Model-Based Verification of the DMAMAC Protocol for Real-time Process Control

Admar Ajith Kumar Somappa
Bergen University College
University of Agder
aaks@hib.no

Andreas Prinz
University of Agder
andreas.prinz@uia.no

Lars M Kristensen
Bergen University College
lmkr@hib.no

Medium Access Control (MAC) protocols are responsible for managing radio communication that constitute the main energy consumer in wireless sensor-actuator networks. The Dual-Mode Adaptive MAC (DMAMAC) protocol is a recently proposed MAC protocol for process control applications within industrial automation. The goal of the DMAMAC protocol is to improve energy efficiency along with addressing real-time requirements for process control applications. The DMAMAC protocol switches between two operational modes that correspond to the two main states in process control: the transient state and the steady state. The state-switch is a safety critical function of the DMAMAC protocol (along with other functional properties) motivating the application of formal verification techniques. In this article, we use timed automata and the Uppaal tool to verify the design of the DMAMAC protocol. We have created a time-based model in Uppaal that constitutes a formal specification of the DMAMAC protocol. Using this model, we have successfully verified key properties of the DMAMAC protocol, thereby increasing confidence in the design of the protocol.

Model checking, Timed automata, Medium Access Control Protocols, Wireless Sensor Actuator Networks

1. INTRODUCTION

A Wireless Sensor Actuator Network (WSAN) (Akyildiz and Kasimoglu (2004)) consists of sensors and actuators that use radio to send, relay, and receive information. WSANs are used in a wide range of domains including process- and factory automation, smart home automation, and health-care. Feedback-based control loops that use wired or wireless solutions are collectively known as Networked Control Systems (NCS) (Hespanha et al. (2007)). NCS mainly use wired communication systems, but are increasingly adopting wireless communication. The salient feature of a wireless solution is the reduction in cost and size compared to the use of wired networks. The use of wireless communication, however, also has shortcomings and it has not yet become the de-facto replacement for existing wired solutions. The limitations of wireless solutions include low-bandwidth, energy efficiency, signal interference, and packet-loss. Energy efficiency in particular is an important concern when devices are battery powered.

Wireless solutions are made up of a collection of protocols that cater for different functions. Medium Access Control (MAC) is one of the functions that are critical to the proper operation of the entire WSAN. MAC protocols govern the communication and control the use of the radio on each node in the network. The radio module is the dominant consumer of energy in wireless nodes. The Dual-Mode Adaptive MAC (DMAMAC) protocol is a new MAC protocol recently proposed in (Kumar S. et al. (2014)) for process control applications. The protocol is aimed to provide an energy efficient solution. The DMAMAC protocol was proposed for NCSs with real-time and energy efficiency requirements. In particular, it targets process control applications that fluctuate between two states of operation: steady and transient. Fig. 1 shows a typical process control with two states. The transient state corresponds to the process state with large and frequent change in measurements of physical quantities, resulting in a high data rate. The steady state refers to the process state with measurements contained within a controlled range of values, thus requiring less data transfer. An example is process control for chemical reactors. The varying physical quantities are temperature and pressure which are measured by sensors. This can be controlled by varying the inflow of chemicals to the chemical reactors and using coolants, controlled by actuators. The state-switch is a safety critical feature of the DMAMAC

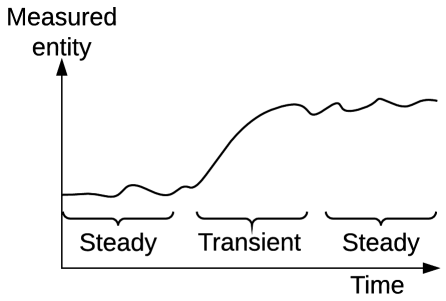

Figure 1: *Process control states*

protocol and can benefit from formal verification to ensure proper functioning.

Model-checking is a powerful technique for verification of protocol designs. Model-checking allows for exhaustive verification and has been widely used for verification of related protocols (see, e.g., (Fehnker et al. (2012, 2007); Tschirner et al. (2008)). Verification in the early design phase can be used to ensure the behavioral correctness of protocols. Model-checking assists in discovering design faults by exhaustively traversing all possible execution traces of a given model. Furthermore, model-checking tools can provide error-traces to failure states, thus assisting in resolving any discovered design issues. Uppaal (David et al. (2011)) is a modelling and verification tool-suite that supports model checking of real-time systems. In addition to model-checking and verification, Uppaal also supports simulation which can be used to provide useful insights into the working of a protocol.

In this article, we apply the Uppaal tool to analyse qualitative features of the DMAMAC protocol. We present a formal specification of the DMAMAC protocol in the form of a network of timed automata and verify safety properties related to the absence of faulty states. Additionally, we verify real-time properties including switch delay and maximum data delay. The timed modelling of the DMAMAC protocol is based on a Finite State Machine (FSM) representation of the sensors, actuators, and the sink node in the WSAN network configuration under consideration.

1.1. Related Work

Uppaal has been widely used to model and verify communication protocols (see, e.g., (Fehnker et al. (2012); Tschirner et al. (2008); Fehnker et al. (2007)). The Lightweight Medium Access Control (LMAC) (Fehnker et al. (2007)) protocol is the closest MAC protocol modelling related to the work presented in this article. The LMAC and the DMAMAC protocols are two distinct protocols with distinct goals, and differ significantly in their base features. The LMAC protocol is a self-organising protocol with nodes selecting their own slots (time duration for data transfer). The focus in the LMAC protocol verification is on efficient slot selection and collision detection. In the DMAMAC protocol, the slot scheduling is done statically and offline prior to deployment. The focus of the DMAMAC protocol is to provide an energy efficient solution along with efficient switching between the two operational modes. It requires a different model to represent the features of the DMAMAC protocol than the one used for the LMAC protocol. In (Tschirner et al. (2008)), the authors have focused mainly on modelling the Chipcon CC2420 transceiver. This work is related in terms of their use of a packet collision model and how collisions are observed. We use a collision model similar to (Tschirner et al. (2008); Fehnker et al. (2007)). With the extension of Statistical Model-Checking (SMC) features, Uppaal can also be used to assess performance related queries as shown in the case study (David et al. (2011)) of the Lightweight Medium Access Control (LMAC) protocol.

1.2. Outline

The rest of the article is organised as follows. In Sect. 2 we briefly introduce the DMAMAC protocol. For extensive details, we refer to (Kumar S. et al. (2014)). Section 3 describes in detail the constructed Uppaal model of the DMAMAC protocol. As part of this, we briefly introduce the constructs of timed automata as implemented in Uppaal, and perform some initial validation of the protocol model. In Sect. 4 we complete the validation of the constructed model. The verification of the protocol for different deployment configurations is discussed in Sect. 5. Finally in Sect. 6 we sum up the conclusions and discuss future work. The reader is assumed to be familiar with the basic concepts and operation of MAC protocols, including superframes and slots, and the principles of Time Division Multiple Access (TDMA) and Carrier Sense Multiple Access (CSMA).

2. DMAMAC PROTOCOL

The DMAMAC protocol (Kumar S. et al. (2014)) has two operational modes catering for the two states of process control applications: transient mode and steady mode. The protocol is based on Time-Division Multiple Access (TDMA) for data communication and a Carrier Sense Multiple Access (CSMA)-TDMA hybrid for alert message communication. The basic functioning of the protocol is based on the GinMAC protocol (Suriyachai et al. (2010)) proposed for industrial monitoring and control. The network topology of the DMAMAC protocol consists of sensor nodes, actuator nodes, and a sink. The sensor nodes are wireless nodes with sensing capability which sense a given area and update the sink by sending the sensed data.

