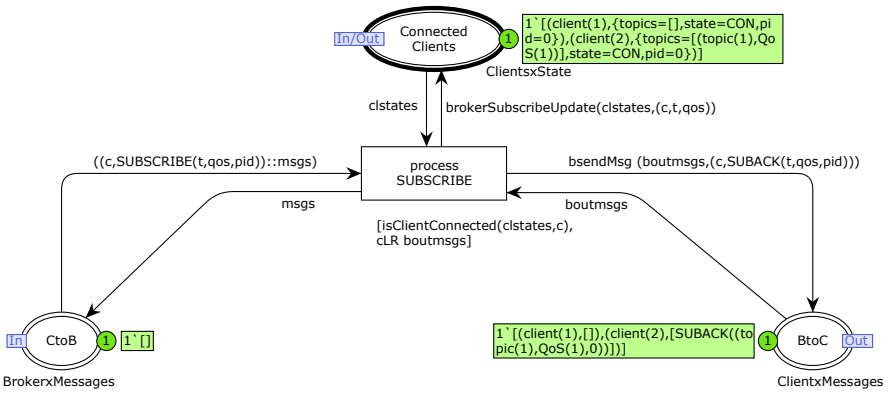

Fig. 10. ProcessSUBSCRIBE module after occurrence of ProcessSUBSCRIBE.

3.7 Publishing Phase

The publishing process in the considered scenario requires two steps to be completed. First a client sends a `PUBLISH` in a specific topic, with a specific `QoS`, which is received by broker. The broker will answer back with the corresponding acknowledgement, depending on the quality of service previously set. Second, the broker, that stores information for all clients, will propagate the `PUBLISH` sent by the client to any clients subscribing to that topic. We have modelled the clients and broker sides using different submodules depending on the quality of service that is being applied for sending and receiving. In our example, `client(1)` will publish in `topic(1)` with a `QoS(1)`. This means that the broker will acknowledge back with a `PUBACK` to `client(1)`, and will create a `PUBLISH` message for `client(2)`, which is subscribed to this topic with a `QoS(1)`. In this case, there is no downgrading for the `client(2)`, so the publication process will be similar to step 1, i.e, `client(2)` will send back a `PUBACK` to the broker.

Figure 11 shows the situation in the model where `client(1)` has sent a `PUBLISH` with a `QoS(1)` for the `topic(1)`. Similar to the subscription process, the place CtoB holds the message that the broker will receive, and the place Publishing keeps the information (`PID` and `topic` in this case) of the packet that needs to be acknowledged. The transition TimeOut models the behaviour for the re-transmission of packets. Quality of service level 1 assures that the message will be received at least once. The TimeOut transition will be enabled to re-send the message until the client has received the acknowledgement from the broker.

The `Broker` module models the logic for both receiver and sender behaviours. Figure 12 shows a marking corresponding to the state where the broker has processed the `PUBLISH` request made by `client(1)`, and it has generated both the answer to this client and the `PUBLISH` message for `client(2)` (in this case, only one client is subscribed to the topic). The port place BPID (Broker PID), at top right of Fig. 12, will hold a packet identifier for each message that the broker re-publishes to the clients. The port place Publishing keeps information

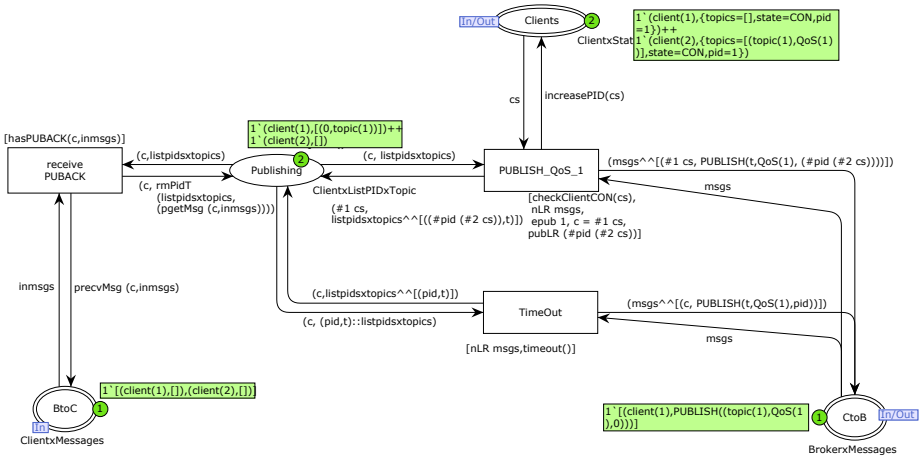

Fig. 11. PUBLISH_QoS_1 module after the PUBLISH_QoS_1 occurrence.

for all the clients that will acknowledge back the publish messages transmitted by the broker. Again, a TimeOut is modelled which, in this case, creates `PUBLISH` messages for all the clients subscribed to the topic in question. In the BtoC place (bottom right of Fig. 12), one can see that both messages have been sent, one for the original sender `client(1)` (`PUBACK` packet), and one for the only receiver `client(2)` (`PUBLISH` packet).