The actuator nodes are wireless nodes equipped with actuators, which act on the data performing a physical operation. It is also possible to have wireless nodes with both sensors and actuators. The sink is a computationally powerful (relative to the nodes) wire powered node which collects the sensed data, performs data processing on it, and then sends the results to corresponding actuators.

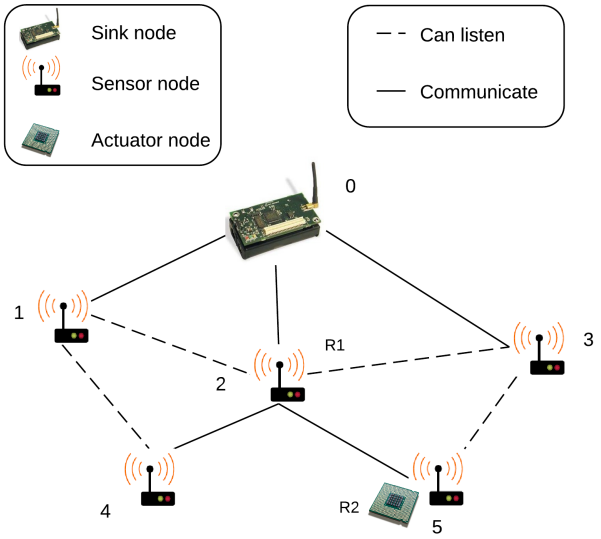

Figure 2: *The network topology for DMAMAC protocol*

Similar to the GinMAC protocol, the network deployment for the DMAMAC protocol is based on a tree topology as shown in Fig. 2. The solid lines between nodes represent data communication. The dashed lines represent nodes which can hear each other, but which have no direct data communication with eachother. Each level in the tree topology is ranked (marked with “R#”), with the sink having the lowest rank number and the farthest leaf nodes having the highest rank number. This ranking is exploited in the alert message sending procedure. Firstly, we discuss the key assumptions that were made to support the design of the protocol. Further, we explain in brief the working of the two operational modes and the respective superframes they use.

- The nodes are assumed to be time synchronized via an existing time synchronization protocol. Thus, the time synchronization mechanism is not defined as a part of the protocol.
- The sink is assumed to be powerful, and it can reach all nodes with one hop.
- A pre-configured static network topology with no mobility is assumed.
- A single slot accommodates both a data packet and a corresponding acknowledgement.

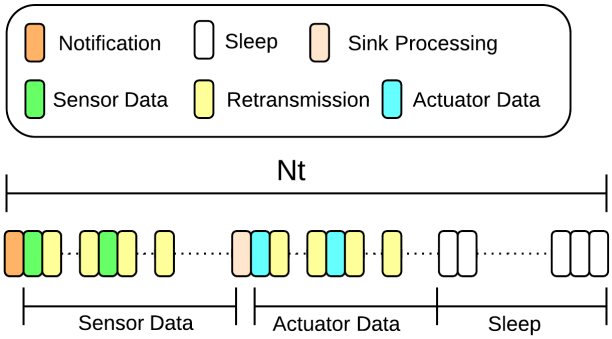

Figure 3: *The transient superframe of the DMAMAC protocol*

2.1. Transient mode

The transient mode is designed to imitate the transient state operation in process control. During transient state, the process changes rapidly generating data at a faster rate relative to the steady mode. During the transient mode operation, the DMAMAC protocol uses the transient superframe shown in Fig. 3. The superframe includes a data part for data transfer from the sensors to the sink, followed by a data part with data being sent from the sink to the actuators, and then a sleep part. The data part also includes a notification message slot from the sink to all nodes, and a sink processing slot. A typical transient mode operation cycle is described below:

- A notification message is sent from the sink to all the nodes. The notification message includes control data like state-switch message, and time-synchronization. Time-synchronization is an integral part of TDMA based protocols.
- The data part is executed with data transmission from sensors to sink to actuators.
- The sleep part is executed where all sensors and actuators enter sleep mode in order to improve energy efficiency. This part represents the situation where all nodes are in sleep mode. Individually, the nodes are in sleep mode when they are not performing other tasks.

2.2. Steady Mode

The steady mode operation is designed to operate during the steady state of the process control application. The steady superframe used in the steady mode operation is shown in Fig. 4. In addition to the parts that also exist in the transient superframe, the steady superframe contains an alert part. The alert part is used to ensure that the state-switch from steady to transient occurs whenever a sensor detects a threshold interval breach in its reading. This threshold is set by the sink when the switch from transient to steady is

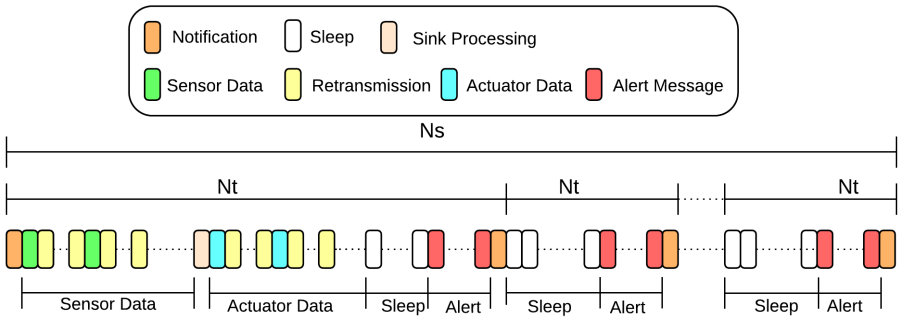

Figure 4: *The steady superframe of the DMAMAC protocol*

made. Note that w.r.t (Kumar S. et al. (2014)) a slightly modified steady mode superframe is used. There are notification slots placed at the end of each transient (Nt) part. This is done to facilitate immediate application of alert, and making a state-switch. In the alert part, one slot is allocated to each level or to nodes with the same rank. All the nodes in the same rank have the possibility to send an alert message in this slot. The alert sending method is described later. A typical steady mode operation cycle is described as follows:

- A notification message is followed by the data and the sleep part, similar to the working in transient mode operation.
- (Alert part) Sensor nodes that have alert messages to be communicated use appropriate slots provided for each rank to notify parents about the alert. This is relayed towards the sink which then makes the switch to the transient state. In an absence of alert, sensor nodes still wake up on their alert receive slot and then enter sleep mode until the next notification slot.
- In the alert part, the notification slot is placed at the end. This is to ensure a quick transition between the two states. All regular nodes wake up in this slot, and receive a notification message from the sink. Alert notification to change superframes is sent here.

2.3. Change of superframes

A process switches between two states: transient and steady. The DMAMAC protocol follows these states via its transient and steady mode operation. There are two switches possible here: transient to steady and steady to transient. The latter is a critical switch since the data rate in transient is higher and it is important to accommodate the higher data rate in transient state. The switch from transient to steady is decided by the sink, which determines if the process is in steady state based on previous readings. When the sink decides to make the switch, it informs all the nodes in the network to change their mode of operation. The message is sent via a notification message from the sink. When the sink node switches from transient to steady, it defines a threshold interval within which the sensor readings should lie, and informs the sensors about this threshold interval. During the entire steady mode operation, the sensors constantly monitor for a threshold breach. When there is a breach, the sensor node waits until its alert slot, and notifies its parent which in turn forwards it towards the sink. The sink then informs the nodes in the network to switch to transient in its immediate next notification message.

2.4. Alert Message

An alert message is the message created by the sensor nodes to notify the sink that a state-switch is required. The sensor nodes choose a random delay in the slot before transmitting the alert message. At the completion of the time duration of the random delay, the nodes sense the channel to prevent collision. If a node during channel assessment detects another node sending an alert message, then it just drops its alert message. Collisions are still possible, e.g., when two nodes choose the same random delay or when two senders cannot listen to each other but the can receiver listen to both. Nodes check for change of operational mode following the sending of the alert. If no change occurs (because of collision) then they save the alert and send the alert again in the next alert slot.