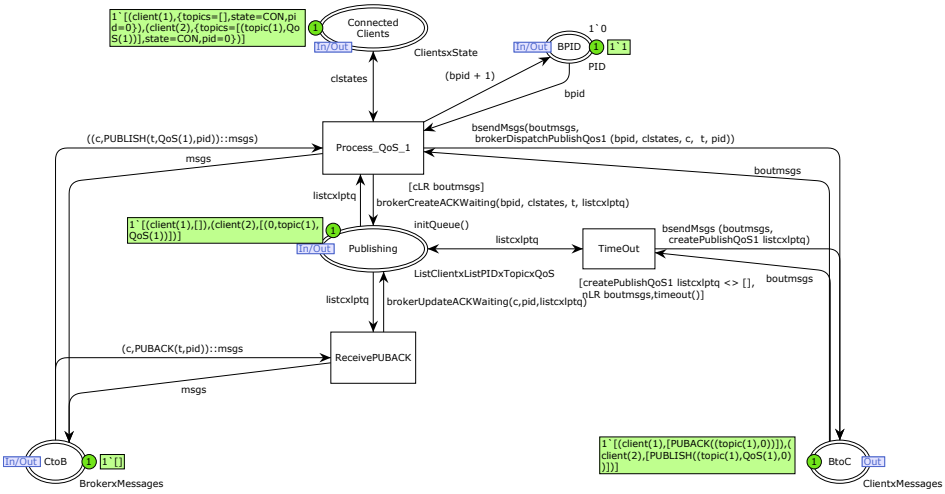

Fig. 12. Process_QoS_1 module after the Process_QoS_1 occurrence.

To finish the process, `client(2)` will notice that there has been a message published in `topic(1)`. Since `client(2)` is subscribed to this topic with `QoS(1)`, it must send a `PUBACK` acknowledgement to the broker to confirm that it has received the published message. Figure 13 shows the Receive_QoS_1 submodule in the clients side. The transition Receive_QoS_1 has been fired meaning that `client(2)` has received the publish message from the broker, and has sent the corresponding `PUBACK`. When the broker detects the incoming `PUBACK` message, it will check if there is some confirmation pending in the Publishing place (in Fig. 12 where `client(2)` is waiting for a `PID = 0` in `topic(1)` with `QoS(1)`.

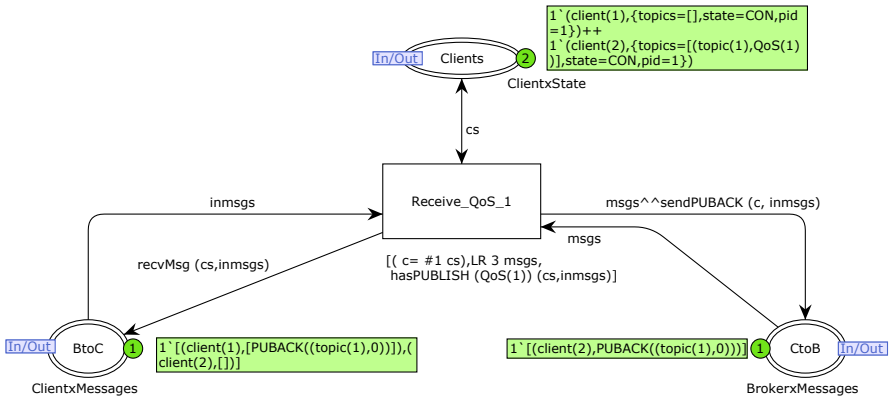

Fig. 13. Receive module after the transition Receive_QoS_1 occurrence

3.8 Findings

In the course of constructing the CPN model based on the informal MQTT specification, we encountered several parts that were vaguely defined and which could lead developers to obtain different implementations. The most significant issues are detailed below.