3. THE DMAMAC UPPAAL MODEL

Uppaal (David et al. (2011)) is a tool-set based on timed automata for model-checking of real-time systems. It is an integrated tool environment that supports modeling, simulation, validation, and verification. An abstract representation of a real-time system in the form of a model is structured as a network of timed automata. The query language of Uppaal allows for verification of safety, reachability, liveness, and time-bounded properties. In Uppaal, models are constructed as a network of templates based on timed automata. Templates are used to represent independent entities (e.g. a sensor node). Uppaal consists of two simulators: a symbolic and a concrete simulator. The symbolic simulator is used to

inspect the execution of the model step by step. For certain queries, Uppaal outputs traces which can be viewed in the symbolic simulator. This is useful for pin-pointing error locations and sequences of events that lead to errors/faults. The symbolic simulator also shows all the templates in the model and message sequence charts (MSC) can be used to represent communications between different processes. The symbolic simulator also allows interactive step-wise simulation of the model. Along with features similar to the symbolic simulator, the concrete simulator has the added advantages of firing transitions at a specified time. For extensive detail on modeling in Uppaal, we refer to (Behrmann et al. (2004)).

3.1. Model design decisions and assumptions

We use a non-deterministic timed automata model to verify the properties of the DMAMAC protocol. The constructed model has several sources of non-determinism including the alert delay for sending alert messages in nodes, and the decision made by the sink to change from the transient mode to the steady mode. Given the design of alert messages, also collisions are possible when sending alert messages. We use a simplified collision model, detailed later in this section. The sink and sensor/actuator nodes have separate timed automata models. Local clocks are used for each automaton. One of the clocks used is a global clock used for the common network time reflecting the assumption on time synchronisation between the nodes. The main aim of the verification of the DMAMAC protocol model is to check that the two modes of operations are working correctly given the presence of non-deterministic choices (like collision) during execution and the delays that may occur.

Below, we discuss the assumptions and design decisions made during the construction of the Uppaal model for the DMAMAC protocol.

- Packets are abstractly modeled without payload. The messages or packets exchange mechanism is represented by channel synchronization in the Uppaal model.
- A time synchronization mechanism is provided using clock variables in Uppaal. This can be considered as a way of implementing the time synchronization between nodes assumed by the DMAMAC protocol.
- An exact model of CSMA results in a rather complex model. Instead, we use a representative CSMA procedure, which imitates the service and effects of actual CSMA on the working of the protocol. The effects include skipping packet transmission on detection of ongoing transmission and also collision. This makes our model and verification independent of the particular CSMA procedure that may be used in conjunction with DMAMAC.
- The collision caused due to the use of CSMA has effects on the state-switch procedure. A simple collision model is used, where we record collision when two or more nodes send packets at the same time. Collision results in failure of the packets, thus affecting the state-switch procedure.

A channel synchronization variable *choice* is used to force enabled transmissions. This is a modeling artifact and is not part of the protocol as such. In Uppaal, execution of models can stay in a location indefinitely even after outgoing edges are enabled. To force the model execution to continue via enabled outgoing edges, an urgent channel synchronization is required.

3.2. Sink Model

The sink model is shown in Fig. 5. We have used colors in the automaton location to differentiate between states. Both the sink and the node automatons begin in an initial location **Start**. The sink initiates the network startup procedure of the network using a broadcast synchronization channel *startup* on the edge towards the **StateController** location. The function INITIALIZE() is used to set proper values to local and global variables. The sink reaches the **StateController** location upon having executed the startup procedure of the network. The node automatons synchronize with the channel variable *startup*, and reach the **StateController** location.

The **StateController** location represents an event handler which handles the transition between different states in the state-machine. The sink model uses a local clock variable "x", which is active in all states indicated by "$x' == 1$". This is used as an invariant on all states to represent continuously running time. It also includes an invariant $x \leq currentMaxSlots * 10$ to prevent it from being in the state beyond the maximum timeframe of the active superframe (transient or steady). A typical slot time in WSANs is 10 milliseconds, and thus we use the unit "ms" for time in our model. Given the time unit of "ms", the variable *currentMaxSlots* is multiplied by 10 to obtain the slot-time. The use of the **StateController** location is also similar to the *self-message* handler in the commonly used OMNeT++ (Varga and Hornig (2008)) framework for the MAC protocols. The function of this particular handler is to check the self-message that it receives, and to act on the message by choosing an appropriate next state. Also, it determines the next state which is then sent as a self-message. The automaton changes between the different locations (states) in the model

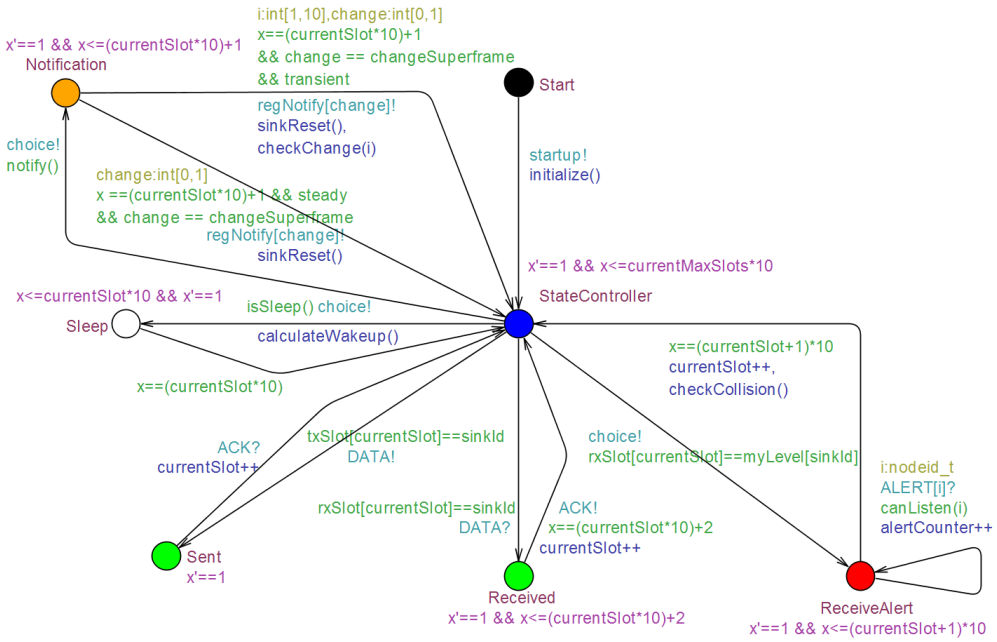

Figure 5: *Uppaal Model of the sink*

based on the local variable *currentSlot*, and the local clock variable x.

The **Notification** location is reached when the sink is due to send a notification message. The notification message is sent by the sink, and received by other nodes in the network. The notification message sending is represented by the broadcast channel *regNotify[change]*, where *change* carries a message of the status of the Boolean variable *changeSuperframe*. The *changeSuperframe* variable is true when the sink needs to indicate to the nodes a change in superframes to switch the mode of operation. For both the steady to transient switch and the transient to steady switch, the sink uses *changeSuperframe* to indicate the change.

The switch from transient to steady is decided by the sink. There are two separate notification edges for transient mode and steady mode. In the transient mode, the sink decides if it has to switch to steady based on the random selection statement ($i : int[1, 10]$) and the obtained change value is sent over the channel. In the absence of real inputs a random selection is used. The edge with select statements ($i : int[1, 10]$) and ($change : int[0, 1]$) is used in transient mode. The second select statement ($change : int[0, 1]$) is a modeling artifact used to be able to send the value of *changeSuperframe* over the channel via the synchronization variable *regNotify[change]*. The guard $change == changeSuperframe$ makes sure that the select statement selects the same value as the *changeSuperframe* variable.

For the steady mode the sink only conveys change of superframe status based on the receiving of alert from nodes. The notification is done via the edge with only one select statement ($change : int[0, 1]$). Also, as a symbolic representation, we have used a guard $x == (currentSlot * 10) + 1$ on these edges, and an invariant $x \leq (currentSlot * 10) + 1$ on location **Notification** to indicate a delay of 1 ms for sending notification message. Both of these edges uses a function SINKRESET(), which resets the sink variables at the beginning of a new superframe including changing of superframes (*currentMaxSlots* reset).

The **Sleep** location is reached when the sink or nodes do not have any active operations to be conducted in the current slot. The edge to **Sleep** is guarded by a Boolean function ISSLEEP() which checks if the current slot is a sleep slot. We use the urgent broadcast channel *choice* (model artifact) to force this transition whenever ISSLEEP() evaluates to true. In the absence of this channel variable, the model can continue to be in the location **StateController** even when ISSLEEP() is true. The location **Sleep** has an invariant $x <= currentSlot * 10$ which indicates that during execution, the control can be in the location as long as the time does not exceed the value *currentSlot*, which holds the value of the slot at which the sink should wake up for its next event. This is set by the function CALCULATEWAKEUP() when **Sleep** is reached.

The location **Sent** is reached when the sink sends data in its data slot. The **Received** location is reached when the sink receives any sensor

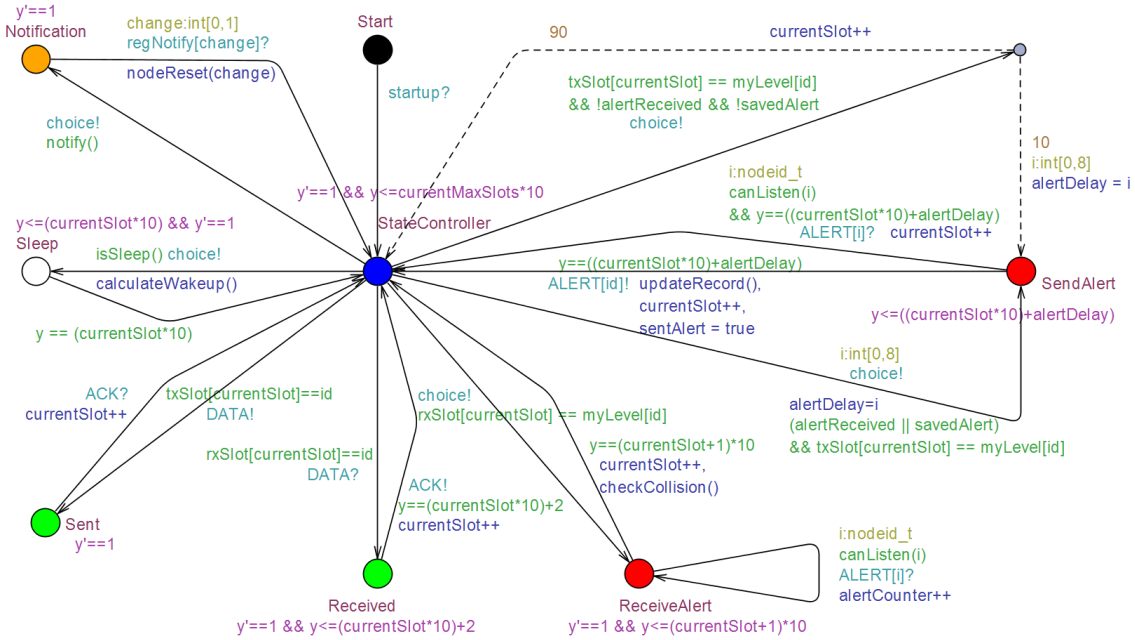

Figure 6: *Uppaal Model of a regular sensor/actuator node*

data, and then sends an ACK via channel synchronization. Location **Received** has an invariant $x \leq (currentSlot * 10) + 2$. This invariant is used to add a delay of 2 ms as a representation for the time required for data communication. A follow up guard on the ACK sending edge $x == (currentSlot * 10) + 2$ makes sure that the delay is applied. Upon sending or receiving of ACK synchronization, the local variable *currentSlot* is incremented.

Lastly, the location **ReceiveAlert** is reached when it is the sink's turn to receive an alert. This is determined by the alert levels defined in the *myLevel* array variable represented by the guard $rxSlot[currentSlot] == myLevel[sinkId]$. The sink stays in the location for an entire slot duration (10 ms), and waits for any alerts from nodes it can listen to. The CANLISTEN(I) function is used as a guard to make sure that the sink listens to alerts from only those nodes that are in its listening range (same function used for nodes). The variable i given as input to the function is the result of a select statement $i : nodeid\_t$, which allows the node to listen to any node that is transmitting. The guard makes sure that the sink can listen to that particular node. At the completion of the alert slot, the sink checks if any collision has occurred via the function CHECKCOLLISION(), and gives back the control to the **StateController**.

3.3. Sensor/Actuator Node Model

The sensor/actuator node model is shown in Fig. 6. The node model is similar to the sink model except for the notification handling procedure. Also, the node template consists of an extra location for sending alert messages. The notification part of the node model is simpler, since nodes only receive notifications. Location **Notification** is reached when the node is in its notification slot (receive) in either of the two types of superframes. The nodes then synchronize on the channel variable *regNotify[change]* from the sink, and reset the node variables using the function NODERESET() based on the value of the variable *change*.

The node model works similar to the sink model for sleep, sent (data), received (data), and alert receive. This means that in locations **Sleep**, **Sent**, **Received**, and **ReceiveAlert** the node and the sink model have the same modeling elements. Further, the location **SendAlert** which handles the crucial part of alert message sending is required by the protocol for the switch of operational mode from steady to transient. Based on the protocol specification, a node can send an alert message when the sensed data crosses the threshold interval. This threshold interval is set by the sink depending on the particular process being controlled. We imitate this event using a probability weight-based selection for sending alert messages as reflected in the edge towards **SendAlert** (shown with dashed lines). The edge with probability weight 90 represents no alert to be sent. The one with probability weight 10 represents the choice to send an alert. The guards on the edge make sure it is the alert slot of the node, and that the node does not already have an alert to be sent. When the node chooses to send an alert, an alert delay is chosen within the interval $[0, 8]$ via the select statement $i : int[0, 8]$. The chosen value is assigned to the *alertDelay* variable. The node then waits in

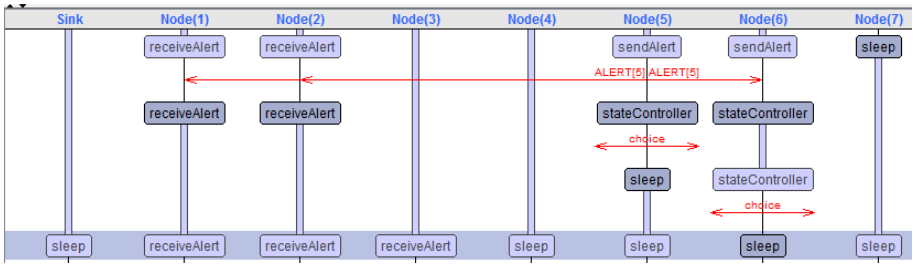

Figure 7: *Message sequence chart for carrier sense during Alert*

Listing 1: Carrier Sense trace

```
(Sleep, receiveAlert , ReceiveAlert , ReceiveAlert , Sleep, StateController , StateController , Sleep)
choice: Node(5)[][3]−> // Delay of 3 ms chosen by node 5
(Sleep, ReceiveAlert , ReceiveAlert , ReceiveAlert , Sleep, SendAlert, StateController , Sleep)
choice: Node(6)[][3]−> // Delay of 3 ms chosen by node 6
(Sleep, ReceiveAlert , ReceiveAlert , ReceiveAlert , Sleep, SendAlert, SendAlert, Sleep)
ALERT[5]: Node(5)−> Node(1)[5]Node(2)[5]Node(6)[5] // Alert send by node 5 is heard by node 2 and node 6
(Sleep, ReceiveAlert , ReceiveAlert , ReceiveAlert , Sleep, StateController , StateController , Sleep)
```

the location **SendAlert** for the duration of the alert delay and performs carrier sense prior to sending the alert message. This is represented by the edge with a the guard function CANLISTEN(I), where the node synchronizes to the broadcast channel *ALERT[i]* sent by other nodes in the vicinity (listening range) to skip sending a message.