- There is a gap in the specification related to the MQTT protocol being described to run over TCP/IP, or other transport protocols that provide ordered, lossless and bidirectional connections. The QoS level 0 description establishes that message loss can occur, but the specification is not clear as to whether this is related to termination of TCP connections and/or clients connecting and disconnecting from the broker.
- It is specified that the receiver (assuming the broker role) is not required to complete delivery of the application message before sending the PUBACK (for QoS1) or PUBREC or PUBCOMP (for QoS2) and the decision of handling this is up to the implementer. For instance, in the case of QoS level 2, the specification provides two alternatives with respect to forward the

publish request to the subscribers: 1) The broker will forward the messages when it receives the PUBLISH from the original sender; or 2) The broker will forward the messages after the reception of the PUBREL from the original sender. Even it is assured that either choice does not modify the behaviour of the QoS level 2, this could lead to different implementation decisions and therefore consequent interoperability problems.
- The documentation specifies that when the original sender receives the PUBACK packet (with a QoS level 1), ownership of the application message is transferred to the receiver. It is unclear how to determine that the original sender has received the PUBACK packet. The same applies for QoS level 2 and the PUBREC packet.

4 Model Validation and Verification

During development of the MQTT protocol model we used single-step and automatic simulation to test the proper operation of the model. To perform a more exhaustive validation of the model, we have conducted state space exploration of the model and verified a number of behavioural properties.

We have conducted the verification of properties using an incremental approach consisting of three 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. The main motivation underlying this incremental approach is to be able to control the effect of the state explosion problem. Errors in the model will often manifest themselves in small configurations and leading to a very large state space. Hence, by incrementally adding the protocol features, we can mitigate the effect of this phenomenon. We identified several modelling errors in the course of conducting this incremental model validation based on the phases of the MQTT protocol.

In addition, we have developed a mechanism to be able to explore different scenarios and check the behavioural properties against them fully automatically. This has been done by providing the model with a set of parameterized options, which we can easily change. This feature allows us to first modify add or remove new configurations, and secondly to run them automatically. For each new modification in the parameters, we always run the six incremental executions and check the behavioural properties. Among others, one can quickly change the number of clients, the roles that such clients can perform (either subscriber, publisher, or both), switch between acyclic or cyclic version (where reconnection of clients is allowed) or enable/disable the possibility to retransmit packets (by means of timeouts).

To obtain a finite state space, we have to limit the number of clients and topics, and also bound the packet identifiers. It can be observed that there is no interaction between clients and brokers across topics as the protocol treats each topic in isolation. Executing the protocol with multiple topics is equivalent to running multiple instances of the protocol in parallel. We therefore

only consider a single topic for the model validation. Initially, we consider two clients. The packet identifiers are incremented throughput 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. Note that the use of bounds does not guarantee that the client uses packet identifiers up to bound. It is the guard on the transitions sending packets from the clients that ensures that these local bounds are enforced. Finally, we enforce an upper bound on the number of messages that can be in the message queues on the places CtoB and BtoC.

Below we describe each step of the model validation and the behavioural properties verified. The properties verified in each step include the properties from the previous step. We summarise the experimental results at the end. For the actual checking of properties, we have used the state and action-oriented variant of CTL supported by the ASK-CTL library of CPN Tools.

Step 1 – Connect and Disconnect. In the first step, we consider only the part of the model related to clients connecting and disconnecting to the broker. We consider the following behavioural properties:

S1-P1-ConsistentConnect The clients and the broker have a consistent view of the connection state. This means that if the clients side is in a connect state, then also the broker has the client recorded as connected.

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

S1-P3-ConsistentTermination In each terminal state (dead marking), clients are in a disconnect state, the broker has recorded the clients as disconnected, no clients are recorded as subscribed on both clients and broker sides, and there are no outstanding messages in the message buffers.

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

The two properties S1-P3 and S1-P4 imply partial correctness of the protocol as it states that the protocol can always be terminated, and if it terminates, then it will be in a correct state. The state space obtained in this step is acyclic when we do not allow reconnections. This together with S1-P3 implies the stronger property of eventual correct termination. This is, however, more a property of how the model has been constructed as in a real implementation there is nothing forcing a client to disconnect.

Step 2 – Subscribe and Unsubscribe. In the second step, we add the ability for the clients to subscribe and unsubscribe (in addition to connect and disconnect from step 1). For subscribe and unsubscribe we additionally consider the following properties:

S2-P1-CanSubscribe For each of the clients, there exists states in which both the clients and the broker sides consider the client to be subscribed.
S2-P2-ConsistentSubscription If the broker side considers the client to be subscribed, then the clients side considers the client to be subscribed.
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 considers the client to be unsubscribed.

It should be noted that for property S2-P2, the antecedent of the implication deliberately refers to the broker side. This is because the broker side unsubscribes the client upon reception of the unsubscribe message, whereas the client side does not remove the topic from the set of subscribed topics until the subscribe acknowledgement message is received from the broker. Hence, during unsubscribe, we may have the situation that the broker has unsubscribed the client, but the subscribe acknowledgement has not yet been received on the client side.

Step 3 – Publish and QoS levels. In this step we also consider publication of data for each of the three quality of service levels. As we do not model the concrete data contained in the messages, we use the packet identifiers attached to the message published to identity the packets being sent and received by the clients. In order to reduce the effect of state explosion, we verify properties for each QoS level in isolation. To make it simpler to check properties related to data being sent, we record for each client the packet identifiers of messages sent. For all three service levels, we consider the following properties:

S3-P1-PublishConnect A client only publishes a message if it is in a connected state.
S3-P2-CanPublish For each client there exists executions in which the client publishes a message.
S3-P3-CanReceive For each client there exists executions in which the client receives a message.
S3-P4-Publish Any data (packet identifiers) received on the client side must also have been sent on the client side.
S3-P5-ReceiveSubscribed A client only receives data if it is subscribed to the topic, i.e., the client side considers the client to be subscribed.

It should be noted that it is possible for a client to publish to a topic without being subscribed. The only requirement is that the client is connected to the broker. What data can correctly be received depends on the quality of service level considered. We therefore have one of the following three properties depending on the quality of service considered.

S3-P6-Publish-QoS0 The data (packet identifiers) received by the subscribing clients must be a subset of the data (packet identifiers) sent by the clients.

S3-P7-Publish-QoS1 The data sent on the client side must be a subset of the multi-set of packets received by the subscribing clients.

S3-P8-Publish-QoS2 The data received by each client is identical to the packet identifiers sent by the clients.

To check the above properties related to data received, we accumulate the packet identifiers received such that they can be compared to the packet identifiers sent. To simplify the verification of data received, we force (using priorities) both clients to be subscribed before data exchange takes places since otherwise the data that can be received depends on the time at which the clients were subscribed and unsubscribed.

Table 1 summarises the validation statistics where each configuration (scenario) is represented by a row comprised of Clients, Roles and Version. We report the size of the state space (number of states / number of arcs) and the number of dead markings (written below the state space size). We do not show the dead markings for the cyclic configurations as they are always 0. The columns S3.1, S3.2 and S3.3 correspond to the results considering QoS level 0, QoS level 1 and QoS level 2, respectively. Cells containing a hyphen represent configurations where the state space exploration and model checking did not complete within 12 hours which we used as a cut-off point.

Table 1. Summary of configurations and experimental results for model validation

N° Clients	Roles	Version	State space (states/arcs) Number of dead markings				
			Step 1	Step 2	Step 3		
					S3.1	S3.2	S3.3
2	1 sub / 1 pub	Acyclic	35/48 1	258/480 4	622/1074 21	1312/2616 21	3234/6394 21
	2 sub-pub		35/48 1	1849/4120 16	4282/8840 70	11462/23934 70	43791/85682 76
	1 sub / 1 pub	Cyclic	24/38	271/547	1149/2265	2376/5045	5996/12267
	2 sub-pub		24/38	2954/6798	8138/17714	20362/43572	79913/159254
3	2 sub / 1 pub	Acyclic	163/292 1	9529/25408 16	31765/76848 165	103176/262254 165	-
	1 sub / 2 pub		163/292 1	1262/2862 4	10360/21604 90	46721/120321 90	-
	2 sub / 1 pub	Cyclic	84/175	12650/35875	87450/235887	254095/679920	-
	1 sub / 2 pub		84/175	1057/2662	23817/59342	101794/279871	-

5 Conclusions and Related Work

We have presented a formal CPN model based on the most recent specification of the MQTT protocol (version 3.1.1 [3]). The constructed CPN model represents a formal and executable specification of the MQTT protocol. While performing an

exhaustive review of the MQTT specification to develop the model, we found several issues that might lead to not interoperable implementations. Consequently, this may add extra complexity for interoperability in the heterogeneous ecosystem that surrounds the application of a protocol such as MQTT.

The model has been built using a set of general CPN modelling patterns ensuring modular organisation of the protocol roles and protocol processing logic. Furthermore, we incorporated parameterization that makes it easy to change, among others, the number of clients and topics without having to make changes in the CPN model structure. In addition, we have applied modelling patterns related to the input and output message queues of the clients (publishers and subscribers) and brokers. These modelling patterns apply generally for modelling distributed systems that include one-to-one and one-to-many communication.