We use a representative carrier sense mechanism in the model. Nodes skip sending an alert message when another node within their listening range is sending an alert message with the same alert delay. In reality, carrier sense would involve listening to the channel for a small duration before sending the packets. Also, in a case where two nodes start carrier sense at the same instant, their packets would collide since they would start sending at the same instant after the carrier sense delay. In the carrier sense mechanism presented here, we represent a case in which two nodes can hear each other and have the same alert delay by one of the two nodes skiping the sending the alert message. When the nodes do not hear eachother and the receiver can hear both, the packets collide at the receiver. An example message sequence of carrier sense is shown in Fig. 7. In this example, Node(5) and Node(6) are trying to send alert with the same alert delay (3ms) as shown in List. 1. In the listing, we have added comments with prefix "//" to add more detail. When Node(5) begins to send the alert, Node(6) senses the sending and skips sending alert via the edge guarded by CANLISTEN(I) function.

In a case where the channel is free, the nodes send an alert at the time instant after the chosen alert delay. The sending is represented using the send part of the broadcast channel variable *ALERT[id]!*. The local variable *currentSlot* is updated, along with the variable *sentAlert*, and function UPDATERECORD(). The variable *sentAlert* is used by the node to remember that it has sent an alert. In a case where no superframe change occurs after an alert was sent, a node updates its local variable *savedAlert*. The UPDATERECORD() function updates a global array variable *alertTimeRecord[]* which stores the alert delay chosen by each node in the given round. This is used to check if collision has happened. In certain cases when the alert messages fail to reach the sink due to collision, the *savedAlert* variable is used to save the alert and are sent in the next round. During this, the probability edge is not used. Instead, the nodes directly move to the location **SendAlert** via the edge (solid line) with the guard $(alertReceived||savedAlert)$. The variable *alertReceived* represents the case when nodes have to forward an alert received from other nodes towards the sink. The nodes then choose a new delay value from the interval $[0, 8]$ for sending the alert message again.

3.4. Collision

We use a simple collision model similar to the one used in (Tschirner et al. (2008)) and (Fehnker et al. (2007)). In the LMAC (Fehnker et al. (2007)) protocol, when two nodes send a packet in the same slot it is considered as a collision. In the DMAMAC protocol model, collision is counted when a node receives at least two alert messages with the same alert delay in the same alert slot. In our model, we assume that apart from its child nodes, the parents can also listen to nodes in the vicinity (similar to real networks). We define statically which other nodes a given node can listen to. Based on the representative carrier sense model, the collision occurs at a node only when it receives two alert messages from nodes (of the same rank) that cannot listen to each other, and had chosen the same alert delay within the alert slot. A message sequence chart showing a collision occurrence at the sink is shown in Fig. 8.

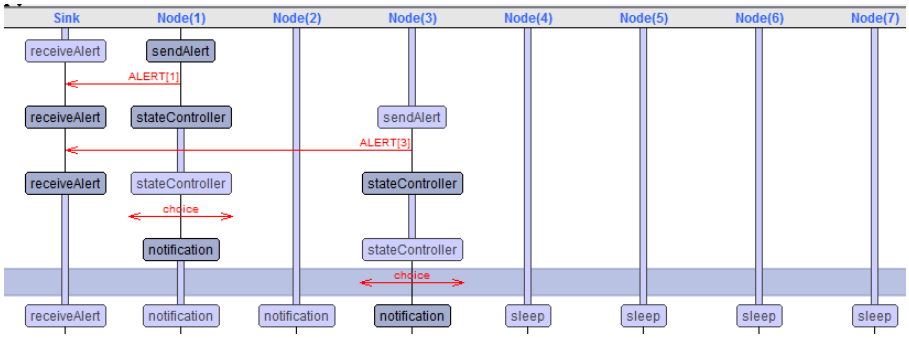

Figure 8: *Message sequence chart for collision at the sink*

Listing 2: Collision Trace

```
(ReceiveAlert, StateController, Notification, StateController, Sleep, Sleep, Sleep, Sleep)
choice: Node(1)[][7]−> // Alert delay of 7 ms chosen by node 1
(ReceiveAlert, SendAlert, Notification, StateController, Sleep, Sleep, Sleep, Sleep)
choice: Node(3)[][7]−> // Alert delay of 7 ms chosen by node 3
(ReceiveAlert, SendAlert, Notification, SendAlert, Sleep, Sleep, Sleep, Sleep)
ALERT[1]: Node(1)−> Sink[1] // Alert sent by node 1 to the sink
(ReceiveAlert, StateController, Notification, SendAlert, Sleep, Sleep, Sleep, Sleep)
ALERT[3]: Node(3)−> Sink[3] // Alert sent by node 3 to the sink
(ReceiveAlert, StateController, Notification, StateController, Sleep, Sleep, Sleep, Sleep)
```

The trace corresponding to the MSC in Fig. 8 is shown in List. 2. In the considered scenario, Node(1) and Node(3) choose the same alert delay of 7 (ms) independently. Since they cannot listen to each other their alert packets end up colliding at the sink. This prevents a change of the superframe (mode of operation). Node(1) and Node(3) detect this and save the alert. The saved alert is used to resend the alert in the next round (with a new delay) to make sure the superframe changes. Note that there could be a situation where collision occurs at lower levels (and even at the sink), but still the change of superframe occurs because of another alert message reaching the sink. For our model, we have created the topology in such a way that both cases exist in different configurations as discussed in Sect. 5. In reality, the receiver nodes do not detect collision: in certain cases nodes receive parts of packets that collide (difficult to decode them) and in other cases they receive nothing at all. In that respect, we rely on a representative model of collision detection designed to be consistent with the effects of collision on the change of superframe.

3.5. Network Topology

The node topology used for the verification of the models is shown in Fig. 9. We use 5 sensor nodes, 2 sensor-actuator nodes, and a sink in the tree topology considered. We consider a small topology to keep the state-space small which is needed in order to conduct exhaustive verification. The current node topology has 3 ranks but since the sink only listens (and does not send alerts), we have 2 alert slots in the DMAMAC protocol. This representative topology allows for both carrier sense and collision, has both sensors and actuators with data communication for both types of nodes, and also has multiple hops. A topology based assumption for listening range is that a higher level (lower rank) node is listening to all of its children nodes, and also sometimes to other nodes in the vicinity. A real node has a listening range based on its receiver sensitivity, and distance with other nodes in the vicinity that varies with topology. Given that our main aim is to check the working of the protocol, we define the listening range in the topology manually instead of calculating it dynamically based on multiple factors like node position, path-loss, and receiver sensitivity as is typically done in network simulators for quantitative analysis.

In the topology used for the evaluation of the DMAMAC protocol the functionalities that need to be verified are covered. The DMAMAC protocol is used for applications with offline scheduling. This means that scheduling is done prior to deployment and all slot allocations are known prior to deployment. The topology in general is well-planned, and no random deployment is used. A real topology would be much larger than the one considered here. In the current topology, we have 3 levels and a maximum of 2 hops. For a qualitative analysis this covers the error scenarios that could potentially exist with multiple-hops.

The schedule for the considered node topology is shown in Fig. 10. The schedule shows both sender/receiver identification (node/sink). The sending part is marked by TX and receiving part is marked by RX. For notification messages, only the sender identification (sink) is marked. The schedule only represents the steady superframe. In the transient superframe only the first *Nt* part is used with

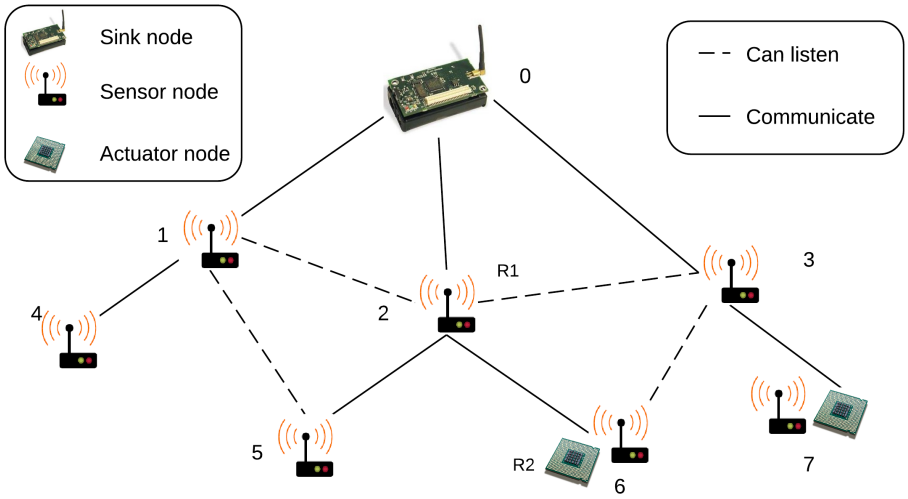

Figure 9: *Node topology*

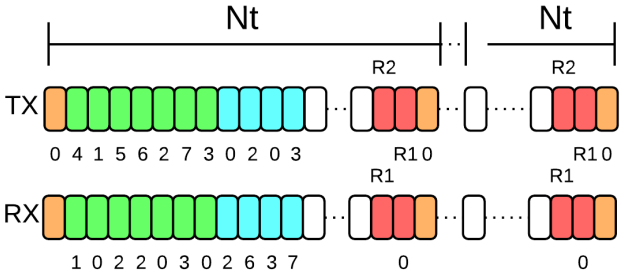

Figure 10: *Superframe structure based on the schedule and node topology*

the alert parts replaced by sleep. Note that we use 10 ms as slot duration and that time is in general is measures in “ms” (milliseconds).

3.6. Configurations

Multiple configurations of the DMAMAC protocol can be analysed based on values that can be varied in the model. Firstly, steady and transient: the Uppaal model can start in either steady or transient mode and this could have an effect on some of the verification properties (as discussed in Sec. 5). Another important factor affecting the configurations is the range of possibilities for the variable *alertDelay*. In the protocol, we have used the range [0,8] to reduce collision. Due to state-space issues we use only $alertDelay[1,1]$ for exhaustive queries, e.g., deadlock query. The $alertDelay[1,1]$ in itself covers all possibilities including possibility of state-switch, collision and CSMA, and hence all the qualitative aspects of the protocol. For other non-exhaustive queries, we use up to $alertDelay[1,4]$ configuration to further validate the verification procedure. The only difference between $alertDelay[1,1]$ and the other considered configurations is the applicability of property *sink mode* and *consistent node mode* of the verification properties and is further discussed in Sec. 5. Also, the select statement interval $[1,10]$ used to decide the switch from transient to steady mode by the sink is reduced to $[1,2]$ to keep the state-space low for all the queries. The reduction of the interval only means that in transient mode there is a 50% probability to switch to steady mode, and thus does not affect the qualitative results.

4. MODEL VALIDATION

We first validate the constructed Uppaal model of the DMAMAC protocol by checking some basic behavioural properties related to the operation of the model. The purpose is to obtain a high degree of confidence in the constructed model prior to verifying key properties of the protocol in the next section. During construction of the DMAMAC protocol, we validated the operation of the model via MSCs obtained from step-by-step execution of the model in the Uppaal simulator. These properties were related to data transmission between nodes, data transmission between the nodes and the sink, sending/receiving of alert message, possibility of collision, and carrier sense when sending alert messages.

Below we validate properties of the model related to data communication and collisions using the verification engine of Uppaal. For this, we express the properties to be validated in the form of Uppaal queries. Queries in Uppall are written in a restricted variant of Computation Tree Logic (CTL) in which path formulas cannot be nested. Specifically, the following path formulae are supported by Uppaal: A□ (always globally), E♢ (reachable), E♢ (always eventually), and E□ (exists globally).

For validation purposes, we first check the operation of the model with respect to data communication

between neighbouring nodes and between the sink and its neighbouring nodes. We check that if two nodes i and j are such that the parent node of node j is i, then these will eventually communicate. Furthermore, it should be such that any child node of the sink node should eventually communicate with the sink. Formally, these two properties can be expressed as the set of queries below. Here, $nodeid\_t$ is the type used to represent node identifiers in the model, $parent[i]$ is used to obtain the parent node of node i, and $sinkId$ denotes the identity of the sink. The property is expressed by reference to the location **Sent** and location **Received** which are reached by the communicating nodes upon synchronization over the channel *DATA*.

Node data communication

∀ i,j ∈ nodeid_t such that parent[j]==i:
A◇ (Node(i).Sent && Node(j).Received)

Sink data communication

∀ i ∈ nodeid_t such that parent[i]==sinkId:
A◇ (Node(i).Sent && Sink.Received)

It should be noted that we do not check the property that two neighbouring nodes always have the possibility to communicate. This is due to the fact that Uppall does not support nesting of CTL path formulae.

The second property that we validate is related to collisions which plays an important role in the DMAMAC protocol in relation to the sending of alert messages. In this case, we check that it is possible to have collision happening on all nodes and on the sink. Collision cannot be guaranteed to happen and hence we verify only the possibility of collision occurring. Formally, these two properties are expressed as the following set of queries:

Node collisions ∀ i ∈ nodeid_t : E◇ Node(i).collision

Sink collisions E◇ Sink.collision

Finally, we also validate that there are no deadlocks in the model. In Uppaal, this can be expressed via the query below where *deadlock* is a built-in state property in Uppaal.

No deadlock A□ !deadlock

The above queries related to data communication, collision, and deadlocks were all verified on both the transient and the steady variant of the model. This in turn increased confidence in the proper operation of the model.

5. PROTOCOL VERIFICATION

We now consider verification of the key functional properties of the DMAMAC protocol. As explained earlier, the constructed model comes in two variants: one variant with the protocol starting in the transient mode and one variant with the protocol starting in the steady mode. We first consider common properties that are independent of whether the protocol starts in the transient or in the steady mode. Then we consider properties specific for the transient mode case followed by properties specific for the steady mode case. Finally, we verify two real-time properties of the protocol related to upper bounds on mode switch delay and data transmission delay.