For the validation of the model, we have conducted simulation and state space exploration in order to verify an extensive list of behavioural properties and thereby validate the correctness of the model. In particular, our modelling approach makes it possible to apply an incremental verification technique where the functionality of the protocol is gradually introduced and properties are verified in each incremental step. A main advantage of the modelling patterns used for communication and message queues is that they avoid intermediate states and hence contributes to making state space exploration feasible.

There exists previous work on modelling and validation of the MQTT protocol. In [11], the authors uses the UPPAAL SMC model checker [7] to evaluate different quantitative and qualitative (safety, liveness and reachability) properties against a formal model of the MQTT protocol defined with probabilistic timed automata. Compared to their work, we have verified a larger set of behavioural properties using the incremental approach adding more operations in each step. In [13], tests are conducted over three industrial implementations of MQTT against a subset of the requirements specified in the MQTT version 3.1.1 standard using the TTCN-3 test specification language. In comparison to our work, test-based approaches do not cover all the possible executions but only randomly generated scenarios. With the exploration of state spaces, we considered all the possible cases. In [2], the authors first define a formal model of MQTT based on timed message-passing process algebra, and they conduct analysis of the three QoS levels. In contrast, our work is not limited to the publishing/subscribing process, but considers all operations of the MQTT specification.

We are planning to extend the features supported by the model in order to be able to simulate more sophisticated scenarios. For instance, we will allow the model to deal with persistence of data, so clients can receive the messages on reconnections lost suddenly in the middle of some operation. Furthermore, we plan to improve the mechanism to simulate loss of packets as an extension of the timeout system already implemented. In addition to aiding in the development of compatible MQTT implementations, the CPN MQTT model may also be used as basis for testing of MQTT implementations. As part of future work, we plan to explore model-based testing of MQTT protocol implementations following the approach presented in [16].

References

1. M. A. Adamski, A. Karatkevich, and M. Wegrzyn. *Design of embedded control systems*, volume 267. Springer, 2005.
2. B. Aziz. A formal model and analysis of an IoT protocol. *Ad Hoc Networks*, 36:49–57, 2016.
3. 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.
4. L. Baresi, C. Ghezzi, and L. Mottola. On accurate automatic verification of publish-subscribe architectures. In *Proceedings of the 29th international conference on Software Engineering*, pages 199–208. IEEE Computer Society, 2007.
5. J. Billington and M. Diaz. *Application of Petri nets to Communication Networks: Advances in Petri nets*, volume 1605. Springer Science & Business Media, 1999.
6. S. Chen, H. Xu, D. Liu, B. Hu, and H. Wang. A vision of IoT: Applications, challenges, and opportunities with china perspective. *IEEE Internet of Things journal*, 1(4):349–359, 2014.
7. A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, Aug 2015.
8. J. Desel, W. Reisig, and G. Rozenberg, editors. *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3018 of *Lecture Notes in Computer Science*. Springer, 2004.
9. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.
10. D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Intl. SPIN Workshop on Model Checking of Software*, pages 166–180. Springer, 2003.
11. M. Houimli, L. Kahloul, and S. Benaoun. Formal specification, verification and evaluation of the MQTT protocol in the Internet of Things. In *Mathematics and Information Technology*, pages 214–221. IEEE, 2017.
12. K. Jensen and L. Kristensen. Coloured Petri Nets: A Graphical Language for Modelling and Validation of Concurrent Systems. *Communications of the ACM*, 58(6):61–70, 2015.
13. K. Mladenov. *Formal verification of the implementation of the MQTT protocol in IoT devices*. Master thesis, University of Amsterdam, 2017.
14. MQTT essentials part 3: Client, broker and connection establishment. https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe.
15. A. Rodriguez, L. M. Kristensen, and A. Rutle. Complete CPN model of the MQTT Protocol. via Dropbox. http://www.goo.gl/6FPVUq.
16. R. Wang, L. Kristensen, H. Meling, and V. Stolz. Application of Model-based Testing on a Quorum-based Distributed Storage. In *Proc. of PNSE'17*, volume 1846 of *CEUR Workshop Proceedings*, pages 177–196, 2017.
17. F. Wortmann and K. Flüchter. Internet of things. *Business & Information Systems Engineering*, 57(3):221–224, 2015.
18. L. Zanolin, C. Ghezzi, and L. Baresi. An approach to model and validate publish/-subscribe architectures. In *Proc. of the SAVCBS*, volume 3, pages 35–41, 2003.