5.1. Common Properties

Given the dual-mode operation of the DMAMAC protocol, the important properties relate to the nodes operating in different modes, and switching between them. Firstly, we check the operating mode properties. We make sure that the sink is exclusively either in the steady mode or in the transient mode at all times. Following this, we check that all nodes follow the operating mode of the sink consistently. Formally, these properties are expressed as follows:

Sink mode

A□ (Sink.steady && !Sink.transient) || (!Sink.Steady && Sink.transient)

Consistent node mode

∀ i ∈ nodeid_t:
A□ (Node(i).transient == Sink.transient || Node(i).steady == Sink.steady)

Next, we investigate properties of the protocols related to collision and its effect on the change of operational modes. The queries refer to the *changeSuperframe* variable which indicates whether the network should change mode in the next superframe. Collisions may have different effects depending on the configuration under consideration. For configurations with $alertDelay[1,1]$ where all the nodes will pick the same alert delay, collision at the sink should not result in a change of superframe or operational modes. For configuration with $alertDelay[1,2]$, we may have both collision and change of superframe since, e.g., two nodes may pick an alert delay of 1 (which will result in a collision) while a single third node picks an alert delay of 2. The latter choice will result in the sink being notified of a required change of mode. Formally, properties related to collisions and change of mode are specified as follows:

Collision and mode switch

E◇(Sink.collision && Sink.changeSuperframe)

Collision and no mode switch

E◇(Sink.collision && !Sink.changeSuperframe)

Following the discussion above, we expect that the first property is false in configurations where all nodes must choose the same alert delay while it is true in configurations where different alert delays

Listing 3: Collision and change of superframe together

```
(ReceiveAlert, StateController, StateController, StateController, Sleep, Sleep, Sleep, Sleep)
choice: Node(1)[][1]-> // Node 1 chose alert delay of 1 ms
(ReceiveAlert, SendAlert, StateController, StateController, Sleep, Sleep, Sleep, Sleep)
choice: Node(3)[][1]-> // Node 3 chose alert delay of 1 ms
(ReceiveAlert, SendAlert, StateController, SendAlert, Sleep, Sleep, Sleep, Sleep)
choice: Node(2)[][2]-> // Node 1 chose alert delay of 2 ms
(ReceiveAlert, SendAlert, SendAlert, SendAlert, Sleep, Sleep, Sleep, Sleep)
ALERT[3]: Node(3)->Sink[3] // Node 3 sends alert with its alert delay
(ReceiveAlert, SendAlert, SendAlert, StateController, Sleep, Sleep, Sleep, Sleep)
ALERT[1]: Node(1)->Sink[1] // Node 1 sends alert with its alert delay
(ReceiveAlert, StateController, SendAlert, StateController, Sleep, Sleep, Sleep, Sleep)
ALERT[2]: Node(2)->Sink[2] // Lastly, Node 2 sends alert with its alert delay (higher than others two)
(ReceiveAlert, StateController, StateController, StateController, Sleep, Sleep, Sleep, Sleep)
```

can be chosen. This implies that the protocol design ensures that the DMAMAC protocol can change mode even in the presence of collisions. The second property is expected to be true as we may (in all configurations) have the situation that the choice of alert delay causes collisions such that the sink may not be notified of the required change of mode in the current superframe. Of course, the sink may be notified via retransmission of the alert in a later superframe, eventually causing a mode switch (see below).

An example trace demonstrating co-existence of collisions and change of superframe is shown in listing 3. In this example, the nodes 1 and 3 choose the same alert delay (1 ms) and cannot listen to each other. The transmissions therefore collide at the sink. But node 2 which has chosen a different alert delay (2 ms) successfully alerts the sink thus inducing change of superframe. This alert delay choice is done when the model changes location from **StateController** to **SendAlert**.

Finally, we verify a property related to the critical change of state in the protocol from steady to transient. When the data requirement is higher, the protocol should be able to detect and switch accordingly. Also, when a switch fails due to collisions, there should be a possibility to re-use the failed alert to induce change of operational modes. The failed alert is used as a saved alert in the next alert round. We use the query which searches for one example where this occurs.

Critical change of state

$\exists$ i $\in$ nodeid_t:
E$\Diamond$ Node(i).savedAlert && Sink.steady && Sink.changeSuperframe

It should be noted that given the nature of the model, the collisions could occur forever preventing the change of superframe. This means that we cannot show that a state switch will eventually happen.

5.2. Transient model variant

We now consider properties specific for the variant of the model where the sink and nodes starts in transient mode. In the model, transient and steady are boolean variables. In the case where the controller process stays in the transient state permanently, the protocol needs to stay in transient mode of operation to suit the application needs. The given query below checks if there is a path where the system invariantly is in the transient mode.

Remain transient E$\Box$ transient

The second property represents the reachability of steady mode from the starting state, i.e., that it is possible for the system to change mode from transient to steady.

Steady switch E$\Diamond$ steady

5.3. Steady model variant

For the variant of the model that starts in the steady mode, we verify the dual properties of the variant that starts in the transient mode. These two properties are listed below:

Remain steady E$\Box$ steady

Transient switch E$\Diamond$ transient

The two properties check that it is possible to remain in steady mode and that it is possible to switch to transient mode.

5.4. Real-time properties

We now consider real-time properties related to mode switch delay and data communication delay. In order to verify these properties, we use a modified version of the Uppaal model where we have included the use of two watch templates (Watch1 and Watch2) in order to record elapsed time.

The first real-time property that we consider is the switch delay, i.e., the time difference between a detection of threshold breach and the mode switch happening. This switch is required to happen within the duration of a superframe (transient superframe length). This property is specified as follow:

Switch delay

A$\Box$ Watch1.switchDelay $\leq$ superframeLength

We verified the switch delay property by considering a single node farthest from the sink. By symmetry,

the property applies to other nodes at the same level, and also to the parent nodes which (by the tree topology) will have a smaller maximum switch delay.

The second real-time property concerns the data communication delay. It is the time elapsed between the first data sent in the superframe until the last data received. This is required to be within the same superframe. The property is expressed as follows:

Data delay

A□ Watch2.dataDelay $\leq$ superframeLength

All properties listed above evaluate to the expected results. Details on the execution time for the queries based on different configurations of the model are shown in table 1. The verification was conducted on a PC with 4 GB RAM, 2.30 GHz 2-core processor. The query **Collision and mode switch** could be verified only on the configuration with $alertDelay[1,1]$ and resulted in memory exhaust in other configurations. Other queries were verified also on configuration $alertDelay[1,2]$, and $alertDelay[1,4]$ with $i : int[1,2]$.

6. CONCLUSION AND PERSPECTIVES

In this article, we have detailed the modelling and verification process of the DMAMAC protocol. The DMAMAC protocol is designed for process control applications and we have used the Uppaal model-checking tool for modelling and verification. The model consists of a network of timed automata with multiple nodes and a sink operating according to the DMAMAC protocol.

We have explained the model in Uppaal including its modelling elements and templates in detail. The constructed timed automata model includes generic MAC slot operations including data sending and receiving, notification, and sleep. This means that the model can be extended to represent other MAC protocols with similar (and extra) slotting within their superframe. To illustrate the generality of the constructed model, the finite state machine for the protocol model and the possible extensions are shown in Fig. 11. The diagram is divided into three parts: the generic part, DMAMAC extensions, and other possible extensions. The generic part consists of notification and data transfer, which is generally part of a wide range of MAC protocols for WSANs. The DMAMAC extensions with alert sending and receiving parts are specifically relevant for DMAMAC. Given the generic structure, the model can be extended to include other MAC protocol slot types or state types including *Channel Sense*, *Backoff* and *Link establishment*. S-MAC (Ye et al. (2002)) is a one such MAC protocol that uses Request To Send (RTS), Clear To Send (CTS), and carrier sense. The current model can be easily extended to model S-MAC with re-use of generic parts.

We have validated the basic operation of the constructed model using message sequence charts highlighting the most important features, and operations of the protocol including data transfer, alert message functioning, carrier sense, and possibility of collision. Further, we validated the proper operation of the model using the verification engine of Uppaal. The validated model was then verified for the switch procedure and safety properties, including absence of deadlock and other faulty states. The key real-time properties in the form of upper bounds on switch delay and data delay were also verified. Two variants of the model were used for verification and validation, one starting with the transient mode of operation and the other starting with steady mode. Different configurations of the model with varying alert delay were used as a basis for the verification. For the verification, we used a representative node topology that covers all important features of the protocol including existence of sensors and actuators, multi-hop, alert messages, and possibility of collision. The DMAMAC protocol model in Uppaal satisfied the properties considered which increases confidence on the design of the protocol. As a proposed future work, a stochastic model of the DMAMAC protocol to verify the quantitative properties including collision probability, expected switch delay, and energy consumption could provide further insights to the working of the protocol.

Currently, we are also in the process of developing a prototype implementation of the DMAMAC protocol on real hardware. The Uppaal model constructed in this paper serve as an important specification in terms of ensuring the proper and correct implementation of the protocol logic and frame processing.

REFERENCES

Akyildiz, I. F. and I. H. Kasimoglu (2004). Wireless Sensor and Actor Networks: Research challenges. *Ad Hoc Networks 2*(4), 351–367.

Behrmann, G., A. David, and K. Larsen (2004). A tutorial on Uppaal. In M. Bernardo and F. Corradini (Eds.), *Formal Methods for the Design of Real-Time Systems*, Volume 3185 of *Lecture Notes in Computer Science*, pp. 200–236. Springer Berlin Heidelberg.

David, A., K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, J. Vliet, and Z. Wang (2011). Statistical Model Checking for networks of Priced Timed Automata. In *Proceedings of the 9th International Conference on Formal Modeling*

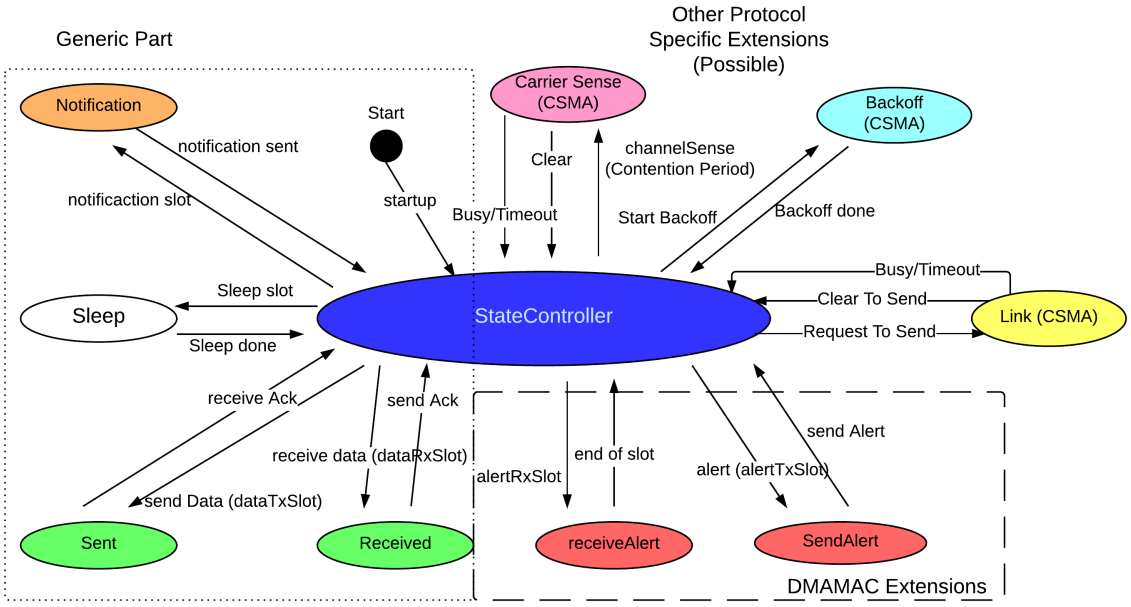

Figure 11: *Generic model with possible extensions*

and Analysis of Timed Systems (FORMATS), Volume 6919 of *LNCS*, pp. 80–96. Springer Berlin Heidelberg.

Fehnker, A., R. van Glabbeek, P. Hfner, A. McIver, M. Portmann, and W. Tan (2012). Automated analysis of AODV using Uppaal. In C. Flanagan and B. Knig (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Volume 7214 of *Lecture Notes in Computer Science*, pp. 173–187. Springer Berlin Heidelberg.

Fehnker, A., L. van Hoesel, and A. Mader (2007). Modelling and Verification of the LMAC protocol for Wireless Ssensor Networks. In J. Davies and J. Gibbons (Eds.), *Integrated Formal Methods*, Volume 4591 of *Lecture Notes in Computer Science*, pp. 253–272. Springer Berlin Heidelberg.

Hespanha, J. P., P. Naghshtabrizi, and Y. Xu (2007). A survey of recent results in Networked Control Systems. Volume 95, pp. 138–162.

Kumar S., A. A., K. Øvsthus, and L. M. Kristensen (2014, August). Towards a Dual-Mode Adaptive Mac Protocol (DMA-MAC) for feedback-based Networked Control Systems. In *The 2nd International Workshop on Communications and Sensor Networks*.

Suriyachai, P., J. Brown, and U. Roedig (2010). Time-critical data delivery in Wireless Sensor Networks. In *Proceedings of DCOSS*, pp. 216–229.

Tschirner, S., L. Xuedong, and W. Yi (2008). Model-based Validation of QoS properties of Biomedical Sensor Networks. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, New York, NY, USA, pp. 69–78. ACM.

Varga, A. and R. Hornig (2008). An overview of the OMNeT++ simulation environment. In *SIMUTOOLS*, pp. 60:1–60:10.

Ye, W., J. Heidemann, and D. Estrin (2002). An energy-efficient mac protocol for wireless sensor networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Volume 3, pp. 1567–1576 vol.3.

Property / Query	Result	CPU Time (s)	Resident Mem. (KB)	Virtual Mem. (KB)
Configuration : alertDelay[1,1], i:[1,2]				
Common queries				
Sink mode	Not Satisfied	718.837	1,795,596	3,595,148
Consistent node mode	Satisfied	876.586	1,772,436	3,573,820
Collision and mode switch	Not Satisfied	711.287	1,797,488	3,599,136
Collision and no mode switch	Satisfied	3.214	21,044	49,512
Critical change of state	Satisfied	33.868	113,084	238,116
Transient specific queries				
Remain transient	Satisfied	0.015	13,820	56,924
Steady switch	Satisfied	0.64	13,888	40,168
Steady specific queries				
Remain steady	Satisfied	0.032	17,424	58,892
Transient switch	Satisfied	1.965	19,308	48,796
Real-time queries				
Switch delay	Satisfied	273.048	440,676	891,152
Data delay	Satisfied	231.302	450,664	905,284
Configuration: alertDelay[1,2], i:[1,2]				
Common queries				
Sink mode	N/A	N/A	Memory exhausted	Memory exhausted
Consistent node mode	N/A	N/A	Memory exhausted	Memory exhausted
Collision and mode switch	Satisfied	8.331	62,292	128,008
Collision and no mode switch	Satisfied	8.346	61,616	129,064
Critical change of state	N/A	N/A	Memory exhausted	Memory exhausted
Transient specific queries				
Remain transient	Satisfied	0.02	13,836	40,092
Steady switch	Satisfied	0.562	13,848	57,012
Steady specific queries				
Remain steady	Satisfied	0.046	36,596	82,700
Transient switch	Satisfied	16.708	66,116	137,096
Real-time queries				
Switch delay	Satisfied	1200.225	1,799,596	3,601,164
Data delay	Satisfied	1068.014	1,799,624	3,622,604

Table 1: *Performance of the protocol verification using Uppaal*